THE ROLE OF CROSS SECTIONAL GEOMETRY IN THE PASSIVE TRACER PROBLEM

Manuchehr Aminian

A dissertation submitted to the faculty at the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Mathematics in the College of Arts and Sciences.

Chapel Hill
2016

Approved by:

Roberto Camassa

Richard McLaughlin

David Adalsteinsson

Laura Miller

Katherine Newhall

ABSTRACT

Manuchehr Aminian: The role of cross sectional geometry in the passive tracer problem
(Under the direction of Roberto Camassa and Richard McLaughlin)

This dissertation is concerned with how the longitudinal moments (mean, variance, skewness) of a tracer distribution undergoing an advective-diffusive process in Poiseiulle flow depend in a nontrivial way upon the cross section of the pipe.

The main focus of this dissertation is on the distribution's skewness, which is the simplest statistic to describe upstream/downstream asymmetry in the tracer distribution. The results of both analysis and numerics show that the distribution's skewness depends significantly on the cross section of the pipe. Typically, cross sections with an exaggerated aspect ratio (e.g., thin ellipses or rectangles) result in negative skewess in the distribution, that is, having a sharp front and a long tail upstream. The opposite is true for nearly circular or square cross sections, with a long tail downstream and the bulk of the distribution upstream. As a result, there are "golden" aspect ratios for each class of cross section – critical aspect ratios which maintain the initial symmetry through the advective timescale – and other critical aspect ratios which symmetrize the distribution at a faster rate than any other aspect ratio on diffusive timescales.

Dedicated to all my friends and family.

ACKNOWLEDGEMENTS

I would like to recognize the significant role my friends played in motivating me to get to work, and the patience of my advisors when I didn't work.

On a more serious note, the significant progress we've made in the work related to this dissertation would not have happened if not for the support of the others in our research group: Francesca Bernardi, Dan Harris, and my advisors, Rich McLaughlin and Roberto Camassa.

Also, the friendly and inclusive environment in the UNC graduate mathematics program kept me mostly sane my years here in Chapel Hill; I'm not sure I would have survived in other departments, based on the horror stories I've heard.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

Introduction

The study of passive tracers under the influence of laminar fluid flow was first brought to the limelight by G.I. Taylor, whose paper in 1953 [1] demonstrated with experiment and theory how the effective diffusivity, as measured by the rate of growth of mean squared displacement of tracer relative to its mean, is much more rapid than would be expected due to draw molecular diffusion when put under the influence of laminar pipe flow. The fundamental result is that the enhancement of diffusivity is proportional to R^2U^2/κ, where R is the pipe radius, U is the characteristic speed of the fluid flow, and κ the molecular diffusivity.

Dynamically, the tracer is asymptotically Gaussian at both very short times and very long times, as measured relative to the diffusive timescale $t_{\mathrm{d}} \propto R^2/\kappa$. However, he observed that at intermediate times the distribution of tracer was highly non-Gaussian.

Since then, many others have studied this result. One main tool came from Aris [2], who showed that the tracer T, whose evolution is modeled by the advection diffusion equation, can more readily be studied by the evolution of its longitudinal (flow-wise) moments, which themselves obey a hierarchy of driven diffusion equations. He, along with many others [3, 4, 5], derived results, both exact and asymptotic in time, for the case of the circular pipe and infinite parallel plate ("channel") flow. In another vein, the long time effective diffusivity was studied for a pipe of general rectangular or elliptical cross section [2, 6].

More recently, the problem has been revisited with the tools of homogenization theory, which have been used to derive more detailed predictions in the channel and circular pipe cases with arbitrary point source release [7, 8], and in the case of pulsatile (time-oscillatory) flows [9].

The problem in shallow rectangular and rectangular-like channels has received renewed interest in the past decade due to the advent of "lab-on-a-chip" devices, which have promise to greatly boost the efficiency and reduce the cost of many clinical trials. The channels in these devices are etched out with a device, which allows free control of the cross section. This naturally leads to the question

of the influence of the cross section on the distribution of the tracer. Several papers have addressed this, but are usually only interested in effective diffusivity [10, 11, 12, 13]. Attention has also been paid in the arena of blood flow and drug delivery [14], despite the necessary assumptions of laminar, Newtonian fluid flow being dubious for blood.

However, very little attention has been paid since the early papers of Aris [2], Chatwin [3], and Barton [5] towards the asymmetry induced by the fluid flow. In the two simplest cases of circular pipe and infinite channel, it turns out that the tracer exhibits opposite signs of the skewness (the centered, normalized third moment) when examining the cross-sectional average.

In other words, despite the flow solutions being mathematically similar, they produce opposite asymmetries in the distribution! This was in fact the original motivation for this dissertation work.

Following from this original question, we were motivated to examine how the skewness behaves for different classes of cross sections to see if we could connect the purely positive skewness of the circular cross section to the purely negative skewness in the channel. To do this, we first examined the family of rectangular and elliptical cross sections, establishing both short time and long time [1] asymptotics of the Aris moment equations. This revealed that the short time skewness is in fact zero for all ellipses, and sign-indefinite for the rectangles, with a "golden" aspect ratio (the ratio of short to long sides) of $\lambda \approx 0.53335$ which has similar statistical behavior to the ellipses. We followed through with the long time analysis to demonsrate that both the rectangles and ellipses have sign-indefinite skewness at long time, separated by an aspect ratio $\lambda \approx 0.49$. We wrote a Monte Carlo method, which shows strong agreement across a wide range of benchmarks, yielding additional support to our results. Finally, we considered a number of extensions to other cross sections which permitted analysis and/or simulation, such as the equilateral triangle, the class of regular polygons, and perturbations of the ellipses, which exhibit a wide range of behaviors.

In broad strokes, the main results of this dissertation work are that the longitudinal skewness of a passive tracer in a laminar fluid flow has strong dependence on the shape of the cross section of the pipe. Depending on the application of interest, this permits a degree of control in how the tracer is delivered to its final destination.

[1] This is measured relative to the generalized diffusive timescale a^2/κ, with a the short transverse length scale

CHAPTER 2

Notation and Setup

In this chapter, we derive the partial differential equation for the fluid velocity in a generic domain and give the solution in a few classes of domains. We also introduce the advection diffusion equation for the tracer.

Derivation of the model equations

First we derive the partial differential equations for the fluid flow. In general, the Navier-Stokes equations describes the evolution of the fluid velocity field $\mathbf{u} = (u, v, w)$ in space and time:

$$\rho\left(\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u}\cdot\nabla\mathbf{u}\right) = -\nabla p + \mu\nabla^2\mathbf{u}, \tag{2.1}$$

with density ρ, pressure field p, and dynamic viscosity μ. Nondimensionalizing the variables as $\mathbf{x} = a\mathbf{x}'$, $\mathbf{u} = U\mathbf{u}'$, $t = (a/U)t'$, $p = (\mu U/a)p'$, with characteristic length and velocity scales a and U respectively, and using a constant density $\rho = \rho_0$ results in the nondimensionalized form

$$\begin{aligned} \mathrm{Re}\left(\frac{\partial \mathbf{u}'}{\partial t'} + \mathbf{u}'\cdot\nabla'\mathbf{u}'\right) &= -\nabla' p' + \nabla^{2'}\mathbf{u}' \\ \nabla'\cdot\mathbf{u}' &= 0. \end{aligned} \tag{2.2}$$

The coefficient $\mathrm{Re} = \rho_0 aU/\mu$ is the Reynolds number. We assume $\mathrm{Re} \ll 1$ and work with the reduced equations (dropping primes)

$$\begin{aligned} \nabla^2\mathbf{u} &= \nabla p, \\ \nabla\cdot\mathbf{u} &= 0. \end{aligned} \tag{2.3}$$

We work with an infinitely long pipe extending in the x direction, with some fixed cross section in the transverse directions. With this setup, assume a constant pressure gradient in the x direction only, that is, $\nabla p = (p_x, 0, 0)$. Additionally there is a no-slip boundary condition on the walls of the

pipe, that is, $\mathbf{u} = \mathbf{0}$ on the boundaries. Explicitly we have

$$\nabla^2 u = p_x, \quad u|_{\partial\Omega} = 0, \tag{2.4a}$$

$$\nabla^2 v = 0, \quad v|_{\partial\Omega} = 0, \tag{2.4b}$$

$$\nabla^2 w = 0, \quad w|_{\partial\Omega} = 0, \text{ with} \tag{2.4c}$$

$$u_x + v_y + w_z = 0. \tag{2.4d}$$

It is a fact that Laplace's equation with zero boundary conditions only has the zero solution, so $v = w = 0$. Incompressibility gives us that $u_x = 0$, so that $u = u(y, z)$. We need only to solve

$$\frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = p_x, \quad u|_{\partial\Omega} = 0, \quad p_x \text{ constant}. \tag{2.5}$$

We note that, while the equations are nondimensional here, an almost identical derivation holds for the dimensional flow, with an extra factor of $1/\mu$ multiplying the pressure gradient. The justification for dropping the inertial terms in the Navier-Stokes equations comes down to a Reynolds-number-like argument, where one would assume inertial effects are negligible compared to the viscous and pressure effects. Functionally the two forms are essentially the same, and we may use the dimensional or nondimensional form as appropriate.

Flow solution in the infinite channel

Working in dimensional variables with a cross section of infinite parallel plates (a "channel")

$$\Omega = \{(y, z) : -a \leq y \leq a\}, \tag{2.6}$$

the boundary conditions for the flow problem

$$\frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = \frac{p_x}{\mu}, \quad u|_{y=\pm a} = 0 \tag{2.7}$$

imply there is no z dependence, so we have an ordinary differential equation for $u(y)$:

$$\frac{d^2 u}{dy^2} = \frac{p_x}{\mu}, \quad u(\pm a) = 0, \tag{2.8}$$

which has the solution

$$u(y) = -\frac{a^2 p_x}{2\mu}\left(1 - (y/a)^2\right). \tag{2.9}$$

We set $U = -a^2 p_x/\mu$ and include an extra factor of 2 in nondimensional form (for convenience) to get a nondimensional flow

$$u(y) = 1 - y^2. \tag{2.10}$$

Flow solution in the circular pipe

Working in dimensional variables with a circular cross section

$$\Omega = \{(y, z) : y^2 + z^2 \leq a^2\}, \tag{2.11}$$

the flow problem is easiest solved using polar coordinates $(y, z) \to (r, \theta)$. Additionally assuming a radially symmetric solution $u(r)$ (justified after the fact by uniqueness of solution to the PDE) the problem reduces to an ODE for $u(r)$:

$$\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial u}{\partial r}\right) = \frac{p_x}{\mu}, \quad u(a) = 0. \tag{2.12}$$

This has a functionally similar solution to the parallel plate case:

$$u = -\frac{a^2 p_x}{4\mu}\left(1 - (r/a)^2\right) = -\frac{a^2 p_x}{4\mu}\left(1 - (y/a)^2 - (z/a)^2\right). \tag{2.13}$$

Letting $U = -a^2 p_x/\mu$ and multiplying by a factor of two gives the nondimensional flow

$$u = \frac{1}{2}(1 - r^2) = \frac{1}{2}(1 - y^2 - z^2). \tag{2.14}$$

Flow solution in the rectangular pipe

In this case, the dimensional domain is

$$\Omega = \{(y, z) : -a \leq y \leq a,\ -b \leq z \leq b\}. \tag{2.15}$$

The flow problem does not reduce as much as in the previous cases:

$$\frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} = \frac{p_x}{\mu}, \quad u|_{y=\pm a} = u|_{z=\pm b} = 0. \tag{2.16}$$

In this case, the solution can be written either as an eigenfunction expansion in the form

$$u(y,z) = \sum_{m,n=1}^{\infty} u_{mn} \cos((m-1/2)\pi y/a) \cos((n-1/2)\pi z/b) \tag{2.17}$$

or in a single series, as a correction of the channel flow. The rough interpretation here is that the channel flow $u_c(y)$ is the limit of the rectangular case if we send the far walls to infinity, that is, $b \to \infty$. This form of the solution is

$$u(y,z) = u_c(y) + \sum_{k=1}^{\infty} c_k \cos((k-1/2)\pi y/a) \cosh((k-1/2)\pi z/a). \tag{2.18}$$

This second form is more convenient in nearly all cases, so we derive the formulae for the coefficients c_k. First, the Poisson equation itself is satisfied, since the PDE is linear, $u_c(y)$ satisfies the equation, and the terms in the summation $\cos((k-1/2)\pi y/a) \cosh((k-1/2)\pi z/a)$ are harmonic. The boundary conditions at $y = \pm a$ are satisfied independently by every term in the expression. Requiring $u|_{z=\pm b} = 0$,

$$0 = u_c(y) + \sum_{k=1}^{\infty} c_k \cos((k-1/2)\pi y/a) \cosh((k-1/2)\pi b/a). \tag{2.19}$$

For convenience define $\phi_k = \cos((k-1/2)\pi y/a$. Multiplying both sides by ϕ_j, integrating over $[-a,a]$, and using the smoothness of the solution to allow the exchange of integration and summation leads to the formula for the coefficients:

$$-\int_{-a}^{a} u_c(y)\phi_j dy = \int_{-a}^{a} \phi_j \sum_{k=1}^{\infty} c_k \cosh((k-1/2)\pi z/a)\phi_k \tag{2.20a}$$

$$-\int_{-a}^{a} u_c(y)\phi_j dy = \sum_{k=1}^{\infty} c_k \cosh((k-1/2)\pi b/a) \int_{-a}^{a} \phi_k \phi_j dy \tag{2.20b}$$

$$\frac{-\int_{-a}^{a} u_c(y)\phi_j \, dy}{\int_{-a}^{a} \phi_j^2 dy} = c_j \cosh((j-1/2)\pi b/a) \tag{2.20c}$$

$$\Rightarrow c_j = \frac{-a^2 p_x}{\mu} \frac{2(-1)^j}{\pi^3 (j-1/2)^3 \cosh((j-1/2)\pi b/a)}. \tag{2.20d}$$

The form of the coefficient gives rapidly converging series, especially for small aspect ratios $\lambda = a/b \ll 1$ even without the factor of $(j-1/2)^{-3}$.

To analyze the behavior for large aspect ratios, let $\zeta = z/b$, $\alpha = (j-1/2)\pi b/a$. Then the aspect ratio dependence in the sum is written as

$$\frac{\cosh(\alpha\zeta)}{\cosh(\alpha)}, \quad -1 \le \zeta \le 1, \tag{2.21}$$

and for $\alpha \to \infty$ (i.e., sending the inverse aspect ratio $b/a \to \infty$) this converges pointwise to zero for $\zeta \in (-1,1)$ and to one for $\zeta = \pm 1$. (This can be shown with standard analysis techniques.)

Multiplying by two and setting $U = -a^2 p_x \mu$ gives the nondimensional form ($\lambda = a/b$)

$$\begin{aligned} u(y,z) &= 1 - y^2 + \sum_{k=1}^{\infty} \tilde{c}_k \cos((k-1/2)\pi y)\cosh((k-1/2)\pi z/\lambda), \\ \tilde{c}_k &= \frac{4(-1)^k}{\pi^3 (k-1/2)^3 \cosh((k-1/2)\pi/\lambda)}. \end{aligned} \tag{2.22}$$

Flow in elliptical pipes

With an elliptical cross section, the domain is defined as

$$\Omega = \left\{ (y,z) : \frac{y^2}{a^2} + \frac{z^2}{b^2} \le 1 \right\}. \tag{2.23}$$

The flow solution will take the form

$$u = k_1 \left(1 - (y/a)^2 - (z/b)^2\right), \tag{2.24}$$

which enforces the boundary conditions. To find the scaling, substitute into the PDE and solve:

$$k_1(-2/a^2 - 2/b^2) = \frac{p_x}{\mu} \quad \Rightarrow \quad k_1 = \frac{-a^2 p_x}{\mu} \frac{1}{2(1+(a/b)^2)}. \tag{2.25}$$

Multiplying by two and setting $U = -a^2 p_x/\mu$ yields the nondimensional form for arbitrary aspect ratio $\lambda = a/b$:

$$u(y,z) = \frac{1}{1+\lambda^2}(1 - y^2 - \lambda^2 z^2). \tag{2.26}$$

Advection diffusion equation

A general tracer density $T(\mathbf{x}, t)$ can possibly influence the fluid flow, and would enter the Navier-Stokes equations through additional forcing terms. A key assumption behind this field is that the tracer is *passive*: that is, it has no affect on the fluid flow itself, and is only advected along by it. This allows us to separately solve for the fluid flow $\mathbf{u}$, then use this to analyze the tracer distribution. If we also assume a simple molecular diffusion, one can derive the advection diffusion equation

$$\frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T = \kappa \nabla^2 T. \tag{2.27}$$

In our case with steady laminar flow $\mathbf{u} = u(y, z)\mathbf{i}$, this reduces to

$$\frac{\partial T}{\partial t} + u(y, z)\frac{\partial T}{\partial x} = \kappa \nabla^2 T. \tag{2.28}$$

A reasonable assumption in the case of pipe flow is that no tracer exits the pipe; this are no-flux, or Neumann, boundary conditions. If we consider an arbitrary point on the boundary of the pipe, the directional derivative of T in the direction perpendicular to the boundary must be zero:

$$D_{\mathbf{n}} T = \mathbf{n} \cdot \nabla T = 0. \tag{2.29}$$

In the case of the channel, circular pipe, and rectangular this boundary condition is:

$$\begin{aligned}
\left.\frac{\partial T}{\partial y}\right|_{y=\pm a} &= 0 \quad \text{(Channel)} \\
\left.\frac{\partial T}{\partial r}\right|_{r=a} &= 0 \quad \text{(Circular pipe)} \\
\left.\frac{\partial T}{\partial y}\right|_{y=\pm a} = \left.\frac{\partial T}{\partial z}\right|_{z=\pm b} &= 0 \quad \text{(Rectangular pipe)} \\
\left.\frac{y}{a^2}\frac{\partial T}{\partial y} + \frac{z}{b^2}\frac{\partial T}{\partial z}\right|_{(y,z)\in\partial\Omega} &= 0 \quad \text{(Elliptical pipe)}
\end{aligned} \tag{2.30}$$

The boundary conditions for the elliptical case can be handled similar to the circular pipe, but require using elliptical coordinates. This will be explored in depth in chapter 3, where we analyze the long time asymptotics of the Aris equations.

CHAPTER 3

Asymptotics of the Aris equations

The Aris equations.

To rigorously describe and predict the phenomenon of effective diffusivity in pipe flow, Aris showed in [2] that found that one could write down a recursive system of partial differential equations for the x-moments of the tracer T. Define the moments

$$T_n(y,z,t) \equiv \frac{\int_{-\infty}^{\infty} x^n T(x,y,z,t)dx}{\int_{-\infty}^{\infty} T(x,y,z,t)dx}, \quad n = 1,2,... \tag{3.1}$$

The equations are derived by taking the advection-diffusion equation, multiplying by x^n, and integrating (similarly for the initial condition). For $n = 0$,

$$\int_{-\infty}^{\infty} \left[\frac{\partial T}{\partial t} + u(y,z)\frac{\partial T}{\partial x} = \kappa \nabla^2 T \right] dx \tag{3.2a}$$

$$\frac{\partial T_0}{\partial t} + u(y,z)\int_{-\infty}^{\infty} \frac{\partial T}{\partial x}dx = \kappa \int_{-\infty}^{\infty} \frac{\partial^2 T}{\partial x^2}dx + \frac{\partial^2 T_0}{\partial y^2} + \frac{\partial^2 T_0}{\partial z^2}, \text{ giving} \tag{3.2b}$$

$$\frac{\partial T_0}{\partial t} = \kappa \nabla_\perp^2 T_0, \qquad T_0(y,z,0) = f_0(y,z), \qquad \left.\frac{\partial T_0}{\partial \mathbf{n}}\right|_{\partial\Omega} = 0, \tag{3.2c}$$

where $\nabla_\perp^2$ is the Laplacian in the transverse directions y and z. Averaging through the cross section and applying the boundary conditions with the divergence theorem gives a conservation equation:

$$\frac{1}{|\Omega|}\int_\Omega \left[\frac{\partial T_0}{\partial t} = \kappa \nabla_\perp^2 T_0 \right] dA, \qquad \frac{1}{|\Omega|}\int_\Omega \left[T_0(y,z,0) = f_0(y,z) \right] dA \tag{3.3a}$$

$$\Rightarrow \frac{d\mathcal{M}_0}{dt} = 0, \qquad \mathcal{M}_0(0) = \frac{1}{|\Omega|}\int_\Omega f_0(y,z)dA = const. \tag{3.3b}$$

A similar argument can be done to arrive at the equation for T_1, using an integration by parts along the way:

$$\frac{\partial T_1}{\partial t} - \kappa \nabla_\perp^2 T_1 = u(y,z) T_0(y,z,t), \qquad T_1(y,z,0) = f_1(y,z), \qquad \left.\frac{\partial T_1}{\partial \mathbf{n}}\right|_{\partial\Omega} = 0. \tag{3.4}$$

In the special case of an initial distribution uniform in the cross section (a function of x only), $T_0(y,z,t) = const.$ and the T_1 equation simplifies to (setting $T_0(y,z,t) = 1$)

$$\frac{\partial T_1}{\partial t} - \kappa \nabla_\perp^2 T_1 = u(y,z), \qquad T_1(y,z,0) = f_1(y,z), \qquad \left.\frac{\partial T_1}{\partial \mathbf{n}}\right|_{\partial\Omega} = 0. \tag{3.5}$$

The n-th moment equation can be derived generally by the same arguments, arriving at

$$\frac{\partial T_n}{\partial t} - \kappa \nabla_\perp^2 T_n = n\, u(y,z) T_{n-1} + n(n-1) T_{n-2}, \qquad T_n(y,z,0) = f_n(y,z), \qquad \left.\frac{\partial T_n}{\partial \mathbf{n}}\right|_{\partial\Omega} = 0, \tag{3.6}$$

for any $n = 0, 1, ...$ Generically denoting the average $\langle g \rangle \equiv \int_\Omega g dA / |\Omega|$, the equations for the cross-sectionally averaged moments $\mathcal{M}_n$ (again taking advantage of the divergence theorem) are

$$\frac{d\mathcal{M}_n}{dt} = \kappa\, n(n-1) \mathcal{M}_{n-2} + n \langle u\, T_{n-1} \rangle, \qquad \mathcal{M}_n(y,z,0) = \langle f_n \rangle. \tag{3.7}$$

Explicitly, the first few full moments equations for the case of cross-sectionally uniform initial data are (with $\mathcal{M}_0 \equiv 1$)

$$\frac{\partial T_1}{\partial t} - \kappa \nabla_\perp^2 T_1 = u \tag{3.8a}$$

$$\frac{\partial T_2}{\partial t} - \kappa \nabla_\perp^2 T_2 = 2\kappa + 2 u T_1 \tag{3.8b}$$

$$\frac{\partial T_3}{\partial t} - \kappa \nabla_\perp^2 T_3 = 6\kappa T_1 + 3 u T_2, \tag{3.8c}$$

and without loss of generality assuming $\langle u(y,z) \rangle = 0$ (by working in the reference frame of the mean velocity) and $\mathcal{M}_0 \equiv 1$ the corresponding cross-sectionally averaged moments analagous to Barton's

[5] equations are

$$\frac{\partial \mathcal{M}_1}{\partial t} = 0, \tag{3.9a}$$
$$\frac{\partial \mathcal{M}_2}{\partial t} = 2\kappa + 2\langle uT_1 \rangle, \tag{3.9b}$$
$$\frac{\partial \mathcal{M}_3}{\partial t} = 6\kappa \mathcal{M}_1 + 3\langle uT_2 \rangle. \tag{3.9c}$$

We often nondimensionalize using the timescale $t = (a^2/\kappa)t'$, $\mathbf{x} = a\mathbf{x}'$, $u = Uu'$, in which case the Aris equations written above are modified by dropping κ and inserting a factor of the Péclet number wherever u is seen.

Short time asymptotics of the Aris equations.

Exact moments without diffusion.

When we work with advection-diffusion in the limit of large Péclet number, there is a range of timescales in which the behavior is essentially advective alone. This can be seen by nondimensionalizing the advection diffusion equation as $\mathbf{x} = a\mathbf{x}'$, $t = (a/U)t'$, $u = Uu'$, which results in the equation

$$\frac{\partial T}{\partial t'} + u'(y', z')\frac{\partial T}{\partial x'} = \frac{1}{\mathrm{Pe}}\Delta' T, \qquad T(\mathbf{x}', 0) = f(\mathbf{x}'), \qquad \left.\frac{\partial T}{\partial \mathbf{n}'}\right|_{\partial\Omega'} = 0, \tag{3.10}$$

with $\mathrm{Pe} = Ua/\kappa$ the Péclet number. In the infinite Péclet limit, the right hand side drops out, and this reduces to an advection equation

$$\frac{\partial T}{\partial t'} + u'(y', z')\frac{\partial T}{\partial x'} = 0, \qquad T(\mathbf{x}', 0) = f(\mathbf{x}'), \qquad \left.\frac{\partial T}{\partial \mathbf{n}'}\right|_{\partial\Omega'} = 0, \tag{3.11}$$

If we neglect the boundary conditions, the advection equation can be solved with method of characteristics

$$T(\mathbf{x}', t') = f(\mathbf{x} - u'(y', z')t'\,\mathbf{i}). \tag{3.12}$$

Now we would like to compute the x moments of this distribution. Change variables to the local

coordinate $\xi = x' - u'(y', z')t'$, drop primes, and calculate the n-th pointwise moment:

$$\begin{aligned} m_0(y,z,t) &= \int_{-\infty}^{\infty} T(\mathbf{x},t)dx, \\ m_n(y,z,t) &= \frac{1}{m_0}\int_{-\infty}^{\infty} x^n f(\mathbf{x} - u(y,z)t\mathbf{i})dx \\ &= \frac{1}{m_0}\int_{-\infty}^{\infty} (\xi + u(y,z)t)^n f(\xi,y,z)d\xi \\ &= \sum_{j=0}^{n}\binom{n}{j}(ut)^{n-j}\frac{\int_{-\infty}^{\infty}\xi^j f(\xi,y,z)d\xi}{m_0} \\ &= (ut)^n + \sum_{j=1}^{n}\binom{n}{j}(ut)^{n-j}m_j(y,z,0), \qquad n \geq 1. \end{aligned} \tag{3.13}$$

In words, due to pure advection, the pointwise moments $m_j(y,z,t)$ are carried by linear combinations of the moments of the initial conditions. Additionally, the n-th moment only depends on the intial moments up to and including itself.

Pointwise statistics of a passive tracer with advection alone.

Denote $m_n(y,z,0) = m_n|_0$. Then for the first few moments we have

$$\begin{aligned} m_0(y,z,t) &= m_0|_0, \\ m_1(y,z,t) &= m_1|_0 + ut, \\ m_2(y,z,t) &= m_2|_0 + 2(ut)\, m_1|_0 + (ut)^2, \\ m_3(y,z,t) &= m_3|_0 + 3(ut)\, m_2|_0 + 3(ut)^2 m_1|_0 + (ut)^3. \end{aligned} \tag{3.14}$$

Define the pointwise central moments

$$\mu_n(y,z,t) = \frac{1}{m_0}\int_{-\infty}^{\infty}(x-\mu_1)^n\, dx, \quad n \geq 1. \tag{3.15}$$

The second and third central moments can then be computed:

$$\mu_2 = \frac{1}{m_0}\int_{-\infty}^{\infty}(x-m_1)^2 T(x,y,z,t)dx = m_2 - 2m_1^2 + m_1^2 = m_2 - m_1^2 \tag{3.16a}$$

$$= m_2|_0 + 2(ut)m_1|_0 + (ut)^2 - [m_1|_0 + ut]^2 \tag{3.16b}$$

$$= m_2|_0 - m_1|_0^2 \tag{3.16c}$$

$$= \mu_2(y,z,0), \tag{3.16d}$$

and

$$\mu_3 = \frac{1}{m_0}\int_{-\infty}^{\infty}(x-m_1)^3 T(x,y,z,t)dx = m_3 - 3m_1m_2 + 2m_1^3 \tag{3.17a}$$

$$= \Big[m_3|_0 + 3(ut)\, m_2|_0 + 3(ut)^2 m_1|_0 + (ut)^3\Big] \tag{3.17b}$$

$$\quad - 3\Big[m_1|_0 + ut\Big]\Big[m_2|_0 + 2(ut)m_1|_0 + (ut)^2\Big] + 2\Big[m_1|_0 + ut\Big]^3$$

$$= m_3|_0 - 3m_1|_0 m_2|_0 + 2m_1|_0^3 \tag{3.17c}$$

$$= \mu_3(y,z,0). \tag{3.17d}$$

The pointwise skewness is then

$$Sk(y,z,t) = \frac{m_3|_0 - 3m_1|_0 m_2|_0 + 2m_1|_0^3}{\big(m_2|_0 - m_1|_0^2\big)^{3/2}} = Sk(y,z,0). \tag{3.18}$$

In short, this says that the pointwise central statistics of the initial condition do not change in the absence of diffusion. This is perhaps unsurprising, since the diffusionless system can be interpreted as an infinite system of independent constant coefficient advection equations, which for each (y,z) only shifts the initial distribution at a constant rate $u(y,z)t$. The story is not as simple for the distribution after averaging in the cross section, which we show below.

Averaged statistics of a passive tracer with pure advection.

Denote angle brackets $\langle\cdot\rangle$ the average in y and z in the cross section:

$$\langle g(y,z)\rangle = \frac{\int_\Omega g(y,z)dA}{\int_\Omega 1 dA}. \tag{3.19}$$

Then we can look at the behavior of the cross-sectionally averaged distribution in a similar manner:

$$\bar{m}_n(t) \equiv \frac{1}{\bar{m}_0} \int_{-\infty}^{\infty} x^n \langle f(x - ut, y, z) \rangle dx \tag{3.20}$$

Exchanging the order of integration and taking the total mass $\bar{m}_0 \equiv 1$ gives

$$\bar{m}_n = \left\langle \int_{-\infty}^{\infty} x^n f(x - ut, y, z) \right\rangle = \langle m_n \rangle \tag{3.21}$$

$$= \left\langle (ut)^n + \sum_{j=1}^{n} \binom{n}{j} (ut)^{n-j} m_j(y, z, 0) \right\rangle \tag{3.22}$$

$$= \langle (ut)^n \rangle + \sum_{j=1}^{n} \binom{n}{j} \langle (ut)^{n-j} m_j|_0 \rangle. \tag{3.23}$$

The first few moments of the averaged, diffusionless tracer distribution are

$$\bar{m}_1 = \langle ut \rangle + \bar{m}_1|_0, \tag{3.24a}$$

$$\bar{m}_2 = \langle (ut)^2 \rangle + \langle ut\, m_1|_0 \rangle + \bar{m}_2|_0, \tag{3.24b}$$

$$\bar{m}_3 = \langle (ut)^3 \rangle + 3\langle (ut)^2\, m_1|_0 \rangle + 3\langle ut\, m_2|_0 \rangle + \bar{m}_3|_0, \tag{3.24c}$$

and the corresponding central statistics $\bar{\mu}_n$ are

$$\begin{aligned} \bar{\mu}_2 &= \bar{m}_2 - \bar{m}_1^2 = \langle u^2 \rangle t^2 + \langle ut\, m_1|_0 \rangle + \bar{m}_2|_0 - \langle ut \rangle^2 - 2\langle ut \rangle \bar{m}_1|_0 - \bar{m}_1|_0^2 \\ &= \Big[\langle u^2 \rangle - \langle u \rangle^2 \Big] t^2 + \Big[\langle u\, m_1|_0 \rangle + 2\langle u \rangle \bar{m}_1|_0 - 2\langle u \rangle m_1|_0 \Big] t + \Big[\bar{m}_2|_0 - \bar{m}_1|_0^2 \Big], \end{aligned} \tag{3.25a}$$

$$\begin{aligned} \bar{\mu}_3 &= \bar{m}_3 - 3\bar{m}_1\bar{m}_2 + 2\bar{m}_1^3 \\ &= \Big[\langle u^3 \rangle - 3\langle u^2 \rangle \langle u \rangle + 2\langle u \rangle^3 \Big] t^3 + \Big[3\langle u^2 m_1|_0 \rangle - 3\langle u^2 \rangle \bar{m}_1|_0 - 3\langle u \rangle \langle u\, m_1|_0 \rangle + 3\langle u \rangle^2 \bar{m}_1|_0 \Big] t^2 \\ &\quad + \Big[3\langle u\, m_2|_0 \rangle - 3\langle u\, m_1|_0 \rangle \bar{m}_1|_0 - 3\langle u \rangle \bar{m}_2|_0 + 3\langle u \rangle \bar{m}_1|_0^2 \Big] t + \Big[\bar{m}_3|_0 - 3\bar{m}_1|_0 \bar{m}_2|_0 + 2\bar{m}_1|_0^3 \Big]. \end{aligned} \tag{3.25b}$$

The resulting skewness can be examined at short and long times. For $t \ll 1$, we get

$$Sk(t) \sim \frac{\bar{m}_3|_0 - 3\bar{m}_1|_0 \bar{m}_2|_0 + 2\bar{m}_1|_0^3}{\left(\bar{m}_2|_0 - \bar{m}_1|_0^2\right)^{3/2}} + \mathcal{O}(t) = Sk(0) + \mathcal{O}(t), \quad t \to 0, \tag{3.26}$$

perturbing off the skewness of the initial condition. For $t \gg 1$, the t^2 and t^3 terms dominate the second and third central moments respectively, giving a constant, generally non-zero result

$$Sk(t) \sim \frac{\langle u^3 \rangle - 3\langle u^2 \rangle \langle u \rangle + 2\langle u \rangle^3}{\left(\langle u^2 \rangle - \langle u \rangle^2\right)^{3/2}} + \mathcal{O}(1/t), \quad t \to \infty, \tag{3.27}$$

and if we take $\langle u \rangle = 0$ by working in a reference frame of the average velocity, this simplifies to

$$Sk(t) \sim \frac{\langle u^3 \rangle}{\langle u^2 \rangle^{3/2}} + \mathcal{O}(1/t), \quad t \to \infty. \tag{3.28}$$

This quantity depends only on the flow, which in turn is given by the solution to the Poisson problem, which is a function of the cross sectional geometry of the pipe. We have termed this the *geometric skewness* as a result. While this derivation was as an infinite time limit, in reality it is seen on advective timescales, which will be much shorter than the diffusive timescale if Pe $\gg 1$.

If we assume the initial condition $f(x, y, z)$ is uniform in the cross section, symmetric about $x = 0$, with variance σ^2, we get $\bar{m}_1|_0 = m_1|_0 = 0$, $\bar{m}_2|_0 = m_2|_0 = \sigma^2$, and $\bar{m}_3|_0 = m_3|_0 = 0$. Taking this with $\langle u \rangle = 0$ greatly simplifies the central moments and skewness:

$$\bar{\mu}_2(t) = \langle u^2 \rangle t^2 + \sigma^2, \tag{3.29a}$$

$$\bar{\mu}_3(t) = \langle u^3 \rangle t^3, \tag{3.29b}$$

$$Sk(t) = \frac{\langle u^3 \rangle t^3}{\left(\sigma^2 + \langle u^2 \rangle t^2\right)^{3/2}}. \tag{3.29c}$$

Dividing the numerator and denominator of the skewness by t^3 gives

$$Sk(t) = \frac{\langle u^3 \rangle}{\left(\sigma^2/t^2 + \langle u^2 \rangle\right)^{3/2}}. \tag{3.30}$$

Then the onset to geometric skewness will occur at on a timescale when

$$\sigma^2/t^2 \ll \langle u^2 \rangle, \quad \text{or} \quad t \gg \sigma/\sqrt{\langle u^2 \rangle}. \tag{3.31}$$

This is essentially the time needed for the flow to overcome the characteristic width σ of the initial condition.

Short time asymptotics with diffusion.

Now we introduce a generic process discussed in [15] to calculate the short time behavior for the moments in the presence of diffusion. The method is based on modifying a two term series in time for T_n to asymptotically obey the averaged moments equations for $\mathcal{M}_n$. In this context, the equations for $\mathcal{M}_n$ can be thought of as a net conservation equation for T_n. In this section, we work with a strip initial condition $\delta(x)$ unless stated otherwise.

Given the exact formulae for the moments in the channel, and the equivalent Poisson summed version, we have done a study of the behavior on short timescales to inform the type of correction necessary.

The Poisson summation shows the structure of the pointwise moment $T_1(y,t)$ at short time can be interpreted as a lattice of scaled heat kernels $G(x-x_k,t)$, with lattice $\{x_k = 2k-1,\, k\in\mathbb{Z}\}$. With this in mind, a formal approach to the short time for general cross section can be formed.

We demonstrate this in the channel and will generalize to generic domain after. We would like a short time approximation for the problem

$$\frac{\partial T_1}{\partial t} - \frac{\partial^2}{\partial y^2}T_1 = u(y), \qquad \left.\frac{\partial T_1}{\partial y}\right|_{y=\pm 1} = 0, \qquad T_1(y,0) = 0. \tag{3.32}$$

Start with a generic time expansion of the first moment $T_1(y,t)$ and seek appropriate coefficients:

$$T_1(y,t) \sim a_1(y)t + a_2(y)\frac{t^2}{2}. \tag{3.33}$$

Substitution into the Aris equation and matching in powers of t gives

$$a_1 = u, \quad a_2 = -\tfrac{\partial^2}{\partial y^2}u = const. \tag{3.34}$$

However, this solution violates the conservation of $\int_{-1}^{1} T_1 dy$ required of the full solution:

$$\begin{aligned} &\int_{-1}^{1}\frac{\partial T_1}{\partial t} - \int_{-1}^{1}\frac{\partial^2}{\partial y^2}T_1 dy = \int_{-1}^{1} u dy && && \frac{d}{dt}\int_{-1}^{1}\left(a_1 t + a_2 t^2/2\right)dy \\ &\frac{d\left(\int_{-1}^{1} T_1 dA\right)}{dt} - \left[\left.\frac{\partial T_1}{\partial \mathbf{y}}\right|_{-1}^{1}\right] = 0 && \text{whereas} && = \int_{-1}^{1}\left(u + (-\frac{\partial^2}{\partial y^2}u)t\right)dy \\ &\Rightarrow \frac{d\left(\int_{-1}^{1} T_1 dy\right)}{dt} = 0, && && = 2(-\frac{\partial^2}{\partial y^2}u)t \neq 0. \end{aligned} \tag{3.35}$$

We seek a term $k(y)$ to add which corrects the conservation requirement, but preserves the short time dynamics, which are primarily advective:

$$\frac{t^2}{2}\int_{-1}^{1} k(y) + \left(-\frac{\partial^2}{\partial y^2}u\right)\, dy = 0 \tag{3.36}$$

$$\int_{-1}^{1} k(y)\, dy = \int_{-1}^{1} \frac{\partial^2}{\partial y^2}u\, dy = \left[\, \frac{\partial u}{\partial y}\Big|_{-1}^{1}\,\right]. \tag{3.37}$$

While any choice of $k(y)$ satisfying the integral requirement will restore conservation, analysis of the channel solution reveals boundary layers which evolve characteristically like heat kernels for $t > 0$, and in the limit $t \to 0$, form a sequence converging to delta functions. Therefore, one choice of correction in this small, but positive time regime would be

$$\tilde{k}(y,t) = c_1 G(y-1^-,t) + c_2 G(y+1^+,t) = c_1 \frac{e^{-\frac{(y-1^-)^2}{4t}}}{\sqrt{4\pi t}} + c_2 \frac{e^{-\frac{(y+1^+)^2}{4t}}}{\sqrt{4\pi t}}. \tag{3.38}$$

For $t \ll 1$, each heat kernel integrates to 1 up to exponentially small corrections. Let $k(y) = \lim_{t\to 0^+} \tilde{k}(y,t)$, so

$$\int_{-1}^{1} k(y)dy = c_1 + c_2 = \left[\, \frac{\partial u}{\partial y}\Big|_{-1}^{1}\,\right]. \tag{3.39}$$

An appropriate choice is then setting

$$c_1 = \frac{\partial u}{\partial y}\bigg|_{1}, \qquad c_2 = -\frac{\partial u}{\partial y}\bigg|_{-1}, \tag{3.40}$$

and if $u = 1/3 - y^2$, this gives $c_1 = c_2 = -2$. Finally, we can verify the modified quadratic term obeys conservation at short time (using $-\nabla^2 u = 2$):

$$\frac{t^2}{2}\int_{-1}^{1} a_2(y)dy = \frac{t^2}{2}\lim_{t\to 0^+}\int_{-1}^{1}\left[-\frac{\partial^2}{\partial y^2}u - 2(G(y-1^-,t) + G(y+1^+,t))\right]dy \tag{3.41a}$$

$$= \frac{t^2}{2}\left[(1-(-1))(2) - 2\left[1+1\right]\right] = 0. \tag{3.41b}$$

The formal asymptotics at $t = 0$ are

$$T_1 \sim u(y)t + \left[-\frac{\partial^2}{\partial y^2}u - 2(\delta(y-1^-) + \delta(y+1^+))\right]\frac{t^2}{2}, \tag{3.42}$$

but they may be extended to $0 < t \ll 1$ by replacing the delta functions with the heat kernels appropriately. In a generic domain, similar arguments lead to the asymptotic

$$T_1(y,z,t) \sim u(y,z)t + \left[-\nabla^2 u + k(y,z)\right] \frac{t^2}{2}. \tag{3.43}$$

The generic correction here can be found starting with the divergence theorem when integrating $-\nabla^2 u$:

$$\int_\Omega -\nabla^2 u \, dA = \int_S -\nabla u \cdot \mathbf{n} \, ds \tag{3.44}$$

Observe that for any $g(y,z)$, with Ω a bounded domain and S is boundary,

$$\int_S g(y,z)\, ds = \int_\Omega g(y,z) \left[\int_S \delta(y-y')\delta(z-z') dy' dz' \right] \, dA, \tag{3.45}$$

so that the boundary integral can be written as an area integral using delta functions laid on the boundary:

$$\begin{aligned} \int_\Omega -\nabla^2 u \, dA = \int_S -\nabla u \cdot \mathbf{n} \, ds &= \int_\Omega -\frac{\partial u}{\partial \mathbf{n}} \left[\int_S \delta(y-y')\delta(z-z') dy' dz' \right] \, dA \\ &= \int_\Omega \left[-\int_S \frac{\partial u}{\partial \mathbf{n}} \delta(y-y')\delta(z-z') dy' dz' \right] \, dA, \end{aligned} \tag{3.46}$$

suggesting the general correction term

$$k(y,z) = + \int_S \frac{\partial u}{\partial \mathbf{n}} \delta(\mathbf{y} - \mathbf{y}') d\mathbf{y}'. \tag{3.47}$$

The same idea is carried forward and applied to the higher moments equations: beginning with a general two-term expansion in time, analyzing the conservation principle for the equation, and adding similar boundary terms where needed to correct. Denoting $\mathcal{L}(u)$ the nondimensional Laplacian [1] , so that typically for us $\mathcal{L}(u) = -2$, and $\tilde{u}$ the lab-frame flow (with $\tilde{u}|_{\partial\Omega} = 0$), the following short time

[1] We leave the term $\mathcal{L}(u)$ generic for the purposes of comparison to other formulae in the literature. There is an unfortunate zoo of conventions used for both the flow formulae and prototypical domain, which make comparison and cross-validation more difficult.

asymptotics are derived:

$$\mathcal{M}_1 \sim 0, \tag{3.48}$$

$$\mathcal{M}_2 \sim 2t + \mathrm{Pe}^{\,2}\langle u^2\rangle t^2 + \frac{1}{3}\mathrm{Pe}^{\,2}\mathcal{L}(u)\langle\tilde{u}\rangle t^3, \tag{3.49}$$

$$\mathcal{M}_3 \sim \mathrm{Pe}^{\,3}\langle u^3\rangle t^3 + \frac{1}{2}\mathrm{Pe}^{\,3}\mathcal{L}(u)(\langle\tilde{u}^2\rangle - 2\langle\tilde{u}\rangle^2)t^4, \tag{3.50}$$

$$\begin{aligned}\mathcal{M}_4 \sim\ & 12t^2 + 12\mathrm{Pe}^{\,2}\langle u^2\rangle t^3 + \Big[\mathrm{Pe}^{\,4}\langle u^4\rangle + 4\mathrm{Pe}^{\,2}\mathcal{L}(u)\langle\tilde{u}\rangle\Big]t^4 \\ & + \Big[\frac{1}{5}\mathrm{Pe}^{\,4}\mathcal{L}(u)\langle u^3\rangle + \frac{1}{5}\mathrm{Pe}^{\,4}\mathcal{L}(u)\langle\tilde{u}\rangle^3 - \frac{7}{5}\mathrm{Pe}^{\,4}\langle u^2|\nabla u|^2\rangle\Big]t^5.\end{aligned} \tag{3.51}$$

In particular, for the skewness, we substitute to obtain short time behavior

$$Sk(t) \sim \frac{\mathrm{Pe}^{\,3}\Big[\langle u^3\rangle t^3 + \frac{1}{2}\mathcal{L}(u)(\langle\tilde{u}^2\rangle - 2\langle\tilde{u}\rangle^2 t^4\Big]}{\big(2t + \mathrm{Pe}^{\,2}\langle u^2\rangle t^2 + \frac{1}{3}\mathrm{Pe}^{\,2}\mathcal{L}(u)\langle\tilde{u}\rangle t^3\big)^{3/2}}. \tag{3.52}$$

For $\mathrm{Pe} \gg 1$ and taking the leading order short time terms once again gives geometric skewness:

$$\lim_{t\to 0}\lim_{\mathrm{Pe}\to\infty} Sk(t) = \frac{\langle u^3\rangle t^3}{(\langle u^2\rangle t^2)^{3/2}} = \frac{\langle u^3\rangle}{\langle u^2\rangle^{3/2}}. \tag{3.53}$$

We can additionally predict the onset of nonzero skewness. Differentiation of the generic skewness formula gives

$$\dot{Sk}(t) = \frac{2\mathcal{M}_2\dot{\mathcal{M}}_3 - 3\mathcal{M}_3\dot{\mathcal{M}}_2}{2\mathcal{M}_2^{5/2}}, \tag{3.54}$$

so that we need to solve $2\mathcal{M}_2\dot{\mathcal{M}}_3 - 3\mathcal{M}_3\dot{\mathcal{M}}_2 = 0$. Substitution of the asymptotic formulae gives a rootfinding problem

$$c_3t^3 + c_2t^2 - \frac{1}{\mathrm{Pe}^{\,2}}[c_1t + c_0] = 0, \tag{3.55}$$

with coefficients

$$c_0 = 36\langle u^3\rangle \tag{3.56a}$$

$$c_1 = 30\mathcal{L}(u)(\langle\tilde{u}^2\rangle - 2\langle\tilde{u}\rangle^2), \tag{3.56b}$$

$$c_2 = 6\mathcal{L}(u)(\langle u^3\rangle\langle\tilde{u}\rangle - \langle u^2\rangle(\langle\tilde{u}^2\rangle - 2\langle\tilde{u}\rangle^2), \tag{3.56c}$$

$$c_3 = \mathcal{L}(u)\langle\tilde{u}\rangle(\langle\tilde{u}^2\rangle - 2\langle\tilde{u}\rangle^2). \tag{3.56d}$$

Applying perturbative rootfinding techniques gives a prediction for the critical point

$$t^* \sim \mathrm{Pe}^{-1}\sqrt{c_0/c_2} \text{ as } \mathrm{Pe} \to \infty. \tag{3.57}$$

This prediction for the timescale is shown in conjunction with the geometric skewness using large X's in figure 3.2. There appears to be good agreement, though a careful study across aspect ratios and time scales has not been done.

Comparison of short time asymptotics and simulation.

In figure 3.2 we compare the results of the short time analysis developed in this chapter to Monte Carlo simulation. The left panel sweeps over a range of aspect ratios, with a fixed Péclet value of $\mathrm{Pe} = 10^4$. The simulations (dotted) and the asymptotics (3.52) (solid) show excellent agreement at short time. Combining the geometric skewness value with the prediction for the critical point (3.57) are overlaid as large crosses. The effect of nonzero variance $\sigma^2 > 0$ in the initial condition is investigated in the inset panel, where the characteristic width is chosen as $\sigma \approx 0.115$. A delay of the onset of nonzero skewness indicated in (3.31) is consistent, since in nondimensional terms, we have $|u| \propto \mathrm{Pe}$, predicting a timescale $t \gg 0.115/\mathrm{Pe} \approx 10^{-5}$ past which geometric skewness will be seen. This is in contrast to the delta initial data, which generically sees the onset of skewness at a nondimensional timescale $t \propto \mathrm{Pe}^{-2}$ [8].

The geometric skewness as a function of the aspect ratio for the rectangles is shown in the center panel, and horizontal lines are drawn connecting this skewness to the corresponding short time behavior in the presence of diffusion for the same aspect ratios. Computing the aspect ratio with zero geometric skewness was done using a numerical rootfinding method applied to $\langle u(y,z;\lambda)^3\rangle$, giving the value $\lambda \approx 0.53335$ which was used for the simulations.

In the right panel we plot the corresponding set of simulations in the class of ellipses. Interestingly, the short time behavior is symmetric independent of the aspect ratio. This is seen numerically, as well as via direct computation showing $\langle u^3\rangle = 0$ independent of aspect ratio. Because of this, our formal short time asymptotics do not give any useful information about the skewness, while in the rectangles we have information at times up to $t \propto \mathrm{Pe}^{-1}$. It is also noteworthy that the circular pipe (yellow) has a larger positive skewness than the corresponding square. One might postulate the circular cross section in fact produces the largest positive skewness of any cross section.

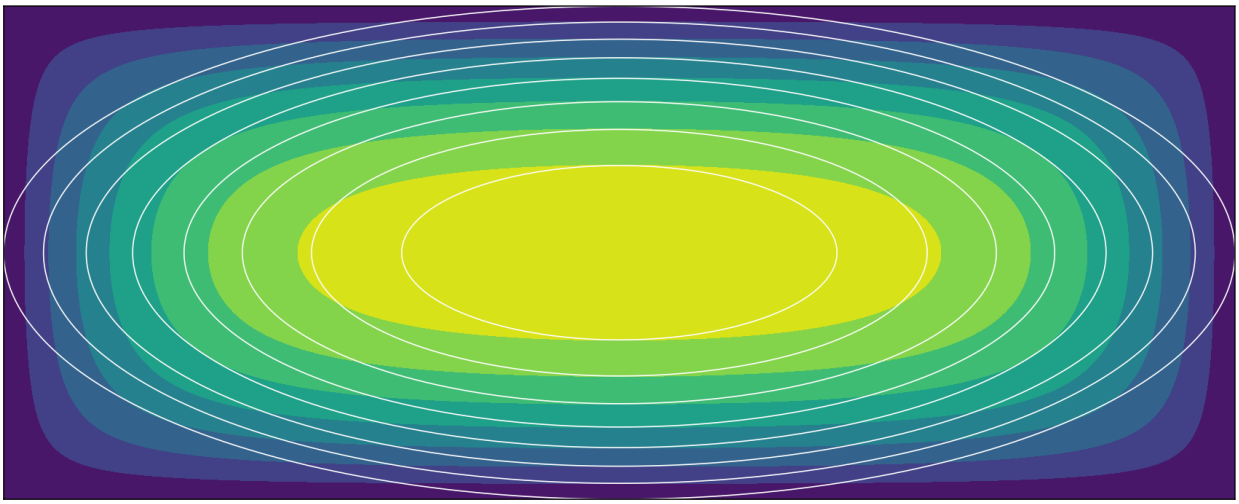

Figure 3.1: Flow profiles for the rectangular (solid colors) and elliptical duct (white lines) of aspect ratio $\lambda = 0.4$, scaled to match the peak velocity.

Our observations from the square, and other geometries such as regular polygons and "racetracks" discussed in chapter 6 lend evidence to this, but it is still an open question to prove mathematically.

Another fundamental question is about the connection, or lack thereof, of the rectangles and ellipses in the limit of aspect ratio $\lambda \to 0$. While both geometries converge to the infinite strip for $\lambda \to 0$ in a pointwise sense, only in the rectangles does a sequence of aspect ratios approaching zero (such as in the figure) produce a sequence of functions $Sk(t;\lambda)$ which appear to converge to the channel's skewness evolution. Similar questions about the effective diffusivity have been asked in the literature [6, 12, 11, 9]. To our knowledge, explanations have mostly been descriptive. The argument is essentially that the Poiseuille flow in the rectangle is nearly uniform with respect to z in the interior, with boundary layers at the far walls $z = \pm 1/\lambda$, whereas in the ellipses, the flow has no boundary layer structure regardless of aspect ratio. We have illustrated this idea in figure 3.1, where we overlay contour maps of the rectangle and ellipse for aspect ratio $\lambda = 0.4$. The rectangular flow has boundary layers present due to the high curvature of the walls, while the elliptical contours have no obvious boundary layer structure, as the sliced flow $u(z_0, y)$ or $u(z, y_0)$ along along lines $z = z_0$ and $y = y_0$ are parabolic, whereas the same is not true for the rectangle. We explore this further in chapter 6 where we introduce a new parameter to smoothly interpolate between ellipses and approximate rectangles to demonstrate this phenomenon is not merely a property of smoothness of the boundary (e.g., the boundary being continuous versus continuously differentiable).

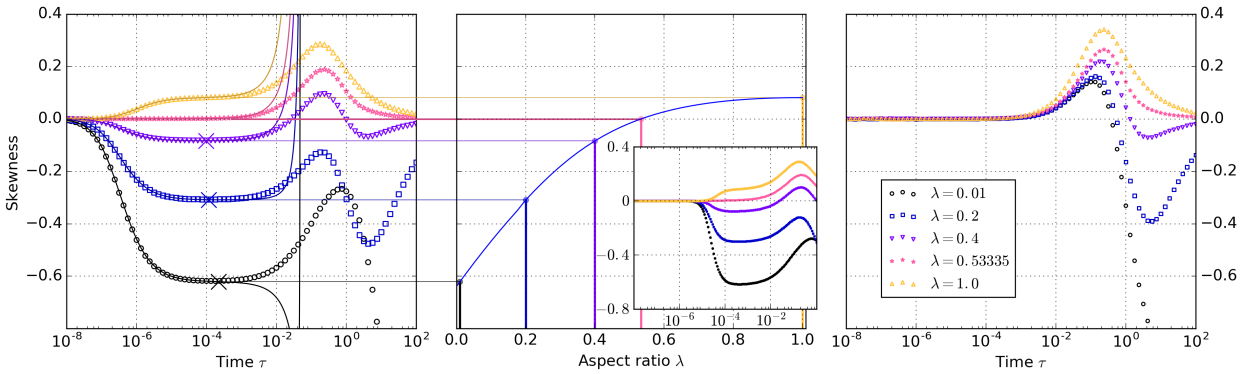

Figure 3.2: Evolution of skewness for the rectangles (left panel) and ellipses (right panel) of varying aspect ratio. Geometric skewness is plotted (center panel) as a function of aspect ratio, with aspect ratios corresponding to the simulations indicated. Simulations done using a finite width initial condition $\sigma \approx 0.115$ in the rectangles (inset) are done with the same aspect ratios. In all cases, $\mathrm{Pe} = 10^4$.

Long time asymptotics of the Aris equations.

The intuitive approach to finding the steady state of a driven diffusion problem is to simply remove the time derivative and solve the corresponding time-independent elliptic boundary value problem:

$$\left(\frac{\partial}{\partial t} - \nabla^2\right) f(y,z,t) = g(y,z), \qquad \left.\frac{\partial f}{\partial \mathbf{n}}\right|_{\partial\Omega} = 0. \tag{3.58}$$

This is the approach originally used by Taylor [1], in spirit. It has been more rigorously developed since then through long time expansions of the moments equations [3, 5, 4] and more recently under the umbrella of homogenization theory [8, 9], to derive the effective diffusivity.

This approach comes with a caveat: the average of the driver needs to be zero, otherwise the problem is inconsistent. This can be seen by integrating both sides of the PDE over Ω and using the divergence theorem:

$$-\int_\Omega \nabla^2 f(y,z)\,dA = \int_\Omega g(y,z)\,dA, \quad \text{but} \tag{3.59}$$

$$-\int_\Omega \nabla^2 f(y,z)\,dA = \int_C \frac{\partial f}{\partial \mathbf{n}}\,ds = 0, \tag{3.60}$$

Where C is the boundary, and the Neumann boundary conditions cause the integral to vanish. Physically speaking, the integrated driving term represents a net increase/decrease in the quantity f in the diffusion equation, and the integrated quantity $\partial f/\partial \mathbf{n}$ represents the net flux into or

out of the domain. For example, if $\int_\Omega g(y,z)\,dA > 0$, and we impose zero Neumann boundary conditions, the solution to the diffusion equation would grow without bound, so there would be no (time-independent) steady state.

Nevertheless, it is straightforward to generalize the steady state idea if we have $\langle g \rangle \neq 0$. If we use the following ansatz

$$f(y,z) = f_0(y,z) + t f_1(y,z) \tag{3.61}$$

for the long time behavior of the driven diffusion problem, substitute, and collect in factors of t, we get two sub-problems

$$-\nabla^2 f_0 = -f_1(y,z) + g(y,z), \qquad \left.\frac{\partial f_0}{\partial \mathbf{n}}\right|_\Omega = 0 \tag{3.62}$$

$$-\nabla^2 f_1 = 0, \qquad \left.\frac{\partial f_1}{\partial \mathbf{n}}\right|_\Omega = 0 \tag{3.63}$$

Solvability of the first problem (3.62) requires that $\langle f_1 \rangle = \langle g \rangle$. Combining this with the fact that the only solutions to (3.63) are $f_1 = const.$ implies that $f_1 \equiv \langle g \rangle$. Thus, the problem (3.62) is just the inconsistent problem (3.58) with the driver modified to be mean-zero. The long time asymptotics take the form

$$f(y,z,t) \sim f_0(y,z) + \langle g(y,z) \rangle t. \tag{3.64}$$

Because of the nature of the Aris equations, successive substitution of these long time asymptotics T_n will produce drivers with increasingly higher order dependence on polynomials in t. To demonstrate, for $T_1(y,z,\tau)$, the driver is $u(y,z)$, which we take to be mean zero. The exact solution in any domain is formally

$$T_1(y,z,\tau) = \mathrm{Pe}\ g(y,z) + \sum_{k=1}^{\infty} c_k e^{-\sigma_k^2 t} \phi_k(y,z), \tag{3.65}$$

where $g(y,z)$ is the solution of the Poisson problem with $u(y,z)$ as the driver, $\phi_k(y,z)$ are the normalized Laplacian eigenfunctions with Neumann boundaries with corresponding eigenvalues $-\sigma_k^2$, and the coefficients c_k are chosen to satisfy the initial condition $f_1(y,z)$:

$$c_k = \int_\Omega \big[f_1(y,z) - \mathrm{Pe}\ g_1(y,z) \big] \phi_k(y,z) dA. \tag{3.66}$$

Note the intial data is solely encoded in the set of coefficients c_k; the loss of information of the initial data typical to diffusive processes comes in the corresponding exponential decay term. We should not expect this initial data to last through to the long time analysis except possibly in correction terms.

Substitution of this solution into the T_2 problem gives

$$\left(\frac{\partial}{\partial t} - \nabla^2\right) T_2 = 2 + 2\mathrm{Pe}\, u(y,z) T_1 = 2 + 2\mathrm{Pe}\, u\Big[\mathrm{Pe}\ g(y,z) \ + \ \sum_{k=1}^{\infty} c_k e^{-\sigma_k^2 t}\phi_k(y,z)\Big] \tag{3.67}$$

The exponentially decaying term produces a term with time dependence $te^{-\sigma_k^2 t}$, which is subdominant to any polynomial terms in t. The long time problem is produced by using the long time asymptotics for T_1 in the driver:

$$\left(\frac{\partial}{\partial t} - \nabla^2\right) T_2 = 2 + 2\mathrm{Pe}^{\,2} u(y,z) g(y,z), \tag{3.68}$$

which is again a Poisson problem with a driver which does not integrate to zero. The long time asymptotics of T_2 will include a linear t term, which will result in a linear-time driver for T_3.

Long time asymptotics of with a polynomial time driver.

Taking the discussion above into account, we derive the solution of the problem with a general driver with time dependence that is polynomial, then apply it to our problems as necessary. We write the long time (neglecting initial data) diffusion problem with a driver which is polynomial in time as

$$\left(\partial_t - \nabla^2\right) f(y,z,t) = \sum_{m=0}^{M} a_m(y,z) t^m, \qquad \left.\frac{\partial f}{\partial \mathbf{n}}\right|_{\partial\Omega} = 0. \tag{3.69}$$

This can be solved by the ansatz

$$f(y,z,t) = \sum_{m=0}^{M+1} f_m(y,z) t^m. \tag{3.70}$$

Substituting and collecting in powers of t gives a set of Poisson equations with Neumann boundary conditions:

$$\begin{aligned} -\nabla^2 f_{M+1}(y,z) &= 0 \\ -\nabla^2 f_M(y,z) &= a_M(y,z) - (M+1) f_{M+1}(y,z) \\ -\nabla^2 f_{M-1}(y,z) &= a_{M-1}(y,z) - M f_M(y,z) \\ &\vdots \\ -\nabla^2 f_0(y,z) &= a_0(y,z) - f_1(y,z). \end{aligned} \tag{3.71}$$

The undetermined constant in the solution of each problem (except the last, for f_0) is chosen to enforce the solvability condition $\langle a_{m-1} - m f_m \rangle = 0$. The leading order term in the solution, $f_{M+1} t^{M+1}$, has a constant solution, with $f_{M+1} = \langle a_M \rangle / (M+1)$, while generically the rest of the f_m are nontrivial functions of space. If it happens that $\langle a_M \rangle = 0$, then f_{M+1} drops out, and the leading order polynomial is one degree lower and *does* have spatial variation. It happens that this is the case when calculating T_3. Also note the degree in t cannot drop further unless $a_M(y,z) \equiv 0$.

Long time behavior of the Aris equations.

Now we successively find the long time asymptotics for the first three moments. The long time T_1 problem is

$$(\partial_t - \nabla^2) T_1 = \mathrm{Pe}\, u(y,z), \qquad \left.\frac{\partial T_1}{\partial \mathbf{n}}\right|_{\partial\Omega} = 0. \tag{3.72}$$

Since we choose work in coordinates where $\langle u \rangle = 0$, the long time asymptotics have no time dependence:

$$T_1(y,z,t) \sim \mathrm{Pe}\, g_1(y,z), \quad t \to \infty \tag{3.73}$$

Where g_1 solves (3.72) without the factor of Pe. The long time T_2 problem is

$$(\partial_t - \nabla^2) T_2 \sim 2 + 2\mathrm{Pe}^2 u(y,z) g_1(y,z), \tag{3.74}$$

and applying the procedure from section 3.3.1, the corresponding long time behavior is

$$T_2 \sim \mathrm{Pe}^2 g_2 + (2 + 2\mathrm{Pe}^2 \langle u g_1 \rangle) t, \tag{3.75}$$

where the function $g_2(y,z)$ is the solution to the problem

$$-\nabla^2 g_2 = 2(ug_1 - \langle ug_1\rangle), \quad \left.\frac{\partial g_2}{\partial \mathbf{n}}\right|_{\partial\Omega} = 0, \quad \langle g_2\rangle = 0. \tag{3.76}$$

We highlight that this formula contains the dimensionless effective diffusivity $\kappa_{\text{eff}} = 1 + \text{Pe}^2\langle ug_1\rangle$. Thus the pointwise variance is asymptotically uniform in the cross section for $t \to \infty$, but will have nontrivial cross sectional structure on intermediate timescales depending on the competition of the terms $g_2(y,z)$ and $\langle ug_1\rangle t$.

Substitution of this T_2 solution into the T_3 problem gives

$$\begin{aligned}(\partial_t - \nabla^2)T_3 &= 6T_1 + 3\text{Pe}\,uT_2 \\ &= 6\text{Pe}\,g_1 + 3\text{Pe}^3 ug_2 + 6\text{Pe}\,\kappa_{\text{eff}} ut\end{aligned} \tag{3.77}$$

with solution (recall $-\nabla^2 g_1 = u$)

$$T_3 \sim -\nabla^{-2}\Big[6\text{Pe}\,g_1 + 3\text{Pe}^3[ug_2 - \langle ug_2\rangle]\Big] + \Big[3\text{Pe}^3\langle ug_2\rangle + 6\text{Pe}\,\kappa_{\text{eff}} g_1\Big]t \tag{3.78}$$

Collecting only the leading order terms of the solutions (3.73, 3.75, 3.78) and substituting into the numerator and denominator of the skewness gives:

$$\begin{aligned}T_3 - 3T_2T_1 + 2T_1^3 &= \Big[3\text{Pe}^3\langle ug_2\rangle + 6\text{Pe}\,\kappa_{\text{eff}} g_1 - 3\big[2\kappa_{\text{eff}}\big]\big[\text{Pe}\,g_1\big]\Big]t = 3\text{Pe}^3\langle ug_2\rangle t, \\ T_2 - T_1^2 &= \Big[2\kappa_{\text{eff}}\Big]t.\end{aligned} \tag{3.79}$$

This gives the overall leading order behavior of the skewness, which is uniform throughout the cross section despite the individual moments possessing dependence on the cross-sectional location at long time:

$$Sk(y,z,t) \sim Sk(t) = \frac{3\text{Pe}^3\langle ug_2\rangle}{\big(2\kappa_{\text{eff}}\big)^{3/2}}\, t^{-1/2} = \frac{3\text{Pe}^3\langle ug_2\rangle}{\big(2\langle ug_1\rangle\big)^{3/2}}\, t^{-1/2}. \tag{3.80}$$

In addition, in the large Péclet limit we get a scaling prediction only dependent on $\langle ug_1\rangle$ and $\langle ug_2\rangle$,

which shows a universality of the dynamics of the skewness at large Péclet;

$$Sk(t) \sim \frac{3\langle ug_2\rangle}{(2\langle ug_1\rangle)^{3/2}} t^{-1/2}. \tag{3.81}$$

In summary, obtaining the leading order behavior of the skewness requires solving the problems

$$-\nabla^2 u(y,z) = 2, \quad u|_{\partial\Omega} = const., \quad \langle u\rangle = 0 \tag{3.82}$$

$$-\nabla^2 g_1(y,z) = u(y,z), \quad \left.\frac{\partial g_1}{\partial \mathbf{n}}\right|_{\partial\Omega} = 0, \quad \langle g_1\rangle = 0, \tag{3.83}$$

$$-\nabla^2 g_2(y,z) = 2\big(ug_1 - \langle ug_1\rangle\big), \quad \left.\frac{\partial g_2}{\partial \mathbf{n}}\right|_{\partial\Omega} = 0, \tag{3.84}$$

calculating the cross sectional averages $\langle ug_1\rangle$ and $\langle ug_2\rangle$, and substituting these into (3.80). Note that $g_2(y,z)$ is only determined up to a constant through this process, but it is not an issue as the constant washes out when taking the averages.

Exact calculation of long time asymptotics in the channel.

The long time problems can be handled explicitly, as the problem is one-dimensional and the flow is polynomial. The flow is $u = 1/3 - y^2$. With this, the problem

$$-\frac{\partial^2}{\partial y^2} g_1 = u, \quad \left.\frac{\partial g_1}{\partial y}\right|_{y=\pm 1} = 0, \quad \langle g_1\rangle = 0 \tag{3.85}$$

has the solution

$$g_1(y) = \frac{1}{6}\left(\frac{1}{30} - y^2 + \frac{y^4}{2}\right) \tag{3.86}$$

which yields the diffusive enhancement $\langle ug_1\rangle = 8/945$, which agrees with [2, 5, 6, 8, 11] after differences in convention with the prototype domain and flow are taken into account. The driver for the next problem is

$$2(ug_1 - \langle ug_1\rangle) = -\frac{y^6}{6} + \frac{7y^4}{18} - \frac{17y^2}{90} + \frac{17}{1890} \tag{3.87}$$

giving the solution for g_2

$$g_2(y) = \frac{-29}{226800} - \frac{17y^2}{3780} + \frac{17y^4}{1080} - \frac{7y^6}{540} + \frac{y^8}{336}. \tag{3.88}$$

The coefficient for the numerator is

$$\langle ug_2 \rangle = -\frac{64}{467775} \tag{3.89}$$

giving the large Péclet asymptotic for the skewness,

$$Sk(t) \sim -\sqrt{\frac{21}{605}} t^{-1/2}. \tag{3.90}$$

Exact calculation of long time asymptotics in any ellipse.

Here we work in dimensional coordinates to avoid ambiguities which arise from different different nondimensionalizations used in the literature. The long-time problems in the ellipses can be solved exactly using a transformation to elliptical coordinates

$$y = c\cos(\xi)\cosh(\eta), \qquad z = c\sin(\xi)\sinh(\eta), \tag{3.91}$$

with $\eta \in [0, 2\pi)$ the "angular" coordinate, $\xi \in [0, \xi_b]$ the "radial" coordinate, and parameters $c = \sqrt{b^2 - a^2}$ and $\xi_b = \tanh^{-1}(a/b)$ specifying the shape of the ellipse with semi-axes $b > a > 0$. The Laplacian is transformed from rectangular to elliptical coordinates as

$$\frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \to \frac{1}{\mathcal{J}(\xi, \eta)} \left(\frac{\partial^2}{\partial \xi^2} + \frac{\partial^2}{\partial \eta^2} \right), \tag{3.92}$$

with Jacobian

$$\mathcal{J}(\xi, \eta) = \frac{b^2 - a^2}{2} (\cosh(2\xi) - \cos(2\eta)). \tag{3.93}$$

The flow solution

$$u = \frac{-a^2 p_x}{\mu} \left[\frac{1}{1 + (a/b)^2} \left(\frac{1}{2} - \frac{y^2}{a^2} - \frac{z^2}{b^2} \right) \right] \tag{3.94}$$

can be written in elliptical coordinates with direct substitution. After multiplying through the Jacobian and expanding the right hand side in a $\cos(2k\eta)$ basis, the long time moment problems to

be solved take the generic form

$$\kappa\left(\frac{\partial^2}{\partial\xi^2}+\frac{\partial^2}{\partial\eta^2}\right)g_m(\xi,\eta)=\sum_{k=0}^{K}\phi_{2k}(\xi)\cos(2k\eta) \tag{3.95a}$$

$$g_m(\xi,0)=g_m(\xi,2\pi),\quad \left.\frac{\partial g_m}{\partial\xi}\right|_{\xi=\xi_b}=0. \tag{3.95b}$$

Importantly, the right hand side is a finite sum in cosines (e.g., $K=2$ for the first moment), so there is a related finite-sum separation of variables solution for $g_m(\xi,\eta)$:

$$g_1(\xi,\eta)=\sum_{k=0}^{K}\gamma_{2k}(\xi)\cos(2k\eta). \tag{3.96}$$

The subproblem for each γ_{2k} requires solving an ODE

$$\gamma_{2k}''(\xi)-(2k)^2\gamma_{2k}(\xi)=\phi_{2k}(\xi), \tag{3.97a}$$

$$\gamma_{2k}'(\xi_b)=\gamma_{2k}'(0)=0. \tag{3.97b}$$

The new boundary condition at $\xi=0$ comes in as a smoothness requirement; any other boundary condition at $\xi=0$ would result in nondifferentiable corners in the solution.

To see this, we look at $\partial g_m/\partial y$ approaching the interfocal line segment $\xi\equiv 0$. Since the original problem is a well-posed Poisson problem with analytic forcing and boundary, derivatives of all orders exist in the interior; in particular $\partial g_m/\partial y$ at $(y,z)=(0,z)$ is well defined for any z lying between the foci. The corresponding point in elliptic coordinates is located at both $(\xi,\eta)=(0,\eta_+)$ and $(0,\eta_-)$, with $\eta_+\in(0,\pi)$ and $\eta_-=2\pi-\eta_+$. The differential mapping can be inverted to get an expression for $\partial g_m/\partial y$ in terms of $\partial g_m/\partial\xi$ and $\partial g_m/\partial\eta$:

$$\frac{\partial g_m}{\partial y}=\frac{1}{\mathcal{J}(\xi,\eta)}\left(\cosh(\xi)\sin(\eta)\frac{\partial g_m}{\partial\xi}+\sinh(\xi)\cos(\eta)\frac{\partial g_m}{\partial\eta}\right), \tag{3.98}$$

with $\mathcal{J}$ the Jacobian, and evaluating at η_+ and substituting the generic solution,

$$\frac{\partial g_m}{\partial y}(0,\eta_+) = \left(\frac{2\sin(\eta_+)}{(b^2-a^2)(1-\cos(2\eta_+))}\right)\frac{\partial g_m}{\partial \xi}(0,\eta_+) \tag{3.99}$$

$$= \mathcal{C}(\eta_+)\sum_{k=0}^{K}\frac{\partial \gamma_{2k}}{\partial \xi}(0)\cos(2k\eta_+), \tag{3.100}$$

whereas evaluating at η_- gives

$$\frac{\partial g_m}{\partial y}(0,\eta_-) = \frac{2}{(b^2-a^2)(1-\cos(2\eta_-))}\left(\sin(\eta_-)\frac{\partial g_m}{\partial \xi}(0,\eta_-)\right) \tag{3.101}$$

$$= \frac{-2\sin(\eta_+)}{(b^2-a^2)(1-\cos(2\eta_+))}\left(\sum_{k=0}^{K}\frac{\partial \gamma_{2k}}{\partial \xi}(0)\cos(2k\eta_-)\right) \tag{3.102}$$

$$= -\mathcal{C}(\eta_+)\sum_{k=0}^{K}\frac{\partial \gamma_{2k}}{\partial \xi}(0)\cos(2k\eta_+). \tag{3.103}$$

$$= -\frac{\partial g_m}{\partial y}(0,\eta_+). \tag{3.104}$$

The two expressions must be equal to avoid a derivative jump. Subtracting the two and taking an inner product with $\cos(2m\eta_+)$ gives, for each k,

$$0 = 2\mathcal{C}(\eta_+)\frac{\partial \gamma_{2k}}{\partial \xi}(0), \tag{3.105}$$

and since $\mathcal{C}(\eta_+) \neq 0$, consistency requires each of the subproblems have an additional Neumann condition at $\xi = 0$.

The full formulae for the drivers and solutions for g_1 and g_2 are detailed in the appendix. After collecting the full equations, the dimensional quantity $\langle u g_1 \rangle$, related to the effective diffusivity, evaluates to

$$\begin{aligned}\langle u g_1 \rangle &= \frac{p_x^2}{\mu^2\kappa}\left[\frac{a^4b^4(5a^4+14a^2b^2+5b^4)}{2304(a^2+b^2)^3}\right] \\ &= \frac{a^6 p_x^2}{\mu^2\kappa}\left[\frac{5\lambda^4+14\lambda^2+5}{2304\,\lambda^2(1+\lambda^2)^3}\right].\end{aligned} \tag{3.106}$$

This formula agrees with formulae in the the literature for effective diffusivity in the elliptical pipe [2, 6], after taking into account differences in conventions.

Similarly, $\langle ug_2 \rangle$, which affects the sign of the long time skewness, evaluates to

$$\begin{aligned} \langle ug_2 \rangle &= \frac{-p_x^3}{\mu^3 \kappa^2} \left[\frac{-a^6 b^6 (5a^4 - 22a^2 b^2 + 5b^4}{138240(a^2+b^2)^3} \right] \\ &= \frac{a^{10} p_x^3}{\mu^3 \kappa^2} \left[\frac{-(5\lambda^4 - 22\lambda^2 + 5)}{138240\, \lambda^4 (1+\lambda^2)^3} \right]. \end{aligned} \tag{3.107}$$

This is sign indefinite. The root of this equation lying in $[0, 1]$ has the exact value

$$\lambda^* = \sqrt{(11 - 4\sqrt{6})/5} \approx 0.49031. \tag{3.108}$$

For $\lambda < \lambda^*$, the long time decay is negative, and it is positive for $\lambda > \lambda^*$. The other roots are at $\lambda = -\lambda^*$ and their reciprocals $\lambda = \pm 1/\lambda^*$, which is a reassuring result from a physical point of view. Put together, this gives a resulting large Péclet nondimensional prediction

$$Sk(t; \lambda) = \frac{-3\sqrt{2}(5 - 22\lambda^2 + 5\lambda^4)}{5\lambda \left(\frac{5 + 14\lambda^2 + 5\lambda^4}{1+\lambda^2} \right)^{3/2}} \tag{3.109}$$

In the case of the circular pipe, the nondimensional, large Péclet skewness then decays as

$$Sk(t; 1) \sim \sqrt{\frac{3}{50}} t^{-1/2}. \tag{3.110}$$

Calculation of long time coefficients in the rectangular duct.

In the rectangular duct, the flow cannot, to our knowledge, be expressed in closed form. We begin with an analytical approach to calculate g_1 and $\langle ug_1 \rangle$, which we can compare to previous results on effective diffusivity. Then we will opt to use a finite element solver to follow through to the prediction for g_2 and the coefficient $\langle ug_2 \rangle$. This finite element approach will also allow us another check against our exact long time predictions, where we have them.

Analytical approach. With the single-series formula for the flow, it is straightforward to calculate g_1 and the diffusive enhancement $\langle ug_1 \rangle$ as in [6]. Because of the varied choices of nondimensionalization in the literature, we choose to work in dimensional coordinates here. Once we have derived a formula, we can validate against previous results (e.g., [6], or more recently [11]) and investigate the

behavior of the diffusivity in various limiting scenarios.

For convenience, we rewrite the dimensional, mean-zero formulas here:

$$u(y,z) = u_c(y) + \sum_{k=1}^{\infty} c_k \big[\cos((k-1/2)\pi y/a) \cosh((k-1/2)\pi z/a) - \beta_k \big], \tag{3.111a}$$

$$u_c = \frac{-a^2 p_x}{\mu} \left(\frac{1}{3} - \frac{y^2}{a^2} \right), \tag{3.111b}$$

$$c_k = \frac{-a^2 p_x}{\mu} \frac{2(-1)^k}{\pi^3 (k-1/2)^3 \cosh((k-1/2)\pi b/a)}, \tag{3.111c}$$

$$\beta_k = \frac{(-1)^{k+1} \lambda \sinh((k-1/2)\pi/\lambda)}{\pi^2 (k-1/2)^2} \tag{3.111d}$$

It is convenient for the purposes of solving the Poisson problem to convert the single series $\tilde{u}(y,z) \equiv u(y,z) - u_c(y)$ into a traditional Fourier series. Using the usual inner product techniques, the end result is

$$\tilde{u} = \frac{-a^2 p_x}{\mu} {\sum_{m,n}}' \tilde{u}_{mn} \cos(m\pi y/a) \cos(n\pi z/b), \tag{3.112a}$$

$$\tilde{u}_{mn} = \frac{-16(a/b)(-1)^{m+n}}{\pi^5 (1 + \delta_{m0} + \delta_{n0})} \sum_{k=1}^{\infty} \frac{\tanh((k-1/2)\pi b/a)}{\big[k-1/2\big]\big[(k-1/2)^2 - m^2\big]\big[(k-1/2)^2 + (a/b)^2 n^2\big]}, \tag{3.112b}$$

where the primed sum is over all $m, n \geq 0$ and $(m,n) \neq (0,0)$. The analagous dimensional form for the $g_1(y,z)$ problem is:

$$-\kappa \nabla^2 g_1(y,z) = u(y,z), \quad \frac{\partial}{\partial \mathbf{n}} g_1|_{\partial\Omega} = 0, \quad \langle g_1 \rangle = 0. \tag{3.113}$$

With u_c a polynomial, $\tilde{u}$ expressed as a cosine series, and linearity of the problem, the solution can be calculated termwise:

$$g_1 = g_{c1}(y) + \tilde{g}_1(y,z), \tag{3.114}$$

with

$$g_c = \frac{-a^4 p_x}{\mu \kappa} \left(\frac{7}{180} - \frac{(y/a)^2}{6} + \frac{(y/a)^4}{12} \right), \tag{3.115}$$

$$\tilde{g} = \frac{-a^4 p_x}{\mu \kappa} {\sum_{m,n}}' \frac{\tilde{u}_{mn}}{\pi^2 (m^2 + (a/b)^2 n^2)} \cos(m\pi y/a) \cos(n\pi z/b). \tag{3.116}$$

The diffusive enhancement is of the form

$$\kappa_{\text{enh.}} = \kappa + \langle u g_1 \rangle. \tag{3.117}$$

The full average expands out,

$$\langle u g_1 \rangle = \langle u_c g_{c1} \rangle + \langle u_c \tilde{g} \rangle + \langle \tilde{u} g_{c1} \rangle + \langle \tilde{u} \tilde{g}_1 \rangle, \tag{3.118}$$

with terms

$$\langle u_c g_c \rangle = \frac{a^6 p_x^2}{\mu^2 \kappa} \cdot \frac{8}{945} \tag{3.119a}$$

$$\langle u_c \tilde{g}_1 \rangle = \frac{a^6 p_x^2}{\mu^2 \kappa} \sum_{m=1}^{\infty} \frac{-2(-1)^m \tilde{u}_{m0}}{(m\pi)^4} \tag{3.119b}$$

$$\langle \tilde{u} g_{c1} \rangle = \frac{a^6 p_x^2}{\mu^2 \kappa} \sum_{m=1}^{\infty} \frac{-2(-1)^m \tilde{u}_{m0}}{(m\pi)^4} \tag{3.119c}$$

$$\langle \tilde{u} \tilde{g}_1 \rangle = \frac{a^6 p_x^2}{\mu^2 \kappa} {\sum_{m,n}}' \frac{(1 + \delta_{m0} + \delta_{n0}) \tilde{u}_{mn}^2}{4\pi^2 (m^2 + (a/b)^2 n^2)} \tag{3.119d}$$

Numerically calculating these series, we see agreement with [6, 11] when modifying the pressure gradient so that the average lab-frame velocity is constant amongst all aspect ratios.

Numerical approach. Because the rectangular ducts ultimately require series truncation and evaluation at some point, we have additionally written code to solve the Poisson problems for u, g_1, and g_2 (3.82 – 3.84) and calculate the averages $\langle u g_1 \rangle$ and $\langle u g_2 \rangle$ using a black box finite element solver. We used *Mathematica 10*, which has a finite element package which can be imported via the command `«NDSolve‘FEM‘`, which includes the procedures `ToElementMesh` to construct a mesh on a specified domain and `NDSolveValue` to solve the problem with specified boundary conditions and parameters. The analytical tools of Mathematica combined with these procedures allows one to, in principle, work with any geometry they can define mathematically.

For the purpose of finding the "golden" aspect ratios λ^* in both the rectangle and ellipse, we have written two routines, one for each class of geometry, whose input is the aspect ratio and whose output is the numerical value of $\langle u g_2 \rangle$. The main routine successively solves the problems for u, g_1,

and g_2 as described above, then outputs the numerical value for $\langle ug_2 \rangle$. The approximate λ^* is then found by hooking this routine into a numerical rootfinder. We have done convergence studies to ensure the convergence of the digits $\lambda^* \approx 0.49038$ in the rectangle. In the ellipse, we have found $\lambda^* \approx 0.49031$, which agrees with the exact prediction of equation (A.14).

We have also compared the the predictions using the finite element solver in the rectangles with Monte Carlo simulation in figure 5.7, found in chapter 5 on our Monte Carlo approach. In the log-log plots, the computed coefficient $3\langle ug_2 \rangle / (2\langle ug_1 \rangle)^{3/2}$ represents a y intercept, which sees good agreement with the Monte Carlo past the first (nondimensional) diffusive timescale $t = 1$, except for very small aspect ratios, when the long time theory is not yet close to being valid (this occurs strictly for $t \propto 1/\lambda^2$).

CHAPTER 4

Poisson summation of channel formulae

Motivation

The exact solution of the first three moments equations for the tracer problem in the infinite channel contain many sums of the form

$$\begin{aligned} &\sum_{n=1}^{\infty} \frac{(-1)^n}{(n\pi)^p} e^{-(n\pi)^2 t} \cos(n\pi y), \text{ for } p \text{ even, and} \\ &\sum_{n=1}^{\infty} \frac{(-1)^n}{(n\pi)^p} e^{-(n\pi)^2 t} \sin(n\pi y), \text{ for } p \text{ odd,} \end{aligned} \tag{4.1}$$

for $p = 0, 1, ..., 12$. These don't have closed-form expressions (except at $t = 0$ and $t \to \infty$), so we can only hope to truncate the series and evaluate numerically. However, for t small, the number of terms necessary for convergence to within a chosen ε may be large for small p. Since this is an alternating series, supposing we truncate the term to $N - 1$ terms and fix y and t, a rough estimate for the error is

$$\begin{aligned} \left| \left(\sum_{n=1}^{\infty} - \sum_{n=1}^{N-1} \right) \frac{(-1)^n}{(n\pi)^p} e^{-(n\pi)^2 t} \cos(n\pi y) \right| &\leq \left| \frac{(-1)^N}{(N\pi)^p} e^{-(N\pi)^2 t} \cos(N\pi y) \right| \\ &\leq \frac{1}{(N\pi)^p} e^{-(N\pi)^2 t}. \end{aligned} \tag{4.2}$$

For t sufficiently small, use the first order Taylor expansion of the exponential to try to get a bound on N to achieve ϵ accuracy:

$$\begin{aligned} &\frac{1}{(N\pi)^p}(1 - (N\pi)^2 t) = \varepsilon \\ &(\varepsilon \pi^p) N^p + (\pi^2 t) N^2 - 1 = 0. \end{aligned} \tag{4.3}$$

We have, a rootfinding problem for N. If we can neglect the N^p term (under the correct assumptions on $\varepsilon\pi^p$ relative to the other terms) then we have a basic requirement that we need to keep at least $N \approx \pi/\sqrt{t}$ terms to achieve this accuracy. For instance, if $t = 10^{-6}$, $N \approx 3.1 \times 10^3$

terms are necessary. The issue becomes more complicated when we need to also start worrying about finite precision arithmetic and Péclet number dependence. For example, we have observed that finite precision issues occur where the series solution diverges for small t independent of increasingly large N, whereas a two-term Taylor expansion does not see this issue.

With this idea in mind, it is useful to construct an equivalent formulas, for $t \ll 1$ using Poisson summation. By its nature, the Poisson summation converges exponentially fast for $t \ll 1$ (compared with exponentially fast convergence for $t \gg 1$ in the original series), which allows us to keep only a few terms of the sum to accurately resolve the moments.

Derivation of the seed identity

The first goal here is to use the Poisson summation formula to rewrite the sum

$$\sum_{n=1}^{\infty} (-1)^n e^{-n^2\pi^2 t} \cos(n\pi y). \tag{4.4}$$

The basic Poisson summation result (c.f. [16], section 3.1.5) relates an infinite sum with summand function f to its Fourier-transformed pair $\hat{f}$:

$$\sum_{n=-\infty}^{\infty} f(n) = \sum_{m=-\infty}^{\infty} \hat{f}(2\pi m), \tag{4.5}$$

with the Fourier transform pair defined as:

$$\begin{cases} \hat{f}(k) = \displaystyle\int_{-\infty}^{\infty} e^{ikx} f(x) dx, \\ f(x) = \dfrac{1}{2\pi} \displaystyle\int_{-\infty}^{\infty} e^{-ikx} \hat{f}(k) dk. \end{cases} \tag{4.6}$$

To use this, first, rewrite the Poisson formula (4.5) in terms of singly-infinite sums:

$$\sum_{n=1}^{\infty} f(n) + f(-n) = -f(0) + \hat{f}(0) + \sum_{m=1}^{\infty} \hat{f}(2\pi m) + \hat{f}(-2\pi m), \tag{4.7}$$

then rewrite the summand (4.4) in a compatible form:

$$(-1)^n e^{-n^2\pi^2 t}\cos(n\pi y) = e^{in\pi}e^{-n^2\pi^2 t}\left(\frac{e^{in\pi y}+e^{-in\pi y}}{2}\right) \tag{4.8a}$$

$$= \frac{1}{2}e^{in\pi}e^{-n^2\pi^2 t}e^{in\pi y} + \frac{1}{2}e^{-in\pi}e^{-n^2\pi^2 t}e^{-in\pi y} \tag{4.8b}$$

$$= \frac{1}{2}e^{-n^2\pi^2 t}e^{in(\pi y+\pi)} + \frac{1}{2}e^{-n^2\pi^2 t}e^{-in(\pi y+\pi)} \tag{4.8c}$$

$$= \frac{1}{2}e^{-n^2\pi^2 t}e^{i(\pi y+\pi)n} + \frac{1}{2}e^{-n^2\pi^2 t}e^{-i(\pi y+\pi)n} \tag{4.8d}$$

$$= f(n) + f(-n), \tag{4.8e}$$

where the function is

$$f(n) = \frac{1}{2}e^{-\gamma n^2}e^{i\alpha n}, \text{with} \quad \gamma = \pi^2 t, \quad \alpha = \pi y + \pi. \tag{4.9}$$

Note (4.8b) uses the fact that $e^{in\pi} = e^{-in\pi}$ for $n \in \mathbb{Z}$. So, the original summand can be written as the left-hand side in (4.7). We just need to find the Fourier transform of $f(n)$:

$$\begin{aligned}\hat{f}(k) = \int_{-\infty}^{\infty} e^{ikx}f(x)dx &= \int_{-\infty}^{\infty} e^{ikx}\frac{1}{2}e^{-\gamma n^2}e^{i\alpha n}dx \\ &= \frac{1}{2}\sqrt{\frac{\pi}{\gamma}}e^{-\frac{(k+\alpha)^2}{4\gamma}}.\end{aligned} \tag{4.10}$$

We have $f(0) = 1/2$ and $\hat{f}(0) = \frac{1}{2}\sqrt{\pi/\gamma}e^{-\alpha^2/(4\gamma)}$, so substitution yields the formula:

$$\sum_{n=1}^{\infty}(-1)^n e^{-n^2\pi^2 t}\cos(n\pi y) = -\frac{1}{2} + \frac{1}{2}\sqrt{\frac{\pi}{\gamma}}e^{-\frac{\alpha^2}{4\gamma}} + \frac{1}{2}\sqrt{\frac{\pi}{\gamma}}\left[\sum_{m=1}^{\infty} e^{-\frac{(2\pi m+\alpha)^2}{4\gamma}} + e^{-\frac{(-2\pi m+\alpha)^2}{4\gamma}}\right], \tag{4.11}$$

or rewritten explicitly with $\gamma = \pi^2 t$ and $\alpha = \pi y + \pi$,

$$\sum_{n=1}^{\infty}(-1)^n e^{-n^2\pi^2 t}\cos(n\pi y) = -\frac{1}{2} + \frac{1}{2}\sqrt{\frac{1}{\pi t}}e^{-\frac{(y+1)^2}{4t}} + \frac{1}{2}\sqrt{\frac{1}{\pi t}}\left[\sum_{m=1}^{\infty} e^{-\frac{(2m+y+1)^2}{4t}} + e^{-\frac{(-2m+y+1)^2}{4t}}\right]. \tag{4.12}$$

Observe the terms on the right are in fact one dimensional heat kernels. Defining

$$G(y,t) = \frac{e^{\frac{-y^2}{4t}}}{\sqrt{4\pi t}}, \tag{4.13}$$

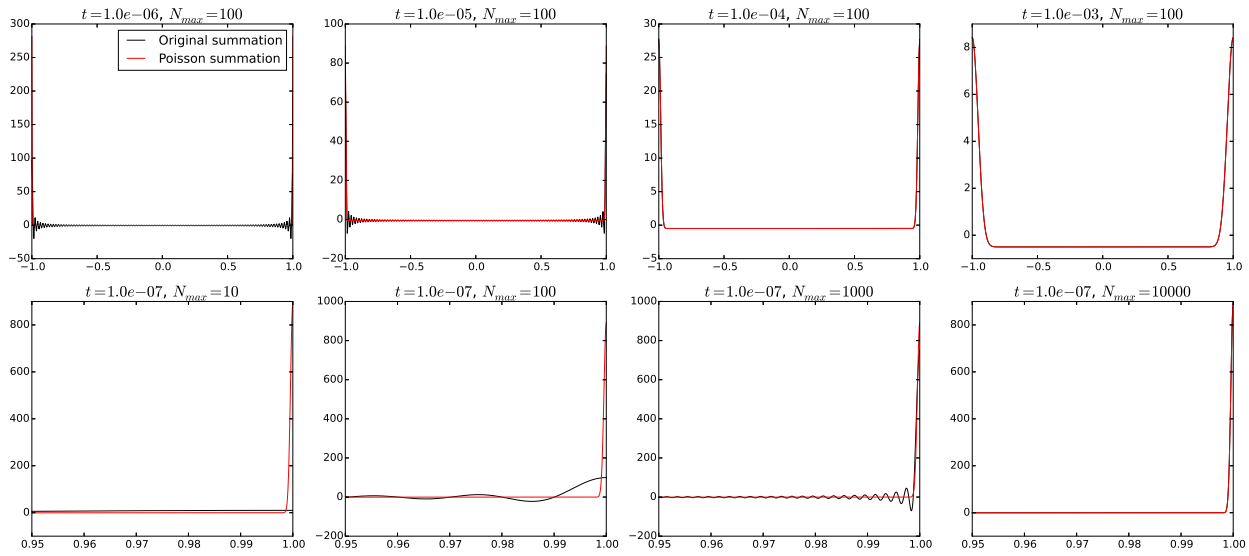

Figure 4.1: Comparison of the original summation to the expression recast via Poisson summation in (4.12). Top row: both original and Poisson versions truncated to 100 terms for times $t = 10^{-6}, ..., 10^{-3}$ on the y-interval $[-1, 1]$. Bottom row: demonstration of the terms needed for convergence scaling like $N_{\max} \sim \sqrt{t}$ in the boundary layer. Here, $t = 10^{-7}$, and $N_{\max}$ is varied from 10 to 10^4.

the sum can be written in the much simpler form

$$\sum_{n=1}^{\infty} (-1)^n e^{-n^2\pi^2 t} \cos(n\pi y) = -\frac{1}{2} + \sum_{m=-\infty}^{\infty} G(y + 2m + 1, t). \tag{4.14}$$

This will be a "seed" identity which we build up from, as similar formulas when multiplying the summand on the left by by $(n\pi)^{-p}$, for positive integer p, can be derived by taking integrals and derivatives of this expression.

Expressing in terms of a lattice of heat kernels also gives us the interpretation of the sum as the solution of the homogeneous diffusion equation on the real line, whose initial condition is a lattice of Dirac delta functions $\delta(y + 2m + 1)$. It can also be interpreted as a solution to the Neumann problem on the interval $[-1, 1]$ using the method of images, but the initial condition here, $-1/2 + \delta(y + 1) + \delta(y - 1)$, has to be interpreted loosely.

To put some faith in this formula, figure 4.1 compares the left and right-hand side expressions for various times and truncation indices $N_{\max}$ on the left hand series, evaluating on the y interval $[-1, 1]$. Only the ± 1 images are kept on the right hand side. The top row fixes $N_{\max} = 100$ and varies t to demonstrate the usefulness of the re-summation for $t \ll 1$. The bottom row fixes $t = 10^{-7}$ and

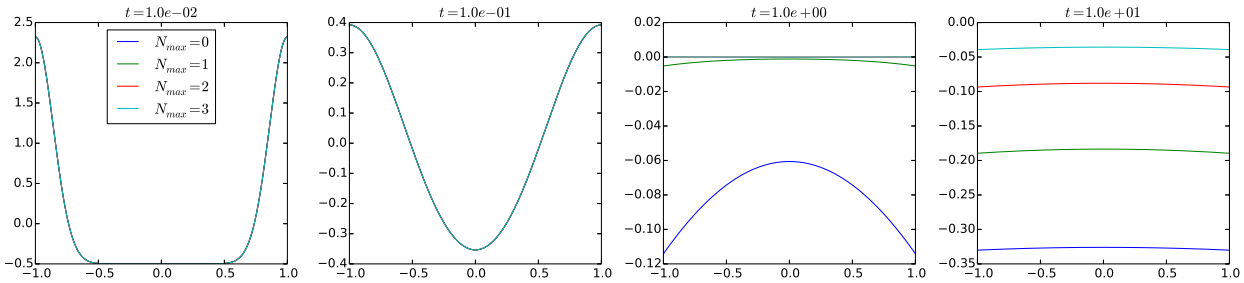

Figure 4.2: Behavior of (4.14) when keeping a small number of images at small to intermediate time. The effect of the extra images is not seen until order one time.

demonstrates the need for $N_{\max} \approx 10^4$ terms in the original sum before the substantial oscillations near the boundary can be removed.

As mentioned, only the terms $G(y+1,t)$ and $G(y-1,t)$ need be kept for substantial accuracy for $t \ll 1$. This is reminiscent of the formal short time asymptotics we had developed for T_1 for the channel problem:

$$T_1 \sim ut + (-1 + 2G(y,-1) + 2G(y,1))\,\frac{t^2}{2}, \tag{4.15}$$

However, it turns out the true nature of the short time behavior for T_1 is slightly different. This will establish rigorously later.

To demonstrate the need to keep only the ± 1 images in resolving the dynamics on $[-1,1]$, in figure 4.2 we look at the effect of including *any* of the additional images in (4.14) outside of $[-1,1]$ No visible effect can be seen until order one time. This can be shown explicitly. For instance, for the image centered at $y=3$, the greatest contribution is felt at $y=1$, to the tune of

$$G(y-3,t) = \frac{1}{\sqrt{4\pi t}} e^{-1/t}, \tag{4.16}$$

which is exponentially small for t small. Images farther out will have significantly smaller contribution (decaying as the square of the distance to the nearest boundary) until order one time.

Poisson summation of T_1

With the basic identity (4.14) established, the next step is to bootstrap this to calculate sums of the form:

$$\sum_{n=1}^{\infty}(-1)^n \frac{e^{-(n\pi)^2 t}}{(n\pi)^p}\cos(n\pi y), \quad p \text{ even}, \tag{4.17}$$

until we can write down an equivalent form for $T_1(y,t)$, the first moment in the infinite channel problem. Odd values of p, which only occur with sine terms, will be addressed later. The method of attack is to repeatedly integrate both sides of (4.14) with respect to t, which we detail for the $p=2$ case below, then summarize for $p=4$, after which we compare the results to $T_1(y,t)$, and analyze at a few approximations.

The $p=2$ case

Integrating the left-hand side of (4.12) and exchanging the order of integration and sum gives

$$\sum_{n=1}^{\infty}(-1)^n \left[\int_0^t e^{-n^2\pi^2 s}ds\right]\cos(n\pi y) = \sum_{n=1}^{\infty}(-1)^n \left[\frac{e^{-n^2\pi^2 t}}{-(n\pi)^2} - \frac{1}{-(n\pi)^2}\right]\cos(n\pi y) \tag{4.18a}$$

$$= \frac{1}{4}\left(y^2 - \frac{1}{3}\right) - \sum_{n=1}^{\infty}(-1)^n \frac{e^{-n^2\pi^2 t}}{(n\pi)^2}\cos(n\pi y). \tag{4.18b}$$

The polynomial part of (4.18b) can be verified by finding its Fourier expansion on $[-1,1]$. Integrating the right-hand side requires integrating the heat kernel in time (replacing t with s understood):

$$\int_0^t G(y,s)ds = \int_0^t \frac{1}{\sqrt{4\pi s}} e^{\frac{-y^2}{4s}} ds \tag{4.19a}$$

$$= \int_{\infty}^{|y|/\sqrt{4t}} \left(\frac{w}{|y|\sqrt{\pi}}\right) e^{-w^2} \left(\frac{-|y|^2}{2w^3}dw\right) \qquad (\text{with } w = |y|/\sqrt{4s}) \tag{4.19b}$$

$$= \frac{|y|}{\sqrt{4\pi}} \int_{|y|/\sqrt{4t}}^{\infty} \frac{1}{w^2} e^{-w^2} dw \tag{4.19c}$$

$$= \frac{|y|}{\sqrt{4\pi}} \left[\frac{-1}{w} e^{-w^2}\Big|_{|y|/\sqrt{4t}}^{\infty} - 2\int_{|y|/\sqrt{4t}}^{\infty} e^{-w^2} dw\right] \tag{4.19d}$$

$$= \frac{\sqrt{t}}{\sqrt{\pi}} e^{-\frac{|y|^2}{4t}} - \frac{|y|}{\sqrt{\pi}} \int_{|y|/\sqrt{4t}}^{\infty} e^{-w^2} dw \tag{4.19e}$$

$$= 2tG(y,t) - \frac{1}{2}|y|\,\mathrm{Erfc}\left(\frac{|y|}{\sqrt{4t}}\right), \tag{4.19f}$$

where $\mathrm{Erfc}(x) = \frac{2}{\sqrt{\pi}}\int_x^{\infty} e^{-s^2} ds$ is the complementary error function. Note that the use of absolute value in the transformation (4.19b) is necessary so that the lower limit $s=0$ always corresponds

with $w = +\infty$ rather than $w = -\infty$. Splitting into cases of $y < 0$ and $y > 0$ would yield the same result. As the erfc term will continue to show up, define

$$Er(y,t) = \text{Erfc}\left(\frac{|y|}{\sqrt{4t}}\right). \tag{4.20}$$

Then (4.14) integrates to

$$\begin{aligned}\int_0^t -\frac{1}{2} + \sum_{m=-\infty}^{\infty} G(y+2m+1,t)ds \\ = -\frac{1}{2}t + \sum_{m=-\infty}^{\infty} 2tG(y+2m+1,t) - \frac{1}{2}|y+2m+1|Er(y+2m+1,t).\end{aligned} \tag{4.21}$$

As we move forward, we will always be dealing with expressions with the sums of various images on the lattice $2\mathbb{Z}-1$. We also will be mainly looking at approximating the full sum by only keeping the ± 1 terms. With this in mind, we denote a primed sum as the full sum, excluding the ± 1 terms:

$$\begin{aligned}\sum{}' f(m) &\equiv \left[\sum_{m=-\infty}^{\infty} f(2m+1)\right] - f(-1) - f(1) \\ &= \sum_{m=1}^{\infty} f(-2m-1) + f(2m+1).\end{aligned} \tag{4.22}$$

Collecting everything to this point, we have the identity

$$\begin{aligned}\sum_{n=1}^{\infty}(-1)^n \frac{e^{-n^2\pi^2 t}}{(n\pi)^2}\cos(n\pi y) = \frac{1}{4}\left(y^2 - \frac{1}{3}\right) + \frac{t}{2} - 2tG(y+1,t) - 2tG(y-1,t) \\ + \frac{1}{2}|y+1|Er(y+1,t) + \frac{1}{2}|y-1|Er(y-1,t) \\ + \sum{}' -2tG(y+m,t) + \frac{1}{2}|y+m|Er(y+m,t).\end{aligned} \tag{4.23}$$

A first attempt at approximating would involve dropping $\sum'$ and the Er terms. This would give the expression:

$$\sum_{n=1}^{\infty}(-1)^n \frac{e^{-n^2\pi^2 t}}{(n\pi)^2}\cos(n\pi y) \approx \frac{1}{4}\left(y^2 - \frac{1}{3}\right) + \frac{t}{2} - 2tG(y,-1) - 2tG(y,1). \tag{4.24}$$

In figure 4.3 we compare these two to the original summation. There seems to be agreement on small

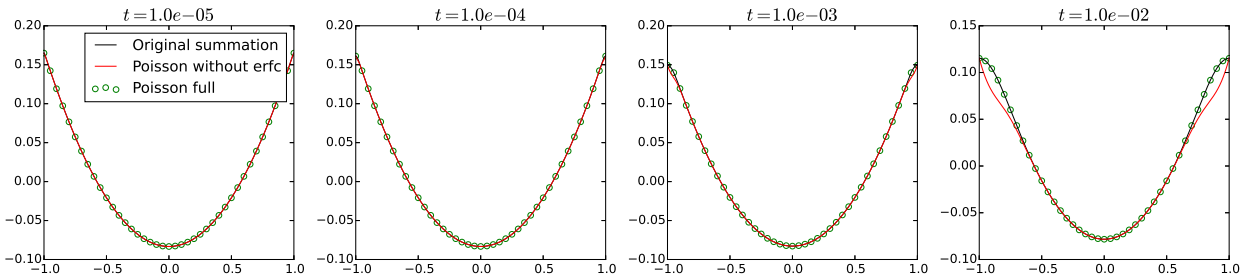

Figure 4.3: Evaluation of the left and right-hand sides of (4.23) and (4.24) with $N_{\max} = 10^4$ for the original summation. The effect of neglecting the $Er(\cdot)$ terms is seen by $t = 10^{-2}$.

timescales, but the approximation fails relatively quickly, since the $Er(\cdot)$ terms become significant when their arguments become order 1. For example, for $Er(y-1,t)$, the argument of the Erfc becomes order 1 at $\frac{y-1}{\sqrt{4t}} \approx 1 \to t \approx (y-1)^2/4$. This idea of dropping Erfc terms is addressed further in section 4.3.4.

The $p = 4$ case

Now repeat for $p = 4$. Integrate both sides of (4.23) in time. The left side integrates to:

$$\begin{aligned}\sum_{n=1}^{\infty}(-1)^n\left(\int_0^t \frac{e^{-n^2\pi^2 s}}{(n\pi)^2}ds\right)\cos(n\pi y) &= \sum_{n=1}^{\infty}(-1)^n\left(\frac{-e^{-n^2\pi^2 t}}{(n\pi)^4}+\frac{1}{(n\pi)^4}\right)\cos(n\pi y)\\ &= -\frac{1}{24}\left(\frac{y^4}{2}-y^2+\frac{7}{30}\right)-\sum_{n=1}^{\infty}(-1)^n\frac{e^{-n^2\pi^2 t}}{(n\pi)^4}\cos(n\pi y).\end{aligned} \tag{4.25}$$

The polynomial terms in the right-hand side integrate to:

$$\int_0^t \frac{1}{4}\left(y^2-\frac{1}{3}\right)+\frac{s}{2}ds = \frac{1}{4}\left(y^2-\frac{1}{3}\right)t+\frac{t^2}{4}, \tag{4.26}$$

and the functional form of the images in (4.23) are seen integrate to:

$$\int_0^t -2sG(y,s)+\frac{1}{2}|y|Er(y,s)ds = \mathcal{P}_1^{(4)}(y,t)G(y,t)+\mathcal{P}_2^{(4)}(y,t)Er(y,t), \tag{4.27}$$

with $\mathcal{P}_1^{(4)}$ and $\mathcal{P}_2^{(4)}$ being the "polynomials"

$$\mathcal{P}_1^{(4)}(y,t) = -\frac{1}{3}(|y|^2t+4t^2), \tag{4.28a}$$

$$\mathcal{P}_2^{(4)}(y,t) = \frac{1}{12}(|y|^3+6|y|t). \tag{4.28b}$$

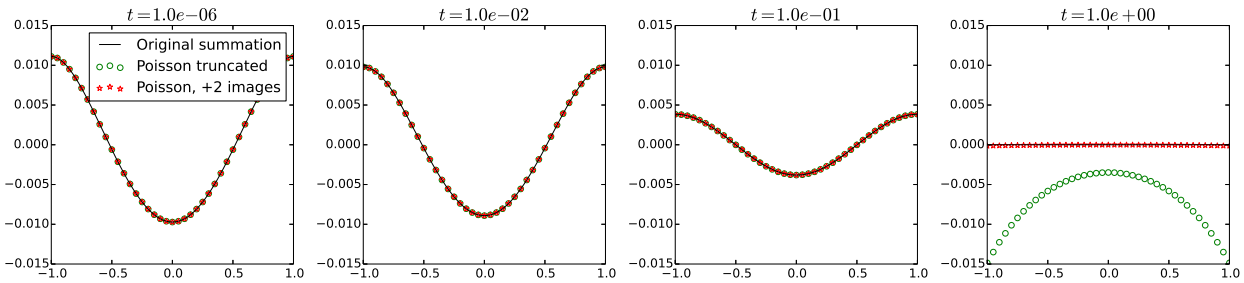

Figure 4.4: Evaluation of the left and right-hand sides of (4.30) with $N_{\max} = 10^4$ for the original summation (black), no extra images as in (4.31) (green), and two extra images kept (red).

Define the "composite" image $\mathcal{I}^{(4)}$:

$$\mathcal{I}^{(4)}(y,t) = -\left[\mathcal{P}_1^{(4)}(y,t)G(y,t) + \mathcal{P}_2^{(4)}(y,t)Er(y,t)\right] \tag{4.29}$$

Collecting all terms, With $\mathcal{I}^{(4)}(y,t)$ defined as in (4.27), the full summation is written as

$$\begin{aligned}\sum_{n=1}^{\infty}(-1)^n \frac{e^{-n^2\pi^2 t}}{(n\pi)^4}\cos(n\pi y) = -\frac{1}{24}\left(\frac{y^4}{2} - y^2 + \frac{7}{30}\right) - \frac{t}{4}\left(y^2 - \frac{1}{3}\right) - \frac{t^2}{4} \\ + \mathcal{I}^{(4)}(y-1,t) + \mathcal{I}^{(4)}(y+1,t) + {\sum}' \mathcal{I}^{(4)}(y+m,t).\end{aligned} \tag{4.30}$$

As before, this can be truncated to the ± 1 images, giving the approximation

$$\begin{aligned}\sum_{n=1}^{\infty}(-1)^n \frac{e^{-n^2\pi^2 t}}{(n\pi)^4}\cos(n\pi y) \approx -\frac{1}{24}\left(\frac{y^4}{2} - y^2 + \frac{7}{30}\right) - \frac{t}{4}\left(y^2 - \frac{1}{3}\right) - \frac{t^2}{4} \\ + \mathcal{I}^{(4)}(y-1,t) + \mathcal{I}^{(4)}(y+1,t).\end{aligned} \tag{4.31}$$

We compare the left and right sides of (4.30) in figure 4.4. This shows good agreement until order one time. Additionally, including one extra pair of images $\mathcal{I}^{(4)}(y \pm 3, t)$ corrects the behavior at $t = 1$.

Poisson summation for the first moment in the channel

Before continuing on to higher p, we use the formulae we have to rewrite the exact pointwise tracer mean T_1 given in [15]. The pointwise mean of the tracer distribution in the channel from a

strip initial condition (neglecting the Péclet number) has the solution

$$T_1(y,t) = \frac{1}{6}\left(\frac{y^4}{2} - y^2 + \frac{7}{30}\right) + 4\sum_{n=1}^{\infty} \frac{(-1)^n}{(n\pi)^4} e^{-(n\pi)^2 t}\cos(n\pi y). \tag{4.32}$$

If we multiply (4.30) by four and move the first term to the left, we get the generic expression T_1 (with $\mathrm{Pe} = 1$ and the images defined in (4.29)):

$$T_1(y,t) = \left(\frac{1}{3} - y^2\right) t - t^2 + \sum_{m=-\infty}^{\infty} 4\mathcal{I}^{(4)}(y + 2m + 1, t). \tag{4.33}$$

To get asymptotic short time behavior, keep only the ± 1 images as in (4.31), the result of which is:

$$T_1(y,t) \sim \left(\frac{1}{3} - y^2\right) t - t^2 - 4\mathcal{I}^{(4)}(y+1,t) - 4\mathcal{I}^{(4)}(y-1,t). \tag{4.34}$$

If we are looking to compare to our formal asymptotics, we drop the Er terms from the images $\mathcal{I}^{(4)}$. After substituting $\mathcal{P}_1^{(4)}$, this can be reduced to

$$T_1(y,t) \approx \left(\frac{1}{3} - y^2\right) t - t^2 + \frac{4}{3}\Big[\big[(y+1)^2 t + 4t^2\big] G(y+1,t) + \big[(y-1)^2 t + 4t^2\big] G(y-1,t)\Big]. \tag{4.35}$$

This is noticeably different than our "boundary-delta" approach, only contains $t^2 G(y,t)$ terms. A first sanity check for this formula is to see in what way this violates "mass" conservation:

$$\begin{aligned}
&\int_{-1}^{1} \left(\frac{1}{3} - y^2\right) t - t^2 + \frac{4}{3}\Big[\big[(y+1)^2 t + 4t^2\big] G(y+1,t) + \big[(y-1)^2 t + 4t^2\big] G(y-1,t)\Big] dy \\
&\qquad = -2t^2 + \frac{4}{3}\left[2t^2 \mathrm{Erf}\left(\frac{1}{\sqrt{t}}\right) - \frac{4e^{-1/t} t^{3/2}}{\sqrt{\pi}}\right] + \frac{4}{3}(4t^2)\mathrm{Erf}\left(\frac{1}{\sqrt{t}}\right) \\
&\qquad = \left[-2 + 8\mathrm{Erf}\left(\frac{1}{\sqrt{t}}\right)\right] t^2 - \frac{16}{3\sqrt{\pi}} e^{-1/t} t^{3/2},
\end{aligned} \tag{4.36}$$

which is asymptotic to $6t^2$, since $e^{-1/t} \to 0$ and $\mathrm{Erf}(1/\sqrt{t}) \to 1$ exponentially fast for $t \to 0$. Unlike our boundary delta approximation, this does not satisfy the conservation law $\int_{-1}^{1} C_1 dy \to 0$ exponentially fast as $t \to 0$. To this point we haven't fully looked at the importance of including Erfc terms, either. Both of these ideas will be addressed in section 4.3.4 below.

Verifying conserved quantities

The last question of the $\mathcal{O}(t^2)$ violation of "mass" conservation when truncating the Poisson sum version of T_1 to raises the question of conservation for all of the identities developed so far. If the identities hold, they should behave identically, but how the truncation of the Poisson sum behaves needs to be addressed. To evaluate the full summations, it will be necessary to evaluate expressions of the form

$$\int_{-1}^{1} \sum_{m=-\infty}^{\infty} f(y+2m+1)dy, \tag{4.37}$$

for f being images of some kind centered on the lattice $2\mathbb{Z}-1$. Assuming exchanging the sum and integral is valid, and changing variables for each integral lets us write

$$\begin{aligned} \int_{-1}^{1} \sum_{m=-\infty}^{\infty} f(y+2m+1)dy &= \sum_{m=-\infty}^{\infty} \int_{-1}^{1} f(y+2m+1)dy \\ &= \int_{-\infty}^{\infty} f(y)dy. \end{aligned} \tag{4.38}$$

Geometrically, the idea is that the area of the right tail of $G(y+1,t)$, say, is accounted for by the contribution of the images $\int_{-1}^{1} G(y+3,t)dy$, $\int_{-1}^{1} G(y+5,t)dy$, and so on. Similarly, the left tail of $G(y-1,t)$ is accounted for by the images at $y=3$, $y=5$, and so on.

This is convenient for us since it lets us sidestep the need to calculate infinite sums that may or may not have closed solutions when tackled directly. For example, for the seed formula (4.14), the left hand side integrates to zero termwise due to the cosines, and the right hand side integrates to zero, since

$$\begin{aligned} &\int_{-1}^{1} -\frac{1}{2} + \sum_{m=-\infty}^{\infty} G(y+2m+1,y)dy \\ &\quad = -2 * \frac{1}{2} + \left(\int_{-\infty}^{\infty} G(y,y_0)dy \right) \\ &\quad = -1+1 = 0. \end{aligned} \tag{4.39}$$

For the $p=2$ case, with (4.23), the left side integrates to zero as usual, and the right side integrates

	$t = 10^{-6}$	$t = 10^{-3}$	$t = 10^{-1}$	$t = 10^{0}$
$\lvert \int_{-1}^{1} (4.34)\ dy \rvert$	0^*	0^*	1.8×10^{-9}	5.5×10^{-2}
$\lvert \int_{-1}^{1} (4.35)\ dy \rvert$	6.0×10^{-12}	6.0×10^{-6}	6.0×10^{-2}	3.6×10^{0}

Table 4.1: Absolute error of the "mass" for the truncated Poisson summations (4.34), and the truncation when dropping $Er(\cdot)$ terms, (4.35) evaluated at different times. Dropping the Er terms is seen to violate the total mass condition in a quadratic fashion.

to zero since:

$$\begin{aligned}
&\int_{-1}^{1} \frac{1}{4}\left(y^2 - \frac{1}{3}\right) + \frac{t}{2} + \left[\sum_{m=-\infty}^{\infty} -2tG(y+2m+1,t) + \frac{1}{2}|y+2m+1|Er(y+2m+1,t)\right] dy \\
&\quad = 0 + 2 * \frac{t}{2} + \int_{-\infty}^{\infty} -2tG(y,t) + \frac{1}{2}|y|Er(y,t)dy \\
&\quad = t - 2t + t = 0.
\end{aligned} \tag{4.40}$$

Finally, the $p = 4$ case (4.30) integrates to zero:

$$\begin{aligned}
&\int_{-1}^{1} -\frac{1}{24}\left(\frac{y^4}{2} - y^2 + \frac{7}{30}\right) - \frac{1}{4}\left(y^2 - \frac{1}{3}\right)t - \frac{t^2}{4} \\
&\quad - \left[\sum_{m=-\infty}^{\infty} \mathcal{P}_1^{(4)}(y+2m+1,t)G(y+2m+1,t) + \mathcal{P}_2^{(4)}(y+2m+1,t)Er(y+2m+1,t)\right] dy \\
&\qquad = 0 + 0 - 2 * \frac{t^2}{4} + \int_{-\infty}^{\infty} \mathcal{P}_1^{(4)}(y,t)G(y,t) + \mathcal{P}_2^{(4)}(y,t)Er(y,t)dy \\
&\qquad = -\frac{1}{2}t^2 + 2t^2 - \frac{3}{2}t^2 = 0.
\end{aligned} \tag{4.41}$$

Now let's return to the question of approximating T_1 at short time. In contrast to (4.35), we now keep the Er terms centered at ± 1 as in (4.34). Table 4.3.4 compares the two approximations by looking at their integrals' rate of separation from zero. The 0^* terms are those which are zero up to hundreds of digits, and the asymptotic behavior of $6t^2$ previously calculated is apparent for (4.35). Considering this along with the evidence in the error of the pointwise solution seen in figure 4.3, it seems necessary to keep Er terms. What this suggests for the formal short time asymptotics in a generic domain is unclear. But what is clear that there is not a direct correspondence between the formal method and the Poisson resummation, except in the presence of heat kernels. It may be possible to more carefully analyze the short time behavior of $\mathcal{P}_1^{(4)}(y,t)G(y,t)$ and $\mathcal{P}_2^{(4)}(y,t)Er(y,t)$ to get a better picture, but this has not been explored yet.

Summary

In this section we have built up the identities needed to write the separation-of-variables solution for $T_1(y,t)$, in terms of a lattice sum of functions $\mathcal{I}^{(4)}(y,t)$ defined in equation (4.29). Work has been done to validate the identities directly, and various truncations of the lattice sum have been analyzed. Specifically, we have

- Used the Poisson summation formula to rewrite the sum $\sum(-1)^n e^{-(n\pi)^2 t}\cos(n\pi y)$ in terms of what turn out to be 1D heat kernels on a lattice (equation 4.14).
- Bootstrapped this result to obtain identities for the sums $\sum(-1)^n(n\pi)^{-p}e^{-(n\pi)^2 t}\cos(n\pi y)$ for $p=2$ (equation 4.23) and $p=4$ (equation 4.30). Essentially, this process is done by recursively integrating the $p-2$ identity in time and rearranging the resulting expressions.
- Validated these results by comparing partial sums over a range of timescales, both as functions of y, and analytically demonstrating preservation of "mass."
- Applied the $p=4$ formula (4.30) to get an equivalent expression for T_1, the first moment of the tracer problem in the channel. In this case, truncating the Poisson summation and dropping Er terms gives a similar, but distinct, expression to the "boundary delta" methodology. Specifically, equation (4.35) does not satisfy mass conservation asymptotically at the same rate in t as the "boundary delta" method does, which shows that Er terms are necessary if an exponential rate of mass is desired of the truncated Poisson sum as $t \to 0$.

Poisson summation of the second moment in the channel

In this section, we would like to develop the necessary identities to write down an expression for T_2, the second moment of tracer in the channel problem. For this, we'll need to solve the $p=6$ case, then derive similar formulas involving sines for odd p:

$$\sum_{n=1}^{\infty}\frac{(-1)^n}{(n\pi)^p}e^{-(n\pi)^2 t}\sin(n\pi y). \tag{4.42}$$

In short, these odd cases can be found by differentiating with respect to y. For the $p=1$ formula, we can differentiating the $p=2$ expression. The $p=5$ case required in T_2 will be found using the $p=6$ case.

The $p = 6$ case

From the $p = 4$ case (4.30), we repeat the same process. Integrating the left hand side in time gives:

$$\begin{aligned}\sum_{n=1}^{\infty}(-1)^n\left(\int_0^t \frac{e^{-n^2\pi^2 s}}{(n\pi)^4}ds\right)\cos(n\pi y) &= \sum_{n=1}^{\infty}(-1)^n\left(\frac{-e^{-n^2\pi^2 t}}{(n\pi)^6}+\frac{1}{(n\pi)^6}\right)\cos(n\pi y)\\ &= \frac{1}{288}\left(\frac{y^6}{5}-y^4+\frac{7y^2}{5}-\frac{31}{105}\right)-\sum_{n=1}^{\infty}(-1)^n\frac{e^{-n^2\pi^2 t}}{(n\pi)^6}\cos(n\pi y).\end{aligned} \tag{4.43}$$

The polynomial terms on the right integrate to:

$$\begin{aligned}\int_0^t -\frac{1}{24}\left(\frac{y^4}{2}-y^2+\frac{7}{30}\right)-\frac{s}{4}\left(y^2-\frac{1}{3}\right)-\frac{s^2}{4}ds&\\ = -\frac{t}{24}\left(\frac{y^4}{2}-y^2+\frac{7}{30}\right)-\frac{t^2}{8}\left(y^2-\frac{1}{3}\right)-\frac{t^3}{12}.&\end{aligned} \tag{4.44}$$

The images individually integrate to:

$$\int_0^t \mathcal{P}_1^{(4)}(y,s)G(y,s)+\mathcal{P}_2^{(4)}(y,s)Er(y,s)ds = \mathcal{P}_1^{(6)}(y,t)G(y,t)+\mathcal{P}_2^{(6)}(y,t)Er(y,t)\equiv\mathcal{I}^{(6)}(y,t), \tag{4.45}$$

with the "polynomials" $\mathcal{P}_1^{(6)}$ and $\mathcal{P}_2^{(6)}$:

$$\begin{aligned}\mathcal{P}_1^{(6)}(y,t) &= -\frac{1}{60}\big(|y|^4t+18|y|^2t^2+32t^3\big),\\ \mathcal{P}_2^{(6)}(y,t) &= \frac{1}{240}\big(|y|^5+20|y|^3t+60|y|t^2\big).\end{aligned} \tag{4.46}$$

Then the fully constructed solution is:

$$\begin{aligned}\sum_{n=1}^{\infty}(-1)^n\frac{e^{-n^2\pi^2 t}}{(n\pi)^6}\cos(n\pi y) = \frac{1}{288}\left(\frac{y^6}{5}-y^4+\frac{7y^2}{5}-\frac{31}{105}\right)+\frac{t}{24}\left(\frac{y^4}{2}-y^2+\frac{7}{30}\right)&\\ +\frac{t^2}{8}\left(y^2-\frac{1}{3}\right)+\frac{t^3}{12}+\sum_{m=-\infty}^{\infty}\mathcal{I}^{(6)}(y+2m+1,t).&\end{aligned} \tag{4.47}$$

The results are compared in figure 4.5. Moving forward, the notation will become even more bulky. Noticing a pattern in the polynomial terms, we define polynomials $q^{(k)}$ based on the recursion

$$\partial_{yy}q^{(k+2)} = q^{(k)}, \quad \partial_y q^{(k+2)}|_{\pm1} = 0, \quad \int_{-1}^{1} q^{(k+2)}dy = 0, \tag{4.48}$$

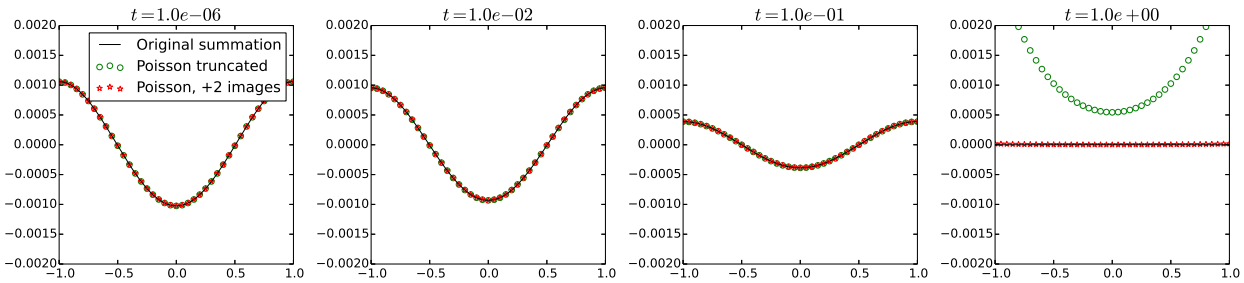

Figure 4.5: Evaluation of the left and right-hand sides of (4.47) with $N_{\max} = 10^4$ for the original summation (black), only the ± 1 images (green), and the ± 1, ± 3 images (red). As with previous cases, only two images are needed until order one time.

beginning with $q^{(0)} = -1/2$. This gives us the polynomials

$$q_2 = -(y^2 - 1/3)/4, \tag{4.49a}$$

$$q_4 = -(y^4/2 - y^2 + 7/30)/24, \tag{4.49b}$$

$$\vdots$$

which can be used to simplify the $p = 6$ (and further) formulas:

$$\sum_{n=1}^{\infty} (-1)^n \frac{e^{-n^2\pi^2 t}}{(n\pi)^6} \cos(n\pi y) = -\left(q^{(6)} + q^{(4)} t + q^{(2)} \frac{t^2}{2} + q^{(0)} \frac{t^3}{6} \right) + \sum_{m=-\infty}^{\infty} \mathcal{I}^{(6)}(y + 2m + 1, t). \tag{4.50}$$

The $p = 5$ case

Once the $p = 6$ identity is found, the $p = 5$ identity can be found by differentiating the expression termwise in y.

$$-\frac{\partial}{\partial y}\left(\sum_{n=1}^{\infty} (-1)^n \frac{e^{-n^2\pi^2 t}}{(n\pi)^6} \cos(n\pi y) \right) = \tag{4.51a}$$

$$-\frac{\partial}{\partial y}\left[-\left(q^{(6)} + q^{(4)} t + q^{(2)} \frac{t^2}{2} + q^{(0)} \frac{t^3}{6} \right) \right] + \sum_{m=-\infty}^{\infty} \left(-\frac{\partial}{\partial y} \right) \mathcal{I}^{(6)}(y + 2m + 1, t),$$

giving

$$\sum_{n=1}^{\infty} (-1)^n \frac{e^{-n^2\pi^2 t}}{(n\pi)^5} \sin(n\pi y) = q^{(5)} + q^{(3)} t + q^{(1)} \frac{t^2}{2} + \sum_{m=-\infty}^{\infty} \mathcal{I}^{(5)}(y + 2m + 1, t), \tag{4.52}$$

with images

$$\mathcal{I}^{(5)}(y,t) = \mathcal{P}_1^{(5)}(y,t)G(y,t) + \mathcal{P}_2^{(5)}(y,t)Er(y,t), \tag{4.53}$$

$$\mathcal{P}_1^{(5)}(y,t) = \frac{ty}{12}(10t+y^2), \qquad \mathcal{P}_2^{(5)}(y,t) = -\frac{1}{48}\,\mathrm{Sign}(y)\left(12t^2+12ty^2+y^4\right). \tag{4.54}$$

The $p = 8$ case

We continue from the $p = 6$ case (4.47). Integrating the left hand side in time gives:

$$\begin{aligned}\sum_{n=1}^{\infty}(-1)^n\left(\int_0^t \frac{e^{-n^2\pi^2 s}}{(n\pi)^6}ds\right)\cos(n\pi y) &= \sum_{n=1}^{\infty}(-1)^n\left(\frac{-e^{-n^2\pi^2 t}}{(n\pi)^8}+\frac{1}{(n\pi)^8}\right)\cos(n\pi y)\\ &= \frac{1}{8640}\left(\frac{3y^8}{28}-y^6+\frac{7y^4}{2}-\frac{31y^2}{7}+\frac{127}{140}\right)-\sum_{n=1}^{\infty}(-1)^n\frac{e^{-n^2\pi^2 t}}{(n\pi)^8}\cos(n\pi y)\\ &= q^{(8)}(y)-\sum_{n=1}^{\infty}(-1)^n\frac{e^{-n^2\pi^2 t}}{(n\pi)^8}\cos(n\pi y).\end{aligned} \tag{4.55}$$

The polynomial terms on the right integrate to:

$$\begin{aligned}&\int_0^t -\left(q^{(6)}+q^{(4)}s+q^{(2)}\frac{s^2}{2}+q^{(0)}\frac{s^3}{6}\right)ds\\ &\qquad = -\left(q^{(6)}t+q^{(4)}\frac{t^2}{2}+q^{(2)}\frac{t^3}{6}+q^{(0)}\frac{t^4}{24}\right).\end{aligned} \tag{4.56}$$

The images individually integrate to:

$$\int_0^t \mathcal{P}_1^{(6)}(y,s)G(y,s)+\mathcal{P}_2^{(6)}(y,s)Er(y,s)ds = \mathcal{P}_1^{(8)}(y,t)G(y,t)+\mathcal{P}_2^{(8)}(y,t)Er(y,t), \tag{4.57}$$

with the polynomials $\mathcal{P}_1^{(8)}$ and $\mathcal{P}_2^{(8)}$:

$$\mathcal{P}_1^{(8)}(y,t) = -\frac{1}{2520}\left(384t^4+348t^3|y|^2+40t^2|y|^4+t|y|^6\right) \tag{4.58a}$$

$$\mathcal{P}_2^{(8)}(y,t) = \frac{1}{10080}\left(840t^3|y|+420t^2|y|^3+42t|y|^5+|y|^7\right). \tag{4.58b}$$

The composite image is defined as

$$\mathcal{I}^{(8)}(y,t) = -\Big(\mathcal{P}_1^{(8)}(y,t)G(y,t) + \mathcal{P}_2^{(8)}(y,t)Er(y,t)\Big). \tag{4.59}$$

Putting all together we get.

$$\begin{aligned}\sum_{n=1}^{\infty}(-1)^n\frac{e^{-n^2\pi^2 t}}{(n\pi)^8}\cos(n\pi y) = q^{(8)} + q^{(6)}t + q^{(4)}\frac{t^2}{2} + q^{(2)}\frac{t^3}{6} + q^{(0)}\frac{t^4}{24} \\ + \sum_{m=-\infty}^{\infty}\mathcal{I}^{(8)}(y+2m+1,t).\end{aligned} \tag{4.60}$$

The $p = 7$ case

As with the $p = 5$ case, we obtain the identity for $p = 7$ by differentiating the $p = 8$ formula (4.60) with respect to y. The end result is

$$\sum_{n=1}^{\infty}(-1)^n\frac{e^{-n^2\pi^2 t}}{(n\pi)^7}\sin(n\pi y) = q^{(7)} + q^{(5)}t + q^{(3)}\frac{t^2}{2} + q^{(1)}\frac{t^3}{6} + \sum_{m=-\infty}^{\infty}\mathcal{I}^{(7)}(y+2m+1,t), \tag{4.61}$$

with polynomials $q^{(p)} = -\partial/\partial y\big(q^{(p+1)}\big)$, and images

$$\mathcal{I}^{(7)}(y,t) = \mathcal{P}_1^{(7)}(y,t)G(y,t) + \mathcal{P}_2^{(7)}(y,t)Er(y,t), \tag{4.62a}$$

$$\mathcal{P}_1^{(7)}(y,t) = \frac{ty}{360}(132t^2 + 28ty^2 + y^4), \tag{4.62b}$$

$$\mathcal{P}_2^{(7)}(y,t) = -\frac{1}{1440}\,\mathrm{Sign}(y)\left(120t^3 + 180t^2y^2 + 30ty^4 + y^6\right). \tag{4.62c}$$

Expression for T_2

The expression for the second moment T_2 in [15], setting the tracer's initial variance $\sigma^2 = 0$ and Péclet number $\mathrm{Pe} = 1$, is

$$T_2(y,t) = 2t + \mathcal{Q}_1(y,t;n) + \sum_{n=1}^{\infty}\mathcal{Q}_2(y,t;n)\cos(n\pi y) + \mathcal{Q}_3(y,t;n)\sin(n\pi y). \tag{4.63}$$

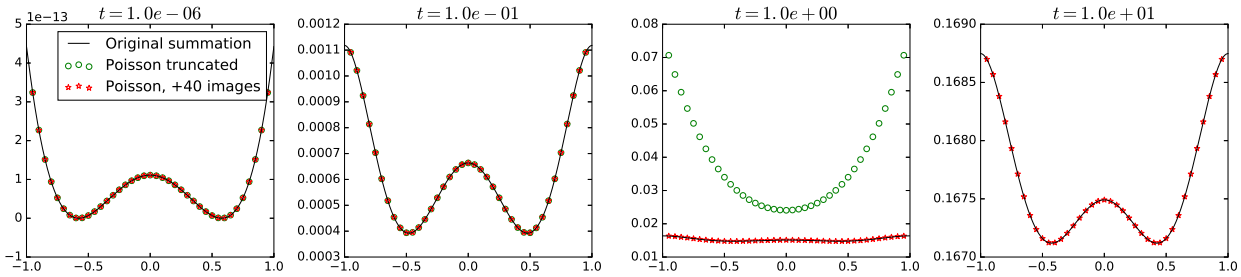

Figure 4.6: Evaluation of (4.64) with $N_{\max} = 10^4$ for the original summation (black), and the Poisson summation equivalents keeping only the ± 1 images (green circles), and the first forty images centered around $[-1, 1]$ (red stars). Green circles not shown in the final panel.

The coefficients $\mathcal{Q}_2$ and $\mathcal{Q}_3$ need to be unpacked, as they contain various powers of $n\pi$. We get an expression

$$\begin{aligned} T_2(y,t) = \mathcal{F}_1(y,t) + \sum_{n=1}^{\infty} \mathcal{F}_2(y,t)\frac{(-1)^n}{(n\pi)^6} e^{-(n\pi)^2 t}\cos(n\pi y) + \mathcal{F}_3(y,t)\frac{(-1)^n}{(n\pi)^8} e^{-(n\pi)^2 t}\cos(n\pi y) \\ + \sum_{n=1}^{\infty} \mathcal{F}_4(y,t)\frac{(-1)^n}{(n\pi)^5} e^{-(n\pi)^2 t}\sin(n\pi y) + \mathcal{F}_5(y,t)\frac{(-1)^n}{(n\pi)^7} e^{-(n\pi)^2 t}\sin(n\pi y), \end{aligned} \tag{4.64}$$

with the coefficients

$$\begin{aligned} \mathcal{F}_1 &= 2t + \frac{1}{226800}(-413 + 3840t - 1020y^2 + 3570y^4 - 2940y^6 + 675y^8), \\ \mathcal{F}_2 &= \frac{34}{3} - 4t + 2y^2, \qquad \mathcal{F}_3 = -128, \\ \mathcal{F}_4 &= \frac{4}{3}(y^3 - y), \qquad \mathcal{F}_5 = -4y. \end{aligned} \tag{4.65}$$

In this form, the derived identities (4.52, 4.47, 4.61, 4.60) can be substituted directly. In figure 4.6 we compare the results. Towards the interest of capturing behavior into the diffusive timescale, we replace the panel at $t = 10^{-2}$ with one at $t = 10$. We need to use forty images at $y = \pm 1, \pm 3, ..., \pm 39$ to obtain good agreement at $t = 10$. Keeping only the images at $y = \pm 1$ is only satisfactory until $t = 10^{-1}$.

Third moment in the channel and comparison for centered statistics

At this point, the procedure should be clear. We give the formulae for $p = 9$ through 12, as are needed to calculate T_3. Once we have these, we can construct the Poisson summation versions of the the variance and skewness. For brevity, we provide the explicit formulae for the image functions $\mathcal{I}^{(p)}$

here, but the forms of the polynomials $q^{(p)}$ are given later in section 4.6.

For $p = 10$, we obtain

$$\sum_{n=1}^{\infty}(-1)^n\frac{e^{-n^2\pi^2t}}{(n\pi)^{10}}\cos(n\pi y) = -\left(\sum_{k=0}^{5}q^{(10-2k)}(y)\frac{t^k}{k!}\right) + \sum_{m=-\infty}^{\infty}\mathcal{I}^{(10)}(y+2m+1,t), \tag{4.66}$$

with the images

$$\mathcal{I}^{(10)}(y,t) = \mathcal{P}_1^{(10)}(y,t)G(y,t) + \mathcal{P}_2^{(10)}(y,t)Er(y,t), \tag{4.67a}$$

$$\mathcal{P}_1^{(10)}(y,t) = -\frac{1}{181440}(6144t^5 + 7800t^4y^2 + 1380t^3y^4 + 70t^2y^6 + ty^8), \tag{4.67b}$$

$$\mathcal{P}_2^{(10)}(y,t) = \frac{1}{725760}(15120t^4|y| + 10080t^3|y|^3 + 1512t^2|y|^5 + 72t|y|^7 + |y|^9). \tag{4.67c}$$

For $p = 12$, we obtain

$$\sum_{n=1}^{\infty}(-1)^n\frac{e^{-n^2\pi^2t}}{(n\pi)^{12}}\cos(n\pi y) = \sum_{k=0}^{6}q^{(12-2k)}(y)\frac{t^k}{k!} + \sum_{m=-\infty}^{\infty}\mathcal{I}^{(12)}(y+2m+1,t), \tag{4.68}$$

with the images

$$\mathcal{I}^{(12)}(y,t) = -\left(\mathcal{P}_1^{(12)}(y,t)G(y,t) + \mathcal{P}_2^{(12)}(y,t)Er(y,t)\right), \tag{4.69a}$$

$$\mathcal{P}_1^{(12)}(y,t) = -\frac{122880t^6 + 202320t^5y^2 + 48720t^4y^4 + 3752t^3y^6 + 108t^2y^8 + ty^{10}}{19958400} \tag{4.69b}$$

$$\mathcal{P}_2^{(12)}(y,t) = \frac{332640t^5|y| + 277200t^4|y|^3 + 55440t^3|y|^5 + 3960t^2|y|^7 + 110t|y|^9 + |y|^{11}}{79833600} \tag{4.69c}$$

Differentiating these equations in y gives the formula for $p = 9$:

$$\sum_{n=1}^{\infty}(-1)^n\frac{e^{-n^2\pi^2t}}{(n\pi)^{9}}\sin(n\pi y) = -\left(\sum_{k=0}^{4}q^{(9-2k)}(y)\frac{t^k}{k!}\right) + \sum_{m=-\infty}^{\infty}\mathcal{I}^{(9)}(y+2m+1,t), \tag{4.70a}$$

$$\mathcal{I}^{(9)}(y,t) = \mathcal{P}_1^{(9)}(y,t)G(y,t) + \mathcal{P}_2^{(9)}(y,t)Er(y,t), \tag{4.70b}$$

$$\mathcal{P}_1^{(9)}(y,t) = \frac{ty}{20160}(2232t^3 + 740t^2y^2 + 54ty^4 + y^6), \tag{4.70c}$$

$$\mathcal{P}_2^{(9)}(y,t) = -\frac{\text{Sign}(y)}{80640}(1680t^4 + 3360t^3y^2 + 840t^2y^4 + 56ty^6 + y^8), \tag{4.70d}$$

and similarly for $p = 11$:

$$\sum_{n=1}^{\infty}(-1)^n \frac{e^{-n^2\pi^2 t}}{(n\pi)^{11}} \sin(n\pi y) = \sum_{k=0}^{5} q^{(11-2k)}(y)\frac{t^k}{k!} + \sum_{m=-\infty}^{\infty} \mathcal{I}^{(11)}(y+2m+1,t), \tag{4.71a}$$

$$\mathcal{I}^{(11)}(y,t) = -\left(\mathcal{P}_1^{(11)}(y,t)G(y,t) + \mathcal{P}_2^{(11)}(y,t)Er(y,t)\right), \tag{4.71b}$$

$$\mathcal{P}_1^{(11)}(y,t) = \frac{46320t^5y + 21120t^4y^3 + 2352t^3y^5 + 88t^2y^7 + ty^9}{1814400}, \tag{4.71c}$$

$$\mathcal{P}_2^{(11)}(y,t) = -\frac{\mathrm{Sign}(y)\left[30240t^5 + 75600t^4y^2 + 25200t^3y^4 + 2520t^2y^6 + 90ty^8 + y^{10}\right]}{7257600}, \tag{4.71d}$$

As with T_2, the formula for T_3 in [15] needs to be unpacked to compare to the Poisson sum. The end result is, with $\sigma^2 = 0$ and $\mathrm{Pe} = 1$,

$$T_3(y,t) = \mathcal{G}_0(y,t) + \sum_{n=1}^{\infty}\sum_{k=2}^{6} \mathcal{G}_{2k}(y,t)\frac{e^{-(n\pi)^2 t}\cos(n\pi y)}{(n\pi)^{2k}} + \mathcal{G}_{2k+1}(y,t)\frac{e^{-(n\pi)^2 t}\sin(n\pi y)}{(n\pi)^{2k+1}} \tag{4.72}$$

with coefficients

$$\begin{aligned}\mathcal{G}_0 = \frac{1}{30}t(7 - 30y^2 + 15y^4) \\ - \frac{4076777}{13621608000} + \frac{8447y^2}{4989600} - \frac{713y^4}{907200} - \frac{y^6}{1200} + \frac{13y^8}{12096} \\ - \frac{211y^{10}}{453600} + \frac{y^{12}}{14784} + \frac{244t}{155925} - \frac{8ty^2}{945} + \frac{4ty^4}{945},\end{aligned} \tag{4.73a}$$

$$\mathcal{G}_4 = 24t, \tag{4.73b}$$

$$\mathcal{G}_5 = 0, \tag{4.73c}$$

$$\mathcal{G}_6 = -\frac{1}{3}y^2 + \frac{2}{3}y^4 - \frac{1}{3}y^6 - \frac{8}{15}t, \tag{4.73d}$$

$$\mathcal{G}_7 = -\frac{28}{5}y + 4y^3 + \frac{8}{5}y^5 + 2ty - 2ty^3, \tag{4.73e}$$

$$\mathcal{G}_8 = \frac{691}{20} + 3t^2 + \frac{9}{2}y^2 + \frac{23}{4}y^4 - 31t - 3ty^2, \tag{4.73f}$$

$$\mathcal{G}_9 = 44y - 76y^3 + 6ty, \tag{4.73g}$$

$$\mathcal{G}_{10} = -\frac{3705}{2} - \frac{231}{2}y^2 + 231t, \tag{4.73h}$$

$$\mathcal{G}_{11} = 231y, \tag{4.73i}$$

$$\mathcal{G}_{12} = 14640, \tag{4.73j}$$

$$\mathcal{G}_{13} = 0. \tag{4.73k}$$

Now that we have expressions for the first three moments in (4.33), (4.64), and (4.72), we can calculate variance and skewness. Figure 4.7 shows a detailed picture of the variance and skewness evolution. In the variance, boundary layers form on the wall at early timescales. The peaks in the variance grow and migrate towards the center in time. On later timescales this structure remains, and increases at a linear rate consistent with the effective diffusivity independent of y. Interestingly, the centerline sees minimum variance for all time. The noise at $t = 10^{-6}$ in both the original and Poisson sum versions is presumed to be due to critical numerical cancellations. There is not much that can be done about this with the original sum, but it may be possible with the Poisson sum to simplify the expression for the variance by re-collecting the expression in terms of $G(y + 2m + 1, t)$ and $Er(y + 2m + 1, t)$ and looking for any cancellations *a priori*.

For the skewness in figure 4.7, we have re-introduced the Péclet number into the formulae and set

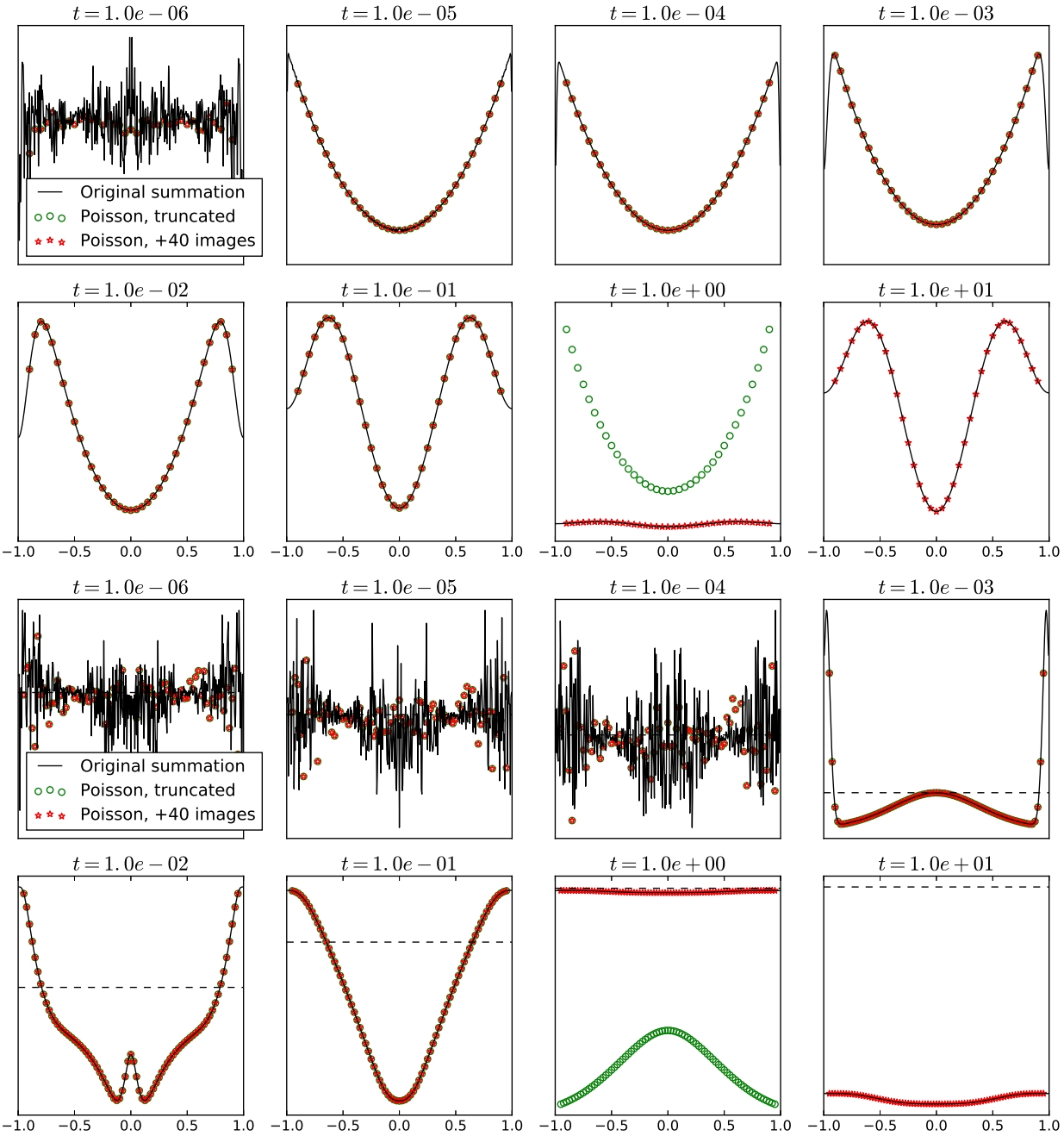

Figure 4.7: Evaluation of the channel variance (top rows) and skewness (bottom rows) with $N_{\max} = 10^4$ for the original summation (black), and the Poisson summation equivalents keeping only the ± 1 images (green circles), and the first forty images centered around $[-1, 1]$ (red stars). The line $Sk = 0$ (dashed) is included for reference. Green circles not shown in the final panel. The variance uses $\mathrm{Pe} = 1$, while the skewness uses $\mathrm{Pe} = 10^3$ to emphasize Poisson sum's resolution of the fine scale structure.

$\mathrm{Pe} = 10^3$ to match the fine scale structure we see in simulations elsewhere. In this case, the numerical issues are even more apparent. We have verified that this issue is independent of (nonzero) Péclet number, again suggesting an issue of numerical cancellation issues in forming the skewness. Despite this problem, it is clear starting at $t = 10^{-3}$ that a boundary layer has formed due to diffusive pumping which affects the third centered moment (the numerator of the skewness) to dominate the variance. Fine scale structure is resolved in intermediate timescales, and the long time asymptotic behavior being independent of y is apparent at $t = 10$.

Summary of identities

This section summarizes all the identities developed in a table for easier access, in addition to restating the notation used. The identities for p even were derived recursively using the expression:

$$\sum_{n=1}^{\infty} \frac{(-1)^n}{(n\pi)^{p+2}} e^{-(n\pi)^2 t} \cos(n\pi y) = \sum_{n=1}^{\infty} \frac{(-1)^n}{(n\pi)^{p+2}} \cos(n\pi y) - \int_0^t \sum_{n=1}^{\infty} \frac{(-1)^n}{(n\pi)^p} e^{-(n\pi)^2 s} \cos(n\pi y) ds, \quad (4.74)$$

and beginning the recursion at $p = 0$ by replacing the last sum via Poisson summation formula (4.14). The end result is generically a polynomial in y and t plus a lattice sum of images $\mathcal{I}^{(p)}$:

$$\begin{aligned} \sum_{n=1}^{\infty} \frac{(-1)^n}{(n\pi)^p} e^{-(n\pi)^2 t} \cos(n\pi y) &= \mathcal{P}_0^{(p)}(y,t) + \sum_{m=-\infty}^{\infty} \mathcal{I}^{(p)}(y + 2m + 1, t), \\ \mathcal{I}^{(p)} &= (-1)^p \left(\mathcal{P}_1^{(p)}(y,t) G(y,t) + \mathcal{P}_2^{(p)}(y,t) Er(y,t) \right), \end{aligned} \quad (4.75)$$

where the leading $(-1)^p$ in $\mathcal{I}^{(p)}$ is due to the subtraction in (4.74). The two basic "image functions" are the one dimensional heat kernel G, and a function related to its integral, Er,

$$G(y,t) = \frac{1}{\sqrt{4\pi t}} e^{-\frac{y^2}{4t}}, \qquad Er(y,t) = \mathrm{Erfc}\left(\frac{|y|}{\sqrt{4t}} \right), \quad (4.76)$$

and $\mathcal{P}_0^{(p)}(y,t)$, $\mathcal{P}_1^{(p)}(y,t)$, and $\mathcal{P}_2^{(p)}(y,t)$ are three "polynomials" indexed by the power p of the denominator $(n\pi)^p$ in the original summand. The coefficients $\mathcal{P}_1^{(p)}(y,t)$ and $\mathcal{P}_2^{(p)}(y,t)$ are found by applying the time integral while disregarding any leading coefficients:

$$\int_0^t \mathcal{P}_1^{(p)}(y,s) G(y,s) + \mathcal{P}_2^{(p)}(y,s) Er(y,s) ds, \quad (4.77)$$

and collecting the resulting expression in terms of G and Er.

The case of p odd (with sines instead of cosines) can be handled by differentiating the $p+1$ case in y and negating:

$$-\frac{\partial}{\partial y}\left(\sum_{n=1}^{\infty}\frac{(-1)^n}{(n\pi)^{p+1}}e^{-(n\pi)^2 t}\cos(n\pi y)\right)=\sum_{n=1}^{\infty}\frac{(-1)^n}{(n\pi)^p}e^{-(n\pi)^2 t}\sin(n\pi y) \tag{4.78}$$

which gives functionally similar formulas for p odd, with $\mathcal{P}_0^{(p)}$, $\mathcal{P}_1^{(p)}$, and $\mathcal{P}_2^{(p)}$ as negative derivatives of their $p+1$ counterparts:

$$\sum_{n=1}^{\infty}\frac{(-1)^n}{(n\pi)^p}e^{-(n\pi)^2 t}\sin(n\pi y)=\mathcal{P}_0^{(p)}(y,t)+\sum_{m=-\infty}^{\infty}\mathcal{I}^{(p)}(y+2m+1,t), \tag{4.79a}$$

$$\mathcal{I}^{(p)}(y,t)=\mathcal{P}_1^{(p)}(y,t)G(y,t)+\mathcal{P}_2^{(p)}(y,t)Er(y,t). \tag{4.79b}$$

The leading polynomials $\mathcal{P}_0^{(p)}$, for p even, can be solved recursively as

$$\begin{aligned}&\mathcal{P}_0^{(2k+2)}(y,t)=q^{(2k)}(y)-\int_0^t\mathcal{P}_0^{(2k)}(y,s)ds,\quad \text{for } k=1,2,...,\\&\mathcal{P}_0^{(0)}(y,t)=q^{(0)}=-1/2,\end{aligned} \tag{4.80}$$

and the expression $q^{(2k)}(y)$ is the time-independent series in (4.74),

$$q^{(2k)}(y)=(-1)^{k+1}\sum_{n=1}^{\infty}\frac{(-1)^n}{(n\pi)^{2k}}\cos(n\pi y). \tag{4.81}$$

The $q^{(2k)}$ themselves are found in closed form by solving a recursive set of Poisson problems with Neumann boundary conditions and a zero-mean constraint:

$$\partial_{yy}q^{(2k+2)}=q^{(2k)},\qquad \partial_y q^{(2k+2)}\Big|_{\pm1}=0,\qquad \int_{-1}^{1}q^{(2k+2)}dy=0. \tag{4.82}$$

This is equivalent due to the uniqueness of solution of this class of problem. Essentially, the series expression is a eigenfunction solution of the problem. The fact that the drivers are polynomials and the problem is one dimensional implies that the system of problems stays within the space of polynomials (since solving the problem only requires integrating in y twice.)

It may be possible to obtain similar recursive formulae for the leading polynomials $\mathcal{P}_1^{(p)}$ and

p	$q^{(p)}(y)$
0	$-1/2$
1	$y/2$
2	$\big(-y^2+1/3\big)/4$
3	$\big(y^3-y\big)/12$
4	$\big(-y^4+2y^2-7/15\big)/48$
5	$\big(6y^5-20y^3+14y\big)/1440$
6	$\big(31/21-7y^2+5y^4-y^6\big)/1440$
7	$\big(24y^7-168y^5+392y^3-248y\big)/241920$
8	$\big(-3y^8+28y^6-98y^4+124y^2-127/5\big)/241920$
9	$\big(10y^9-120y^7+588y^5-1240y^3+762y\big)/7257600$
10	$\big(-y^{10}+15y^8-98y^6+310y^4-381y^2+2555/33\big)/7257600$
11	$\big(12y^{11}-220y^9+1848y^7-8184y^5+16764y^3-10220y\big)/958003200$
12	$\big(-y^{12}+22y^{10}-231y^8+1364y^6-4191y^4+5110y^2-1414477/1365\big)/958003200$

Table 4.2: Polynomials $q^{(p)}$, $p=0,1,...,12$ necessary for Poisson summation of the solutions for the first three moments in the channel.

$\mathcal{P}_2^{(p)}$, aside from integrating the individual images in time and collecting in terms of the functions G and Er. There is obviously some structure in the problem, due to the similarity in the leading polynomials across values of p. This is a future direction to explore.

For reference we have tabulated in Table 4.6 the functions $q^{(p)}$. The polynomials $\mathcal{P}_1^{(p)}$ and $\mathcal{P}_2^{(p)}$ have been defined in the earlier subsections for the values of p as needed to calculate the channel skewness, and following the procedure described in this subsection is enough to derive expressions for other values of p as needed.

CHAPTER 5

Monte Carlo simulation

This chapter discusses our use of a Monte Carlo method to numerically solve the advection-diffusion equation and obtain statistics of the tracer $T(\mathbf{x}, t)$.

We have opted to use Fortran 90 for the numerical simulations because of its well-known speed and efficiency for scientific computation (though in principle any another compiled language could be used). The appendix contains the full source code necessary to compile and run these simulations, aside from two external software packages used.

The first external software used is for the Mersenne Twister pseudorandom number generator, introduced by Matsumoto and Nishimura [17], and implemented in Fortran by Nishimura, available at [18]. The second is a version of the HDF5 software [19], which is a file format that allows one to store large amounts of data, along with metadata (such as problem parameters) in a single file. Tools to read from HDF files exist in most major high level programming languages.

Brownian motions and their connection to advection-diffusion problems

In one dimension, a Brownian motion $B(t)$ with drift and variance parameters μ and σ^2 is a random function defined through the following properties:

$$B(t) - B(0) \sim \mathcal{N}(\mu t, \sigma^2 t) \tag{5.1a}$$

$$B(t_2) - B(t_1) = B(t_2 - t_1) - B(0) \quad (\text{with } t_2 \geq t_1) \tag{5.1b}$$

$$B(t) = \mu t + \sigma W(t), \tag{5.1c}$$

where $\mathcal{N}(\mu t, \sigma^2 t)$ is a Gaussian distribution with mean μt and variance $\sigma^2 t$, $W(t)$ is "simple" Brownian motion with $\mu = 0$, $\sigma = 1$. One can form a probability density function:

$$p(x, t) = \lim_{\Delta x \to 0} \frac{\mathbb{P}(x \leq B(t) \leq x + \Delta x)}{\Delta x}. \tag{5.2}$$

Using this definition and the properties of Brownian motion, it can be shown [20, 21] that $p(x,t)$ satisfies the one-dimensional free space advection-diffusion equation

$$\frac{\partial p}{\partial t} + \mu \frac{\partial p}{\partial x} = \frac{\sigma^2}{2} \frac{\partial^2 p}{\partial x^2}, \tag{5.3a}$$

$$p(x,0) = \delta(x), \tag{5.3b}$$

where we assume without loss of generality $B(0) = 0$, and $\delta(x)$ is the Dirac delta distribution. This suggests that a Monte Carlo method for solving the advection diffusion equation (5.3) is to generate a large number of independent sample paths of a Brownian motion and use the law of large numbers to approximate the probability density.

It is convenient to understand the Brownian motion with drift and variance parameters as the solution to a stochastic differential equation (SDE). This term is a misnomer, as Brownian motions are generically nondifferentiable for all t. This can be seen directly from the difference quotient:

$$\frac{B(t+h) - B(t)}{h} = \frac{B(h) - B(0)}{h} = \frac{\mu h + W(h)}{h} \tag{5.4a}$$

$$= \mu + \frac{\sqrt{h} W(1)}{h} = \mu + W(1) \frac{1}{\sqrt{h}}, \tag{5.4b}$$

so that the quotient diverges when sending $h \to 0$. If we wish to generate sample paths on a sequence of t values $\{t_0 = 0, t_1, ..., t_n\}$, we can again use the properties of Brownian motion to write the difference equation

$$\begin{aligned} &X(t_0) = 0, \\ &X(t_k) - X(t_{k-1}) = \mu(t_k - t_{k-1}) + \sigma \left[W(t_k) - W(t_{k-1}) \right], \qquad k = 1, ..., n. \end{aligned} \tag{5.5}$$

This is essentially the Euler-Maruyama timestepping scheme for SDEs [22], the generalization of the forward Euler method in ordinary differential equations. It is possible to regard SDEs in a more abstract setting and derive higher-order timestepping schemes, but for our purposes it is enough to use the above formulation.

In the standard notation for SDEs, this is written in shorthand as

$$dX(t) = \mu dt + \sigma dW(t). \tag{5.6}$$

This idea can be extended in a relatively straightforward way to handle other initial and boundary conditions; if for instance we required the solution to the PDE in the bounded domain $[-a, a]$ with some non-point source initial condition $f(x)$:

$$\frac{\partial p}{\partial t} + \mu \frac{\partial p}{\partial x} = \frac{\sigma^2}{2} \frac{\partial^2 p}{\partial x^2}, \tag{5.7a}$$

$$p(x, 0) = f(x), \qquad \left.\frac{\partial p}{\partial x}\right|_{x=\pm a} = 0 \tag{5.7b}$$

the corresponding sample path would pull the initial condition from the PDF $f(x)$ (without loss of generality $\int_{\mathbb{R}} f(x)dx = 1$), that is, $X(0) \sim f(x)$, and impose "reflective" boundary conditions at $x = \pm a$, the analogue of Neumann boundary conditions.[1] Specifically, if a sample path were to exit the domain $[-a, a]$, it would reflect back into the domain in an elastic way – preserving the sampled distance $|X(t_k) - X(t_{k-1})|$ before imposing the boundary condition, and reflecting inwards from the tangent plane at the boundary at the same angle relative to the local surface normal vector.

The theory follows through if we move to two spatial dimensions. For example, the free space two-dimensional advection-diffusion with constant advection vector $\underline{\mu} = \mu_1 \mathbf{i} + \mu \mathbf{j}$ and isotropic diffusion κ has the form

$$\frac{\partial p}{\partial t} + \underline{\mu} \cdot \nabla p = \kappa \nabla^2 p, \tag{5.8a}$$

$$p(x, y, 0) = \delta(x)\delta(y), \tag{5.8b}$$

and the SDE analogue is

$$dX(t) = \mu_1 dt + \sqrt{2\kappa} dW_1(t), \tag{5.9a}$$

$$dY(t) = \mu_2 dt + \sqrt{2\kappa} dW_2(t), \tag{5.9b}$$

$$X(0) = Y(0) = 0, \tag{5.9c}$$

with W_1 and W_2 being independent simple Brownian motions. In this case, the X and Y equations

[1] The analogue of Dirichlet boundary condition is known as an "absorbing" boundary condition.

are independent and the diffusion could be handled as two one-dimensional equations in succession. With a variable coefficient advection term, as in the tracer problem in three dimensions, with initial data $f(x) = \delta(x)$ (say), the corresponding set of equations is

$$dX(t) = u(Y(t), Z(t))dt + \sqrt{2\kappa}dW_1(t), \tag{5.10a}$$

$$dY(t) = \sqrt{2\kappa}dW_2(t), \tag{5.10b}$$

$$dZ(t) = \sqrt{2\kappa}dW_3(t), \tag{5.10c}$$

$$X(0) = 0,\ (Y, Z) \sim U(\Omega), \tag{5.10d}$$

where $U(\Omega)$ denotes a uniform random distribution in the cross section.[2] Again, W_2 and W_3 are independent except when needing to respect reflecting boundary conditions.

Implementation of the Monte Carlo method

In the subsections below, we detail all the aspects of the implementation of the Monte Carlo method for our problem.

In the numerics the nondimensional equations are used, so that the domains are bounded in $(y, z) \in [-1, 1] \times [-1/\lambda, 1/\lambda]$. Time is nondimensionalized relative to the diffusive timescale $t = (a^2/\kappa)\tau$. In this setup the SDE takes the vector form

$$d\mathbf{X}(t) = \mathrm{Pe}\, u(Y(t), Z(t))\mathbf{i}\, dt + \sqrt{2}d\mathbf{W}(t), \tag{5.11}$$

with Pe the Péclet number, a specified initial condition $f(x, y, z)$, and reflecting boundary conditions on Ω.

Implementing this is broken down into several parts: specifying the initial condition $f(x, y, z)$, (pre)calculating the flow $u(y, z)$, implementing Brownian motion and enforcing reflective boundary conditions $d\mathbf{W}$, and calculating the various statistics of the particle ensemble $\mathbf{X}(t)$.

Specifying the initial conditions

To compare with theory and experiment, we usually consider a few special cases of initial data:

[2] However, in practice, we may opt to initialize (Y, Z) in a regular grid in the cross section instead of using a random initial condition.

1. “Strip" initial data, $f(x, y, z) = \delta(x)$,

2. Gaussian initial data in x alone; with mean zero and some variance σ^2: $f(x, y, z) = G(x; \sigma^2)$;

3. Point source initial data, $f(x, y, z) = \delta(x)\delta(y - y_0)\delta(z - z_0)$.

Specifically working towards replicating experiment, we also want to be able to diffuse any initial condition before turning on the flow. To cover all cases, we allow one to specify the following in the code in regards to the initial condition:

- The approximate number of particles n_c desired for the lattice in the cross section Ω;
- The width of the lattice in the x direction (only considered if $n_x > 1$);
- The number of lattice strips to place in the x direction, n_x;
- The initial position (y_0, z_0) (only considered if $n_c = 1$);
- The amount of nondimensional time τ_{diff} to diffuse the initial condition formed above (without advection).

If $n_c = 1$, a point source initial condition is assumed. Otherwise, the numbers $p = \lfloor \sqrt{n_c \lambda} \rfloor$ and $q = \lceil \sqrt{n_c / \lambda} \rceil$ (with floor $\lfloor \cdot \rfloor$ and ceiling $\lceil \cdot \rceil$ functions) are the chosen as the number of discretization points in the y and z directions respectively, so that both the resulting lattice is approximately uniformly spaced in y and z, and the total number of particles is still $pq \approx n_c$. For elliptical domains, n_c is scaled up within the code by the ratio of the area of the rectangle and inscribed ellipse, and points outside the ellipse are thrown out, so that the original value of n_c is approximately preserved.

The following parameter settings give desired initial conditions:

1. For a strip $f = \delta(x)$, one would choose any n_c, set $n_x = 1$, and $\tau_{\text{diff}} = 0$.

2. For a Gaussian initial condition with variance σ^2, form the strip initial condition above, and chose $\tau_{\text{diff}} = \sigma^2/2$, using the fact that the heat kernel $G(x, \tau)$ satisfies $\sigma^2 \equiv \int x^2 G(x, \tau) dx = 2\tau$;

3. For a point source $f = \delta(x)\delta(y - y_0)\delta(z - z_0)$, set $n_c = 1$, choose any n_x and (y_0, z_0), and $\tau_{\text{diff}} = 0$.

We have also implemented the ability to import an arbitrary initial condition from a file in HDF format. This is useful if we wish to replicate experimental initial conditions, or create more exotic initial conditions in a high-level programming language; for instance, Python or Matlab/Octave.

Calculation of the flow

In the case of the infinite channel and any elliptical pipe, the mean-zero flow is calculated directly during the simulation whenever needed, as they are polynomial, and inexpensive to evaluate:

$$u_c(y) = \frac{1}{3} - y^2, \tag{5.12}$$

$$u(y,z) = \frac{1}{1+\lambda^2}\left(\frac{1}{2} - y^2 - \lambda^2 z^2\right). \tag{5.13}$$

In the rectangular duct, it is inefficient to directly evaluate the truncated sum

$$u(y,z) = u_c(y) + \sum_{k=1}^{N} a_k(\lambda)\big[\cos((k-1/2)\pi y)\cosh((k-1/2)\pi z) - b_k(\lambda)\big] \tag{5.14}$$

on the fly, as it requires N evaluations of (hyperbolic) cosine. Instead, we precompute the flow on a fine grid $(y_{i,j}, z_{i,j})$. Then, on any timestep, given a position (y,z), the indices i,j for the surrounding rectangle $[y_{i,j}, y_{i+1,j}] \times [z_{i,j}, z_{i,j+1}]$ are found, and bilinear interpolation is used to approximate the flow value $u(y,z)$.

The choice of a fine grid for the precomputed flow can be modified internally; both a uniform grid and one which uses Chebyshev nodes have been implemented. The uniform grid has the advantage that lookups can be done fast by using integer division. Using Chebyshev nodes requires a slower binary search due to non-uniform spacing, but fewer total grid points are necessary to resolve the boundary layer of the flow for small aspect ratios λ, so it may be advantageous in some cases.

Implementation of diffusion and enforcing boundary conditions

At the heart of any Monte Carlo is the generation of pseudorandom numbers, and thus a pseudorandom number generator. We use a Fortran module from [18] that implements the Mersenne Twister, a well known, high quality uniform random number generator [17] (the details of which are outside the scope of this dissertation).

Simulating Brownian motion, however, requires normally distributed random numbers with arbitrary mean and variance; $X \sim \mathcal{N}(\mu, \sigma^2)$. Using basic properties of normal random variables,

we can let $Y \sim \mathcal{N}(0,1)$ and defining $X = \mu + \sigma Y$ gives us the needed distribution to the random variable. Therefore, we need to only generate samples from the standard normal distribution. To do this, we implement Marsaglia's polar variant [23] of the Box-Muller method. These methods take pairs of independently distributed uniform random numbers (v_1, v_2) and applies a transformation which produces independently distributed normal random numbers (w_1, w_2). We briefly compare the two algorithms.

Algorithm 1 Box-Muller method.

Let $\text{unif}(a,b)$ be a uniform random number generator $U(a,b)$.
Set $v_1 = \text{unif}(0,1)$, $v_2 = \text{unif}(0,1)$.
Set $R = \sqrt{-2\log(v_1)}$.
Set $w_1 = R\cos(2\pi v_2)$, $w_2 = R\sin(2\pi v_2)$.
Output (w_1, w_2).

Algorithm 2 Marsaglia's polar variant of the Box-Muller method.

Let $\text{unif}(a,b)$ be a uniform random number generator $U(a,b)$.
Set $v_1 = 1$, $v_2 = 1$, $q = v_1^2 + v_2^2$.
while $q > 1$ **do**
    Set $v_1 = \text{unif}(-1,1)$, $v_2 = \text{unif}(-1,1)$
    Set $q = v_1^2 + v_2^2$.
end while
Set $s = \sqrt{-2\log(q)/q}$
Set $w_1 = v_1 s$, $w_2 = v_2 s$.
Output (w_1, w_2).

Marsaglia's polar variant avoids an evaluation of sine and cosine, but because of the rejection method needed to produce a uniform random pair (v_1, v_2) in the unit disk, it requires $4/\pi \approx 1.27$ times the number of calls to the uniform random number generator. Empirically, the polar variant is faster, though conceivably the opposite could be true if the uniform generator is very slow. Other algorithms exist which may provide further speedup, but the polar variant is sufficient for our purposes.

Imposing reflective boundary conditions is more involved. Generally, if the diffusion $d\mathbf{W}(t_k)$ would push a path out of the domain, one needs to appropriately reflect it back in the domain to accurately capture the dynamics. Briefly, one needs to find the location of the intersection, calculate the local normal vector, reflect the component of the vector exiting the domain back in, and possibly repeat. This is described in algorithm 5.2.3. An illustration of this process is provided in figure 5.1.

Algorithm 3 Enforcing reflective boundary conditions for a particle trajectory.

$(y_1, z_1) \leftarrow (Y_i(t_k), Z_i(t_k))$
$(y_0, z_0) \leftarrow (Y_i(t_{k-1}), Z_i(t_{k-1}))$
while $u(y_1, z_1) < 0$ **do**

(*Find the first point of intersection.*)
Let $f(s) = y_0 + s(y_1 - y_0)$
Let $g(s) = z_0 + s(z_1 - z_0)$
$s_0 \leftarrow \min\{s : (f(s), g(s)) \in \partial\Omega,\ s \in [0, 1]\}$
$(y_0, z_0) \leftarrow (f(s_0), g(s_0))$

(*Reflect across the tangent line of the boundary.*)
$\underline{v} \leftarrow \mathbf{n}(y_0, z_0)$
$\underline{w} \leftarrow \langle y_1 - y_0, z_1 - z_0 \rangle$
$\underline{w} \leftarrow \underline{w} - 2\text{proj}_{\underline{v}}\underline{w}$

(*Update the final point.*)
$(y_1, z_1) \leftarrow (y_0, z0) + \underline{w}$
end while
$(Y_i(t_k), Z_i(t_k)) \leftarrow (y_1, z_1)$

In general, this process will result in a nontrivial spatial coupling between the random processes W_2 and W_3, but the analytical form of this coupling is generally unimportant.

Simplifications are possible in the rectangular and channel cases. In the channel case, if we have a particle exiting the domain in the positive direction $y_1 > 1$, reflect it by calculating $y_1 \leftarrow 2 - y_1$; if exiting the domain on the negative direction $y_1 < -1$, calculate $y_2 \leftarrow -2 - y_1$ (see figure 5.1 for a schematic).

In the rectangular case, the algorithm simplifies in a similar way because of the "separable" nature of the domain. However, extra care needs to be taken to corner cases, where multiple reflections may be necessary in a single timestep. For example, if $y_1 > 1$ and $z_1 > 1/\lambda$, then the calculation of the *first* boundary crossing, s_0, is not immediate. One needs to find the first of the two intersections as described in algorithm 5.2.3, that is, taking $\{s : f(s) = 1\}$ and $\{s : g(s) = 1/\lambda\}$ and applying the reflection at location with the smallest value of s.

In the elliptical case, more effort is required with every aspect of the calculation. The main feature here is that the interior and exterior of the domain can be partitioned based on the sign of

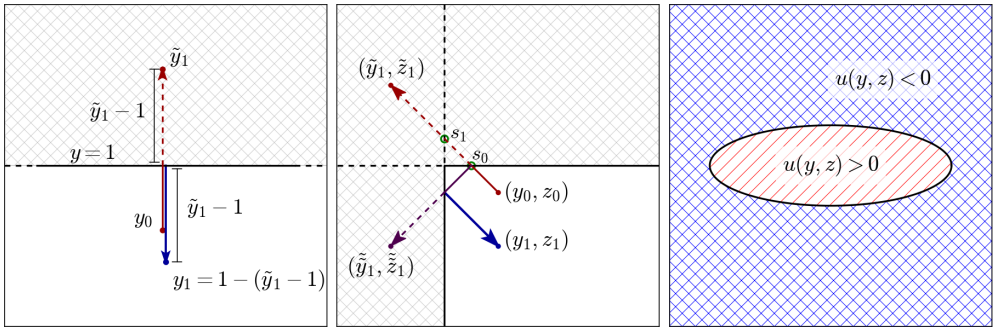

Figure 5.1: Illustration of a reflecting boundary condition in one dimension (left panel) and two dimensions in the case of the duct near the corner (right panel). The exterior of the domain is indicated with hatches. The channel reflections have a simple formula for preserving the total distance traveled. The rectangle requires dealing with corner cases, where one needs to find the minimum of intersection times $\{s_0, s_1\}$ (green circles) and multiple reflections. The right panel illustrates that the domain can be implicitly defined where a function $u(y, z) > 0$.

the lab-frame flow (with zero Dirichlet boundary conditions) (see right panel of figure 5.1):

$$\Omega = \{(y, z) : u(y, z) > 0\}, \quad \mathbb{R}^2 \setminus \bar{\Omega} = \{(y, z) : u(y, z) < 0\}, \quad \partial\Omega = \{(y, z) : u(y, z) = 0\}. \tag{5.15}$$

Therefore, calculating the point of intersection of an exiting path $(y_0, z_0) \to (y_1, z_1)$ in the ellipse involves solving

$$u(f(s), g(s)) = \frac{1}{1 + \lambda^2}(1 - f^2(s) - \lambda^2 g^2(s)) = 0 \tag{5.16}$$

for s, which when simplified is a quadratic which takes the form

$$ps^2 - 2qs - r = 0, \tag{5.17}$$

with coefficients

$$\begin{aligned} p &= (y_1 - y_0)^2 + \lambda^2 (z_1 - z_0)^2, \\ q &= y_0^2 + \lambda^2 z_0^2 - y_0 y_1 - \lambda^2 z_0 z_1, \\ r &= 1 - y_0^2 - \lambda^2 z_0^2, \end{aligned} \tag{5.18}$$

and has the solutions

$$s = \frac{q}{p} \pm \sqrt{\left(\frac{q}{p}\right)^2 + r}. \tag{5.19}$$

Because of the convexity of the domain, only one of the solutions for s will lie in $[0, 1]$, so there is no

room for ambiguity. Both solutions are checked and the one in the interval is taken.

A similar method is used when we deal with more general domains whose boundary can be implicitly described as a level set $u(y, z) = 0$ for a prescribed flow function u, or when dealing with a general polygonal domains, but other complications arise there. This is described in detail in chapter 6.

Calculation of statistics

The nature of Monte Carlo allows one to approximate the statistics of the distribution by binning the particles in the appropriate way.

- To approximate $\int_\Omega T(\mathbf{x}, \tau) dA$ for a fixed τ, first choose the number of bins n_b, then:

  1. Find the minimum and maximum values $x_{\min}$ and $x_{\max}$ of the ensemble of particles $\{X_i\}$;
  2. Create a uniform grid of $n+1$ points from $x_{\min}$ to $x_{\max}$ with spacing $h = (x_{\max} - x_{\min})/n$;
  3. Count the number of particles $\{X_i\}$ in each bin $[x_{i-1}, x_i]$, denote it c_i;
  4. Normalize the values c_i so that $h \sum_{i=1}^{N} c_i = 1$, or more generally, use a numerical integration procedure to normalize to integral to one.
  5. For analysis, save the array $\{c_i,\, i = 1, ..., n\}$, and the corresponding array of bin centers $\{b_i,\, i = 1, ..., n\}$, with $b_i = (x_{i-1} + x_i)/2$.

  If a binning procedure is available in the environment (for example, `hist` in Matlab/Octave, or `numpy.histogram` or `matplotlib.hist` in Python) then one should use that instead of the manual version described above. Otherwise, binning with uniform bins boils down to a linear mapping of the position X_i to the appropriate bin j.

- To approximate the centered statistics of the cross-sectionally averaged distribution,

$$\mu_1 = \int x \left[\int_\Omega T(\mathbf{x}, \tau) dA \right] dx, \qquad \mu_k = \int (x - \mu_1)^k \left[\int_\Omega T(\mathbf{x}, \tau) dA \right] dx, \tag{5.20}$$

  one in fact only needs to calculate the discrete statistics without regards to Y_i or Z_i:

$$\mu_1 \approx \frac{1}{N} \sum_{i=1}^{N} X_i, \qquad \mu_k \approx \frac{1}{N} \sum_{i=1}^{N} (X_i - m_1)^k. \tag{5.21}$$

Technically speaking these are biased estimators for the statistics. Unbiased estimators for the statistics (e.g., dividing by $N-1$ instead of N for the variance) can be used if desired, but since N is typically larger than 10^6, it is relatively unimportant.

- Approximating pointwise statistics

$$\mu_1(y,z,t) = \frac{\int xT(\mathbf{x},t)dx}{\int T(\mathbf{x},t)dx}, \qquad \mu_k(y,z,t) = \frac{\int xT(\mathbf{x},t)dx}{\int T(\mathbf{x},t)dx}, \tag{5.22}$$

is more involved. Our approach is to:

1. Create a two dimensional uniform grid containing the cross sectional domain with $n_{by}+1$ and $n_{bz}+1$ points in the y and z directions respectively;
2. Define a linear mapping $I : \mathbb{R}^2 \to \{1, ..., n_{by}n_{bz}\}$ (e.g., row-major or column-major counting) to assign each point a bin number j based on its cross-sectional coordinate (Y_i, Z_i);
3. For each bin j, collect the all particles in that bin, $S_j \equiv \{i : I(Y_i, Z_i) = j\}$, and calculate the discrete x-statistics of that set:

$$\mu_{1j} \approx \frac{1}{|S_j|}\sum_{i\in S_j} X_i, \qquad \mu_{kj} \approx \frac{1}{|S_j|}\sum_{i\in S_j}(X_i - m_{1j})^k; \tag{5.23}$$

4. For analysis, save the locations of the bins as needed to map μ_{kj} to the corresponding grid location.

We are not aware of any procedures which would allow you to do this in high-level environments. The collection of bins is done in two steps. First, the list of indexed coordinates is sorted according to their index (their bin number) [3] , and a count is done of the number of particles in each bin. Then, using the bin counts, the contiguous subsets of the list can be immediately passed into the moments subroutine, and the statistics are assigned to the appropriate bin.

[3]The sorting can be done efficiently, since the elements to sort over (the bin index) are integer valued.

Validation and convergence of numerics

In the case of the infinite channel, there are exact formulae [15] for the first three longitudinal moments of both the pointwise and the cross-sectionally averaged distribution which we can use to benchmark our code. In the circular pipe, there are exact formulae for the first two pointwise moments, and for the first three cross-sectionally averaged moments.

Validation in the infinite channel

We first benchmark the code on the channel case.

In figure 5.3.1, we plot numerics (colors) to exact results (black) for a few sample cases. In the left panel, we work with $N = 10^6$ particles and compare the skewness between the numerics and exact formula for a few values of the Péclet number. In the right panels, we visualize the pointwise mean, variance, and skewness, the y coordinate in the ordinate direction, for $\mathrm{Pe} = 10^4$ and $t = 10^{-2}$. The pointwise statistics are approximated by using 100 bins in the cross section interval $[-1, 1]$. We see good "eyeball norm" agreement in these cases.

For a more rigorous error analysis, we perform a study on the absolute error $|Sk(t) - Sk_{\text{exact}}(t)|$ in figure 5.3.1. We sweep over the parameter space in number of particles $N = 10^3, 10^4, ..., 10^8$, and Péclet number $\mathrm{Pe} = 10^2, 10^4, 10^6$, with an initially growing, but ultimately hard bound on the timestep, $\Delta t \leq 10^{-3}$. The left panel includes the results for all (N, Pe), with the error independent of Pe and generally decreasing in N. The right panel measures $\max_{t_i}\{|Sk(t_i) - Sk_{\text{exact}}(t_i)|\}$ for each pair (N, Pe), and plots this error as a function of N. A decay rate of $N^{-0.5}$ is seen, which is expected with Monte Carlo methods.

The caveat to the expected decay in the above study is the anomalous growth in the error for $10^{-4} \leq t \leq 10^{-1}$ which is seen independent of increasing N. This is also seen in the right panel, with a stagnation in the error for $N = 10^8$. This can be explained as a result of the maximum timestep Δt being slightly too large to resolve the intermediate time dynamics. In figure 5.3.1 we study the same time-varying error as before, but now fixing $N = 10^7$ and $\mathrm{Pe} = 10^4$, and varying the maximum timestep $\Delta t_{\max} = 10^{-1}, 10^{-2}, ..., 10^{-5}$.

Validation in the circular pipe

Next we move on to benchmarking against the circular pipe case. In this case there are formulae for only the first two longitudinal moments of the fully distribution, and the first three for the cross-sectionally averaged distribution. The same parameter sweep is performed. However, there

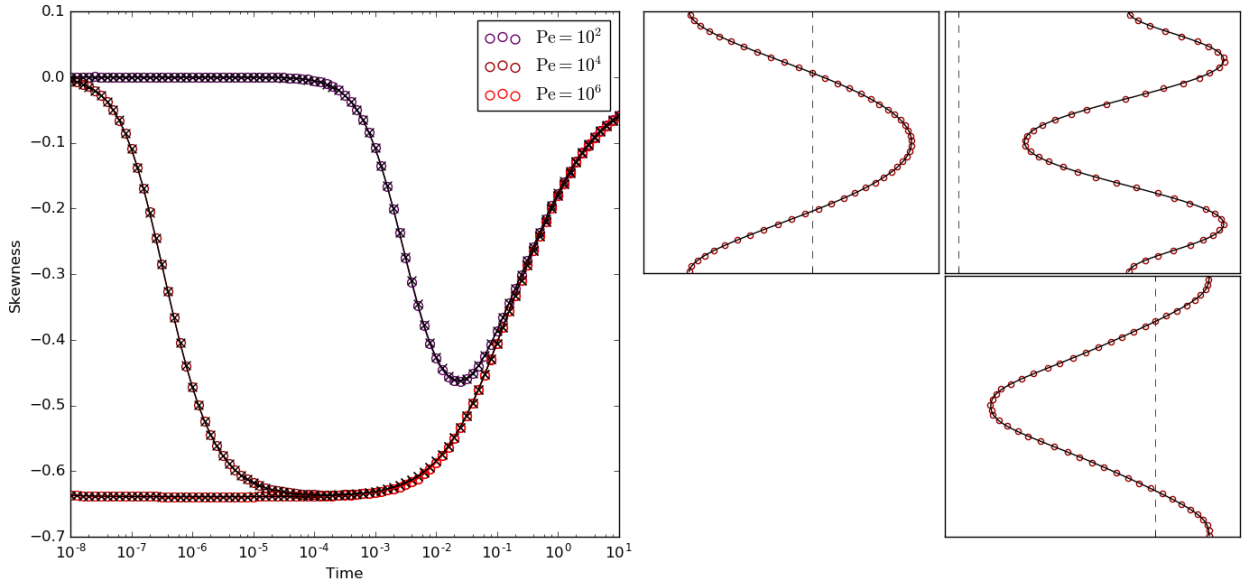

Figure 5.2: Left panel: Time evolution of the numerical (shades of red) and exact skewness (black) of the averaged distribution in the channel for Péclet values Pe $= 10^2, 10^4, 10^6$ and number of particles $N = 10^6$. Right panels, clockwise from top left: snapshots of the numerical and exact pointwise mean, variance, and skewness for Pe $= 10^4$, at $t = 10^{-2}$. Dashed black lines indicate the zero line of the mean, variance, and skewness.

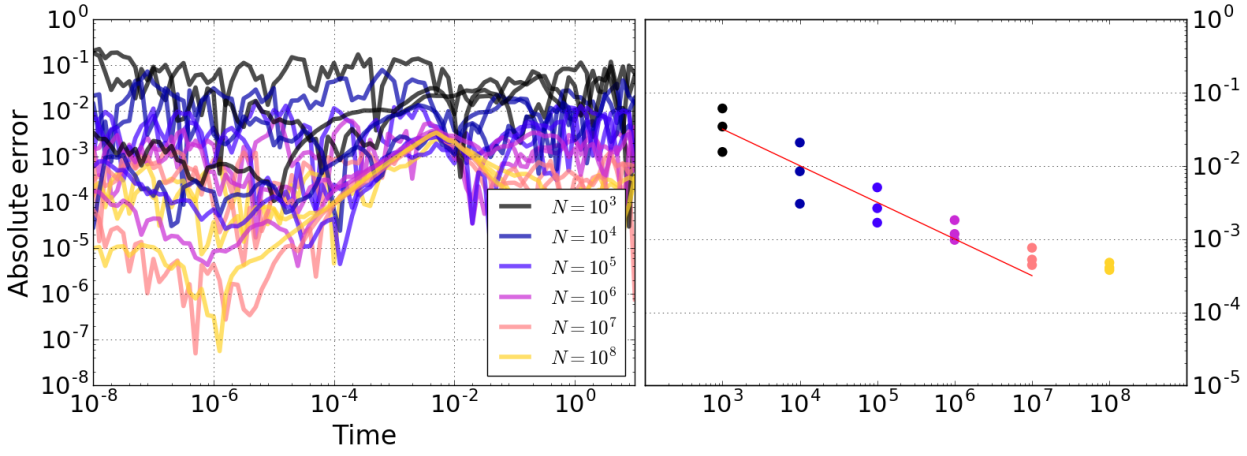

Figure 5.3: Analyzing convergence of numerics with increasing N. Left panel: absolute error in the averaged skewness $|Sk(t) - Sk_{\text{exact}}(t)|$ in the channel for Péclet values Pe $= 10^2, 10^4, 10^6$ and numbers of particles $N = 10^3, ..., 10^8$. Right panel: the corresponding average error of $|Sk(t_i) - Sk_{\text{exact}}(t_i)|$ is plotted versus the number of particles, with a power law fit. The expected scaling of error as $1/\sqrt{N}$ is generally observed.

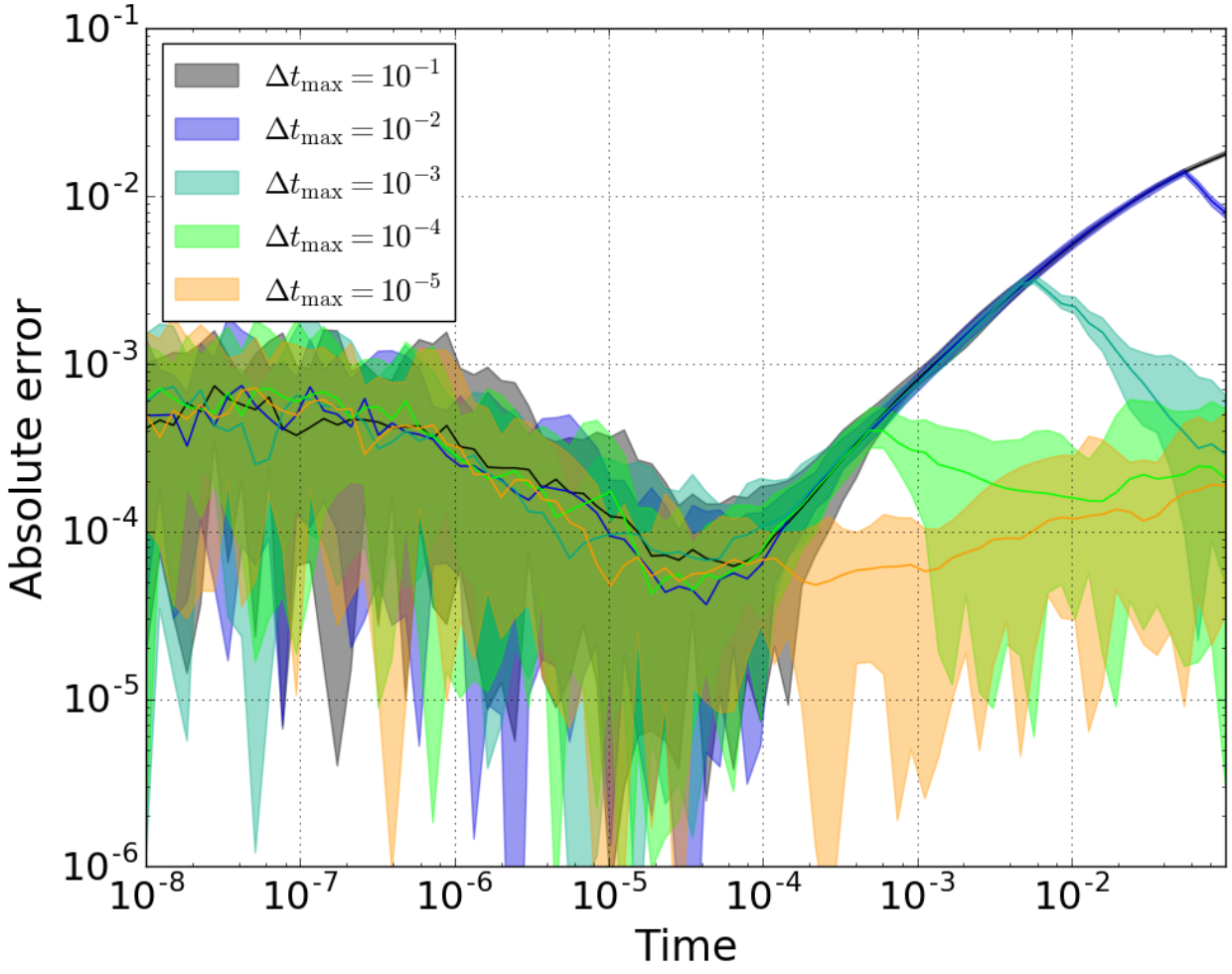

Figure 5.4: Analyzing convergence of numerics with $N = 10^7$ and $\mathrm{Pe} = 10^4$ fixed, with decreasing $\Delta t_{\max}$. Ten simulations are run for each $\Delta t_{\max}$, and the range of errors is shown. The error behaves nonrandomly for $\Delta t_{\max} \geq 10^{-4}$, growing at an approximately linear rate in time until the hard cap is hit. The error then decays as both the numerics and exact solution begin decaying to zero around $t \approx 10^{-2}$.

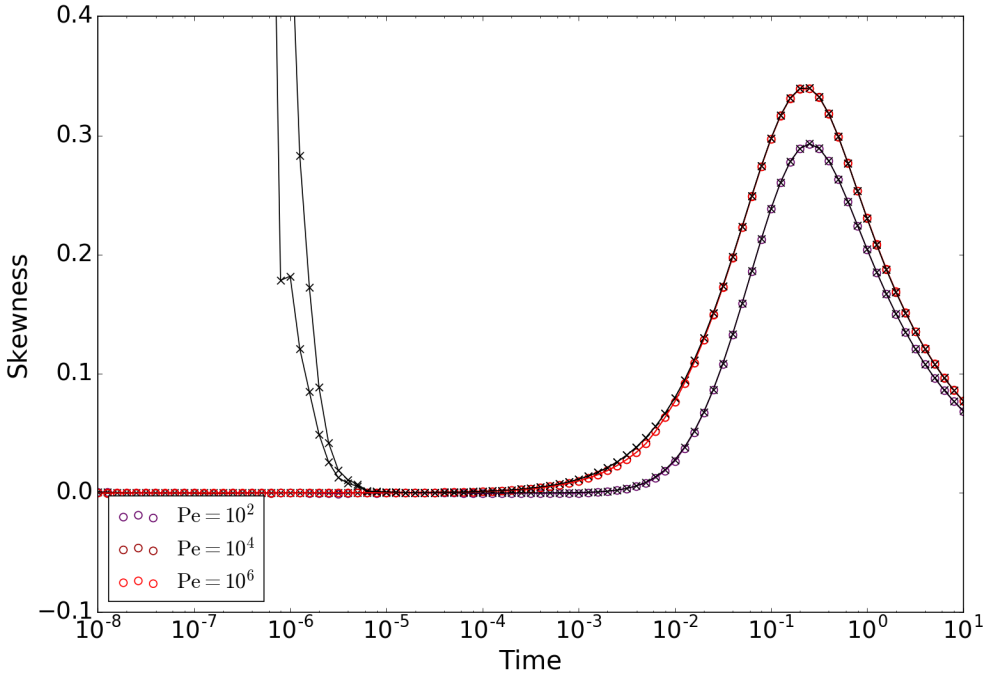

Figure 5.5: Time evolution of the numerical (shades of red) and exact (black) skewness in the circular pipe for Péclet values Pe $= 10^2, 10^4, 10^6$ and number of particles $N = 10^6$. General agreement is seen across a range of Péclet values, except at short time, where the exact formulae have numerical cancellation issues, and a deviation near $\tau \approx 10^{-3}$.

are 100 bins in each direction, and the grid contains "inactive" bins that lie outside the unit disk, so there are approximately $100^2\pi/4$ total bins in this case. Hence, we should see similar errors compared to the channel with a factor of $\sqrt{100^2\pi/4} \approx 88$ more particles.

Figure 5.3.2 compares the exact formulae from [5] with the numerical results across a range of Péclet values and time scales, with $N = 10^6$ particles. The numerical canellation issues are seen at the shortest times with the exact formulae. The numerics also deviate slightly at the onset of skewness for large Péclet near $\tau \approx 10^{-3}$, which is seen in figure 5.3.2 to be analagous to the channel's error behavior with $\Delta t_{\max} = 10^{-3}$, and should be cured if this value is taken one or two orders of magnitude smaller.

The mean errors are again plotted, after excluding the region $\tau < 10^{-4}$ with divergence issues in the exact formulae. Similar issues are seen as with the channel with the choice of $\Delta t_{\max} = 10^{-3}$, which degrades the rate of convergence for large N. Nevertheless, for small to moderate N, the expected convergence rate $N^{-1/2}$ (illustrated with the red line) is observed.

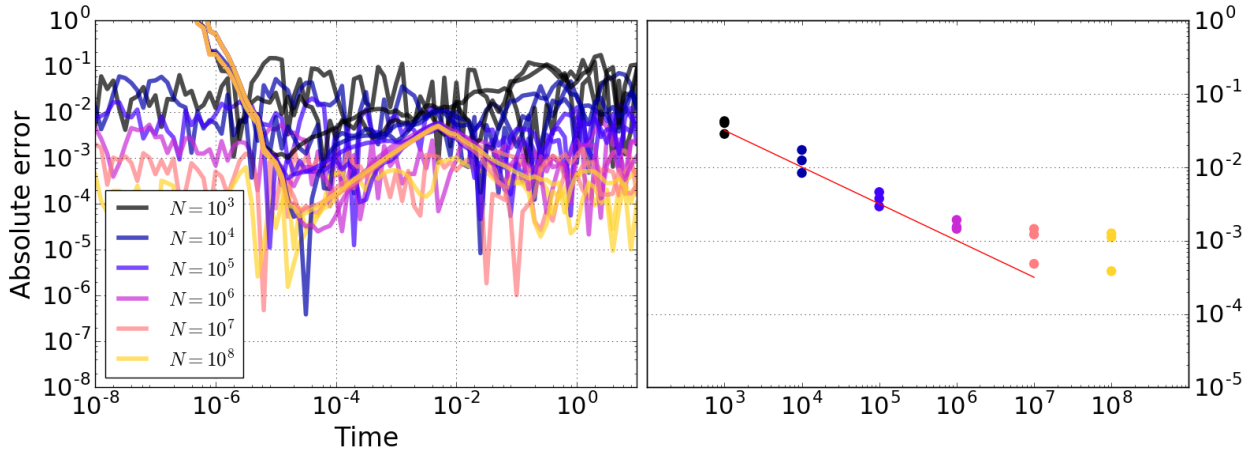

Figure 5.6: Analyzing convergence of numerics in the circular pipe with increasing N. Left panel: absolute error in the averaged skewness $|Sk(t) - Sk_{\text{exact}}(t)|$ in the pipe for Péclet values $\text{Pe} = 10^2, 10^4, 10^6$ and numbers of particles $N = 10^3, ..., 10^8$. Right panel: the corresponding average error of $|Sk(t_i) - Sk_{\text{exact}}(t_i)|$ is plotted versus the number of particles, with a power law fit. The expected scaling of error as $1/\sqrt{N}$ is generally observed.

Validation through long time asymptotics

Some validation can be done in domains without formulas for the full evolution of the tracer moments.

In chapter 3, the generic long time (past the diffusive timescale) asymptotics for the first three moments in any domain. For the variance, this is the relaxation to diffusivity enhanced by a term proportional to a domain-dependent constant times the square of the Péclet number. The coefficient can be calculated in closed form in the channel and circular pipe. There is also an exact formula for any ellipse, and the rectangle can be expressed as a double sum. Some care must be taken for the choice of parameters, or the limiting behavior for small aspect ratio can give several different behaviors.

For the skewness, we have shown there is typically a $t^{-1/2}$ decay, whose coefficient depends on the solution of a sequence of elliptic PDEs relating to the flow $u(y, z)$. In the infinite channel, circular pipe, and ellipse, these coefficients again have closed form solution. For the rectangular domain we have opted to use a built-in finite element package in Mathematica to numerically solve the PDEs necessary to calculate the coefficient, and obtained convergence up to the point necessary to benchmark against the Monte Carlo code.

In figure 5.7 we try to validate the numerics by matching the expected long time behavior. The

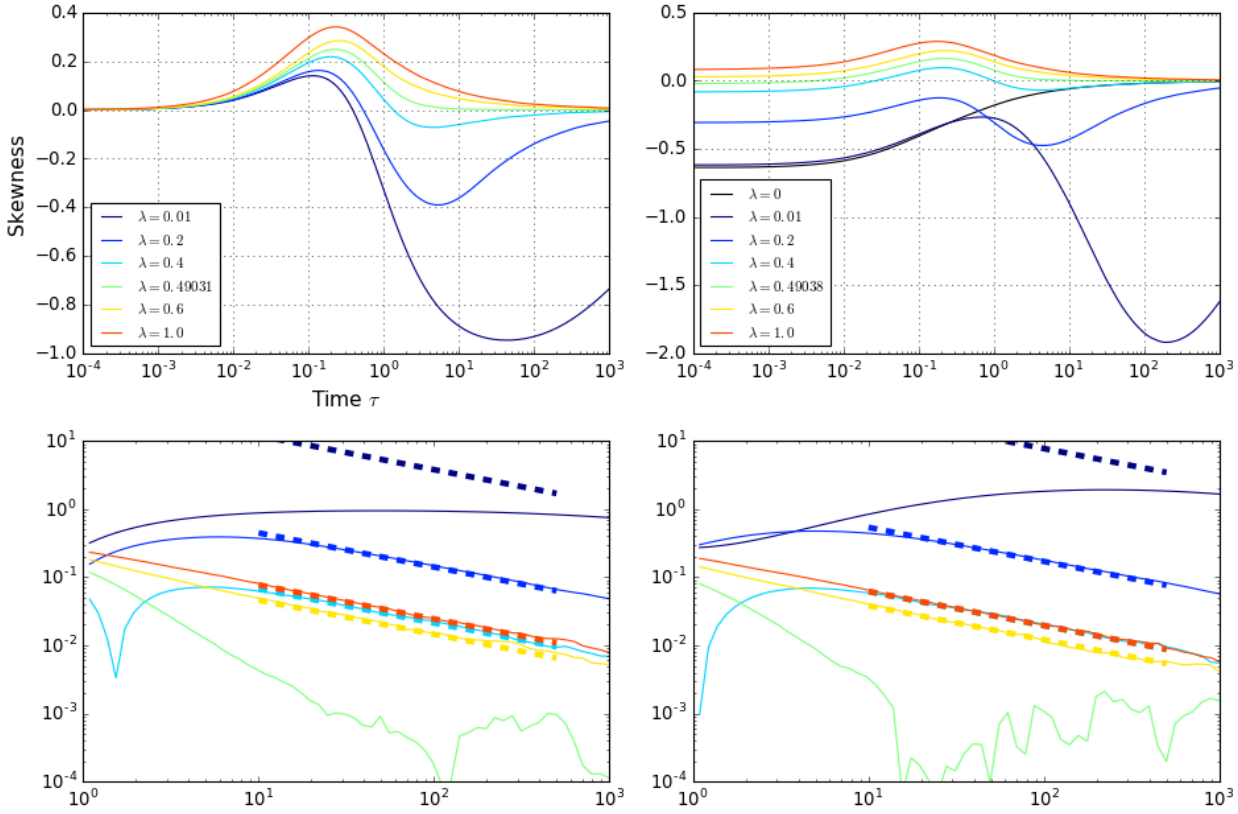

Figure 5.7: Behavior of the skewness Sk in the numerics for various geometries. Top row: skewness evolution for ellipses (left) and rectanges (right) of varying aspect ratio with Pe $= 10^4$. Bottom row: log-log plot of $|Sk|$ versus time for the ellipses (left) and rectangles (right).

ellipses and rectangles are simulated with Pe $= 10^4$ and varying aspect ratios λ. The bottom row shows the long time behavior of the absolute-value skewness in a log-log plot to verify the expected asymptotic is followed. In the bottom left panel, the predicted asymptotic decay rates (dashed) generally agree strongly with the numerics. The first exception is the cases $\lambda = 0.01$, whose final diffusive timescale is at $t = 10^4$, which is when the asymptotics are valid. The other exception is $\lambda = \lambda^* \approx 0.49031$, at which the coefficient $\langle ug_2 \rangle$ vanishes, and the leading order asymptotic behavior is instead $t^{-3/2}$, for which we don't have a prediction. The bottom right shows the similar behavior in the rectangles, though the coefficient $\langle ug_2 \rangle$ is computed in Mathematica as described above.

Results in the rectangular and elliptical domains

The general results for the skewness of the cross-sectionally averaged tracer in the rectangular domains are seen in the right column of figure 5.7. The averages of 100 runs of 10^6 particles are shown, with Pe $= 10^4$. The simulations demonstrate agreement with the short time theory of section 3.2 at $\tau \approx 10^{-4}$, and the long time skewness predictions of section 3.3 can be seen in the bottom right. There is a wealth of nontrivial behavior on intermediate timescales for which we have no theory. For example, there is a positive influence on the skewness at roughly $\tau \approx 10^{-1}$ independent of the aspect ratio. Intermediate aspect ratios exhibit sign changes in the skewness, as a result. There is a narrow band of aspect ratios which have negative short time skewness and positive long time skewness. Lastly, despite the good matching between the channel ($\lambda = 0$) and most "channel-like" ($\lambda = 0.01$) on short times, the skewness separates strongly past the diffusive time $\tau = 1$. There is an expectation that the statistics for $\lambda \ll 1$ approach the channel statistics, but examining the simulations, the nature of this convergence is not be trivial.

In figure 5.8 we display the numerical results for the pointwise skewness $Sk(y, z, \tau)$ for a range of times and aspect ratios, spanning from $\tau \approx 10^{-3}$ to $\tau = 2.5$, and $\lambda = 0.2$, $\lambda = \lambda^* \approx 0.49$, and $\lambda = 1$. The same color scale is used amongst all panels for consistency, with red positive, blue negative, and white zero skewness.

There is interesting fine spatial structure in the skewness at short times which we have no theory for. The intuition in the infinite channel case is with the idea of diffusive pumping. The tracer initially bends according to the flow profile, and due to the finite extent of the domain, the tracer distribution near the walls will naturally have a strong peak behind the mean flow line, with a long forward tail due to lateral diffusion from the center. Similarly, in the center, the tracer will have a

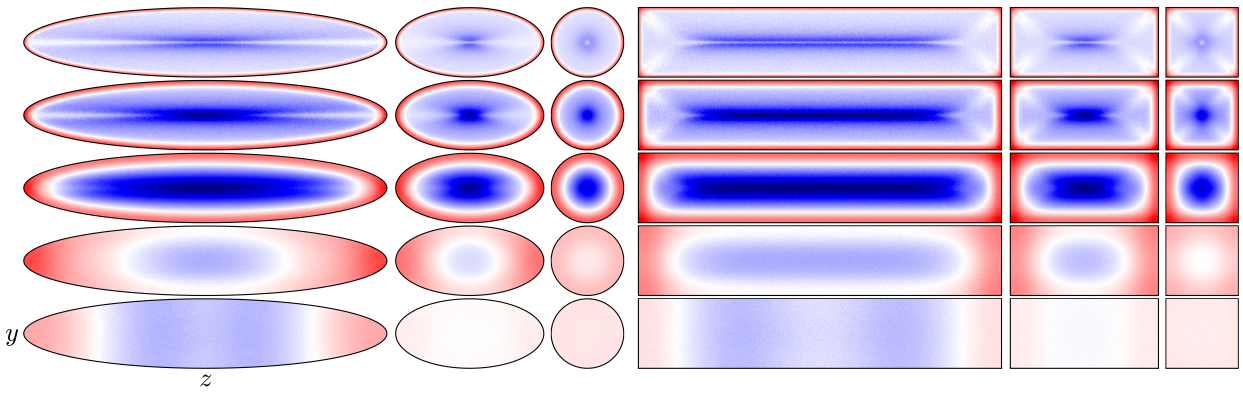

Figure 5.8: Snapshots of the pointwise skewness in the ellipses and rectangles. Aspect ratios $\lambda = 0.2$, $\lambda = \lambda^*$, and $\lambda = 1$ are used, for times $\tau = 0.0014$, 0.008, 0.046, 0.44, and 2.5. Nontrivial

strong peak ahead of the mean flow line, with a long backward tail due to lateral diffusion from the walls. Somewhere in the interior, these effects balance out to result in points of zero skewness. This is observed in the exact solution and numerics for the channel (see figure 5.3.1, bottom right panel).

In the rectangles and ellipses, this same behavior is generally observed, but because of the added dimension, the zero-skewness contours assume take a nontrivial shape. What governs their particular shape, specifically for $\lambda \neq 1$, is unclear. Even for the infinite channel case, we have no theory in regards to the behavior of the implicit function $Sk(y,t) = 0$ in (t,y) space.

At the longest time, the predicted spatial independence $Sk(y,z,t) \sim Sk(t)$ is observed for $\lambda = 1$, and beginning to be observed for the other aspect ratios. The deviation from constant is explained by the fact that the final diffusive timescale $\tau = 1/\lambda^2$ must be reached for the long time theory to be valid; this is at $\tau \approx 4$ for $\lambda = \lambda^*$ and $\tau = 25$ for $\lambda = 0.2$.

CHAPTER 6

Numerics and asymptotics in other domains

Introduction

In this section we discuss a few other classes of domains which have been studied. First, we extend the elliptical domains by introducing a second parameter independent of the aspect ratio. This allows us, to a moderate degree, interpolate cross sections from an ellipse of a given aspect ratio to the analagous rectangle. Using Mathematica, we have numerically calculate the geometric skewness and demonstrated it can be continuously changed from the circle value (zero) to square-like domains, where it is positive. We have also extended the Monte Carlo code for the ellipses to handle this case, though the calculations are somewhat trickier.

Next, we examine the equilateral triangle cross section, where there is a polynomial formula for the flow, which results in a positive value for the short time skewness, and a negative value for the long time skewness. In this case, the Monte Carlo code has been extended to handle a general convex, polygonal boundary.

Finally, we address the cases of the regular n-gons, for $n = 3, 4, ...$; the equilateral triangle, the square, and so on. Except for $n = 3$, a closed form formula for the flow is not known, so we only examine the asymptotics numerically.

Racetrack cross sections

The "racetracks," as we have termed them, are an extension of the elliptical cross sections. The method is to implicitly perturb the boundary by modifying the flow directly. Since there is a formula for the flow, the Monte Carlo code has be extended.

Derivation of the flow

The flow solution $u(y, z)$ is the solution of the Poisson problem $\nabla^2 u = const.$ with specified domain boundary conditions.

In general, the usual approach to the Poisson problem is to begin with a specified domain and boundary conditions, then seek the solution. The idea for this section is to instead *begin* with

a solution to the Poisson equation, then modify it with some choice of harmonic function. The boundary will then be implicitly defined as the zero level set of this new function, which automatically satisfies Dirichlet boundary conditions, and the domain will be the interior of this boundary.

Our approach is to use an ellipse flow solution and modify it with a harmonic polynomial. Putting aside overall multiplicative constants, the new (non-dimensional) flow solution can be written as

$$u(y,z) = 1 - c_1\left(y^2 + c_2^2 z^2\right) - c_3 P(y,z) \tag{6.1}$$

with harmonic polynomial $P(y,z)$ and coefficients c_1, c_2, and c_3. Due to the Cauchy-Riemann equations, the only polynomials of two variables which are harmonic are combinations of the real and imaginary parts of complex polynomials $f(w) = w^n = (y+iz)^n$. The simplest polynomial of this class with four-fold symmetry is

$$P(y,z) = \mathrm{Re}\left[(y+iz)^4\right] = y^4 - 6y^2z^2 + z^4. \tag{6.2}$$

Using this $P(y,z)$, we are left with specifying the undetermined constants. We impose the conditions $u(1,0) = u(0,1/\lambda) = 0$ to maintain the aspect ratio. [1]

$$\left.\begin{array}{rl} u(1,0) & = 1 - c_1 - c_3 = 0 \\ u(0,1/\lambda) & = 1 - \frac{c_1 c_2^2}{\lambda^2} - \frac{c_3}{\lambda^4} = 0 \end{array}\right\} \Rightarrow \left\{\begin{array}{ll} c_1 & = \frac{1-\lambda^4}{1-\lambda^2 c_2^2} \\ c_3 & = 1 - c_1 = \frac{\lambda^2(\lambda^2 - c_2^2)}{1-\lambda^2 c_2^2} \end{array}\right. \tag{6.3}$$

The count is two equations and three coefficients, leaving us $c_2 \equiv s$ to use as a shape parameter. Collecting everything, the flow is

$$u(y,z;s) = 1 - \left(\frac{1-\lambda^4}{1-\lambda^2 s^2}\right)\left(y^2 + s^2 z^2\right) - \left(\frac{\lambda^2(\lambda^2 - s^2)}{1-\lambda^2 s^2}\right)(y^4 - 6y^2z^2 + z^4), \tag{6.4}$$

[1] An alternate choice is to set $u(0,1/\lambda) = u(1,1/\lambda) = 0$, so that the far boundary and corner are pinned down and the long walls can vary more freely, but we do not investigate this here.

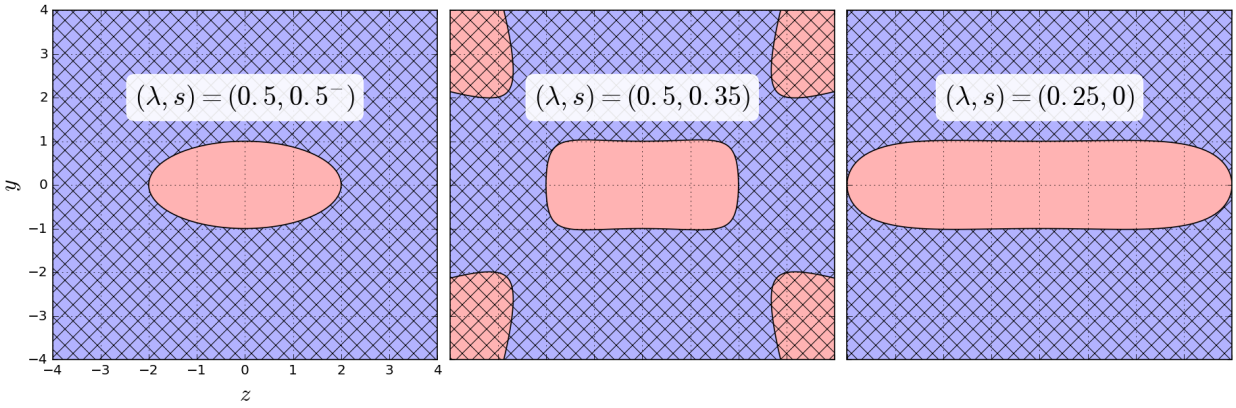

Figure 6.1: Illustration of the racetrack for the choices of parameter pairs shown. Note that for $s < \lambda$ it is common for the domain to be come non-convex, as can be seen with $(\lambda, s) = (0.5, 0.35)$. The hatched regions indicate the exterior of the domain, and red and blue indicate regions where $u > 0$ and $u < 0$ respectively.

and by construction, this choice of u satisfies the Poisson problem

$$\begin{aligned} &\Delta u = const., \qquad u|_{\partial\Omega(s)} = 0, \\ &\Omega(s) \equiv \{(y, z) \,:\, u(y, z; s) \geq 0\}. \end{aligned} \tag{6.5}$$

The domains for a few choices of parameter pairs (λ, s) are shown in figure (6.2.1). Due to the nature of the harmonic perturbation, there are regions disconnected from the main region with $u > 0$. For this reason it is not quite enough to define the domain as the set where $u(y, z) > 0$, but rather as the path-connected component where $u(y, z) > 0$. For particular values of s (depending on λ), the four "probes" on the exterior come towards the corners of the domain, which can make the numerics difficult, as the timestep needs to be sufficiently small to avoid "jumping" into the nonphysical domain. There are a few observations to be made:

1. Problem (6.5) reduces to the ellipse when $s = \lambda$.

2. It is not clear that the implicitly defined domain $\Omega(s)$ is bounded for all pairs (λ, s). If the harmonic component (whose level sets are unbounded) is large relative to the elliptical component, we expect the level sets of their sum to be unbounded. For example, this happens in the regime $\lambda \approx 1$ and $s \ll \lambda$.

Algorithm 4 To enforce reflective boundary conditions for a boundary implicitly by the flow.

for $i = 1, 2, ..., n_{particles}$ **do**
  $(y_1, z_1) \leftarrow (Y_i(t_k), Z_i(t_k))$
  $(y_0, z_0) \leftarrow (Y_i(t_{k-1}), Z_i(t_{k-1}))$
  while $u(y_1, z_1) < 0$ **do**
    (*Find the first point of intersection with a rootfinder.*)
    Let $f(s) = y_0 + s(y_1 - y_0)$
    Let $g(s) = z_0 + s(z_1 - z_0)$
    $s_0 \leftarrow \min\{s : u(f(s), g(s)) = 0,\ s \in [0, 1]\}$
    $(y_0, z_0) \leftarrow (f(s_0), g(s_0))$

    (*Reflect across the tangent line of the boundary.*)
    $\underline{v} \leftarrow \nabla u(y_0, z_0)$
    $\underline{w} \leftarrow \langle y_1 - y_0, z_1 - z_0 \rangle$
    $\underline{w} \leftarrow \underline{w} - 2\text{proj}_{\underline{v}}\underline{w}$

    (*Save the new position*)
    $(y_1, z_1) \leftarrow (y_0, z0) + \underline{w}$
  end while
end for

Monte Carlo simulation for the racetrack

As before, the Monte Carlo simulation relies on taking sample paths of the stochastic differential equation

$$\begin{aligned} dX(t) &= \text{Pe}\, u(Y(t), Z(t))dt + dW_1(t), \\ dY(t) &= dW_2(t), \\ dZ(t) &= dW_3(t). \end{aligned} \tag{6.6}$$

The flow is a polynomial by construction, so it can be evaluated directly. The boundary is defined implicitly by $u(y, z) = 0$, and, except for a measure-zero set of parameters (λ,s), there exists a region for which $u < 0$ in the exterior surrounding the domain. This gives us a way to detect boundary crossings, and allows us to use a similar algorithm (detailed below) to perform reflections off the boundary.

With the polynomial form of the flow, the exact gradient ∇u is calculated beforehand and evaluated in its own subroutine. Finding the solution(s) of $u(f(s), g(s)) = 0$ require a numerical rootfinder for general $u(y, z)$.

While the flow formula is well behaved for any (λ, s) (except for $\lambda = s$ as a removable discontinuity), the domain shape varies noticeably depending on the pair chosen. First, one must restrict to $s \leq \lambda$,

otherwise the domain will not be as expected, as the “pinning" of the domain to $(y, z) = (1, 0)$ and $(0, 1/\lambda)$ will occur on branches of the exterior hyperbolae instead of the main racetrack. Secondly, for sufficiently small s and moderate λ, the domain can become unbounded. This corresponds to the “probes" illustrated in the center panel of figure 6.2.1 touching the racetrack.

We do not have a predictive criteria for the curve $s = f(\lambda)$ for which this will occur. One may make an attempt by parameterizing the boundary in polar coordinates. This does have a simple expression, but the angle θ where the corner occurs depends on both λ and s. We have resorted to a trial and error method; sampling the lower triangle $s \leq \lambda$ and throwing out simulations with anomalously large values of skewness which result from unbounded domains.

A summarized result of a parameter sweep in (λ, s) space for the racetracks is shown in figure 6.2.2. A uniform colormap is used, where blue and red correspond to negative and positive skewness, respectively, and white zero skewness, with an approximate zero level curve drawn in black. Only the subset of parameter values with bounded domain and $s \leq \lambda$ are shown. At $t = 0.015$, the after effects of geometric skewness are seen; the ellipses (on the line $\lambda = s$) are all positive, while the rectangular-like shapes (bottom curved boundary) are separated between positive and negative values between $\lambda = 0.4$ and $\lambda = 0.5$. Approaching the diffusive timescales, the behavior becomes nearly independent of the shape parameter, providing evidence that the “golden" long time aspect ratio of $\lambda^* \approx 0.49$ is observed across this class of geometries.

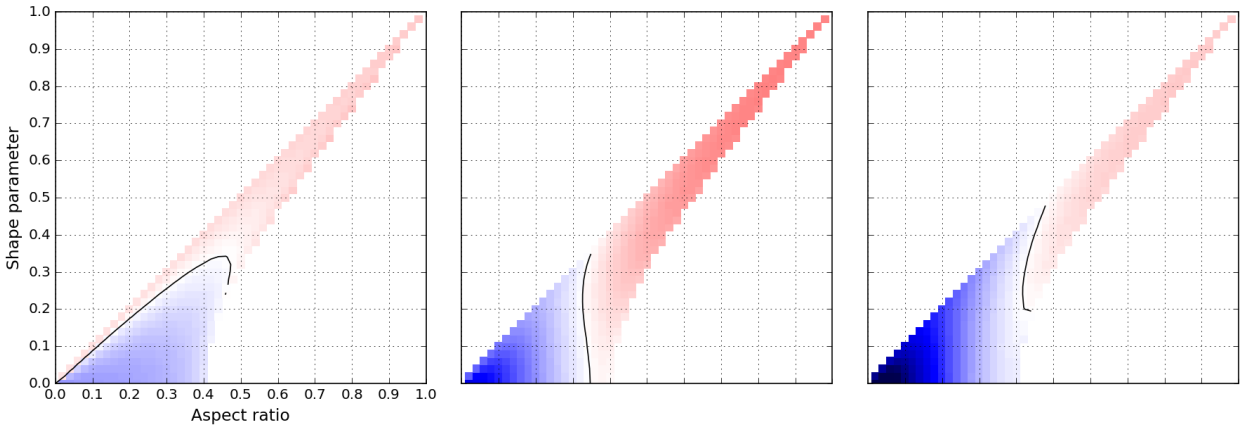

Figure 6.2: Averaged skewness over a range of parameter pairs (λ, s) which satisfy a convexity criterion at $(y, z) = (1, 0)$, plotted at nondimensional times (from left to right) $t \approx 0.015$, 0.97, and 6.28. Positive (negative) skewness is red (blue), with white being zero. An approximate zero skewness contour is overlaid in black. Long time behavior is seen to be nearly independent of the shape parameter.

Triangular cross section

The flow in the equilateral triangle is one of only a few cross setions which has a closed form expression. Additionally, since the boundary is the intersection of three half-planes (i.e., a convex polygon) it is possible to modify the Monte Carlo code to efficiently apply reflecting boundary conditions.

As with the channel, the exact short and long time asymptotics can be calculated, and agree with the simulations, as will be shown below.

Calculation of the flow

In the case of a cross section which is an equilateral triangle, there is an exact formula for the flow:

$$u(y, z) = \frac{1}{12a}(a + y)(2a + \sqrt{3}z - y)(2a - \sqrt{3}z - y), \tag{6.7}$$

where a is now the distance from the centroid (fixed at $(0, 0)$) to the nearest boundaries (or alternatively, the radius of the inscribed circle), and *not* the length of one of the sides (which is $2\sqrt{3}$). It is straightforward to check that this is equivalent to the definition of a in the infinite channel, rectangles, and ellipses.

Each term in the product is in fact one of the three boundaries:

$$y = -a, \quad y = 2a + \sqrt{3}z, \quad y = 2a - \sqrt{3}z \tag{6.8}$$

which verifies the Dirichlet boundary conditions are satisfied. and the Laplacian can be calculated directly to be -2.

Calculation of asymptotics

Since the flow is a polynomial, the short and long time asymptotics can be calculated exactly. The end result for geometric skewness is

$$\frac{\langle u^3 \rangle}{\langle u^2 \rangle^{3/2}} = \frac{1/19250}{(3/700)^{3/2}} = \sqrt{\frac{112}{3267}} \approx 0.185 \tag{6.9}$$

which predicts positive short time skewness. The coefficient of the long time, large Péclet skewness needs to be calculated numerically with a finite element package; the result is weak and negative:

$$\frac{3\langle u g_2 \rangle}{(2\langle u g_1 \rangle)^{3/2}} \approx -0.07076. \tag{6.10}$$

The comparison of the short and long time asymptotics is shown and discussed in section 6.3.3.

Modification of Monte Carlo code

The Monte Carlo code is roughly a modification between the infinite channel and rectangular cases. It is like the channel in that the flow can be evaluated directly, while it is like the rectangle in that there are two cross-sectional directions.

However, it is not quite as trivial to apply boundary conditions as in the channel or rectangle (where only subtractions are needed). Our approach is to describe the triangle as an intersection of the three half-planes $\ell_i(y, z) > 0$, where each ℓ_i is essentially one of the boundaries written above:

$$\ell_1(y, z) = a + y, \tag{6.11a}$$

$$\ell_2(y, z) = 2a - y + \sqrt{3}z, \tag{6.11b}$$

$$\ell_3(y, z) = 2a - y - \sqrt{3}z. \tag{6.11c}$$

or for a generic linear boundary,

$$\ell(y, z) = c_0 + c_1 y + c_2 z, \tag{6.12}$$

with specified coefficients c_0, c_1, c_2. Typically one needs to construct $\ell(y, z)$ from a piece of the boundary specified in another way; for instance, $y = -a$. There is a freedom of the overall sign, i.e., $\ell_1 = \pm(a + y)$. The sign is fixed by the convention that a $\ell(y, z)$ should be positive for (y, z) in the domain. We fix $(0, 0)$ in the domain, so requiring $\ell_1(0, 0) > 0$ means we take the positive sign.

The problem of a reflection about one half plane is effectively the same as the generalization used in the racetrack, but because the boundaries are linear, the implementation is much simpler. For instance, if we had only the half-space advection diffusion problem with boundary $\ell = 0$, after seeing a particle go from $(y_0, z_0) \to (y_1, z_1)$ with $\ell(y_1, z_1) < 0$ (hence, exiting the domain), the main steps are to

1. Calculate the time of intersection by solving

$$\ell\big(y_0 + s(y_1 - y_0), z_0 + s(z_1 - z_0)\big) = 0. \tag{6.13}$$

   In contrast to the racetrack, this now has an exact solution for arbitrary coefficients:

$$s = \frac{c_0 + c_1 y_0 + c_2 z_0}{c_1(y_0 - y_1) + c_2(z_0 - z_1)} = \frac{\ell(y_0, z_0)}{c_1(y_0 - y_1) + c_2(z_0 - z_1)} \tag{6.14}$$

2. Find the normal direction at the intersection point. Again, this has an exact solution, but it is also position independent: $\nabla\ell = \langle c_1, c_2\rangle$.

3. Apply the appropriate reflection about the tangent plane, preserving the distance $|\langle y_1 - y_0, z_1 - z_0\rangle|$.

When moving to multiple linear boundaries, one needs to check the formula (6.14) for all ℓ_i, take the smallest positive value of s and the appropriate ℓ_j, reflect about the line $\ell_j = 0$, and repeat until $\ell_i(y_1, z_1) > 0$ for all i. An illustration of this is shown in figure 6.3.3. The basic demonstration is done in the equilateral triangle on the left panel. Colors are used to visualize the start (red) and end (blue) of a path. To show how this can be generalized, the center and left panels implement the rectangle and an non-regular octagon. The main algorithm is identical; one only needs to specify

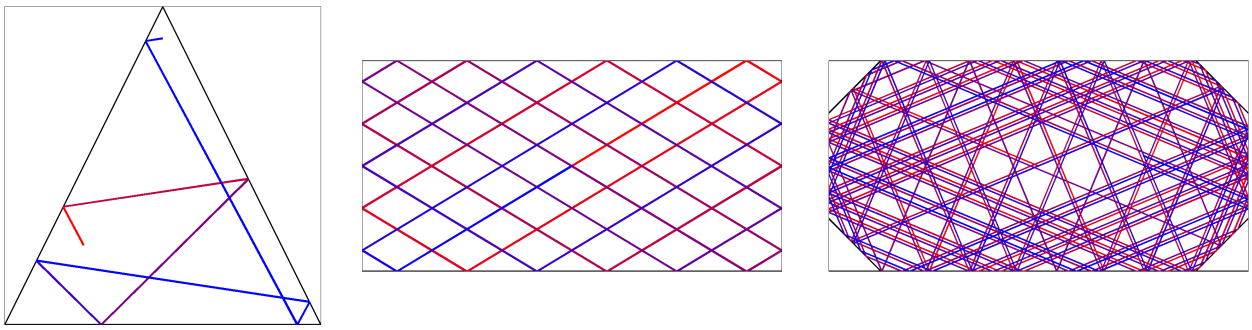

Figure 6.3: Demonstration of the reflection algorithm for some convex polygons. An extremely long trajectory is taken, then the reflection algorithm is applied iteratively until the final position (y_1, z_1) is in the domain. The number of reflections is illustrated in the changing color. Left: equilateral triangle with eight reflections. Center: Reflection in a rectangle $\lambda = 1/2$ whose initial outward trajectory has a rational slope. Right: demonstration in a non-regular octagon of the same aspect ratio.

additional planes ℓ_i. In the rectangle, using a trajectory with a rational slope is known to form a periodic paths. In the octagon, it appears two "phases" occur, with the majority of the trajectory time being rectangular, and transitions between phases occuring when the trajectory encounters one of the four sloped walls. Though not shown here, the total distance traveled prior to (as the length of the line segment), and after applying the reflections (as the sum of lengths of line segments), has been tracked in these tests and verified to be conserved, giving additional support to their validity.

The main barrier to implementing the Monte Carlo in a general convex polygon is calculation of the flow, which would most likely be numerical in nature. If this is implemented, it would prove to be a very flexible tool to repeat the skewness program, as we could approximate any convex[2] boundary using an appropriate polygon. This is one promising route towards dealing with the general question of the skewness.

Asymptotics in the regular polygons

The result of section 6.3.3 that the long time skewness is negative raises an interesting question as to the other general polygons. One may postulate that there may be an even/odd symmetry involved in the sign of the long time skewness, depending on whether n is even or odd in the n-gon (this turns out not to be the case).

[2]Non-convex domains cannot be handled with this method, as the domain can no longer be described as an intersection of half-planes.

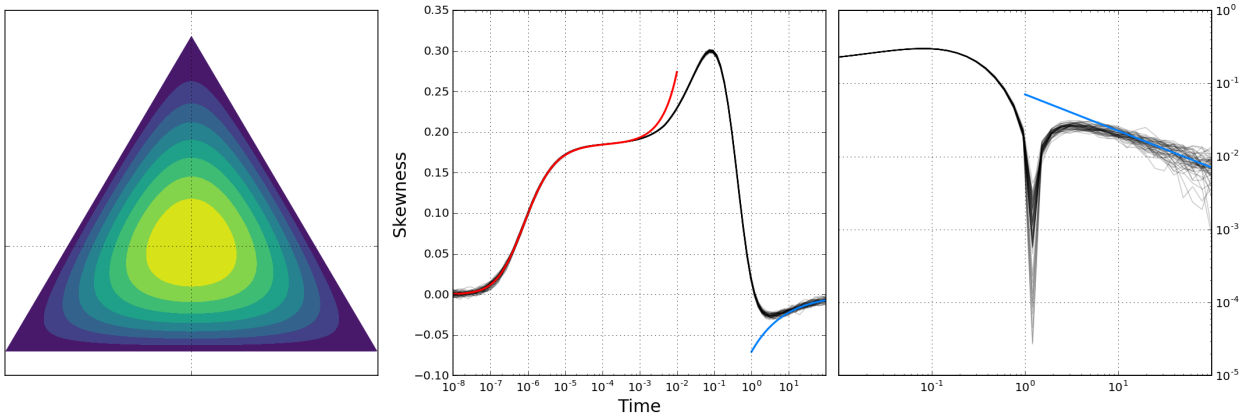

Figure 6.4: Results in the triangular geometry. Left: schematic of the flow profile, with $y = 0$ and $z = 0$ lines. Center: skewness in fifty simulations (black) and short time (red) and long time (blue) asymptotics, with $\mathrm{Pe} = 10^4$. Right: the same simulations and long time asymptotics with log-scaled axes.

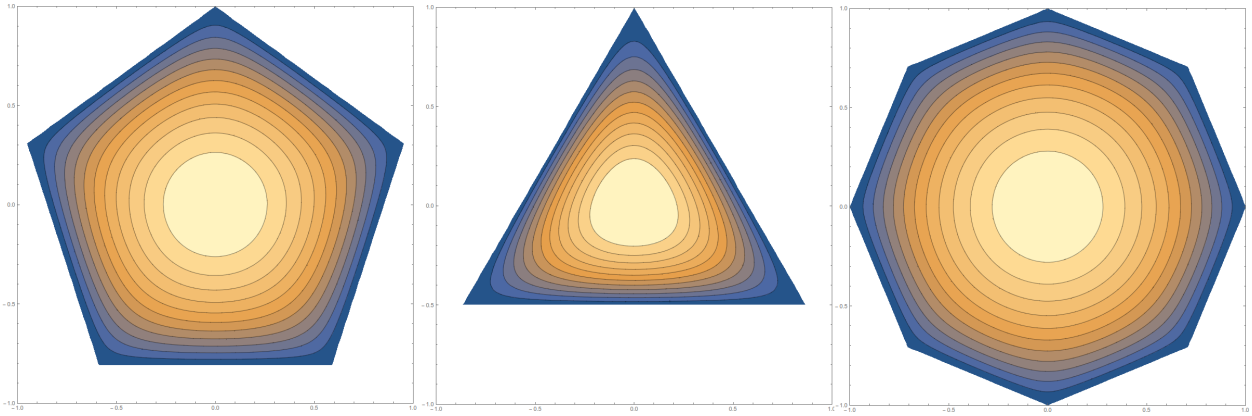

Figure 6.5: Visualization of the numerically computed flow profile in the regular n-gons for $n = 5, 3$, and 8 respectively. Curvature of the level sets is more influential for n small, but is almost immediately washed out for n large.

n	3	4	5	6	7	8	20	100	∞
$\mathcal{S}^{\mathcal{G}}$	0.185	0.0813	0.042	0.024	0.0149	0.0097	3.97×10^{-4}	8.6×10^{-7}	0
LT coeff	-0.035	0.139	0.194	0.215	0.225	0.231	0.243	0.245	0.245

Table 6.1: Geometric skewness and the long time coefficient $3\langle ug_2\rangle/(2\langle ug_1\rangle)^{3/2}$ calculated numerically for the regular n-gons. The coefficients monotonically approach the circle ($n = \infty$) value, disproving the conjecture that there may be an even/odd parity in the behavior of the skewness in the regular polygons.

To answer this question, we have utilized Mathematica's finite element package to construct the regular n-gons and numerically calculate the geometric skewness and the long time coefficient for increasing n. The regular polygons are constructed by inscribing them in the unit circle. That is, their vertices are located at

$$(y_i, z_i) = (\cos(i\pi y/n), \sin(i\pi y/n)), \text{ with } i = 0, ..., n-1. \tag{6.15}$$

We visualize the flows for a few values of n in figure 6.3.3. The flow is solved on an underlying triangular mesh, and the results are smoothly interpolated using the contour mapping program in Mathematica. For $n = 3$ and $n = 5$, curvature of the level sets of the flow extend into the inter significantly, but for $n = 8$, the contours are almost immediately circular. This is one way of understanding the approach of the flow profile (and thus the asymptotic coefficients) to the circular values.

The numerically calculated coefficients are tabulated in table 6.3.4. The triangular result differs from the result calculated in section 6.3.3 as the value of a, the radius of the inscribed circle, is different when constructing the n-gons as described above. While the value of a will grow with increasing n (limiting to 1 as $n \to \infty$), the value will not affect the *sign* of the coefficients, which is what is important.

In both the geometric skewness $\mathcal{S}^{\mathcal{G}}$ and the long time decay coefficient, we see a uniform approach towards the circle's values. Therefore, there seems to be no even/odd parity in this class of problems. Instead, the presence of skewness seems to be induced by large curvature near the walls.

APPENDIX A

Solutions to asymptotics in the ellipse.

The elliptical problems necessary to calculate the long time asymptotics can be solved by transforming into elliptical coordinates. However, the solutions are lengthy trigonometric polynomials. This chapter details the solutions of these problems.

Explicit solution of the $g_1(\xi,\eta)$ problem in the ellipse

The $g_1(y,z)$ problem (3.83) can be expressed in elliptical coordinates as (absorbing length components into the ϕ_k):

$$\kappa\left(\frac{\partial^2}{\partial\xi^2}+\frac{\partial^2}{\partial\eta^2}\right)g_1(\xi,\eta) = -\mathcal{J}u = \frac{p_x}{\mu}\Big[\phi_0(\xi)+\phi_2(\xi)\cos(2\eta)+\phi_4(\xi)\cos(4\eta)\Big], \tag{A.1a}$$

$$g_1(\xi,0) = g_1(\xi,2\pi), \quad \frac{\partial g_1}{\partial\xi}(\xi_b,\eta) = 0. \tag{A.1b}$$

Setting $c^2 = b^2 - a^2$ and $d^2 = b^2 + a^2$, the functions are

$$\phi_0(\xi) = \frac{-c^2d^2}{16}\cosh(2\xi) + \frac{c^4}{16}\cosh(4\xi), \tag{A.2a}$$

$$\phi_2(\xi) = \frac{c^2d^2}{16} - \frac{c^6}{16d^2}\cosh(4\xi), \tag{A.2b}$$

$$\phi_4(\xi) = \frac{-c^4}{16} + \frac{c^6\cosh(2\xi)}{16d^2}, \tag{A.2c}$$

and the subproblems take the forms

$$\begin{aligned} \gamma_0''(\xi) &= \phi_0(\xi), \\ \gamma_2''(\xi) - 4\gamma_2(\xi) &= \phi_2(\xi), \\ \gamma_4''(\xi) - 16\gamma_4(\xi) &= \phi_4(\xi), \end{aligned} \tag{A.3}$$

with homogeneous Neumann boundary conditions at 0 and ξ_b. The solutions are

$$\gamma_0(\xi) = \frac{5c^4 + 4d^4}{768} - \frac{c^2 d^2}{64}\cosh(2\xi) + \frac{c^4}{256}\cosh(4\xi), \tag{A.4}$$

$$\gamma_2(\xi) = -\frac{c^2 d^2}{64} + \frac{c^4}{48}\cosh(2\xi) - \frac{c^6}{192 d^2}\cosh(4\xi), \tag{A.5}$$

$$\gamma_4(\xi) = \frac{c^4}{256} - \frac{c^6}{192 d^2}\cosh(2\xi) + \frac{c^8}{768 d^4}\cosh(4\xi). \tag{A.6}$$

and the solution is

$$g_1(\xi,\eta) = \frac{p_x}{\mu\kappa}\Big[\gamma_0(\xi) + \gamma_2(\xi)\cos(2\eta) + \gamma_4(\xi)\cos(4\eta)\Big]. \tag{A.7}$$

The dimensional value of $\langle u g_1\rangle$ is (with aspect ratio $\lambda = a/b$)

$$\begin{aligned}\langle u g_1\rangle &= \frac{p_x^2}{\mu^2\kappa}\left[\frac{a^4 b^4(5a^4 + 14a^2b^2 + 5b^4)}{2304(a^2+b^2)^3}\right] \\ &= \frac{a^6 p_x^2}{\mu^2\kappa}\left[\frac{5\lambda^4 + 14\lambda^2 + 5}{2304\,\lambda^2(1+\lambda^2)^3}\right].\end{aligned} \tag{A.8}$$

Solution of the $g_2(\xi,\eta)$ problem in the ellipse

The $g_2(y,z)$ problem (3.84) can be expressed in elliptical coordinates as

$$\kappa\left(\frac{\partial^2}{\partial\xi^2} + \frac{\partial^2}{\partial\eta^2}\right) g_2(\xi,\eta) = -2\mathcal{J}\left[u g_1 - \langle u g_1\rangle\right] = \frac{p_x^2}{\mu^2\kappa}\sum_{k=0}^{4}\phi_{2k}(\xi)\cos(2k\eta), \tag{A.9a}$$

$$g_2(\xi,0) = g_2(\xi,2\pi), \quad \frac{\partial g_2}{\partial\xi}(\xi_b,\eta) = 0, \tag{A.9b}$$

with functions (defining for compactness $c^2 = b^2 - a^2$, $d^2 = b^2 + a^2$, $A = ab$)

$$\begin{aligned}\phi_0(\xi) = {} & \frac{-2304A^6c^2 - 3008A^4c^6 - 1032A^2c^{10} - 105c^{14}}{73728d^6}\cosh(2\xi) \\ & + \frac{c^4\left(320A^4 + 232A^2c^4 + 35c^8\right)}{12288d^4}\cosh(4\xi) \\ & - \frac{c^6\left(480A^4 + 304A^2c^4 + 45c^8\right)}{24576d^6}\cosh(6\xi) + \frac{c^8\left(12A^2 + 5c^4\right)}{12288d^4}\cosh(8\xi)\end{aligned} \tag{A.10a}$$

$$\begin{aligned}\phi_2(\xi) = {} & \frac{2304A^6c^2 + 3008A^4c^6 + 1032A^2c^{10} + 105c^{14}}{73728d^6} \\ & - \frac{7c^6d^2}{2048}\cosh(4\xi) + \frac{c^8}{384}\cosh(6\xi) - \frac{c^{10}\left(56A^2 + 15c^4\right)}{24576d^6}\cosh(8\xi)\end{aligned} \tag{A.10b}$$

$$\begin{aligned}\phi_4(\xi) = {} & \frac{-c^4\left(320A^4 + 232A^2c^4 + 35c^8\right)}{12288d^4} + \frac{7c^6d^2}{2048}\cosh(2\xi) \\ & - \frac{5c^{10}}{6144d^2}\cosh(6\xi) + \frac{c^{12}}{4096d^4}\cosh(8\xi)\end{aligned} \tag{A.10c}$$

$$\begin{aligned}\phi_6(\xi) = {} & \frac{c^6\left(480A^4 + 304A^2c^4 + 45c^8\right)}{24576d^6} - \frac{c^8}{384}\cosh(2\xi) \\ & + \frac{5c^{10}}{6144d^2}\cosh(4\xi) - \frac{c^{14}}{24576d^6}\cosh(8\xi)\end{aligned} \tag{A.10d}$$

$$\begin{aligned}\phi_8(\xi) = {} & \frac{-c^8\left(12A^2 + 5c^4\right)}{12288d^4} + \frac{c^{10}\left(56A^2 + 15c^4\right)}{24576d^6}\cosh(2\xi) \\ & - \frac{c^{12}}{4096d^4}\cosh(4\xi) + \frac{c^{14}}{24576d^6}\cosh(6\xi).\end{aligned} \tag{A.10e}$$

The subproblems take the forms

$$\begin{aligned}\gamma_0''(\xi) &= \phi_0(\xi), \\ \gamma_2''(\xi) - 4\gamma_2(\xi) &= \phi_2(\xi), \\ \gamma_4''(\xi) - 16\gamma_4(\xi) &= \phi_4(\xi), \\ \gamma_6''(\xi) - 36\gamma_6(\xi) &= \phi_6(\xi), \\ \gamma_8''(\xi) - 64\gamma_8(\xi) &= \phi_8(\xi),\end{aligned} \tag{A.11}$$

with homogeneous Neumann boundary conditions at 0 and ξ_b. The solutions are

$$\begin{aligned}\gamma_0(\xi) = const. &+ \frac{-2304A^6c^2 - 3008A^4c^6 - 1032A^2c^{10} - 105c^{14}}{294912d^6}\cosh(2\xi) \\ &+ \frac{c^4(320A^4 + 232A^2c^4 + 35c^8)}{196608d^4}\cosh(4\xi) \\ &- \frac{c^6\left(480A^4 + 304A^2c^4 + 45c^8\right)}{884736d^6}\cosh(6\xi) + \frac{c^8\left(12A^2 + 5c^4\right)}{786432d^4}\cosh(8\xi)\end{aligned} \tag{A.12a}$$

$$\begin{aligned}\gamma_2(\xi) = &\frac{-2304A^6c^2 - 3008A^4c^6 - 1032A^2c^{10} - 105c^{14}}{294912d^6} \\ &+ \frac{c^4\left(1376A^4 + 772A^2c^4 + 105c^8\right)}{184320d^4}\cosh(2\xi) - \frac{7c^6d^2}{24576}\cosh(4\xi) \\ &+ \frac{c^8}{12288}\cosh(6\xi) - \frac{c^{10}\left(56A^2 + 15c^4\right)}{1474560d^6}\cosh(8\xi)\end{aligned} \tag{A.12b}$$

$$\begin{aligned}\gamma_4(\xi) = &\frac{c^4\left(320A^4 + 232A^2c^4 + 35c^8\right)}{196608d^4} - \frac{7c^6d^2}{24576}\cosh(2\xi) \\ &\frac{c^8\left(30A^2 + 7c^4\right)}{49152d^4}\cosh(4\xi) - \frac{c^{10}}{24576d^2}\cosh(6\xi) + \frac{c^{12}}{196608d^4}\cosh(8\xi)\end{aligned} \tag{A.12c}$$

$$\begin{aligned}\gamma_6(\xi) = &\frac{-c^6\left(480A^4 + 304A^2c^4 + 45c^8\right)}{884736d^6} + \frac{c^8}{12288}\cosh(2\xi) \\ &- \frac{c^{10}}{24576d^2}\cosh(4\xi) + \frac{c^{12}\left(44A^2 + 9c^4\right)}{258048d^4\left(16A^2 + 3c^4\right)}\cosh(6\xi) - \frac{c^{14}}{688128d^6}\cosh(8\xi)\end{aligned} \tag{A.12d}$$

$$\begin{aligned}\gamma_8(\xi) = &\frac{c^8\left(12A^2 + 5c^4\right)}{786432d^4} - \frac{c^{10}\left(56A^2 + 15c^4\right)}{1474560d^6}\cosh(2\xi) \\ &+ \frac{c^{12}}{196608d^4}\cosh(4\xi) - \frac{c^{14}}{688128d^6}\cosh(6\xi) + \frac{c^{16}\left(136A^2 + 15c^4\right)}{82575360d^8\left(8A^2 + c^4\right)}\cosh(8\xi)\end{aligned} \tag{A.12e}$$

and the full solution is

$$g_2(\xi,\eta) = \frac{p_x^2}{\mu^2\kappa^2}\left[\sum_{k=0}^{4}\gamma_{2k}(\xi)\cos(2k\eta)\right]. \tag{A.13}$$

The key quantity needed for constructing the long time asymptotics for the skewness is $\langle ug_2\rangle$ which evaluates to

$$\begin{aligned}\langle ug_2\rangle &= \frac{-p_x^3}{\mu^3\kappa^2}\left[\frac{-a^6b^6(5a^4 - 22a^2b^2 + 5b^4}{138240(a^2+b^2)^3}\right] \\ &= \frac{a^{10}p_x^3}{\mu^3\kappa^2}\left[\frac{-(5\lambda^4 - 22\lambda^2 + 5)}{138240\,\lambda^4(1+\lambda^2)^3}\right].\end{aligned} \tag{A.14}$$

The prefactor in this expression is equivalent to $a\kappa \mathrm{Pe}^3$, which when non-dimensioqnalized, agrees with the formula in the main body. The root of this equation lying in $[0, 1]$ is $\lambda = \lambda^* = \sqrt{(11 - 4\sqrt{6})/5} \approx 0.49031$. (The other roots are at $\lambda = -\lambda^*$ and their reciprocals $\lambda = \pm 1/\lambda^*$, which is a reassuring result from a physical point of view.)

APPENDIX B

Source code for Monte Carlo simulations.

Necessary packages and compilers

What follows below is the entirety of the source code for the Fortran implementation of Monte Carlo in all the classes of geometries discussed in the main body of the dissertation. A couple packages are needed:

1. An installation of the HDF5 package, along with its compiler `h5fc`, a wrapper for the popular compilers `gfortran` or `ifort`

2. The Mersenne Twister Fortran module `mtmod.f`, freely available online [18] (and also included in the source code below).

3. To read and visualize the outputs (in HDF format), we have opted to use Python, with the packages `h5py` (for reading), and `numpy` and `matplotlib` for the necessary mathematical operations and graphics. It is possible to use other high level languages (such as Matlab), which have similar capabilities (and in the case of Matlab, has an HDF reader built-in).

To compile, one should either use the included makefile, or more generically run a command of the form (for example, for the channel code)

```
h5fc $(folder1)/*.f90 $(folder2)/*.f90 ... -o channel_mc $(COMPILERFLAGS)
```

with suggested compiler flags being (for gfortran) `-fbackslash`, `-fbounds-check`, `-funroll-loops`, and an optimization flag such as `-O2` (or `-O3`). A single parameter file format is used across all the Monte Carlo codes. A template of this is included below, and the options are described in greater detail the main body of the dissertation. Often it is the case that a large number of simulations should be run at once with the same set of parameters (except the seed for the random number generator). Included is a Python script for automating this process for an LSF cluster. If one only wishes to run a single simulation, it can be run by typing (for example, in the channel)

```
./channel_mc parameters_mc.txt
```

and if all is well, the program will give feedback as to the inputted simulation parameters, and expected time to completion.

./

This section contains necessary auxilliary files in the main directory: the makefile, the parameter file, and the batch submit script written in Python which automates submitting many jobs at once to a LSF cluster.

./makefile

```
# Makefile for geometric skewness code.
#

# -----
# Subdirectories. Make sure HOME is correct, in particular.
# --------
HOME = $(shell pwd)
UTILS = $(HOME)/utils
COMPS = $(HOME)/computation
MONTE = $(HOME)/monte
MODS = $(HOME)/modules

# --------------
#
# Fortran compiler, you _need_ to use h5fc (the hdf5 gfortran/ifort wrapper) for the MC code.
#
# -----------------------

FC        = h5fc
#FC        = gfortran

# -------
# Compiler flags. fbackslash allows for 'backspacing' in writes,
# letting you have a dynamic progress bar (for instance).
# ---------------

OTHER2 = -fbackslash -fbounds-check -funroll-loops -O3

OTHER = $(OTHER2)
# Uncomment to enable profiling
#OTHER = $(OTHER2) -pg

CLEANUP = rm -f ./*.o ./*.mod

# ---------
# END OF CONFIGURABLES
# -----------------------

# exe names, object names.
```

```
EXES = channel_mc duct_mc ellipse_mc triangle_mc racetrack_mc

channel_mc: $(MONTE)/channel_mc.f90
	$(FC) $(MODS)/*.f90 $(UTILS)/*.f90 $(COMPS)/*.f90 -o $@ $(MONTE)/$@.f90 $(OTHER)
	$(CLEANUP)

duct_mc: $(MONTE)/duct_mc.f90
	$(FC) $(MODS)/*.f90 $(UTILS)/*.f90 $(COMPS)/*.f90 -o $@ $(MONTE)/$@.f90 $(OTHER)
	$(CLEANUP)

ellipse_mc: $(MONTE)/ellipse_mc.f90
	$(FC) $(MODS)/*.f90 $(UTILS)/*.f90 $(COMPS)/*.f90 -o $@ $(MONTE)/$@.f90 $(OTHER)
	$(CLEANUP)

triangle_mc: $(MONTE)/triangle_mc.f90
	$(FC) $(MODS)/*.f90 $(UTILS)/*.f90 $(COMPS)/*.f90 -o $@ $(MONTE)/$@.f90 $(OTHER)
	$(CLEANUP)

racetrack_mc: $(MONTE)/racetrack_mc.f90
	$(FC) $(MODS)/*.f90 $(UTILS)/*.f90 $(COMPS)/*.f90 -o $@ $(MONTE)/$@.f90 $(OTHER)
	$(CLEANUP)

clean:
	rm -f $(EXES) ./*.o ./*.mod
```

```
./parameters_mc.txt

! Do NOT modify the spacing of this file;
! the parameters are read in by line number in the code. Modifying this will screw things up.
! -------------------
!
! Parameters relating to setting the initial condition
!
0.5d0                ! Aspect ratio (ignored in channel)
0.4d0          ! Shape parameter (for racetrack only)
1.0d4                ! Peclet
100000                ! Number of walkers in transverse direction in initial condition discretiz
1               ! Number of points in the x direction
0.0d0                ! Longitudinal width of initial condition (relative to width 2 in channel)
0.0d0                ! Initial y position (only for nGates=1)
5.0d0                ! Initial z position
.false.                ! Whether or not to save particle position histories
121               ! Number of bins to use in the short direction for ptwise stats
.false.         ! Save 2d histogram looking into the y direction. probably slow.
0.0d0                ! Amount of time to let the initial condition diffuse before turning on the
.false.         ! Whether to read the initial condition in from an h5 file. This will ignore all
IC_beta_reverse_rect.h5       ! Name of the h5 file with the full initial condition.
```

```
!
! -------------------
!
! Parameters relating to the timestepping
!
expo          ! tstep_type, determining the target times. One of 'unif', 'expo', 'supplied'
1.0d-8          ! tmin, the first nonzero time.
1.0d-4            ! Maximum internal timestep allowed. (if bigger than target time's dt, is ok)
1.0d-1               ! Tfinal
81                    ! ntt, number of target time points (including zero)
tsteps_sample.h5        ! If tstep_type is 'supplied', reads times from this file and ignores t
fill                    ! The seed for the RNG; if not an integer, then one is generated by bat
```

```
./batch_submit.py

# Script to execute a single execute locally.
#

import subprocess,os

# -------------------------
# Parameters specifying the number of runs,
# and location and names for output files.
# -------------------

execute = "local"   # if "local" then run directly, sequentially.
                    # "bsub" submits jobs to Kure/Killdevil.

n = 1               # number of trials to run.
fname_prefix = "c_" # prefix name (appended with numbers on output.)
parent = "./"       # parent folder (should contain folders out/ and err/ if on a cluster)
sim_folder = ""     # simulation folder

exe_loc = "./channel_mc" # Name of the executable (which geometry?)

# ------------------
# More bsub options

memoryreq = 0.1        # in GB
queue = "week"

# --------------------------

folder = parent+sim_folder
file_prefix = folder+fname_prefix
out_suffix = '.h5'
```

```
# ------------------------

def process_parameter_file(time_loc,i):
# Looks at the parameter file, checks if the line
# specifying the RNG seed is filled in with an
# integer. If it is, leave it. If not, replace the line
# with a seed based on the operating system RNG
# (eg, /dev/urandom.) Python handles this part automatically.

     tfile = open(time_loc,'r')
     lines = tfile.readlines()

     # If thing is not an integer, replace the line
     # with a random seed.
     sidx = 31 # Line which should hold the seed.
     seed = lines[sidx].split()
     try:
          int(seed)
     except:
          int1 = int(os.urandom(10).encode('hex'),16)
          int2 = i*int(os.urandom(10).encode('hex'),16)

          # Bitwise XOR. Why? I don't know. Just because.
          int3 = int1^int2

          lines[sidx] = str(int3)[-8:-1]+' \n'
     # end try
     tfile.close()

     tfile = open(time_loc,'w')
     tfile.writelines(lines)
     tfile.close()

# end def

# Creating simulation initialization files.
for i in range(n):
     # Make a four-digit index (0000 through 9999)
     stridx = str(i).zfill(4)

     param_loc = folder+"parameters_mc_"+fname_prefix+stridx+".txt"
     out_loc = file_prefix+stridx

     # Copy the parameter files to the appropriate place, and
     # generate seeds if necessary.
     copy_command1 = ["cp","parameters_mc.txt",param_loc]

     subprocess.call(copy_command1)
```

```
     process_parameter_file(param_loc,i)

# end if

# Running simulations after the files are created.
for i in range(n):

     stridx = str(i).zfill(4)

     param_loc = folder+"parameters_mc_"+fname_prefix+stridx+".txt"
     out_loc = file_prefix+stridx+out_suffix

     exe_command = [exe_loc,param_loc,out_loc]

     if (execute=="bsub"):
          bsub_prefix = ["bsub","-n","1","-o",parent+"out/"+"out."+stridx,"-e",parent+"err/"+"er

          if (queue == "week"):
               more = ["-q",queue]
          else:
               more = ["-M",str(int(memoryreq)),"-q",queue]
          # end if

          command = bsub_prefix + more + exe_command
     else:
          command = exe_command
     # end if


     subprocess.call(command)

# end for
```

```
 ./monte/
```

```
 ./monte/channel_mc.f90
```

```
program channel_mc
! Program to do Monte Carlo in a channel.

use HDF5
use mtmod
use mod_time
use mod_duration_estimator
use mod_readbuff
```

```
implicit none

    integer, parameter                              :: i64 = selected_int_kind(18)
    integer(kind=i64)                               :: mc_n
    integer                                         :: nt,kt,ny,nGates,nTot,nr,funit,idx,i
    double precision                                :: t,Tfinal,tmin,dtmax,kscale,Pe,mcvar,dz,told

    ! Internal variables for checking if we're at a target time during the timestepping.
    double precision                                :: next_tt
    integer                                         :: tt_idx

    ! Type of geometry.
    character(len=1024)                             :: geometry
    double precision                                :: t_warmup

    ! i/o
    character(len=1024)                                 :: param_file,&
                                                           other_file,filename,tstep_type,ic_fi
    character(len=1024)                                 :: out_msg

    ! For compatibility only.
    double precision                                :: aratio,q

    ! RNG stuff.
    integer(i64)                                    :: mt_seed

    ! Positions, position/statistic histories. Some of these are not used by channel,
    ! but are for compatibility with the 2/3d code.
    double precision                                    :: a    ! Channel width.
    double precision                                    :: y0,z0  ! Initial conditions (point sou
    double precision, dimension(:), allocatable         :: X,Y    ! Position arrays
    double precision, dimension(:,:), allocatable       :: Xbuffer,Ybuffer
    integer(kind=i64)                                   :: buffer_len,bk,inext,rem
    double precision, dimension(:), allocatable         :: means,vars,skews,kurts,t_hist
    double precision, dimension(:,:), allocatable       :: hist_centers,hist_heights

    double precision                                    :: x0width
    integer                                             :: x0n

    ! Moments through slices and distributions
    double precision, dimension(:,:), allocatable       :: means_sl,vars_sl,skews_sl,kurts_sl
    integer                                             :: nbins,bin_count,nhb,nby
    double precision                                    :: dby

    ! HDF
    integer                                             :: rank,h5error
    character(len=1024)                                 :: fname2,descr
```

```
      character(len=1024)                 :: dsetname

      ! HDF variables.
      integer(hid_t)                      :: file_id
      integer(hid_t)                      :: dset_id_X,dset_id_Y,dset_id_Z,&
                                             dset_id_Sk,dset_id_t
      integer(hid_t)                      :: dspace_id_X,dspace_id_Y,dspace_id_Z,&
                                             dspace_id_Sk,dspace_id_t

      integer(hsize_t), dimension(2)      :: data_dims

      ! ------------------------
      logical                             :: save_hist,save_hist2d,use_external_ic

      logical check_ic_channel

      ! Advection/diffusion functions!
      external  :: impose_reflective_BC_rect, u_channel

      ! ----------------------------------------
      ! Internal parameters that you might want to change at some point.

      ! Length of the buffer before writing to disk.
      ! Only relevant if saving the entire position history.
      ! Run time is bottlenecked by this to a severe degree, so in general
      ! this should be made as large as possible while still fitting in memory.
      parameter(buffer_len = 100)

      ! Scale of the channel: [-a,a]. Not much reason to use anything other than 1.0d0.
      parameter(a=1.0d0)
      parameter(geometry = "channel")

      ! Number of bins when looking at the cross-sectionally averaged distribution.
      parameter(nhb = 400)

      ! ------------------------------------------------------------

      ! Number of spatial dimensions for random walks!
      ! nd=2 for channel, nd=3 for duct/pipe.
      integer, parameter  :: nd=2

      ! Read all parameters from file.
      call get_command_argument(1,param_file)
!      call get_command_argument(2,time_file)
      call get_command_argument(2,filename)

      ! NOTE - save_hist2d is not implemented as of 11 Nov 2016.
      ! Only put here for compatibility.
```

```
call read_inputs_mc(param_file,aratio,q,Pe,nGates,x0n,x0width,y0,z0,save_hist,nbins,&
                        save_hist2d,t_warmup,use_external_ic,ic_file,tstep_type,&
                        dt,dtmax,Tfinal,ntt,other_file,mt_seed)

if (filename=="") then
     out_msg = 'missing_args'
     call channel_mc_messages(out_msg)
     go to 1234
end if

   ! Assign the number of bins in each direction.
if (nbins .eq. 0) then
     nby = 0
     dby = 0.0d0
else
     nby = nbins
     dby = 2.0d0*a/nby
end if

! --------------------------------
!
! Based on the input, generate the array target_times, (times to save output)
! and get information for the internal array t_hist.
!

call generate_target_times(dt,tstep_type,Tfinal,other_file)

! Get the value of nt before allocating arrays.
call correct_tstep_info(ntt,nt,target_times,dtmax)

! Initialize the Mersenne Twister RNG with seed read in from the
! input files.

call sgrnd(mt_seed)

! Get the total number of simulations
ny = nGates
nTot = nGates*x0n

! Correct nTot if we are using an external file.
if (use_external_ic) then
     dsetname = "nTot"
     call hdf_read_1d_darray(i,ic_file,dsetname)
     nTot = int(readbuff_double(1))
     deallocate(readbuff_double)
else
! Recalculate nGates and nTot based on the values for ny and nz.
     nGates = ny
```

```
     nTot = nGates*x0n
end if

! ------------------
! Allocate arrays.
!

! Time
allocate(t_hist(nt))

! Positions
allocate(X(nTot),Y(nTot))

! Channel-averaged stats
allocate(means(ntt),vars(ntt),skews(ntt),kurts(ntt))

! Stats on slices
if (.not. (nbins .eq. 0)) then
     allocate(means_sl(ntt,nbins),vars_sl(ntt,nbins),skews_sl(ntt,nbins),kurts_sl(ntt,nbins
end if

! Cross-sectionally averaged distribution
allocate(hist_centers(ntt,nhb),hist_heights(ntt,nhb))

! --------------------------
! Generate initial conditions and internal timestepping.
!

call set_initial_conds_channel_mc(ny,nGates,x0n,nTot,X,Y,y0,a,x0width,t_warmup,use_external

if (.not. check_ic_channel(nTot,Y,a)) then
     write(*,*) "Part of the initial condition lies outside the domain. Exiting."
     go to 1234
end if

call generate_internal_timestepping(ntt,nt,target_times,t_hist,dtmax)

call print_parameters(aratio,q,Pe,nGates,nTot,y0,z0,save_hist,&
                        t_hist,dtmax,nt,ntt,mt_seed,geometry,use_external_ic)

! ---------------------------------------------------------
! Initialize HDF with appropriate dataset, etc.

fname2 = trim(filename)

call hdf_create_file(fname2)

! We need to open the h5 file after hdf_create_file
```

```
! because the interface is "global" amongst all files
! containing the hdf5 module.

call h5open_f(h5error)


if (save_hist) then
     ! Set up dataspaces in the hdf file for:
     ! X, Y.
     !
     ! Allocate memory for the memory buffers here, too.

     allocate(Xbuffer(buffer_len,nTot))
     allocate(Ybuffer(buffer_len,nTot))

     call h5fopen_f(fname2, H5F_ACC_RDWR_F, file_id, h5error)

     data_dims(1) = ntt
     data_dims(2) = nTot
     rank = 2

     dsetname = "X"
     call h5screate_simple_f(rank, data_dims, dspace_id_X, h5error)
     call h5dcreate_f(file_id, dsetname, H5T_NATIVE_DOUBLE, dspace_id_X, &
                    dset_id_X, h5error)

     dsetname = "Y"
     call h5screate_simple_f(rank, data_dims, dspace_id_Y, h5error)
     call h5dcreate_f(file_id, dsetname, H5T_NATIVE_DOUBLE, dspace_id_Y, &
                    dset_id_Y, h5error)

end if

! Save a history of time per iteration for predicting time to completion.
mde_ntt = nt-1
allocate(mde_dts(mde_ntt))

inext = 1
tt_idx = 1

call accumulate_moments_1d(tt_idx,ntt,nTot,X,Y,a,means,vars,skews,&
                kurts,nby,means_sl,vars_sl,skews_sl,kurts_sl)
call make_histogram(nTot,X,nhb,hist_centers(tt_idx,1:nhb),hist_heights(tt_idx,1:nhb))

   tt_idx = 2
   next_tt = target_times(tt_idx)

! -----------
```

```
! Prepare the buffer to save position histories if requested.
bk = 0
if (save_hist) then
     call buffer_op_channel(bk,nTot,buffer_len,Xbuffer,Ybuffer,&
                            X,Y,ntt,inext,dset_id_X,dset_id_Y)
end if


! --------------
! Start the timestepping.

out_msg = 'simul_start'
call channel_mc_messages(out_msg)

do kt=2,nt

     call system_clock(mde_t1,count_rate)    ! Time for progress.

     ! Push forward time.
     t = t_hist(kt)
     dt = t_hist(kt) - t_hist(kt-1)

     !
     ! Primary timestep
     !

     call apply_advdiff1_chan(nTot,X,Y,Pe,dt,a, &
                    u_channel,impose_reflective_BC_rect)

     !
     ! If we're at a target time,
     ! calculate and save moments (and positions, if requested),
     ! then increment tt_idx and update next_tt.
     !
     if ( t .eq. next_tt ) then

          call accumulate_moments_1d(tt_idx,ntt,nTot,X,Y,a,means,vars,skews,&
                         kurts,nby,means_sl,vars_sl,skews_sl,kurts_sl)

          ! Update the histogram centers and heights.
          call make_histogram(nTot,X,nhb,hist_centers(tt_idx,1:nhb),hist_heights(tt_idx,1:n

          !
          ! Write history if requested.
          !
          ! Need to buffer writes to the hard drive so that we don't lock up the
          ! computation with file opens/closes. Ideally the buffer should be as
          ! large as possible while fitting into RAM; modify the relevant parameter
```

```
          ! (buffer_len) hard coded in this program to play around with this.
          !

          if (save_hist) then
               call buffer_op_channel(bk,nTot,buffer_len,&
                                      Xbuffer,Ybuffer,X,Y,ntt,inext,dset_id_X,dset_id_Y)
          end if

          !
          ! Update the target time and array index.
          !
          if (next_tt .lt. Tfinal) then
               tt_idx = tt_idx + 1
               next_tt = target_times(tt_idx)
          end if
     end if

     ! Display percentage progress. The last argument as .true. should be used with gfortra
     ! (or any other compiler that supports "\b"), or .false. with ifort.

     call system_clock(mde_t2,count_rate)    ! Time in milliseconds

     mde_ntc = kt-1
     mde_dts(mde_ntc) = (mde_t2-mde_t1)/dble(count_rate)  ! Time in seconds

     call progress_meter(kt,nt,.true.)

end do

out_msg = 'simul_done'
call channel_mc_messages(out_msg)

if (save_hist) then
! Write the remainder of the buffer, then close the file.

     rem = ntt-inext+1
     if (rem .gt. 0) then
          call hdf_write_to_open_2d_darray(ntt,nTot,inext,rem,Xbuffer(1:rem,1:nTot),dset_id
          call hdf_write_to_open_2d_darray(ntt,nTot,inext,rem,Ybuffer(1:rem,1:nTot),dset_id
     end if

     call h5dclose_f(dset_id_X,h5error)
     call h5dclose_f(dset_id_Y,h5error)
     call h5fclose_f(file_id,h5error)
end if

! Because of the nature of hdf5 mod for fortran,
! we close the interface here, since it gets
```

```
     ! re-opened in the calls below.

     call h5close_f(h5error)

     ! Save all the remaining arrays. It's a lot of fluff so it's been
     ! given its own subroutine.

     call save_the_rest_channel(fname2,geometry,ntt,target_times,means,vars,skews,kurts,&
                   nby,means_sl,vars_sl,skews_sl,kurts_sl,nhb,hist_centers,hist_heights,&
                   Pe,nTot,mt_seed,dtmax,t_warmup)

     ! -------

     deallocate(X,Y)

     deallocate(means,vars,skews,kurts,target_times)

     if (.not. (nbins .eq. 0)) then
          deallocate(means_sl,vars_sl,skews_sl,kurts_sl)
     end if

     deallocate(hist_centers,hist_heights)
     deallocate(t_hist)

     if (save_hist) then
          deallocate(Xbuffer,Ybuffer)
     end if

     out_msg = 'done'
     call channel_mc_messages(out_msg)

     write(*,*) ""

1234 continue

end program channel_mc
```

./monte/duct_mc.f90

```
program duct_mc
! Program to do Monte Carlo in a duct.

use HDF5
use mtmod
use mod_time
use mod_duration_estimator
use mod_ductflow
```

```
use mod_readbuff

implicit none

    ! Array sizes, parameters, local vars
    integer, parameter                                  :: i64 = selected_int_kind(18)

    integer                                             :: i

    integer                                             :: nGates,nTot
    integer                                             :: mc_n,nt,kt,&
                                                             ny,nz,nr,tt_idx

    double precision                                    :: Tfinal,dt,dtmax,Pe,mcvar,aratio,q,next
    double precision                                    :: t,uval,told

    integer(i64)                                          :: mt_seed


    ! Positions, position/statistic histories
    double precision                                    :: a,b,bin_lo,bin_hi,dby,dbz,y0,z0,t_warm
    double precision, dimension(:), allocatable         :: X,Y,Z
    double precision, dimension(:,:), allocatable       :: Xbuffer,Ybuffer,Zbuffer
    integer                                             :: buffer_len,bk,inext,rem
    double precision, dimension(:), allocatable         :: means,vars,skews,kurts,t_hist,X_bin
    double precision, dimension(:,:), allocatable       :: hist_centers,hist_heights

    double precision, dimension(:,:,:), allocatable     :: means_sl,vars_sl,skews_sl,kurts_sl

    double precision, dimension(:,:), allocatable       :: W
    integer                                             :: nd,bin_count,kb,jb,n_bins,nbx,nby,nbz,

    double precision                                    :: x0width
    integer                                             :: x0n


    ! Stuff for 2d histogram looking into the short direction.
    double precision, dimension(:,:,:), allocatable     :: hist2d
    double precision, dimension(:,:), allocatable       :: hist2dcx, hist2dcy ! bin centers.

    ! Type of geometry, only used for filenames and things.
    character(len=1024)                                 :: geometry

    ! i/o
    character(len=1024)                                 :: param_file,&
                                                             other_file,filename,tstep_type,ic_fi
    character(len=1024)                                 :: out_msg
```

```
character(len=1024)                                 :: arrayname,descr
! HDF
integer                                             :: rank,h5error
character(len=1024)                                 :: fname2

character(len=1024)                 :: dsetname

! HDF variables.
integer(hid_t)                      :: file_id
integer(hid_t)                      :: dset_id_X,dset_id_Y,dset_id_Z
integer(hid_t)                      :: dspace_id_X,dspace_id_Y,dspace_id_Z

integer(hsize_t), dimension(2)      :: data_dims

! Flags to save position histories and read IC from a file.
logical                             :: save_hist,save_hist2d,use_external_ic
logical check_ic_duct


! Advection/diffusion functions!
external  :: impose_reflective_BC_rect, u_duct_precomp

! ----------------------------------------
! Internal parameters that you might want to change at some point.

! Number of physical dimensions. No reason to change this.
parameter(nd=3)

parameter(geometry = "duct")


! Length of the buffer before writing to disk.
! Only relevant if saving the entire position history.
! Run time is bottlenecked by this to a severe degree, so in general
! this should be made as large as possible while still fitting in memory.
!
! Some quick numbers; 5e3 buffer size with 1e4 walks
! requires about 1GB RAM. So, you should choose the parameters
! so that it works out.
!
! RAM = kt*buffer*walks
!    ..............=> buffer = RAM/(k*walks)
!                      k = RAM/(buffer*walks)
!
! In our example kt = 1/(5*10**7).
!
parameter(buffer_len = 51)
```

```
! Number of bins when looking at the cross-sectionally averaged distribution.
parameter(nhb = 400)


! -----------------------------------------------------

! Read all parameters from file.
call get_command_argument(1,param_file)
call get_command_argument(2,filename)

call read_inputs_mc(param_file,aratio,q,Pe,nGates,x0n,x0width,y0,z0,save_hist,n_bins,save_h
                        use_external_ic,ic_file,tstep_type,dt,dtmax,Tfinal,ntt,other_file,


if (filename=="") then
     out_msg = 'missing_args'
     call duct_mc_messages(out_msg,nz)
     go to 1234
end if

! Set dimensions of the thing.
a = 1.0d0
b = a/aratio

    ! Assign the number of bins in each direction for ptwise stats.
if (n_bins .eq. 0) then
     nby = 0
     nbz = 0

     dby = 0.0d0
     dbz = 0.0d0
else
        nby = n_bins
        nbz = ceiling(n_bins/aratio) ! Could also make this the same as nby.

     dby = (2.0d0*a)/nby
     dbz = (2.0d0*b)/nbz

end if

nbx = nby

!
! Generate the target times; times at which output is saved.
! Internal timestepping is created after.
!

call generate_target_times(dt,tstep_type,Tfinal,other_file)
```

```
! Get the value of nt before allocating arrays.
call correct_tstep_info(ntt,nt,target_times,dtmax)

! Initialize the Mersenne Twister RNG with seed read in from the
! input files.

call sgrnd(mt_seed)


! Based on the specified number of discretization points,
! calculate the appropriate number of points to seed in
! each direction so that they are approximately uniformly
! spaced.
if (nGates .gt. 1) then
     ny = floor(dsqrt(nGates*aratio))+1
     nz = floor(dsqrt(nGates/aratio))+1
else
     ny = 1
     nz = 1
end if


! If resolution is an issue, exit.
if ( (ny .lt. 7) .and. (nGates .gt. 1) ) then
     out_msg = 'resolution'
     call duct_mc_messages(out_msg,nz)
     go to 1234
end if

! Correct nTot if we are using an external file.
if (use_external_ic) then
     dsetname = "nTot"
     call hdf_read_1d_darray(i,ic_file,dsetname)
     nTot = int(readbuff_double(1))
     deallocate(readbuff_double)
else
! Recalculate nGates and nTot based on the values for ny and nz.
     nGates = ny*nz
     nTot = nGates*x0n
end if

! -----------------
! Allocate arrays.
!

! Internal time
allocate(t_hist(nt))
```

```
! Positions
allocate(X(nTot), Y(nTot), Z(nTot))

! Cross-sectionally averaged stats.
allocate(means(ntt), vars(ntt), skews(ntt),kurts(ntt))

! Pointwise stats
if (.not. (n_bins .eq. 0)) then
     allocate(means_sl(ntt,nby,nbz),vars_sl(ntt,nby,nbz),&
               skews_sl(ntt,nby,nbz),kurts_sl(ntt,nby,nbz))
end if

! If the 2d histogram (looking in the short direction) is desired,
! allocate.
if (save_hist2d) then
     allocate(hist2d(ntt,nbx,nby))
     allocate(hist2dcx(ntt,nbx))
     allocate(hist2dcy(ntt,nby))
end if

! Temporary vector used for binning purposes.
allocate(X_bin(nTot))

! Cross-sectionally averaged distribution.
allocate(hist_centers(ntt,nhb),hist_heights(ntt,nhb))

! Precalculation of the flow.
allocate(ya(ui), za(uj), u_precomp(ui,uj))

! ------------------
! Generate initial conditions and internal timestepping.
!

call set_initial_conds_duct_mc(ny,nz,nGates,x0n,nTot,X,Y,Z, &
                        y0,z0,a,b,x0width,t_warmup,use_external_ic,ic_file)

if (.not. check_ic_duct(nTot,Y,Z,a,b)) then
     write(*,*) "Part of the initial condition lies outside the domain. Exiting."
     go to 1234
end if

call generate_internal_timestepping(ntt,nt,target_times,t_hist,dtmax)

call print_parameters(aratio,q,Pe,nGates,nTot,y0,z0,save_hist,&
                        t_hist,dtmax,nt,ntt,mt_seed,geometry,use_external_ic)

! -----------------------------------------------------
```

```
! Initialize HDF with appropriate dataset, etc.

fname2 = trim(filename)

call hdf_create_file(fname2)

! We need to open the h5 file after hdf_create_file
! because the interface is "global" amongst all files
! containing the hdf5 module.

call h5open_f(h5error)

if (save_hist) then
     ! Set up dataspaces in the hdf file for:
     ! X, Y, Z.
     !
     ! Allocate memory for the memory buffers here, too.

     allocate(Xbuffer(buffer_len,nTot))
     allocate(Ybuffer(buffer_len,nTot))
     allocate(Zbuffer(buffer_len,nTot))

     call h5fopen_f(fname2, H5F_ACC_RDWR_F, file_id, h5error)

     data_dims(1) = ntt
     data_dims(2) = nTot
     rank = 2

     dsetname = "X"
     call h5screate_simple_f(rank, data_dims, dspace_id_X, h5error)
     call h5dcreate_f(file_id, dsetname, H5T_NATIVE_DOUBLE, dspace_id_X, &
                    dset_id_X, h5error)

     dsetname = "Y"
     call h5screate_simple_f(rank, data_dims, dspace_id_Y, h5error)
     call h5dcreate_f(file_id, dsetname, H5T_NATIVE_DOUBLE, dspace_id_Y, &
                    dset_id_Y, h5error)
     dsetname = "Z"
     call h5screate_simple_f(rank, data_dims, dspace_id_Z, h5error)
     call h5dcreate_f(file_id, dsetname, H5T_NATIVE_DOUBLE, dspace_id_Z, &
                    dset_id_Z, h5error)

end if

! Save a history of time per iteration for predicting time to completion.
mde_ntt = nt-1
allocate(mde_dts(mde_ntt))
```

```
! --------------------------------
!
! Precompute values of u on a grid.
! The actual values and arrays are defined in the module mod_ductflow
! and only accessed internally.
!
write(*,"(A25)",advance="no") "Precomputing u values... "
call precompute_uvals_ss(a,b,aratio)
write(*,"(A5)") " done."

! --------------------------------
! Calculate the statistics of the initial condition.
!
inext = 1
tt_idx = 1

call accumulate_moments_2d(tt_idx,ntt,nTot,X,Y,Z,-a,a,-b,b,means,vars,skews,&
          kurts,nby,nbz,means_sl,vars_sl,skews_sl,kurts_sl)

call make_histogram(nTot,X,nhb,hist_centers(tt_idx,1:nhb),hist_heights(tt_idx,1:nhb))

if (save_hist2d) then
     call make_histogram2d(nTot,X,Y,nbx,nby,hist2dcx(tt_idx,1:nbx),&
               hist2dcy(tt_idx,1:nby), hist2d(tt_idx,1:nbx,1:nby))
end if

tt_idx = 2
next_tt = target_times(tt_idx)

! --------------------------------
!
! Prepare the buffer to save position histories if requested.
!
bk = 0

if (save_hist) then
     call buffer_op_duct(bk,nTot,buffer_len,Xbuffer,Ybuffer,Zbuffer,&
                         X,Y,Z,ntt,inext,dset_id_X,dset_id_Y,dset_id_Z)
end if

! --------------
! Start the timestepping.

out_msg = 'simul_start'
call duct_mc_messages(out_msg,nz)

do kt=2,nt
```

```
call system_clock(mde_t1,count_rate)    ! Time for progress.
! Push forward time.
t = t_hist(kt)
dt = t_hist(kt) - t_hist(kt-1)


!
! Primary timestep
!

call apply_advdiff1_duct(nTot,X,Y,Z,Pe,dt,a,b, &
               u_duct_precomp,impose_reflective_BC_rect)

!
! Check if we're at a target time. If we are,
! and calculate and save moments (and positions, if requested),
! then increment tt_idx and update next_tt.
!
if ( t .eq. next_tt ) then

     call accumulate_moments_2d(tt_idx,ntt,nTot,X,Y,Z,-a,a,-b,b,means,vars,skews,&
               kurts,nby,nbz,means_sl,vars_sl,skews_sl,kurts_sl)

     call make_histogram(nTot,X,nhb,hist_centers(tt_idx,1:nhb),hist_heights(tt_idx,1:n

     if (save_hist2d) then
          call make_histogram2d(nTot,X,Y,nbx,nby,hist2dcx(tt_idx,1:nbx),&
                    hist2dcy(tt_idx,1:nby), hist2d(tt_idx,1:nbx,1:nby))
     end if

     ! Write history if requested.
     !
     ! Need to buffer writes to the hard drive so that we don't lock up the
     ! computation with file opens/closes. Ideally the buffer should be as
     ! large as possible while fitting into RAM; modify the relevant parameter
     ! at the end of the variable definitions.
     !

     if (save_hist) then
          call buffer_op_duct(bk,nTot,buffer_len,Xbuffer,Ybuffer,Zbuffer,&
                         X,Y,Z,ntt,inext,dset_id_X,dset_id_Y,dset_id_Z)
     end if

     !
     ! Update the target time and array index.
     !
     if (next_tt .lt. Tfinal) then
          tt_idx = tt_idx + 1
```

```
                next_tt = target_times(tt_idx)
            end if
        end if

        ! Display percentage progress. The last argument as .true. should be used with gfortra
        ! (or any other compiler that supports "\b"), or .false. with ifort.

        call system_clock(mde_t2,count_rate)    ! Time in milliseconds

        mde_ntc = kt-1
        mde_dts(mde_ntc) = (mde_t2-mde_t1)/dble(count_rate)  ! Time in seconds


        call progress_meter(kt,nt,.true.)


    end do

    if (save_hist) then
    ! Write the remainder of the buffer, then close the file.

        rem = ntt-inext+1

        if (rem .gt. 0) then
            call hdf_write_to_open_2d_darray(ntt,nTot,inext,rem,Xbuffer(1:rem,1:nTot),dset_id
            call hdf_write_to_open_2d_darray(ntt,nTot,inext,rem,Ybuffer(1:rem,1:nTot),dset_id
            call hdf_write_to_open_2d_darray(ntt,nTot,inext,rem,Zbuffer(1:rem,1:nTot),dset_id
        end if

        call h5dclose_f(dset_id_X,h5error)
        call h5dclose_f(dset_id_Y,h5error)
        call h5dclose_f(dset_id_Z,h5error)
        call h5fclose_f(file_id,h5error)
    end if

    ! For now, with the hist2d stuff, save it here rather than in "save the rest".

    ! Because of the nature of hdf5 mod for fortran,
    ! we close the interface here, since it gets
    ! re-opened in the calls below.
    call h5close_f(h5error)

    if (save_hist2d) then
        arrayname = "hist2dcx"
        descr = "Array tracking bin centers in the x direction for hist2d"
        call hdf_add_2d_darray_to_file(ntt,nbx,hist2dcx,fname2,arrayname,descr)

        arrayname = "hist2dcy"
```

```
          descr = "Array tracking bin centers in the y direction for hist2d"
          call hdf_add_2d_darray_to_file(ntt,nby,hist2dcy,fname2,arrayname,descr)

          arrayname = "hist2d"
          descr = "Array tracking the density for hist2d"
          call hdf_add_3d_darray_to_file(ntt,nbx,nby,hist2d,fname2,arrayname,descr)
     end if

     ! Save all the remaining arrays. It's a lot of fluff so it's been
     ! given its own subroutine.

     call save_the_rest_duct(fname2,geometry,ntt,target_times,means,vars,skews,kurts,nby,nbz,&
                             means_sl,vars_sl,skews_sl,kurts_sl,nhb,hist_centers,hist_heights,&
                             Pe,nTot,mt_seed,aratio,q,dtmax,t_warmup)

     ! ------
     deallocate(X,Y,Z)

     deallocate(means,vars,skews,kurts,target_times)
     if (.not. (n_bins .eq. 0)) then
          deallocate(means_sl,vars_sl,skews_sl,kurts_sl)
     end if
     deallocate(hist_centers,hist_heights)
     deallocate(t_hist)
     deallocate(ya,za,u_precomp)
     if (save_hist) then
          deallocate(Xbuffer,Ybuffer,Zbuffer)
     end if

     if (save_hist2d) then
          deallocate(hist2d,hist2dcx,hist2dcy)
     end if

     out_msg = 'done'
     call duct_mc_messages(out_msg,nz)


1234 continue

end program duct_mc
```

./monte/ellipse_mc.f90

```
program ellipse_mc
! Program to do Monte Carlo in an ellipse.

use HDF5
```

```
use mtmod
use mod_time
use mod_duration_estimator

implicit none

     ! Array sizes, parameters, local vars
     integer, parameter                                :: i64 = selected_int_kind(18)

     integer                                           :: nGates
     integer                                           :: nTot,nt,kt,ny,nz,tt_idx
     double precision                                  :: Tfinal,dt,dtmax,Pe,aratio,q,next_tt
     double precision                                  :: t

     integer(i64)                                      :: mt_seed

     integer                                           :: maxrefl

     ! Positions, position/statistic histories
     integer                                           :: n_bins,nbx,nby,nbz,nhb
     double precision                                  :: a,b,dby,dbz,t_warmup
     double precision                                  :: y0,z0

     double precision                                  :: x0width
     integer                                           :: x0n

     ! Stuff for 2d histogram looking into the short direction.
     double precision, dimension(:,:,:), allocatable   :: hist2d
     double precision, dimension(:,:), allocatable     :: hist2dcx, hist2dcy ! bin centers.


     double precision, dimension(:), allocatable       :: X,Y,Z
     double precision, dimension(:,:), allocatable     :: Xbuffer,Ybuffer,Zbuffer
     integer                                           :: buffer_len,bk,inext,rem
     double precision, dimension(:), allocatable       :: means,vars,skews,kurts,t_hist

     double precision, dimension(:,:,:), allocatable   :: means_sl,vars_sl,skews_sl,kurts_sl
     double precision, dimension(:,:), allocatable     :: hist_centers,hist_heights


     ! Type of geometry, only used to modify the output header.
     character(len=1024)                               :: geometry

     ! i/o
     character(len=1024)                               :: param_file,other_file,&
                                                            filename,tstep_type,ic_file

     character(len=1024)                               :: out_msg
```

```
character(len=1024)                                 :: arrayname,descr

! HDF
integer                                             :: rank,h5error
character(len=1024)                                 :: fname2

character(len=1024)                                 :: dsetname

! HDF variables.
integer(hid_t)                                      :: file_id
integer(hid_t)                                      :: dset_id_X,dset_id_Y,dset_id_Z
integer(hid_t)                                      :: dspace_id_X,dspace_id_Y,dspace_id_Z

integer(hsize_t), dimension(2)                      :: data_dims

! For saving position histories and read IC from a file.
logical                             :: save_hist,save_hist2d,use_external_ic


! References to functions that go in arguments.
external  :: impose_reflective_BC_ellipse, u_ellipse

! Parameters.
!

parameter(geometry = "ellipse")

! Buffer length, to reduce the number of writes
! onto the HDF files.
! Make this as large as possible to fit in RAM!
!
! 5*10**3 buff * 10**4 walks => ~1GB RAM
!
! RAM = kt*buffer*walks
!    ..............=> buffer = RAM/(k*walks)
!                      k = RAM/(buffer*walks)
!
! In our example kt = 1/(5*10**7).
!
parameter(buffer_len = 20)

! Number of bins when looking at the cross-sectionally averaged distribution.
parameter(nhb = 400)

! Maximum reflections applied before giving up and stopping the point on the boundary.
parameter(maxrefl = 10)
```

```
! Read all parameters from file.
call get_command_argument(1,param_file)
call get_command_argument(2,filename)

call read_inputs_mc(param_file,aratio,q,Pe,nGates,x0n,x0width,y0,z0,save_hist,n_bins,save_h
                        use_external_ic,ic_file,tstep_type,dt,dtmax,Tfinal,ntt,other_file,

if (filename=="") then
     out_msg = 'missing_args'
     call duct_mc_messages(out_msg,nz)
     go to 1234
end if

! Set the dimensions of the thing.
a = 1.0d0
b = a/aratio

   ! Assign the number of bins in each direction for ptwise stats.
if (n_bins .eq. 0) then
     nby = 0
     nbz = 0

     dby = 0.0d0
     dbz = 0.0d0
else
     nby = n_bins
     nbz = ceiling(n_bins/aratio) ! Could also make this the same as nby.

     dby = (2.0d0*a)/nby
     dbz = (2.0d0*b)/nbz
end if

nbx = nby

! Generate the target times; times at which output is saved.
! Internal timestepping is created after.
!

call generate_target_times(dt,tstep_type,Tfinal,other_file)

! Get the value of nt before allocating arrays.
call correct_tstep_info(ntt,nt,target_times,dtmax)

! Initialize the Mersenne Twister RNG with seed read in from the
! input files.

call sgrnd(mt_seed)
```

```
if (nGates .gt. 1) then
     ! Need to adjust this to make the spacing
     ! uniform taking into account the aspect ratio.

     ny = floor(dsqrt(dble(nGates)*aratio))+1
     nz = floor(dsqrt(dble(nGates)/aratio))+1
     !
     ! Because of the rejection method used to generate uniform points,
     ! each of ny,nz needs to be scaled up appropriately so that
     ! the number of points that are actually simulated is genuinely
     ! ~nGates, as input from the user's file.
     !
     ! The factor is the ratio of the circle to its circumscribing square.
     ny = floor(ny/dsqrt(datan(1.0d0)))+1
     nz = floor(nz/dsqrt(datan(1.0d0)))+1
else
     ny = 1
     nz = 1
end if

! If resolution is an issue, exit.
if ( (ny .lt. 7) .and. (nGates .gt. 1) ) then
     out_msg = 'resolution'
     call duct_mc_messages(out_msg,nz)
     go to 1234
end if

! Calculate nGates and nTot (total number of particles).
! In the ellipse this is done with two "sweeps."
! The number of points should be approximately ~(pi/4)*ny*nz.
! Sweep the grid and update the values for nGates and nTot,
! then allocate memory for X,Y,Z, then fill in the values.

call get_pts_in_ellipse(ny,nz,x0n,a,b,nGates,nTot)

! Internal time
allocate(t_hist(nt))

! Positions
allocate(X(nTot), Y(nTot), Z(nTot))

! Channel-averaged stats
allocate(means(ntt), vars(ntt), skews(ntt),kurts(ntt))

! Stats on Y slices (integrated across Z)
if (.not. (n_bins .eq. 0)) then
     allocate(means_sl(ntt,nby,nbz),vars_sl(ntt,nby,nbz),&
```

```
                skews_sl(ntt,nby,nbz),kurts_sl(ntt,nby,nbz))
end if

! If the 2d histogram (looking in the short direction) is desired,
! allocate.
if (save_hist2d) then
     allocate(hist2d(ntt,nbx,nby))
     allocate(hist2dcx(ntt,nbx))
     allocate(hist2dcy(ntt,nby))
end if

! Cross-sectionally averaged distribution
allocate(hist_centers(ntt,nhb),hist_heights(ntt,nhb))

! ------------------------------------------------------
! Initialize HDF with appropriate dataset, etc.

fname2 = trim(filename)

call hdf_create_file(fname2)

! We need to open the h5 file after hdf_create_file
! because the interface is "global" amongst all files
! containing the hdf5 module.

call h5open_f(h5error)

if (save_hist) then
     ! Set up dataspaces in the hdf file for:
     ! X, Y, Z.
     !
     ! Allocate memory for the memory buffers here, too.

     allocate(Xbuffer(buffer_len,nTot))
     allocate(Ybuffer(buffer_len,nTot))
     allocate(Zbuffer(buffer_len,nTot))

     call h5fopen_f(fname2, H5F_ACC_RDWR_F, file_id, h5error)

     data_dims(1) = ntt
     data_dims(2) = nTot
     rank = 2

     dsetname = "X"
     call h5screate_simple_f(rank, data_dims, dspace_id_X, h5error)
     call h5dcreate_f(file_id, dsetname, H5T_NATIVE_DOUBLE, dspace_id_X, &
                    dset_id_X, h5error)
     dsetname = "Y"
```

```
     call h5screate_simple_f(rank, data_dims, dspace_id_Y, h5error)
     call h5dcreate_f(file_id, dsetname, H5T_NATIVE_DOUBLE, dspace_id_Y, &
                    dset_id_Y, h5error)
     dsetname = "Z"
     call h5screate_simple_f(rank, data_dims, dspace_id_Z, h5error)
     call h5dcreate_f(file_id, dsetname, H5T_NATIVE_DOUBLE, dspace_id_Z, &
                    dset_id_Z, h5error)

end if

! ------------------------------------------------
!
! Set initial conditions and internal timestepping.
!

call set_initial_conds_ellipse_mc(ny,nz,x0n,a,b,nGates,nTot,X,Y,Z, &
                          y0,z0,x0width,t_warmup)

call generate_internal_timestepping(ntt,nt,target_times,t_hist,dtmax)

call print_parameters(aratio,q,Pe,nGates,nTot,y0,z0,save_hist,&
                          t_hist,dtmax,nt,ntt,mt_seed,geometry,use_external_ic)

! Save a history of time per iteration for predicting time to completion.
mde_ntt = nt-1
allocate(mde_dts(mde_ntt))

! --------------------------------
!
! Calculate the statistics of the initial condition.
!
inext = 1
tt_idx = 1

call accumulate_moments_2d(tt_idx,ntt,nTot,X,Y,Z,-a,a,-b,b,means,vars,skews,&
          kurts,nby,nbz,means_sl,vars_sl,skews_sl,kurts_sl)

call make_histogram(nTot,X,nhb,hist_centers(tt_idx,1:nhb),hist_heights(tt_idx,1:nhb))

if (save_hist2d) then
     call make_histogram2d(nTot,X,Y,nbx,nby,hist2dcx(tt_idx,1:nbx),&
               hist2dcy(tt_idx,1:nby), hist2d(tt_idx,1:nbx,1:nby))
end if

tt_idx = 2
next_tt = target_times(tt_idx)

! -----------
```

```
! Prepare the buffer to save position histories if requested.
! The subroutine is geometry independent once the initial conditions are set,
! so there's no need to make a "buffer_op_ellipse" subroutine.

bk = 0

if (save_hist) then
     call buffer_op_duct(bk,nTot,buffer_len,Xbuffer,Ybuffer,Zbuffer,&
                         X,Y,Z,ntt,inext,dset_id_X,dset_id_Y,dset_id_Z)
end if

! --------------
! Start the timestepping.

out_msg = 'simul_start'
call duct_mc_messages(out_msg,nz)

do kt=2,nt

     call system_clock(mde_t1,count_rate)    ! Time for progress.

     ! Push forward time.
     t = t_hist(kt)
     dt = t_hist(kt) - t_hist(kt-1)

     call apply_advdiff1_ellipse(nTot,X,Y,Z,Pe,dt,a,b, &
                         u_ellipse,impose_reflective_BC_ellipse,maxrefl)

     ! Check if we're at a target time. If we are,
     ! and calculate and save moments (and positions, if requested),
     ! then increment tt_idx and update next_tt.
     !

     if (t .eq. next_tt) then

          call accumulate_moments_2d(tt_idx,ntt,nTot,X,Y,Z,-a,a,-b,b,means,vars,skews,&
                    kurts,nby,nbz,means_sl,vars_sl,skews_sl,kurts_sl)

          call make_histogram(nTot,X,nhb,hist_centers(tt_idx,1:nhb),hist_heights(tt_idx,1:n

          if (save_hist2d) then
               call make_histogram2d(nTot,X,Y,nbx,nby,hist2dcx(tt_idx,1:nbx),&
                         hist2dcy(tt_idx,1:nby), hist2d(tt_idx,1:nbx,1:nby))
          end if

          !
          ! Write history if requested.
          !
```

```
          ! Need to buffer writes to the hard drive so that we don't lock up the
          ! computation with file opens/closes. Ideally the buffer should be as
          ! large as possible while fitting into RAM; modify the relevant parameter
          ! at the end of the variable definitions.
          !

          if (save_hist) then
               call buffer_op_duct(bk,nTot,buffer_len,Xbuffer,Ybuffer,Zbuffer,&
                              X,Y,Z,ntt,inext,dset_id_X,dset_id_Y,dset_id_Z)
          end if

          !
          ! Update the target time and array index.
          !
          if (next_tt .lt. Tfinal) then
               tt_idx = tt_idx + 1
               next_tt = target_times(tt_idx)
          end if

     end if

     ! Display percentage progress. The last argument as .true. should be used with gfortra
     ! (or any other compiler that supports "\b"), or .false. with ifort.

     call system_clock(mde_t2,count_rate)    ! Time in milliseconds

     mde_ntc = kt-1
     mde_dts(mde_ntc) = (mde_t2-mde_t1)/dble(count_rate)  ! Time in seconds

     call progress_meter(kt,nt,.true.)

end do

out_msg = 'simul_done'
call duct_mc_messages(out_msg,nz)


if (save_hist) then
! Write the remainder of the buffer, then close the file.

     rem = ntt-inext+1

     if (rem .gt. 0) then
          call hdf_write_to_open_2d_darray(ntt,nTot,inext,rem,Xbuffer(1:rem,1:nTot),dset_id
          call hdf_write_to_open_2d_darray(ntt,nTot,inext,rem,Ybuffer(1:rem,1:nTot),dset_id
          call hdf_write_to_open_2d_darray(ntt,nTot,inext,rem,Zbuffer(1:rem,1:nTot),dset_id
     end if
```

```
     call h5dclose_f(dset_id_X,h5error)
     call h5dclose_f(dset_id_Y,h5error)
     call h5dclose_f(dset_id_Z,h5error)
     call h5fclose_f(file_id,h5error)
end if

! Because of the nature of hdf5 mod for fortran,
! we close the interface here, since it gets
! re-opened in the calls below.

call h5close_f(h5error)

if (save_hist2d) then
         write(*,*) nbx,nby,shape(hist2dcx),shape(hist2dcy),shape(hist2d)
     arrayname = "hist2dcx"
     descr = "Array tracking bin centers in the x direction for hist2d"
     call hdf_add_2d_darray_to_file(ntt,nbx,hist2dcx,fname2,arrayname,descr)

     arrayname = "hist2dcy"
     descr = "Array tracking bin centers in the y direction for hist2d"
     call hdf_add_2d_darray_to_file(ntt,nby,hist2dcy,fname2,arrayname,descr)

     arrayname = "hist2d"
     descr = "Array tracking the density for hist2d"
     call hdf_add_3d_darray_to_file(ntt,nbx,nby,hist2d,fname2,arrayname,descr)
end if

! Save all the remaining arrays. It's a lot of fluff so it's been
! given its own subroutine.
!
! The ellipse and duct implementations are identical again here, so no use making another s

call save_the_rest_duct(fname2,geometry,ntt,target_times,means,vars,skews,kurts,nby,nbz,&
                        means_sl,vars_sl,skews_sl,kurts_sl,nhb,hist_centers,hist_heights,&
                        Pe,nTot,mt_seed,aratio,q,dtmax,t_warmup)


! ------
deallocate(X,Y,Z)

deallocate(means,vars,skews,kurts,target_times)

if (.not. (n_bins .eq. 0)) then
     deallocate(means_sl,vars_sl,skews_sl,kurts_sl)
end if

deallocate(hist_centers,hist_heights)
deallocate(t_hist)
```

```
     if (save_hist) then
          deallocate(Xbuffer,Ybuffer,Zbuffer)
     end if

     if (save_hist2d) then
          deallocate(hist2d,hist2dcx,hist2dcy)
     end if

     out_msg = 'done'
     call duct_mc_messages(out_msg,nz)

1234 continue

end program ellipse_mc
```

./monte/racetrack_mc.f90

```
program racetrack_mc
! Program to do Monte Carlo in a racetrack.

use HDF5
use mtmod
use mod_time
use mod_duration_estimator

implicit none

     ! Array sizes, parameters, local vars
     integer, parameter                              :: i64 = selected_int_kind(18)

     integer                                         :: nGates
     integer                                         :: nTot,nt,kt,ny,nz,tt_idx
     double precision                                :: Tfinal,dt,dtmax,Pe,aratio,q,next_tt
     double precision                                :: t

     integer(i64)                                    :: mt_seed

     integer                                         :: maxrefl

     ! Positions, position/statistic histories

     integer                                         :: n_bins,nbx,nby,nbz,nhb
     double precision                                :: a,b,dby,dbz,t_warmup
     double precision                                :: y0,z0

     double precision                                :: x0width ! Longitudinal width of initia
```

```
integer                                         :: x0n     ! number of discretization poi

! Stuff for 2d histogram looking into the short direction.
double precision, dimension(:,:,:), allocatable   :: hist2d
double precision, dimension(:,:), allocatable     :: hist2dcx, hist2dcy ! bin centers.

double precision, dimension(:), allocatable       :: X,Y,Z
double precision, dimension(:,:), allocatable     :: Xbuffer,Ybuffer,Zbuffer
integer                                           :: buffer_len,bk,inext,rem
double precision, dimension(:), allocatable       :: means,vars,skews,kurts,t_hist

double precision, dimension(:,:,:), allocatable   :: means_sl,vars_sl,skews_sl,kurts_sl
double precision, dimension(:,:), allocatable     :: hist_centers,hist_heights

! Type of geometry, only used to modify the output header.
character(len=1024)                               :: geometry

! i/o
character(len=1024)                               :: param_file,other_file,ic_file,filename
character(len=1024)                               :: out_msg

character(len=1024)                               :: arrayname,descr

! HDF
integer                                           :: rank,h5error
character(len=1024)                               :: fname2

character(len=1024)                               :: dsetname

! HDF variables.
integer(hid_t)                                    :: file_id
integer(hid_t)                                    :: dset_id_X,dset_id_Y,dset_id_Z
integer(hid_t)                                    :: dspace_id_X,dspace_id_Y,dspace_id_Z

integer(hsize_t), dimension(2)                    :: data_dims


! For saving position histories.
logical                                 :: save_hist,save_hist2d,use_external_ic


! References to functions that go in arguments.
external  :: impose_reflective_BC_racetrack, u_racetrack

! Parameters.
!

parameter(geometry = "racetrack")
```

```
! Buffer length, to reduce the number of writes
! onto the HDF files.
! Make this as large as possible to fit in RAM!
!
! 5*10**3 buff * 10**4 walks => ~1GB RAM
!
! RAM = kt*buffer*walks
!    ..............=> buffer = RAM/(k*walks)
!                      k = RAM/(buffer*walks)
!
! In our example kt = 1/(5*10**7).
!
parameter(buffer_len = 20)

! Number of bins when looking at the cross-sectionally averaged distribution.
parameter(nhb = 400)

! Maximum reflections applied before giving up and stopping the point on the boundary.
parameter(maxrefl = 10)

! Read all parameters from file.
call get_command_argument(1,param_file)
call get_command_argument(2,filename)


call read_inputs_mc(param_file,aratio,q,Pe,nGates,x0n,x0width,y0,z0,save_hist,&
                        n_bins,save_hist2d,t_warmup,use_external_ic,ic_file,tstep_type,&
                        dt,dtmax,Tfinal,ntt,other_file,mt_seed)

if (filename=="") then
     out_msg = 'missing_args'
     call duct_mc_messages(out_msg,nz)
     go to 1234
end if

! Set the region for collecting pointwise statistics.
a = 1.5d0
b = a/aratio

   ! Assign the number of bins in each direction for ptwise stats.
if (n_bins .eq. 0) then
     nby = 0
     nbz = 0

     dby = 0.0d0
     dbz = 0.0d0
else
```

```
     nby = n_bins
     nbz = ceiling(n_bins/aratio) ! Could also make this the same as nby.

     dby = (2.0d0*a)/nby
     dbz = (2.0d0*b)/nbz
end if

! For stats integrated through the z direction
nbx = nby
!
! Generate the target times; times at which output is saved.
! Internal timestepping is created after.
!

call generate_target_times(dt,tstep_type,Tfinal,other_file)

! Get the value of nt before allocating arrays.
call correct_tstep_info(ntt,nt,target_times,dtmax)

! Initialize the Mersenne Twister RNG with seed read in from the
! input files.

call sgrnd(mt_seed)

! Calculate ny,nz,nGates and nTot (total number of particles).
! In the racetrack, NEED TO DO EVERYTHING IN THIS SUBROUTINE:
!
! 1. Get an approximatation for the area to choose appropriate ny,nz,
!    so that the remaining in the interior is approximately the
!    input nGates.
! 2. Use boundary distance function bdistfun_rt(y,z,aratio,q)
!    to accept/reject points on the new grid.

nTot = nGates*x0n

! Internal time
allocate(t_hist(nt))

! Positions
allocate(X(nTot), Y(nTot), Z(nTot))

! Channel-averaged stats
allocate(means(ntt), vars(ntt), skews(ntt),kurts(ntt))

! Stats on Y slices (integrated across Z)
if (.not. (n_bins .eq. 0)) then
     allocate(means_sl(ntt,nby,nbz),vars_sl(ntt,nby,nbz),&
               skews_sl(ntt,nby,nbz),kurts_sl(ntt,nby,nbz))
```

```
end if

! If the 2d histogram (looking in the short direction) is desired,
! allocate.
if (save_hist2d) then
     allocate(hist2d(ntt,nbx,nby))
     allocate(hist2dcx(ntt,nbx))
     allocate(hist2dcy(ntt,nby))
end if

! Cross-sectionally averaged distribution
allocate(hist_centers(ntt,nhb),hist_heights(ntt,nhb))

! ------------------------------------------------------
! Initialize HDF with appropriate dataset, etc.

fname2 = trim(filename)

call hdf_create_file(fname2)

! We need to open the h5 file after hdf_create_file
! because the interface is "global" amongst all files
! containing the hdf5 module.

call h5open_f(h5error)

if (save_hist) then
     ! Set up dataspaces in the hdf file for:
     ! X, Y, Z.
     !
     ! Allocate memory for the memory buffers here, too.

     allocate(Xbuffer(buffer_len,nTot))
     allocate(Ybuffer(buffer_len,nTot))
     allocate(Zbuffer(buffer_len,nTot))

     call h5fopen_f(fname2, H5F_ACC_RDWR_F, file_id, h5error)

     data_dims(1) = ntt
     data_dims(2) = nTot
     rank = 2

     dsetname = "X"
     call h5screate_simple_f(rank, data_dims, dspace_id_X, h5error)
     call h5dcreate_f(file_id, dsetname, H5T_NATIVE_DOUBLE, dspace_id_X, &
                    dset_id_X, h5error)
     dsetname = "Y"
     call h5screate_simple_f(rank, data_dims, dspace_id_Y, h5error)
```

```
     call h5dcreate_f(file_id, dsetname, H5T_NATIVE_DOUBLE, dspace_id_Y, &
                    dset_id_Y, h5error)
     dsetname = "Z"
     call h5screate_simple_f(rank, data_dims, dspace_id_Z, h5error)
     call h5dcreate_f(file_id, dsetname, H5T_NATIVE_DOUBLE, dspace_id_Z, &
                    dset_id_Z, h5error)

end if

!
! ------------------------------------------------
!
! Set initial conditions and internal timestepping.
!

call set_initial_conds_racetrack_mc(x0n,aratio,q,nGates,nTot,X,Y,Z, &
                         y0,z0,x0width,t_warmup)

call generate_internal_timestepping(ntt,nt,target_times,t_hist,dtmax)

call print_parameters(aratio,q,Pe,nGates,nTot,y0,z0,save_hist,&
                         t_hist,dtmax,nt,ntt,mt_seed,geometry,use_external_ic)

! Save a history of time per iteration for predicting time to completion.
mde_ntt = nt-1
allocate(mde_dts(mde_ntt))

! --------------------------------
!
! Calculate the statistics of the initial condition.
!
inext = 1
tt_idx = 1

call accumulate_moments_2d(tt_idx,ntt,nTot,X,Y,Z,-a,a,-b,b,means,vars,skews,&
          kurts,nby,nbz,means_sl,vars_sl,skews_sl,kurts_sl)

call make_histogram(nTot,X,nhb,hist_centers(tt_idx,1:nhb),hist_heights(tt_idx,1:nhb))

if (save_hist2d) then
     call make_histogram2d(nTot,X,Y,nbx,nby,hist2dcx(tt_idx,1:nbx),&
               hist2dcy(tt_idx,1:nby), hist2d(tt_idx,1:nbx,1:nby))
end if

tt_idx = 2
next_tt = target_times(tt_idx)

! -----------
```

```
! Prepare the buffer to save position histories if requested.
! The subroutine is geometry independent once the initial conditions are set,
! so there's no need to make a "buffer_op_ellipse" subroutine.

bk = 0

if (save_hist) then
     call buffer_op_duct(bk,nTot,buffer_len,Xbuffer,Ybuffer,Zbuffer,&
                         X,Y,Z,ntt,inext,dset_id_X,dset_id_Y,dset_id_Z)
end if

! --------------
! Start the timestepping.

out_msg = 'simul_start'
call duct_mc_messages(out_msg,nz)

do kt=2,nt

     call system_clock(mde_t1,count_rate)    ! Time for progress.

     ! Push forward time.
     t = t_hist(kt)
     dt = t_hist(kt) - t_hist(kt-1)

     call apply_advdiff1_racetrack(nTot,X,Y,Z,Pe,dt,aratio,q, &
                         u_racetrack,impose_reflective_BC_racetrack,maxrefl)

     !
     ! Check if we're at a target time. If we are,
     ! and calculate and save moments (and positions, if requested),
     ! then increment tt_idx and update next_tt.
     !

     if (t .eq. next_tt) then

          call accumulate_moments_2d(tt_idx,ntt,nTot,X,Y,Z,-a,a,-b,b,means,vars,skews,&
                    kurts,nby,nbz,means_sl,vars_sl,skews_sl,kurts_sl)

          call make_histogram(nTot,X,nhb,hist_centers(tt_idx,1:nhb),hist_heights(tt_idx,1:n

          if (save_hist2d) then
               call make_histogram2d(nTot,X,Y,nbx,nby,hist2dcx(tt_idx,1:nbx),&
                         hist2dcy(tt_idx,1:nby), hist2d(tt_idx,1:nbx,1:nby))
          end if

          !
          ! Write history if requested.
```

```
            !
            ! Need to buffer writes to the hard drive so that we don't lock up the
            ! computation with file opens/closes. Ideally the buffer should be as
            ! large as possible while fitting into RAM; modify the relevant parameter
            ! at the end of the variable definitions.
            !

            if (save_hist) then
                 call buffer_op_duct(bk,nTot,buffer_len,Xbuffer,Ybuffer,Zbuffer,&
                                X,Y,Z,ntt,inext,dset_id_X,dset_id_Y,dset_id_Z)
            end if

            !
            ! Update the target time and array index.
            !
            if (next_tt .lt. Tfinal) then
                 tt_idx = tt_idx + 1
                 next_tt = target_times(tt_idx)
            end if

       end if

       ! Display percentage progress. The last argument as .true. should be used with gfortra
       ! (or any other compiler that supports "\b"), or .false. with ifort.

       call system_clock(mde_t2,count_rate)    ! Time in milliseconds

       mde_ntc = kt-1
       mde_dts(mde_ntc) = (mde_t2-mde_t1)/dble(count_rate)  ! Time in seconds

       call progress_meter(kt,nt,.true.)

  end do

  out_msg = 'simul_done'
  call duct_mc_messages(out_msg,nz)


  if (save_hist) then
  ! Write the remainder of the buffer, then close the file.

       rem = ntt-inext+1

       if (rem .gt. 0) then
            call hdf_write_to_open_2d_darray(ntt,nTot,inext,rem,Xbuffer(1:rem,1:nTot),dset_id
            call hdf_write_to_open_2d_darray(ntt,nTot,inext,rem,Ybuffer(1:rem,1:nTot),dset_id
            call hdf_write_to_open_2d_darray(ntt,nTot,inext,rem,Zbuffer(1:rem,1:nTot),dset_id
       end if
```

```
     call h5dclose_f(dset_id_X,h5error)
     call h5dclose_f(dset_id_Y,h5error)
     call h5dclose_f(dset_id_Z,h5error)
     call h5fclose_f(file_id,h5error)
end if

! Because of the nature of hdf5 mod for fortran,
! we close the interface here, since it gets
! re-opened in the calls below.

call h5close_f(h5error)

if (save_hist2d) then
      write(*,*) nbx,nby,shape(hist2dcx),shape(hist2dcy),shape(hist2d)
     arrayname = "hist2dcx"
     descr = "Array tracking bin centers in the x direction for hist2d"
     call hdf_add_2d_darray_to_file(ntt,nbx,hist2dcx,fname2,arrayname,descr)

     arrayname = "hist2dcy"
     descr = "Array tracking bin centers in the y direction for hist2d"
     call hdf_add_2d_darray_to_file(ntt,nby,hist2dcy,fname2,arrayname,descr)

     arrayname = "hist2d"
     descr = "Array tracking the density for hist2d"
     call hdf_add_3d_darray_to_file(ntt,nbx,nby,hist2d,fname2,arrayname,descr)
end if

! Save all the remaining arrays. It's a lot of fluff so it's been
! given its own subroutine.
!
!

call save_the_rest_duct(fname2,geometry,ntt,target_times,means,vars,skews,kurts,nby,nbz,&
                        means_sl,vars_sl,skews_sl,kurts_sl,nhb,hist_centers,hist_heights,&
                        Pe,nTot,mt_seed,aratio,q,dtmax,t_warmup)

! ------
deallocate(X,Y,Z)

deallocate(means,vars,skews,kurts,target_times)

if (.not. (n_bins .eq. 0)) then
     deallocate(means_sl,vars_sl,skews_sl,kurts_sl)
end if

deallocate(hist_centers,hist_heights)
deallocate(t_hist)
```

```
     if (save_hist) then
          deallocate(Xbuffer,Ybuffer,Zbuffer)
     end if

     if (save_hist2d) then
          deallocate(hist2d,hist2dcx,hist2dcy)
     end if

     out_msg = 'done'
     call duct_mc_messages(out_msg,nz)

1234 continue

end program racetrack_mc
```

./monte/triangle_mc.f90

```
program triangle_mc
! Program to do Monte Carlo in a triangle.

use HDF5
use mtmod
use mod_time
use mod_duration_estimator
use mod_triangle_bdry

implicit none

     ! Array sizes, parameters, local vars
     integer, parameter                              :: i64 = selected_int_kind(18)

     integer                                         :: nGates
     integer                                         :: nTot,nt,kt,ny,nz,tt_idx
     double precision                                :: Tfinal,dt,dtmax,Pe,aratio,q,next_tt
     double precision                                :: t

     integer(i64)                                    :: mt_seed

     ! Positions, position/statistic histories
     integer                                         :: n_bins,nbx,nby,nbz,nhb
     double precision                                :: a,b,dby,dbz,t_warmup
     double precision                                :: y0,z0

     double precision                                :: x0width
     integer                                         :: x0n
```

```
! Stuff for 2d histogram looking into the short direction.
double precision, dimension(:,:,:), allocatable   :: hist2d
double precision, dimension(:,:), allocatable     :: hist2dcx, hist2dcy ! bin centers.

double precision, dimension(:), allocatable       :: X,Y,Z
double precision, dimension(:,:), allocatable     :: Xbuffer,Ybuffer,Zbuffer
integer                                           :: buffer_len,bk,inext,rem
double precision, dimension(:), allocatable       :: means,vars,skews,kurts,t_hist

double precision, dimension(:,:,:), allocatable   :: means_sl,vars_sl,skews_sl,kurts_sl
double precision, dimension(:,:), allocatable     :: hist_centers,hist_heights


! Type of geometry, only used to modify the output header.
character(len=1024)                               :: geometry

! i/o
character(len=1024)                               :: param_file,other_file,&
                                                       filename,tstep_type,ic_file


character(len=1024)                               :: out_msg

character(len=1024)                               :: arrayname,descr

! HDF
integer                                           :: rank,h5error
character(len=1024)                               :: fname2

character(len=1024)                               :: dsetname

! HDF variables.
integer(hid_t)                                    :: file_id
integer(hid_t)                                    :: dset_id_X,dset_id_Y,dset_id_Z
integer(hid_t)                                    :: dspace_id_X,dspace_id_Y,dspace_id_Z

integer(hsize_t), dimension(2)                    :: data_dims

! Flags to save position histories and read IC from a file.
logical                                :: save_hist,save_hist2d,use_external_ic
logical check_ic_duct

! References to functions that go in arguments.
external  :: impose_reflective_BC_polygon, u_triangle

! Parameters.
!

parameter(geometry = "triangle")
```

```
! Buffer length, to reduce the number of writes
! onto the HDF files.
! Make this as large as possible to fit in RAM!
!
! 5*10**3 buff * 10**4 walks => ~1GB RAM
!
! RAM = kt*buffer*walks
!    ..............=> buffer = RAM/(k*walks)
!                      k = RAM/(buffer*walks)
!
! In our example kt = 1/(5*10**7).
!
parameter(buffer_len = 20)

! Number of bins when looking at the cross-sectionally averaged distribution.
parameter(nhb = 400)

! -------------------------------------------------------


! Read all parameters from file.
call get_command_argument(1,param_file)
call get_command_argument(2,filename)

call read_inputs_mc(param_file,aratio,q,Pe,nGates,x0n,x0width,y0,z0,save_hist,&
                        n_bins,save_hist2d,t_warmup,use_external_ic,ic_file,tstep_type,&
                        dt,dtmax,Tfinal,ntt,other_file,mt_seed)

! Ignore whatever was input and replace.
aratio = 1.0d0

if (filename=="") then
     out_msg = 'missing_args'
     call duct_mc_messages(out_msg,nz)
     go to 1234
end if

! Set the dimensions of the thing.
a = 1.0d0

   ! Assign the number of bins in each direction for ptwise stats.
if (n_bins .eq. 0) then
     nby = 0
     nbz = 0

     dby = 0.0d0
     dbz = 0.0d0
```

```
else
     nby = n_bins
     nbz = nby

     dby = (2.0d0*a)/nby
     dbz = dby
end if

! For stats integrated through the z direction
nbx = nby
!
! Generate the target times; times at which output is saved.
! Internal timestepping is created after.
!

call generate_target_times(dt,tstep_type,Tfinal,other_file)

! Get the value of nt before allocating arrays.
call correct_tstep_info(ntt,nt,target_times,dtmax)

! Initialize the Mersenne Twister RNG with seed read in from the
! input files.

call sgrnd(mt_seed)

if (nGates .gt. 1) then
     !
     ! Because of the rejection method used to generate uniform points,
     ! each of ny,nz needs to be scaled up appropriately so that
     ! the number of points that are actually simulated is genuinely
     ! ~nGates, as input from the user's file.
     !
     ! The factor is the ratio of the triangle to its circumscribing square.
     !

     nGates = nGates * (12.0d0/dsqrt(27.0d0))

     ny = floor(dsqrt(dble(nGates)))+1
     nz = floor(dsqrt(dble(nGates)))+1

else
     ny = 1
     nz = 1
end if

! If resolution is an issue, exit.
if ( (ny .lt. 7) .and. (nGates .gt. 1) ) then
     out_msg = 'resolution'
```

```
     call duct_mc_messages(out_msg,nz)
     go to 1234
end if

! Calculate nGates and nTot (total number of particles).
! In the triangle this is done with two "sweeps."
! The number of points should be approximately ~(pi/4)*ny*nz.
! Sweep the grid and update the values for nGates and nTot,
! then allocate memory for X,Y,Z, then fill in the values.

call get_pts_in_triangle(ny,nz,x0n,a,nGates,nTot,nl,lls)

! Internal time
allocate(t_hist(nt))

! Positions
allocate(X(nTot), Y(nTot), Z(nTot))

! Channel-averaged stats
allocate(means(ntt), vars(ntt), skews(ntt),kurts(ntt))

! Stats on Y slices (integrated across Z)
if (.not. (n_bins .eq. 0)) then
     allocate(means_sl(ntt,nby,nbz),vars_sl(ntt,nby,nbz),&
               skews_sl(ntt,nby,nbz),kurts_sl(ntt,nby,nbz))
end if

! If the 2d histogram (looking in the short direction) is desired,
! allocate.
if (save_hist2d) then
     allocate(hist2d(ntt,nbx,nby))
     allocate(hist2dcx(ntt,nbx))
     allocate(hist2dcy(ntt,nby))
end if

! Cross-sectionally averaged distribution
allocate(hist_centers(ntt,nhb),hist_heights(ntt,nhb))

! ------------------------------------------------------
! Initialize HDF with appropriate dataset, etc.

fname2 = trim(filename)

call hdf_create_file(fname2)

! We need to open the h5 file after hdf_create_file
! because the interface is "global" amongst all files
! containing the hdf5 module.
```

```
call h5open_f(h5error)

if (save_hist) then
     ! Set up dataspaces in the hdf file for:
     ! X, Y, Z.
     !
     ! Allocate memory for the memory buffers here, too.

     allocate(Xbuffer(buffer_len,nTot))
     allocate(Ybuffer(buffer_len,nTot))
     allocate(Zbuffer(buffer_len,nTot))

     call h5fopen_f(fname2, H5F_ACC_RDWR_F, file_id, h5error)

     data_dims(1) = ntt
     data_dims(2) = nTot
     rank = 2

     dsetname = "X"
     call h5screate_simple_f(rank, data_dims, dspace_id_X, h5error)
     call h5dcreate_f(file_id, dsetname, H5T_NATIVE_DOUBLE, dspace_id_X, &
                    dset_id_X, h5error)
     dsetname = "Y"
     call h5screate_simple_f(rank, data_dims, dspace_id_Y, h5error)
     call h5dcreate_f(file_id, dsetname, H5T_NATIVE_DOUBLE, dspace_id_Y, &
                    dset_id_Y, h5error)
     dsetname = "Z"
     call h5screate_simple_f(rank, data_dims, dspace_id_Z, h5error)
     call h5dcreate_f(file_id, dsetname, H5T_NATIVE_DOUBLE, dspace_id_Z, &
                    dset_id_Z, h5error)

end if

!
! ------------------------------------------------
!
! Set initial conditions and internal timestepping.
!

call set_initial_conds_triangle_mc(ny,nz,x0n,a,nGates,nTot,X,Y,Z, &
                         y0,z0,x0width,t_warmup,use_external_ic,ic_file,nl,lls)

call generate_internal_timestepping(ntt,nt,target_times,t_hist,dtmax)

call print_parameters(aratio,q,Pe,nGates,nTot,y0,z0,save_hist,&
                         t_hist,dtmax,nt,ntt,mt_seed,geometry,use_external_ic)
```

```
! Save a history of time per iteration for predicting time to completion.
mde_ntt = nt-1
allocate(mde_dts(mde_ntt))

! -------------------------------
!
! Calculate the statistics of the initial condition.
!
inext = 1
tt_idx = 1

call accumulate_moments_2d(tt_idx,ntt,nTot,X,Y,Z, &
          -1.0d0,-1.0d0+2*a*dsqrt(3.0d0),-a*dsqrt(3.0d0),a*dsqrt(3.0d0), &
          means,vars,skews,kurts,nby,nbz,means_sl,vars_sl,skews_sl,kurts_sl)

call make_histogram(nTot,X,nhb,hist_centers(tt_idx,1:nhb),hist_heights(tt_idx,1:nhb))

if (save_hist2d) then
     call make_histogram2d(nTot,X,Y,nbx,nby,hist2dcx(tt_idx,1:nbx),&
               hist2dcy(tt_idx,1:nby), hist2d(tt_idx,1:nbx,1:nby))
end if

tt_idx = 2
next_tt = target_times(tt_idx)

! -----------
! Prepare the buffer to save position histories if requested.
! The subroutine is geometry independent once the initial conditions are set,
! so there's no need to make a "buffer_op_ellipse" subroutine.

bk = 0

if (save_hist) then
     call buffer_op_duct(bk,nTot,buffer_len,Xbuffer,Ybuffer,Zbuffer,&
                         X,Y,Z,ntt,inext,dset_id_X,dset_id_Y,dset_id_Z)
end if

! --------------
! Start the timestepping.

out_msg = 'simul_start'
call duct_mc_messages(out_msg,nz)

do kt=2,nt

     call system_clock(mde_t1,count_rate)    ! Time for progress.

     ! Push forward time.
```

```
t = t_hist(kt)
dt = t_hist(kt) - t_hist(kt-1)

call apply_advdiff1_triangle(nTot,X,Y,Z,Pe,dt,a, &
                    u_triangle,impose_reflective_BC_polygon,nl,lls)

!
! Check if we're at a target time. If we are,
! and calculate and save moments (and positions, if requested),
! then increment tt_idx and update next_tt.
!

if (t .eq. next_tt) then

     call accumulate_moments_2d(tt_idx,ntt,nTot,X,Y,Z, &
               -1.0d0,-1.0d0+a*3.0d0,-a*dsqrt(3.0d0),a*dsqrt(3.0d0), &
               means,vars,skews,kurts,nby,nbz,means_sl,vars_sl,skews_sl,kurts_sl)

     call make_histogram(nTot,X,nhb,hist_centers(tt_idx,1:nhb),hist_heights(tt_idx,1:n

     if (save_hist2d) then
          call make_histogram2d(nTot,X,Y,nbx,nby,hist2dcx(tt_idx,1:nbx),&
                    hist2dcy(tt_idx,1:nby), hist2d(tt_idx,1:nbx,1:nby))
     end if

     !
     ! Write history if requested.
     !
     ! Need to buffer writes to the hard drive so that we don't lock up the
     ! computation with file opens/closes. Ideally the buffer should be as
     ! large as possible while fitting into RAM; modify the relevant parameter
     ! at the end of the variable definitions.
     !

     if (save_hist) then
          call buffer_op_duct(bk,nTot,buffer_len,Xbuffer,Ybuffer,Zbuffer,&
                         X,Y,Z,ntt,inext,dset_id_X,dset_id_Y,dset_id_Z)
     end if

     !
     ! Update the target time and array index.
     !
     if (next_tt .lt. Tfinal) then
          tt_idx = tt_idx + 1
          next_tt = target_times(tt_idx)
     end if

end if
```

```
      ! Display percentage progress. The last argument as .true. should be used with gfortra
      ! (or any other compiler that supports "\b"), or .false. with ifort.

      call system_clock(mde_t2,count_rate)    ! Time in milliseconds

      mde_ntc = kt-1
      mde_dts(mde_ntc) = (mde_t2-mde_t1)/dble(count_rate)  ! Time in seconds

      call progress_meter(kt,nt,.true.)

end do

out_msg = 'simul_done'
call duct_mc_messages(out_msg,nz)


if (save_hist) then
! Write the remainder of the buffer, then close the file.

      rem = ntt-inext+1

      if (rem .gt. 0) then
           call hdf_write_to_open_2d_darray(ntt,nTot,inext,rem,Xbuffer(1:rem,1:nTot),dset_id
           call hdf_write_to_open_2d_darray(ntt,nTot,inext,rem,Ybuffer(1:rem,1:nTot),dset_id
           call hdf_write_to_open_2d_darray(ntt,nTot,inext,rem,Zbuffer(1:rem,1:nTot),dset_id
      end if

      call h5dclose_f(dset_id_X,h5error)
      call h5dclose_f(dset_id_Y,h5error)
      call h5dclose_f(dset_id_Z,h5error)
      call h5fclose_f(file_id,h5error)
end if

! Because of the nature of hdf5 mod for fortran,
! we close the interface here, since it gets
! re-opened in the calls below.

call h5close_f(h5error)

if (save_hist2d) then
       write(*,*) nbx,nby,shape(hist2dcx),shape(hist2dcy),shape(hist2d)
      arrayname = "hist2dcx"
      descr = "Array tracking bin centers in the x direction for hist2d"
      call hdf_add_2d_darray_to_file(ntt,nbx,hist2dcx,fname2,arrayname,descr)

      arrayname = "hist2dcy"
      descr = "Array tracking bin centers in the y direction for hist2d"
```

```
          call hdf_add_2d_darray_to_file(ntt,nby,hist2dcy,fname2,arrayname,descr)

          arrayname = "hist2d"
          descr = "Array tracking the density for hist2d"
          call hdf_add_3d_darray_to_file(ntt,nbx,nby,hist2d,fname2,arrayname,descr)
     end if

     ! Save all the remaining arrays. It's a lot of fluff so it's been
     ! given its own subroutine.
     !

     call save_the_rest_duct(fname2,geometry,ntt,target_times,means,vars,skews,kurts,nby,nbz,&
                             means_sl,vars_sl,skews_sl,kurts_sl,nhb,hist_centers,hist_heights,&
                             Pe,nTot,mt_seed,aratio,q,dtmax,t_warmup)


     ! ------
     deallocate(X,Y,Z)

     deallocate(means,vars,skews,kurts,target_times)

     if (.not. (n_bins .eq. 0)) then
          deallocate(means_sl,vars_sl,skews_sl,kurts_sl)
     end if

     deallocate(hist_centers,hist_heights)
     deallocate(t_hist)

     if (save_hist) then
          deallocate(Xbuffer,Ybuffer,Zbuffer)
     end if

     out_msg = 'done'
     call duct_mc_messages(out_msg,nz)

1234 continue

end program triangle_mc
```

```
 ./utils/

 ./utils/buffer_op_channel.f90

subroutine buffer_op_channel(bk,nTot,buffer_len,Xbuffer,Ybuffer,&
                             X,Y,tsteps,inext,dset_id_X,dset_id_Y)
! The basic buffered write operation.

use HDF5
implicit none

     integer, intent(in)                                                  :: nTot,tsteps,buffer
     integer, intent(inout)                                               :: bk,inext
     integer(hid_t), intent(inout)                                        :: dset_id_X,dset_id_

     double precision, dimension(1:nTot), intent(in)                      :: X,Y
     double precision, dimension(1:buffer_len,1:nTot), intent(inout)      :: Xbuffer,Ybuffer


     ! We need to reshape X and Y to put them in the buffer; it doesn't
     ! care about nRounds.
     !
     ! If we really do, each round will be packed in blocks size ng;
     ! the first from indices 1,...,nGates, the second round nGates+1,..,2*nGates, etc.

     bk = bk + 1

     Xbuffer(bk,:) = X
     Ybuffer(bk,:) = Y

     ! If we've hit the end of the buffer, write it to the appropriate
     ! location in the h5 file, and "reset" the buffer (by setting bk=0).
     if (bk .eq. buffer_len) then

          call hdf_write_to_open_2d_darray(tsteps,nTot,inext,buffer_len,Xbuffer,dset_id_X)

          call hdf_write_to_open_2d_darray(tsteps,nTot,inext,buffer_len,Ybuffer,dset_id_Y)

          bk = 0
          inext = inext + buffer_len
     end if


end subroutine buffer_op_channel
```

./utils/buffer_op_duct.f90

```
subroutine buffer_op_duct(bk,nTot,buffer_len,Xbuffer,Ybuffer,Zbuffer,&
                              X,Y,Z,tsteps,inext,dset_id_X,dset_id_Y,dset_id_Z)
! The basic buffered write operation.
! Saves the most recent buffer_len timesteps in an array
! before writing to the .h5 file (otherwise file i/o dominates computation time).

use HDF5
implicit none

     integer, intent(in)                                                :: nTot,tsteps,buffer_len
     integer, intent(inout)                                             :: bk,inext
     integer(hid_t), intent(inout)                                      :: dset_id_X,dset_id_Y,dse

     double precision, dimension(1:nTot), intent(in)                    :: X,Y,Z
     double precision, dimension(1:buffer_len,1:nTot), intent(inout)    :: Xbuffer,Ybuffer,Zbuffer

     bk = bk + 1

     Xbuffer(bk,:) = X
     Ybuffer(bk,:) = Y
     Zbuffer(bk,:) = Z

     ! If we've hit the end of the buffer, write it to the appropriate
     ! location in the h5 file, and "reset" the buffer (by setting bk=0).
     if (bk .eq. buffer_len) then
          call hdf_write_to_open_2d_darray(tsteps,nTot,inext,buffer_len,Xbuffer,dset_id_X)
          call hdf_write_to_open_2d_darray(tsteps,nTot,inext,buffer_len,Ybuffer,dset_id_Y)
          call hdf_write_to_open_2d_darray(tsteps,nTot,inext,buffer_len,Zbuffer,dset_id_Z)

          bk = 0
          inext = inext + buffer_len
     end if

end subroutine buffer_op_duct
```

./utils/channel_mc_messages.f90

```
subroutine channel_mc_messages(msg)
! Is this bad practice?
implicit none

     character(len=1024), intent(in)    :: msg

     character(len=1024)                :: missing_args,simul_start,simul_done,done
```

```
     parameter(missing_args = 'missing_args')
     parameter(simul_start = 'simul_start')
     parameter(simul_done = 'simul_done')
     parameter(done = 'done')

     if (msg .eq. missing_args) then
          write(*,*) "You must specify the name of an output file in the third argument. Exiting
     else if (msg .eq. simul_start) then
          write(*,"(A14)",advance="no") "Simulating... "
          write(*,"(I3,A1)",advance="no") 0,"%"
     else if (msg .eq. simul_done) then
          write(*,"(A8)") "\b\b\b done."
          write(*,"(A19)",advance="no") "Writing to file... "
     else if (msg .eq. done) then
          write(*,*) " done."
     end if

end subroutine channel_mc_messages


 ./utils/check_ic_channel.f90

logical function check_ic_channel(nTot,Y,a)
! Checks that the initial data is contained within the cross section.
implicit none

     integer, intent(in)                              :: nTot
     double precision, dimension(nTot), intent(in)    :: Y
     double precision, intent(in)                     :: a

     integer                                          :: i
     double precision                                 :: ymin,ymax


     ymin = minval(Y)
     ymax = maxval(Y)

     if ((ymin .lt. -a) .or. (ymax .gt. a)) then
          check_ic_channel = .false.
          write(*,*) ymin,ymax,a
     else
          check_ic_channel = .true.
     end if

end function check_ic_channel
```

./utils/check_ic_duct.f90

```
logical function check_ic_duct(nTot,Y,Z,a,b)
! Checks that the initial data is contained within the cross section.
implicit none

     integer, intent(in)                                :: nTot
     double precision, dimension(nTot), intent(in)      :: Y,Z
     double precision, intent(in)                       :: a,b

     integer                                            :: i
     double precision                                   :: ymin,ymax,zmin,zmax


     ymin = minval(Y)
     ymax = maxval(Y)
     zmin = minval(Z)
     zmax = maxval(Z)

     if ((ymin .lt. -a) .or. (ymax .gt. a) .or. (zmin .lt. -b) .or. (zmax .gt. b)) then
          check_ic_duct = .false.
          write(*,*) ymin,ymax,a
          write(*,*) zmin,zmax,b
     else
          check_ic_duct = .true.
     end if

end function check_ic_duct
```

./utils/correct_tstep_info.f90

```
subroutine correct_tstep_info(ntt,nt,target_times,dtmax)
! This is a 'dry run' version of generate_internal_timestepping, which
! calculates the internal time array size, nt.
!
! ntt is the number of target time points which will be saved to file.
! Essentially, the target times will grow exponentially, and the internal
! timestepping will prevent timesteps from exceeding dtmax.
!
! The resulting nt is an _upper bound_ for the number of
! internal timesteps needed.

implicit none

     integer, intent(in)                                :: ntt
     integer, intent(inout)                             :: nt
     double precision, intent(in)                       :: dtmax
```

```
     double precision, dimension(1:ntt), intent(in)    :: target_times

     double precision                   :: dist,tnext,tcurr
     integer                            :: k,tt_idx,idx,i


     idx = 0
     do i=2,ntt
          tcurr = target_times(i-1)
          tnext = target_times(i)
          do while (tcurr .lt. tnext)
               idx = idx + 1
               tcurr = tcurr + dtmax
          end do
     end do

     idx = idx + 1

     nt = idx

end subroutine correct_tstep_info


 ./utils/duct_mc_messages.f90

subroutine duct_mc_messages(msg,ny)
! Is this bad practice?
implicit none

     character(len=1024), intent(in)    :: msg
     integer, intent(in)                :: ny

     character(len=1024)                :: missing_args,resolution,&
                                           simul_start,simul_done,done

     parameter(missing_args = 'missing_args')
     parameter(resolution = 'resolution')
     parameter(simul_start = 'simul_start')
     parameter(simul_done = 'simul_done')
     parameter(done = 'done')

     if (msg .eq. missing_args) then
          write(*,*) "You must specify the name of an output file in the third argument. Exiting
     else if (msg .eq. resolution) then
          write(*,*) ""
          write(*,*) "You need to specify a larger number of points to accurately"
          write(*,*) "resolve the y direction for your aspect ratio. Current rule"
          write(*,*) "of thumb is that you need nGates > 7/sqrt(aratio)."
```

```
          write(*,*)
          write(*,*) "Currently, ny=",ny,"."
          write(*,*)
          write(*,*) "Exiting."
          write(*,*) ""
     else if (msg .eq. simul_start) then
          write(*,"(A14)",advance="no") "Simulating... "
          write(*,"(I3,A1)",advance="no") 0,"%"
     else if (msg .eq. simul_done) then
          write(*,"(A8)") "\b\b\bdone."
          write(*,"(A19)",advance="no") "Writing to file... "
     else if (msg .eq. done) then
          write(*,"(A5)") " done."
     end if

end subroutine duct_mc_messages


 ./utils/findcond.f90

subroutine findcond(n,b,q,aptr)
!
! Given a boolean array b, dimension(n),
! outputs an array aptr where the
! first q elements are integer pointers
! to the elements of b which are .true.
!

implicit none
     integer, intent(in)                :: n
     logical, dimension(n), intent(in)  :: b

     integer, intent(out)               :: q
     integer, dimension(n), intent(out) :: aptr

     integer                            :: i

     q = 0

     do i=1,n
          if (b(i)) then
               q = q+1
               aptr(q) = i
          end if
     end do

end subroutine findcond
```

./utils/generate_internal_timestepping.f90

```
subroutine generate_internal_timestepping(ntt,nt,target_times,t_hist,dtmax)
!
! Generate the array of internal time values based on dtmax and target_times.
!

implicit none

     integer, intent(in)                                :: ntt
     integer, intent(inout)                             :: nt
     double precision, dimension(1:ntt), intent(in)     :: target_times
     double precision, dimension(1:nt), intent(inout)   :: t_hist
     double precision, intent(in)                       :: dtmax

     integer                                            :: idx,i
     double precision                                   :: tcurr,tnext

     idx = 0
     do i=2,ntt
          tcurr = target_times(i-1)
          tnext = target_times(i)
          do while (tcurr .lt. tnext)
               idx = idx + 1
               t_hist(idx) = tcurr
               tcurr = tcurr + dtmax
          end do
     end do

     idx = idx + 1
     t_hist(idx) = target_times(ntt)

end subroutine generate_internal_timestepping
```

./utils/generate_target_times.f90

```
subroutine generate_target_times(tmin,tstep_type,Tfinal,other_file)
! From the input options, fill in an array
! target_times which will be the times on which
! output data is saved.

use mod_readbuff    ! For passing an unallocated array between here and the subroutine
                    ! hdf_read_1d_darray().

use mod_time     ! For time-related variables and arrays.

implicit none
```

```
double precision, intent(inout)                    :: Tfinal
double precision, intent(in)                       :: tmin
character(len=1024), intent(in)                    :: tstep_type,other_file


character(len=1024)                                :: expo,unif,supplied,stt
integer                                            :: tt_idx
double precision                                   :: kscale,dt

logical                                            :: flag

parameter(expo='expo',unif='unif',supplied="supplied",stt="target_times")

if (tstep_type .eq. unif) then
     ! Uniform timestepping using the specified tmin and tfinal with
     ! ntt timesteps.
     allocate(target_times(ntt))
     target_times(1) = 0.0d0

     dt = (Tfinal-tmin)/dble(ntt-1)
     ! Generate the target times
     do tt_idx=2,ntt
          target_times(tt_idx) = tmin + dt*(tt_idx-1)
     end do

else if (tstep_type .eq. expo) then
     ! Exponential timestepping; these are uniformly spaced in a log scale.
     ! This is the usual setting for our purposes.

     allocate(target_times(ntt))
     target_times(1) = 0.0d0

     target_times(2) = tmin
     kscale = dble(Tfinal/tmin)**(1.0d0/dble(ntt-2))

     do tt_idx=3,ntt-1
          target_times(tt_idx) = kscale*target_times(tt_idx-1)
     end do

     target_times(ntt) = Tfinal

else if (tstep_type .eq. supplied) then
     ! User has supplied their own timesteps in the specified h5 file.
     ! Override all other settings and use them.

     call hdf_read_1d_darray(ntt,other_file,stt)
```

```
          flag = (.not. (readbuff_double(1) .eq. 0.0d0))
          if (flag) then
               ntt = ntt + 1
          end if

          allocate(target_times(ntt))

          if (flag) then
               target_times(1) = 0.0d0
               target_times(2:ntt) = readbuff_double
          else
               target_times = readbuff_double
          end if

          Tfinal = target_times(ntt)

          deallocate(readbuff_double)

     else
          write(*,*) "Unrecognized tstep_type. Use either ''unif'', ''expo'', or ''supplied''."
     end if

     Tfinal = target_times(ntt)

end subroutine generate_target_times
```

./utils/get_pts_in_ellipse.f90

```
subroutine get_pts_in_ellipse(ny,nz,x0n,a,b,nGates,nTot)
! In the ellipse, the technique to set the initial condition is to proceed with
! uniform spacing as if we were in the rectangle, then exclude points that lie
! outside the circle. This leaves an issue of not knowing exactly how many
! points will be remaining. This function is a trimmed down version of the
! set_initial_conditions_ellipse where only the *number* of points in the domain
! is counted.
!
! Could probably be done in a single call if I was clever. But this isn't a bottleneck
! in computations.

implicit none

     integer, intent(in)                                    :: ny,nz,x0n
     double precision, intent(in)                           :: a,b
     integer, intent(out)                                   :: nGates,nTot


     integer                                                :: idx,iy,iz
```

```
     double precision                                        :: hy,hz,dist


     ! Sample points in the circumscribing square; throw out any that
     ! lie outside the circle.

     hy = 2.0d0*a/dble(ny-1)
     hz = 2.0d0*b/dble(nz-1)
     idx = 0

     if (nGates .gt. 1) then
          do iz=0,nz-1
               do iy=0,ny-1
                    dist = ((-a + iy*hy)**2)/(a**2) + ((-b + iz*hz)**2)/(b**2)

                    if (dist .le. 1.0d0) then
                         idx = idx + 1
                    end if
               end do
          end do

          nGates = idx
     else
          ! Nothing gets changed.
          nGates = 1
     end if

     nTot = nGates*x0n

end subroutine get_pts_in_ellipse


 ./utils/get_pts_in_triangle.f90

subroutine get_pts_in_triangle(ny,nz,x0n,a,nGates,nTot,nl,lls)
! In the triangle, the technique to set the initial condition is to proceed with
! uniform spacing as if we were in the rectangle, then exclude points that lie
! outside the triangle. This leaves an issue of not knowing exactly how many
! points will be remaining. This function is a trimmed down version of the
! set_initial_conditions_triangle where only the *number* of points in the domain
! is counted.
!
! Could probably be done in a single call if I was clever. But this isn't a bottleneck
! in computations.

implicit none

     integer, intent(in)                                     :: ny,nz,x0n,nl
```

```
      double precision, intent(in)                              :: a
      double precision, dimension(nl,3), intent(in)             :: lls
      integer, intent(out)                                      :: nGates,nTot


      integer                                                   :: idx,iy,iz
      double precision                                          :: hy,hz,rl,rb
      double precision, dimension(3)                            :: tempv
      double precision, dimension(nl)                           :: bvals


      ! Sample points in the circumscribing square; throw out any that
      ! lie outside the triangle.

      rl = -a*dsqrt(3.0d0)
      rb = -1.0d0

      hz = 2.0d0*a*dsqrt(3.0d0)/(nz-1)
      hy = 2.0d0*a*dsqrt(3.0d0)/(ny-1)
      idx = 0

      if (nGates .gt. 1) then
           do iz=0,nz-1
                do iy=0,ny-1

                     tempv(1) = 1.0d0
                     tempv(2) = rb + iy*hy
                     tempv(3) = rl + iz*hz

                     call matvec(nl,3,lls,tempv,bvals)

                     if (all(bvals .ge. 0.0d0)) then
                          idx = idx + 1
                     end if
                end do
           end do

           nGates = idx
      else
           ! Nothing gets changed.
           nGates = 1
      end if

      nTot = nGates*x0n

!     write(*,*) nTot
end subroutine get_pts_in_triangle
```

./utils/get_racetrack_area.f90

```
subroutine get_racetrack_area(q,aratio,area)
! Calculates the area in the racetrack with the given parameters.
! Simple Monte Carlo method with rejection.

use mtmod

implicit none
     double precision, intent(in)       :: q,aratio
     double precision, intent(out)      :: area

     integer, parameter                 :: ntot = 10**9
     integer                            :: nin,i
     double precision                   :: area_rec,yl,yr,zl,zr,my,mz,y,z


     double precision bdistfun_rt
     ! ---------------------

     yl = -1.3d0
     yr = 1.3d0

     zl = yl/aratio
     zr = yr/aratio

     my = yr-yl
     mz = zr-zl

     area_rec = my*mz

     do i=1,ntot
          ! Uniform random
          y = yl + my*grnd()
          z = zl + mz*grnd()

          if (bdistfun_rt(y,z,aratio,q) .ge. 0.0d0) then
               nin = nin + 1
          end if

     end do

     area = area_rec*dble(nin)/ntot

end subroutine get_racetrack_area
```

```
 ./utils/hdf_add_1d_darray_to_file.f90

subroutine hdf_add_1d_darray_to_file(m,A,filename,arrayname,description)
! Given an hdf file already created, takes an array with
! dimensions m,n and writes it to the hdf file.
!
! Baby steps. Test it with the dump_uduct_flow code.


use hdf5
implicit none

     ! Inputs
     integer                              :: m
     double precision, dimension(m)       :: A
     character(len=1024)                  :: filename,arrayname,attrname
     character(len=1024)                  :: description

     ! HDF variables.
     integer(hid_t)                       :: file_id,dset_id,dspace_id, &
                                             attr_id,aspace_id,atype_id
     integer(hsize_t), dimension(1)       :: adims
     integer                              :: rank,error
     integer                              :: arank
     integer(HSIZE_T), dimension(1)       :: data_dims
     integer(size_t)                      :: attrlen

     ! Misc.


     parameter(rank=1)    ! Dimension of array.
     parameter(arank=1)   ! Rank of attribute (size of attribute array?)
     parameter(adims=(/1/))  ! Size of array of hdf attributes. For our purposes,
                          ! only using 1.
     parameter(attrname="Description")

     ! ------------------------------

     ! Do some preliminary work
     data_dims(1) = m


     ! Initialize interface, open the file.
     call h5open_f(error)
     call h5fopen_f(filename, H5F_ACC_RDWR_F, file_id, error)

     ! Create the dataset and dataspace and all that.
     call h5screate_simple_f(rank, data_dims, dspace_id, error)
```

```
      call h5dcreate_f(file_id, arrayname, H5T_NATIVE_DOUBLE, dspace_id, &
                       dset_id, error)

      ! Write array.
      call h5dwrite_f(dset_id, H5T_NATIVE_DOUBLE, A, data_dims, error)

      ! Write the text description for the array.
      ! Turns out this requires making the datatype and whatnot.

      description = trim(description)
      attrlen = len_trim(description)

      call h5screate_simple_f(arank, adims, aspace_id, error)
      call h5screate_simple_f(arank, adims, aspace_id, error)
      call h5tcopy_f(H5T_NATIVE_CHARACTER, atype_id, error)
      call h5tset_size_f(atype_id, attrlen, error)
      call h5acreate_f(dset_id, attrname, atype_id, aspace_id, attr_id, error)

      ! The write happens here.
      call h5awrite_f(attr_id, atype_id, description, adims, error)
      call h5aclose_f(attr_id, error)


      ! Close the dataset, file, and hdf interface.
      call h5dclose_f(dset_id, error)
      call h5fclose_f(file_id, error)
      call h5close_f(error)

      ! EXIT

end subroutine hdf_add_1d_darray_to_file


 ./utils/hdf_add_2d_darray_to_file.f90

subroutine hdf_add_2d_darray_to_file(m,n,A,filename,arrayname,description)
! Given an hdf file already created, takes an array with
! dimensions m,n and writes it to the hdf file.
!
! Baby steps. Test it with the dump_uduct_flow code.


use hdf5
implicit none

      ! Inputs
      integer                             :: m,n
```

```
double precision, dimension(m,n)   :: A
character(len=1024)                :: filename,arrayname,attrname
character(len=1024)                :: description

! HDF variables.
integer(hid_t)                     :: file_id,dset_id,dspace_id, &
                                        attr_id,aspace_id,atype_id
integer(hsize_t), dimension(1)     :: adims
integer                            :: rank,error
integer                            :: arank
integer(HSIZE_T), dimension(2)     :: data_dims
integer(size_t)                    :: attrlen

! Misc.


parameter(rank=2)          ! Dimension of array.
parameter(arank=1)         ! Rank of attribute (size of attribute array?)
parameter(adims=(/1/))     ! Size of array of hdf attributes. For our purposes,
                           ! only using 1.
parameter(attrname="Description")

! Do some preliminary work
data_dims(1) = m
data_dims(2) = n


! Initialize interface, open the file.
call h5open_f(error)
call h5fopen_f (filename, H5F_ACC_RDWR_F, file_id, error)

! Create the dataset and dataspace and all that.
call h5screate_simple_f(rank, data_dims, dspace_id, error)
call h5dcreate_f(file_id, arrayname, H5T_NATIVE_DOUBLE, dspace_id, &
                   dset_id, error)

! Write array.
call h5dwrite_f(dset_id, H5T_NATIVE_DOUBLE, A, data_dims, error)

! Write the text description for the array.
! Turns out this requires making the datatype and whatnot.
description = trim(description)
attrlen = len_trim(description)

call h5screate_simple_f(arank, adims, aspace_id, error)
call h5screate_simple_f(arank, adims, aspace_id, error)
call h5tcopy_f(H5T_NATIVE_CHARACTER, atype_id, error)
call h5tset_size_f(atype_id, attrlen, error)
```

```
      call h5acreate_f(dset_id, attrname, atype_id, aspace_id, attr_id, error)

      ! The write happens here.
      call h5awrite_f(attr_id, atype_id, description, adims, error)
      call h5aclose_f(attr_id, error)

      ! Close the dataset, file, and hdf interface.
      call h5dclose_f(dset_id, error)
      call h5fclose_f(file_id, error)
      call h5close_f(error)

end subroutine hdf_add_2d_darray_to_file
```

./utils/hdf_add_3d_darray_to_file.f90

```
subroutine hdf_add_3d_darray_to_file(m,n,p,A,filename,arrayname,description)
! Given an hdf file already created, takes an array with
! dimensions m,n,p and writes it to the hdf file.
!
! Baby steps. Test it with the dump_uduct_flow code.

use hdf5
implicit none

      ! Inputs
      integer, intent(in)                                 :: m,n,p
      double precision, dimension(m,n,p), intent(in)      :: A
      character(len=1024), intent(in)                     :: filename,arrayname
      character(len=1024), intent(in)                     :: description

      ! HDF variables.
      integer(hid_t)                                      :: file_id,dset_id,dspace_id, &
                                                               attr_id,aspace_id,atype_id
      integer(hsize_t), dimension(1)                      :: adims
      integer                                             :: rank,error
      integer                                             :: arank
      integer(HSIZE_T), dimension(3)                      :: data_dims
      integer(size_t)                                     :: attrlen

      ! Misc.
      character(len=1024) :: attrname

      parameter(rank=3)    ! Dimension of array.
      parameter(arank=1)   ! Rank of attribute (size of attribute array?)
      parameter(adims=(/1/))   ! Size of array of hdf attributes. For our purposes,
                           ! only using 1.
      parameter(attrname="Description")
```

```
    ! Do some preliminary work
    data_dims(1) = m
    data_dims(2) = n
    data_dims(3) = p

    ! Initialize interface, open the file.
    call h5open_f(error)
    call h5fopen_f (filename, H5F_ACC_RDWR_F, file_id, error)

    ! Create the dataset and dataspace and all that.
    call h5screate_simple_f(rank, data_dims, dspace_id, error)
    call h5dcreate_f(file_id, arrayname, H5T_NATIVE_DOUBLE, dspace_id, &
                     dset_id, error)

    ! Write array.
    call h5dwrite_f(dset_id, H5T_NATIVE_DOUBLE, A, data_dims, error)

    ! Write the text description for the array.
    ! Turns out this requires making the datatype and whatnot.

    attrlen = len_trim(trim(description))

    call h5screate_simple_f(arank, adims, aspace_id, error)
    call h5screate_simple_f(arank, adims, aspace_id, error)
    call h5tcopy_f(H5T_NATIVE_CHARACTER, atype_id, error)
    call h5tset_size_f(atype_id, attrlen, error)
    call h5acreate_f(dset_id, attrname, atype_id, aspace_id, attr_id, error)

    ! The write happens here.
    call h5awrite_f(attr_id, atype_id, description, adims, error)
    call h5aclose_f(attr_id, error)

    ! Close the dataset, file, and hdf interface.
    call h5dclose_f(dset_id, error)
    call h5fclose_f(file_id, error)
    call h5close_f(error)

end subroutine hdf_add_3d_darray_to_file
```

./utils/hdf_create_file.f90

```
subroutine hdf_create_file(filename)
! Creates a blank h5 file with the given filename.

    USE HDF5 ! This module contains all necessary modules
```

```
      IMPLICIT NONE

      CHARACTER(LEN=1024)      :: filename
      INTEGER(HID_T)           :: file_id     ! File identifier

      INTEGER                  :: error  ! Error flag

      !
      !    Initialize FORTRAN interface.
      !
      CALL h5open_f(error)

      !
      ! Create a new file using default properties.
      !
      CALL h5fcreate_f(filename, H5F_ACC_TRUNC_F, file_id, error)

      !
      ! Terminate access to the file.
      !
      CALL h5fclose_f(file_id, error)

      !
      !    Close FORTRAN interface.
      !
      CALL h5close_f(error)

end subroutine hdf_create_file


 ./utils/hdf_read_1d_darray.f90

subroutine hdf_read_1d_darray(m,filename,dsetname)
! Read a 1d double array from an h5 file under the corresponding
! dsetname into a temporary buffer from the module mod_readbuff.

use mod_readbuff
use hdf5
implicit none

      integer(HSIZE_T), intent(out)                :: m
      character(len=1024), intent(in)              :: filename,dsetname

!     double precision, dimension(m), intent(out)  :: A

      ! HDF variables.
      integer(hid_t)                              :: file_id,dset_id,dspace_id
      integer                                     :: hdferror,rank
```

```
    integer(HSIZE_T), dimension(1)              :: dims,maxdims

    ! ------------------------------
    ! Initialize the hdf interface.
    call h5open_f(hdferror)

    ! Open the file.
    call h5fopen_f(filename, H5F_ACC_RDONLY_F, file_id, hdferror)
    call h5dopen_f(file_id, dsetname, dset_id, hdferror)

    ! Read the file, figuring out the dimensions, allocating,
    ! then copying over the array.

    ! Getting the dataspace ID
    call h5dget_space_f(dset_id, dspace_id, hdferror)
    call h5sget_simple_extent_ndims_f(dspace_id, rank, hdferror)

    ! Getting dims from dataspace
    call h5sget_simple_extent_dims_f(dspace_id, dims, maxdims, hdferror)


    m = dims(1)
    allocate(readbuff_double(m))
    ! Reading array of size dims.
    call h5dread_f(dset_id, H5T_NATIVE_DOUBLE, readbuff_double, dims, hdferror, h5S_ALL_F, dspa

    ! Close the dataset, file, and hdf interface.
    call h5sclose_f(dspace_id, hdferror)
    call h5dclose_f(dset_id, hdferror)
    call h5fclose_f(file_id, hdferror)
    call h5close_f(hdferror)

end subroutine hdf_read_1d_darray


 ./utils/hdf_write_to_open_2d_darray.f90

subroutine hdf_write_to_open_2d_darray(m,n,i,nrow,array,dset_id)
! As the name suggests; given an opened dataset id,
! modifies the i-th through (i+nrow-1) row of it.
!
! The h5 dataset is assumed to have total dimension (m,n).
!
! Does no error checking whatsoever. Don't be dumb!

use hdf5
implicit none
```

```
! Inputs
integer                                         :: m,n,i,nrow
double precision, dimension(1:nrow,1:n)         :: array
INTEGER(HID_T)                                  :: dset_id          ! Dataset identifier


! HDF things
INTEGER(HID_T)                                  :: dataspace        ! Dataspace identifier
INTEGER(HID_T)                                  :: memspace         ! memspace identifier
integer                                         :: error,rank

integer(hsize_t), dimension(2)                  :: offset,stride,block,steps,dimsm

rank=2
offset = (/i-1,0/)                  ! Which element to start at. HDF counts from zero.
stride = (/1,1/)                    ! Write sequential elements
block = (/1,1/)                     ! No blocks.
steps = (/nrow,n/)                  ! How many times to 'repeat the pattern'
                                    ! in each direction.
                                    ! In this case, the size of the array.

dimsm=(/nrow,n/)                    ! Dimensions of subset to write to dataset.


!
! Get dataset's dataspace identifier and select subset.
!
if (i+nrow-1 .gt. m) then
     write(*,*) "Warning: possibly writing past the end of a HDF dataset."
end if

CALL h5dget_space_f(dset_id, dataspace, error)

CALL h5sselect_hyperslab_f(dataspace, H5S_SELECT_SET_F, &
     offset, steps, error, stride, block)

!
! Create memory dataspace.
!
CALL h5screate_simple_f(rank, dimsm, memspace, error)

!
! Write subset to dataset
!

CALL h5dwrite_f(dset_id, H5T_NATIVE_DOUBLE, array, dimsm, error, &
     memspace, dataspace)
```

```
end subroutine hdf_write_to_open_2d_darray


 ./utils/interp_meshes.f90

subroutine padded_cheb_nodes(n,x,x0,xf)
! Fills array x, length n, with the n-2 Chebyshev nodes
! on the interval (x0,xf), with the first and
! last nodes as x0,xf padded on.
implicit none

     ! In
     integer                              :: n,i
     double precision                     :: x0,xf

     ! In/out
     double precision, dimension(n)       :: x

     ! Internal
     double precision                     :: pi

     parameter(pi = 4.0d0*datan(1.0d0))

     x(1) = x0
     do i=2,n-1
          x(i) = 0.5d0*(x0+xf) + 0.5d0*(xf-x0)*dcos(dble(2*(n-i)-1)/dble(2*(n-2))*pi)
     end do
     x(n) = xf


end subroutine padded_cheb_nodes
!
! ----------------------------------------
!
subroutine uniform_nodes(n,x,x0,xf)
! Fills array x, length n, with the uniformly distributed nodes
! on the interval (x0,xf).
implicit none

     ! In
     integer                              :: n,i
     double precision                     :: x0,xf

     ! In/out
     double precision, dimension(n)       :: x
```

```
      ! Internal
      double precision                    :: dx

      ! Uniform nodes.
      dx = (xf-x0)/(n-1)

      do i=1,n
           x(i) = x0 + (i-1)*dx
      end do

end subroutine uniform_nodes
```

./utils/linear_interp_2d.f90

```
double precision function linear_interp_2d(um,un,u,x,y,x0,y0)
! Bilinear interpolation in 2d, with the accompanying index-locator
! function.
!
! Given an m-by-n array of "exact" u values, rectangular domain
! defined on a grid with x and y values, and interpolation point
! x0, y0, returns the approximate u(x0,y0) value using a 4-point bilinear
! interpolation.

implicit none
      ! Input vars
      integer                             :: um,un
      double precision, dimension(um,un) :: u
      double precision, dimension(um)    :: x
      double precision, dimension(un)    :: y
      double precision                    :: x0,y0

      ! Internal vars
      double precision                    :: u1,u2,u3,u4,p,q
      integer                             :: j,k
      double precision                    :: one
      parameter(one = 1.0d0)

      ! Functions
      integer locate,locate2


      ! If on a uniform grid, should use locate2 instead to reduce
      ! the lookup cost.
      if (.true.) then
           j = locate2(um,x(1),x(um),x0)
           k = locate2(un,y(1),y(un),y0)
      else
```

```
          j = locate(um,x,x0)
          k = locate(un,y,y0)
     end if

     u1 = u(j,k)
     u2 = u(j+1,k)
     u3 = u(j+1,k+1)
     u4 = u(j,k+1)


     p = ( x0 - x(j) )/( x(j+1) - x(j) )
     q = ( y0 - y(k) )/( y(k+1) - y(k) )

     linear_interp_2d = (one-p)*(one-q)*u1 + p*(one-q)*u2 + &
                    p*q*u3 + (one-p)*q*u4


end function linear_interp_2d
!
! ------------------------------------------------------
!
integer function locate(n,x,x0)
! Given 1d double precision array x dimension n,
! monotonically increasing or decreasing,
! and double precision x0, locate the index
! j which satisfies x(j) <= x0 <= x(j+1).
!
! This is not idiot-proof; if x0 lies outside
! the domain of x then you're SOL.
!
! Adapted from "Numerical Recipes." The basic idea
! behaves like bisection; the endpoints of the array
! serve as the positive/negative bounds of the
! "function" x-x0, then the bounds are iteratively
! refined until we reach a bound of (/j,j+1/).
!
! Should converge in log_2(n) steps.

implicit none
     ! Input vars
     integer                              :: n
     double precision, dimension(n)       :: x
     double precision                     :: x0

     ! Internal
     integer                              :: jl,ju,jm
     logical                              :: xisincr
```

```
      ! Upper/lower limits on the containing index
      jl = 0
      ju = n+1

      xisincr = ( x(n) .gt. x(1) )

      do while (ju-jl .gt. 1)

            jm = (ju+jl)/2      ! Compute a midpoint of idx bound

            ! Choose the next bound depending on whether
            ! the array x is monotone increasing or decreasing.
            if ( xisincr .eqv. ( x0 .gt. x(jm) ) ) then
                  jl = jm
            else
                  ju = jm
            end if

      end do

      ! Return lower index bound.
      ! This will only appear if there is a double reflection
      ! that would be necessary in the code with MC, where
      ! jl=0 since x0 < min(x).
      !
      ! Should not occur with proper simulation, though;
      ! dt should be chosen small enough relative to Peclet
      ! that probability of a Brownian motion that large is
      ! vanishingly small.
      locate = max(jl,1)

end function locate
!
! ------------------------
!
integer function locate2(n,xl,xr,x0)
! I THINK WE CAN DO BETTEH!
! https://www.youtube.com/watch?v=NTpptLoUEk8
!
! Assuming the array x is a uniform mesh, we can get the
! precise index with some modular arithmetic. xl and xr are
! the lower and upper bounds of the array.
!
! Also assumes arrays start counting at 1.
implicit none

      ! Input
      double precision    :: xl,xr,x0
```

```
    integer             :: n
    ! Internal
    double precision    :: h

    h = (xr-xl)/(n-2)

    ! Add one because indexing starts at 1.
    locate2 = floor( ((x0-xl) - dmod(x0-xl,h)) / h ) + 1

end function locate2
```

./utils/make_filename_direct.f90

```
subroutine make_filename(mMax, nMax, flow_type, suffix, output_file)

! Makes a succinct filename describing the max summation indices,
! problem type, flow type.
!
! PROBLEM TYPE MUST BE 4 CHARACTERS LONG FOR NOW ("root" or "eval")

    implicit none

    ! Inputs
    integer                 :: mMax, nMax, sm, sn
    character(len=1024)     :: flow_type, suffix, temp1, output_file

    ! This is a "general" implementation, unless you want to sum to more than 10**10
    ! on every single index. (The "I1" in the fortran format descriptor assumes
    ! a 1-digit integer, which is sm and sn, the number of digits in the
    ! truncation indices M and N.

    sm = floor(log10(dble(mMax)))+1             ! Count the number of digits in nMax.
    sn = floor(log10(dble(nMax)))+1             ! Count the number of digits in nMax.

    ! Create the formatting string with this number of digits.
    write(temp1,"(A5,I1,A5,I1,A13)") "(A1,I",sm,",A1,I",sn,",A5,A1,A4,A4)"

    ! Write the max index on the first line, then the array in columns
    ! on the following lines.
    write(output_file,temp1) trim("m"),mMax,trim("n"),nMax,trim(flow_type),"_",trim(suffix),tri

end subroutine make_filename
```

./utils/make_histogram.f90

```
subroutine make_histogram(n,X,nhb,centers,heights)
! Bins the array X with specified bin centers. This subroutine uses
```

```
! equally spaced bins.
!
! In:
!
!    n
!    X(n)      - array of particle positions to bin across the second
!                          dimension, then averaged across the first dimension.
!    nhb                  - number of histogram bins
!
! In/out:
!
!    centers(nhb), heights(nhb)    - centers and heights of the bins. Normalized
!                                    to be a probability density.
!

implicit none

     integer, intent(in)                                    :: n,nhb
     double precision, dimension(n), intent(in)             :: X
     double precision, dimension(nhb), intent(out)          :: centers,heights


     integer, dimension(n)                                  :: Xidx
     double precision                                       :: xmin,xmax,db
     integer                                                :: j

     ! Set up the binning.
     xmin = minval(X)
     xmax = maxval(X)

     if (xmin .eq. xmax) then
          xmin = xmin - 1.0d0
          xmax = xmax + 1.0d0
     end if

     db = (xmax-xmin)/nhb

     do j=1,nhb
          centers(j) = xmin + (j-0.5d0)*db
          heights(j) = 0.0d0
     end do

     call uniform_bins_idx(n,X,xmin,xmax,nhb,Xidx)

     do j=1,n
          heights(Xidx(j)) = heights(Xidx(j)) + 1.0d0
     end do
```

```
end subroutine make_histogram


 ./utils/make_histogram2d.f90

subroutine make_histogram2d(n,X,Y,nhb,nby,hcx,hcy,heights)
! Bins the pair of arrays X,Y in two dimensions, with nhb bins in the
! x direction and nby bins in the y direction. hcx and hcy track
! the locations of bin *centers*, not boundaries. heights is the count,
! non-normalized.
!

implicit none

     integer, intent(in)                                    :: n,nhb,nby
     double precision, dimension(n), intent(in)             :: X,Y
     double precision, dimension(nhb), intent(out)          :: hcx
     double precision, dimension(nby), intent(out)          :: hcy
     double precision, dimension(nhb,nby), intent(out)      :: heights

     integer, dimension(n)                                  :: Xidx,Yidx,Bidx
     double precision                                       :: xmin,xmax,dbx,ymin,ymax,dby
     integer                                                :: j,i

     ! Set up the binning.
     xmin = minval(X)
     xmax = maxval(X)

     if (xmin .eq. xmax) then
          xmin = xmin - 1.0d0
          xmax = xmax + 1.0d0
     end if

     ymin = -1.0d0
     ymax = 1.0d0

     dbx = (xmax-xmin)/nhb
     dby = (ymax-ymin)/nby

     do j=1,nhb
          hcx(j) = xmin + (j-0.5d0)*dbx
     end do

     do j=1,nby
          hcy(j) = ymin + (j-0.5d0)*dby
     end do

     do i=1,nhb
```

```
          do j=1,nby
               heights(i,j) = 0.0d0
          end do
     end do


     call uniform_bins_idx(n,X,xmin,xmax,nhb,Xidx)
     call uniform_bins_idx(n,Y,ymin,ymax,nby,Yidx)

     do j=1,n
          heights(Xidx(j),Yidx(j)) = heights(Xidx(j),Yidx(j)) + 1.0d0
     end do

end subroutine make_histogram2d
```

./utils/my_normal_rng.f90

```
subroutine my_normal_rng(na,array,mean,variance)
! Populates array size n with normally distributed
! random variables with given mean and variance.
!
! Done by generating pairs of uniform random [0,1]
! with Fortran's built-in function, then doing
! the Box-Muller transform to get iid normal vars.
!
! The Mersenne Twister is assumed already initialized/seeded
! in a parent function.

use mtmod ! Mersenne Twister module

implicit none

     integer, intent(in)                                :: na
     double precision, dimension(1:na), intent(inout)   :: array
     double precision, intent(in)                       :: mean,variance

     double precision                                   :: stdev,modulus,modsq,mult,twopi
     double precision, dimension(1:2)                   :: pair
     logical                                            :: nisodd,good
     integer                                            :: krng,nac

     parameter( twopi = 6.283185307179586d0 )

     stdev = dsqrt(variance)

     nac=na
```

```
      nisodd = (mod(nac,2) .eq. 1)
      if (nisodd) then
           nac=nac-1
      end if

      ! Testing "Numerical Recipes" version, avoiding calculating
      ! cosine and sine with a variation on Box-Muller.

      do krng=1,nac,2

           ! Grab pairs of points until you get a pair
           ! that lies in the unit ball.
           good = .false.
           do while (.not. good)

                pair(1) = grnd()
                pair(2) = grnd()
                pair = 2.0d0*pair - 1.0d0
                modsq = pair(1)**2 + pair(2)**2

                ! Logical evaluation
                good = (modsq .lt. 1.0d0)
           end do

           ! Then, apply the formula.
           mult = dsqrt(-2.0d0*dlog(modsq)/modsq)
           array(krng) = mean + stdev*mult*pair(1)
           array(krng+1) = mean + stdev*mult*pair(2)

      end do


      ! Handle the last point with a normal Box-Muller if necessary.
      if (nisodd) then
           nac = nac+1

           pair(1) = grnd()
           pair(2) = grnd()
           modulus = dsqrt(-2.0d0 * dlog(pair(1)))
           array(nac) = mean + stdev*modulus*dcos(twopi*pair(2))
      end if

end subroutine my_normal_rng
```

```
 ./utils/print_parameters.f90

subroutine print_parameters(aratio,q,Pe,nGates,nTot,y0,z0,save_hist,&
                            t_hist,dtmax,nt,ntt,mt_seed,geometry,use_external_ic)

! Spit out a nice header with all the simulation settings.

implicit none
     integer, parameter                                 :: i64 = selected_int_kind(18)

     double precision, intent(in)                       :: aratio,q,Pe,y0,z0,dtmax
     integer, intent(in)                                :: nGates,nTot,nt,ntt
     logical, intent(in)                                :: save_hist,use_external_ic
     double precision, dimension(1:nt), intent(in)      :: t_hist
     integer(kind=i64), intent(in)                      :: mt_seed
     character(len=1024), intent(in)                    :: geometry
     character(len=15)                                  :: num2str

     character(len=1024), parameter                     :: racetrack = "racetrack"
     character(len=1024), parameter                     :: channel = "channel"

     write(*,*) ""
     write(*,"(80A)") REPEAT("=",80)
     write(*,*) ""

     write(*,"(A10,A7)") "Geometry: ",trim(geometry)
     write(*,*) ""

     ! Display differently for point source and uniform.
     if (use_external_ic) then
          write(*,"(A28)") "Initial data read from file."
          write(*,*) ""
     else
          if (nGates .eq. 1) then
               write(*,"(A50)") "Point source initial data."
               write(*,*) ""

               ! Display point source different for channel and pipe/duct.
               if (geometry .eq. "channel") then
                    write(*,"(A50,ES10.3)") "Starting coordinate: ",y0
               else
                    write(*,"(A50,A1,ES10.3,A3,ES10.3,A2)") "Starting coordinates: ","(",y0," ,
               end if

          else
               write(*,"(A50)") "Uniform initial data."
          end if
     end if
```

```
      write(*,*) ""

      write(*,"(A50,ES10.3)") "Peclet: ", Pe
      if (.not. (geometry .eq. channel)) then
           write(*,"(A50,ES10.3)") "Aspect ratio: ",aratio
           write(*,*) ""
      end if
      if (geometry .eq. racetrack) then
           write(*,"(A50,ES10.3)") "Shape parameter: ",q
      end if
      write(*,*) ""

      write(num2str,"(I15)") nTot
      write(*,"(A50,A15)") "Number of particles: ", adjustl(num2str)

      write(*,"(A50,A1,ES10.3,A3,ES10.3,A2)") "Time interval: ","(", t_hist(1), " , ", t_hist(nt)

      write(num2str,"(I15)") ntt
      write(*,"(A50,A15)") "Number of requested timesteps: ",adjustl(num2str)

      write(num2str,"(I15)") nt
      write(*,"(A50,A15)") "Number of internal timesteps: ",adjustl(num2str)
      write(*,"(A50,ES10.3)") "Largest internal timestep: ",dtmax

      write(*,*) ""

      if (save_hist) then
           write(*,"(A40)") "Saving position histories to file."
           write(*,*) ""
      end if

      write(num2str,"(I15)") mt_seed
      write(*,"(A40,A15)") "Mersenne Twister seed: ", adjustl(num2str)

1234  continue

      write(*,*) ""
      write(*,"(80A)") REPEAT("=",80)
      write(*,*) ""



end subroutine print_parameters
```

./utils/progress_meter.f90

```fortran
subroutine progress_meter(k,n,in_place)
! Displays a simple percentage progress meter,
! assuming this is placed inside a do loop, k=1,...,n.

use mod_duration_estimator

implicit none
     integer, intent(in) :: k,n
     logical, intent(in) :: in_place

     mde_pttc = predict_completion(mde_ntt,mde_ntc,mde_dts)

     call mde_pretty_print_time(mde_pttc,mde_pttc_pretty,mde_time_unit)

     if (in_place) then
          ! Use the gfortran functionality to shift the cursor
          ! to the left, to have a "dynamic" percentage.
          ! Does not work with ifort, which is what the case below handles.
          if (k .eq. 2) then
               write(*,"(A27)",advance="no") "   Time remaining:         "
          end if

          if (mod(k*100,n) .lt. 100) then
               write(*,"(A27)",advance="no") "\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b\b
               write(*,"(A4)",advance="no") "\b\b\b\b"
               write(*,"(I3,A1,A3,A16,F5.1,A3)",advance="no") floor(dble(k)/dble(n)*100.0d0),"%"
                                       "   ","Time remaining: ",mde_pttc_pretty,mde_time_unit
          else
               write(*,"(A8)",advance="no") "\b\b\b\b\b\b\b\b"
               write(*,"(F5.1,A3)",advance="no") mde_pttc_pretty,mde_time_unit
          end if


     else
          if (mod(k*100,n) .lt. 100) then
               write(*,"(I3,A1,A3,A16,F5.1,A3)") floor(dble(k)/dble(n)*100.0d0),"%",&
                                       "   ","Time remaining: ",mde_pttc_pretty,mde_time_unit
          end if
     end if

end subroutine progress_meter
```

./utils/read_inputs_direct.f90

```
subroutine read_inputs_direct(input_file,nAratios,aratios,mMax,nMax,flow_type)

implicit none

     ! Internal vars
     integer                                          :: funit

     ! Inputs
     character(len=1024)                              :: input_file

     ! Inputs/outputs
     integer                                          :: nAratios,mMax,nMax
     double precision, dimension(1:1024)              :: aratios
     character(len=1024)                              :: flow_type

     funit = 53 ! Arbitrary

     open(funit,file=input_file)

          read(funit,*) nAratios
          read(funit,*) aratios(1:nAratios)
          read(funit,*) mMax
          read(funit,*) nMax
          read(funit,*) flow_type

     close(funit)

end subroutine read_inputs_direct
```

./utils/read_inputs_mc.f90

```
subroutine read_inputs_mc(param_file,aratio,q,Pe,nGates,x0n,x0width,y0,z0,save_hist,&
                          n_bins,save_hist2d,t_warmup,use_external_ic,ic_file,&
                          tstep_type,dt,dtmax,Tfinal,tsteps,other_file,mt_seed)

! Reads the parameter input file specified on the calling of the program, which
! sets all of the problem and timestepping parameters.

implicit none
     integer, parameter                 :: i64 = selected_int_kind(18)


     character(len=1024), intent(in)    :: param_file

     character(len=1024), intent(out)   :: tstep_type, other_file, ic_file
```

```
double precision, intent(out)      :: aratio, q, Pe, y0, z0, dt, dtmax, Tfinal, x0width
logical, intent(out)               :: save_hist, save_hist2d, use_external_ic
integer, intent(out)               :: nGates, tsteps, x0n, n_bins
integer(i64), intent(out)          :: mt_seed
double precision, intent(out)      :: t_warmup

! Internal
integer                            :: funit,i
character(len=1)                   :: dummy

funit=55
! Initial condition/problem parameters.

open(funit,file=param_file)
     ! Six initial lines to skip

     do i=1,6
          read(funit,*) dummy
     end do

     read(funit,*) aratio     ! Aspect ratio (ignored in channel)
     read(funit,*) q          ! Shape parameter (for racetrack)
     read(funit,*) Pe         ! Peclet number
     read(funit,*) nGates     ! Number of discr. points to use in the transverse direction.
     read(funit,*) x0n        ! Number of discr. points to use in the longitudinal directio
                              ! Total number of points will be (roughly) nGates*x0n.
                              ! Memory requirement is then to leading order 3*nGates*x0n*8
     read(funit,*) x0width    ! IC characteristic width (relative to short side length 2)
     read(funit,*) y0         ! If nGates=1, specify initial y position for point source re
     read(funit,*) z0         ! If nGates=1, specify initial z position for point source re
     read(funit,*) save_hist  ! Flag to save full particle position histories.
     read(funit,*) n_bins     ! Num. of bins to use in the short direction for ptwise stati
                              ! If zero, not saved. Computationally expensive.
     read(funit,*) save_hist2d  ! Flag to save 2d histogram data.

     read(funit,*) t_warmup   ! After setting the initial condition with the given
                              ! parameters above, allow it to diffuse with no
                              ! flow for nondimensional time t_warmup.
                              ! Used for setting a plug IC which is Gaussian in x, or
                              ! diffusing a point source injection.
                              ! No statistics are collected during this time.

     read(funit,*) use_external_ic      ! Look at an external h5 file for the initial condi
     read(funit,*) ic_file              ! Name of h5 file to look at.

     ! Five lines to skip between the initial condition and timestepping
     do i=1,5
```

```
            read(funit,*) dummy
        end do

        ! Timestepping parameters.
        ! Basically, you specify the values of time you want data to be saved,
        ! and the internal timesteps will be chosen as min(t_{n}-t,dtmax).

        read(funit,*) tstep_type ! 'unif', 'expo', or 'supplied'
        read(funit,*) dt         ! Timestep to save statistics for 'unif', initial time for 'e
        read(funit,*) dtmax      ! Maximum internal timestep
        read(funit,*) Tfinal     ! Final time
        read(funit,*) tsteps     ! Total number of timesteps to save.
        read(funit,*) other_file ! If tstep_type .eq. 'supplied', the name of the h5 file
                                 ! containing the specified times.
        read(funit,*) mt_seed    ! Seed for the RNG. Usually filled in with a random
                                 ! seed by batch_submit.py.
    close(funit)

end subroutine read_inputs_mc
```

./utils/save_the_rest_channel.f90

```
subroutine save_the_rest_channel(fname,geometry,ntt,target_times,means,vars,skews,kurts,n_bins,&
                                 means_sl,vars_sl,skews_sl,kurts_sl,nhb,hist_centers,hist_heights,&
                                 Pe,nGates,x0n,x0width,mt_seed)
! Saves the remainder of the calculated data (moments, problem parameters, solver settings,
! etc) in the h5 file.

implicit none
    integer, parameter                  :: i64 = selected_int_kind(18)


    character(len=1024), intent(in)                                 :: fname,geometry
    character(len=1024)                                             :: dsetname, descr
    integer, intent(in)                                             :: ntt, n_bins, nGates, x0n, nh
    double precision, dimension(1:ntt), intent(in)                  :: target_times,means,vars,skew
    double precision, dimension(1:ntt,1:n_bins), intent(in)         :: means_sl,vars_sl,skews_sl,ku
    double precision, dimension(1:ntt,1:nhb), intent(in)            :: hist_centers,hist_heights
    double precision, intent(in)                                    :: Pe, x0width

    integer(i64), intent(in)                                        :: mt_seed

    character(len=1024)                                             :: channel,duct,ellipse
    channel = "channel"
    duct = "duct"
    ellipse = "ellipse"
```

```
! ------------------------------------------

dsetname = "geometry"
descr = "Problem geometry: Channel=0, Duct=1, Ellipse=2, Other=3"
if (geometry .eq. channel) then
     call hdf_add_1d_darray_to_file(1,0,fname,dsetname,descr)
else if (geometry .eq. duct) then
     call hdf_add_1d_darray_to_file(1,1,fname,dsetname,descr)
else if (geometry .eq. ellipse) then
     call hdf_add_1d_darray_to_file(1,2,fname,dsetname,descr)
end if

dsetname = "Time"
descr = "Nondimensionalized time, t = \\frac{H^2}{\\kappa} t'"
call hdf_add_1d_darray_to_file(ntt,target_times,fname,dsetname,descr)


dsetname = "Avgd_Mean"
descr = "Cross-section averaged mean for Nwalkers particles, in X direction."
call hdf_add_1d_darray_to_file(ntt,means,fname,dsetname,descr)

dsetname = "Avgd_Variance"
descr = "Cross-section averaged variance for Nwalkers particles, in X direction."
call hdf_add_1d_darray_to_file(ntt,vars,fname,dsetname,descr)

dsetname = "Avgd_Skewness"
descr = "Cross-section averaged skewness for Nwalkers particles, in X direction."
call hdf_add_1d_darray_to_file(ntt,skews,fname,dsetname,descr)

dsetname = "Avgd_Kurtosis"
descr = "Cross-section averaged kurtosis for Nwalkers particles, in X direction."
call hdf_add_1d_darray_to_file(ntt,kurts,fname,dsetname,descr)

if (.not. (n_bins .eq. 0)) then

     dsetname = "nBins"
     descr = "Number of bins used to calculate statistics across slices."
     call hdf_add_1d_darray_to_file(1,dble(n_bins),fname,dsetname,descr)

     dsetname = "Mean"
     descr = "Mean on slices for Nwalkers particles, in X direction."
     call hdf_add_2d_darray_to_file(ntt,n_bins,means_sl,fname,dsetname,descr)

     dsetname = "Variance"
     descr = "Variance on slices for Nwalkers particles, in X direction."
     call hdf_add_2d_darray_to_file(ntt,n_bins,vars_sl,fname,dsetname,descr)

     dsetname = "Skewness"
```

```
            descr = "Skewness on slices for Nwalkers particles, in X direction."
            call hdf_add_2d_darray_to_file(ntt,n_bins,skews_sl,fname,dsetname,descr)

            dsetname = "Kurtosis"
            descr = "Kurtosis on slices for Nwalkers particles, in X direction."
            call hdf_add_2d_darray_to_file(ntt,n_bins,kurts_sl,fname,dsetname,descr)

        end if

        dsetname = "Hist_centers"
        descr = "Bin centers for the cross-sectionally averaged distribution."
        call hdf_add_2d_darray_to_file(ntt,nhb,hist_centers,fname,dsetname,descr)

        dsetname = "Hist_heights"
        descr = "Bin heights for the cross-sectionally averaged distribution (normalized to PDF)."
        call hdf_add_2d_darray_to_file(ntt,nhb,hist_heights,fname,dsetname,descr)

        dsetname = "Peclet"
        descr = "Peclet number."
        call hdf_add_1d_darray_to_file(1,Pe,fname,dsetname,descr)

        dsetname = "x0width"
        descr = "Initial conditionn longitudinal width."
        call hdf_add_1d_darray_to_file(1,x0width,fname,dsetname,descr)

        dsetname = "nGates"
        descr = "Number of random walkers used in this trial."
        call hdf_add_1d_darray_to_file(1,dble(nGates),fname,dsetname,descr)

        dsetname = "mt_seed"
        descr = "Integer seed used in the Mersenne Twister (RNG)."
        call hdf_add_1d_darray_to_file(1,dble(mt_seed),fname,dsetname,descr)

        dsetname = "timesteps"
        descr = "Number of timesteps."
        call hdf_add_1d_darray_to_file(1,dble(ntt),fname,dsetname,descr)

        dsetname = "x0n"
        descr = "Number of discretization points discretizing the longitudinal IC."
        call hdf_add_1d_darray_to_file(1,dble(x0n),fname,dsetname,descr)

        dsetname = "nhb"
        descr = "Number of bins used for the cross-sectionally averaged distribution."
        call hdf_add_1d_darray_to_file(1,dble(nhb),fname,dsetname,descr)

    end subroutine save_the_rest_channel
```

```
./utils/save_the_rest_duct.f90

subroutine save_the_rest_duct(fname,geometry,ntt,t_hist,means,vars,skews,kurts,nby,nbz,&
                              means_sl,vars_sl,skews_sl,kurts_sl,nhb,hist_centers,hist_heights,&
                              Pe,nTot,mt_seed,aratio,q,dtmax,t_warmup)
! Saves the remainder of the calculated data (moments, problem parameters, solver settings,
! etc) in the h5 file.

implicit none
     integer, parameter                 :: i64 = selected_int_kind(18)


     character(len=1024), intent(in)                                    :: fname,geometry
     character(len=1024)                                                :: dsetname, descr
     integer, intent(in)                                                :: ntt, nby, nbz, nhb, nTo
     double precision, dimension(1:ntt), intent(in)                     :: t_hist,means,vars,&
                                                                           skews,kurts

     double precision, dimension(1:ntt,1:nby,1:nbz), intent(in)         :: means_sl,vars_sl,&
                                                                           skews_sl,kurts_sl

     double precision, dimension(1:ntt,1:nhb), intent(in)               :: hist_centers,hist_heigh
     double precision, intent(in)                                       :: Pe,aratio,q,dtmax,t_war

     integer(i64), intent(in)                                           :: mt_seed

     character(len=1024)                                                :: channel,duct,ellipse

     channel = "channel"
     duct = "duct"
     ellipse = "ellipse"

     ! ---------------------------------------------

     dsetname = "geometry"
     descr = "Problem geometry: Channel=0, Duct=1, Ellipse=2, Other=3"
     if (geometry .eq. channel) then
          call hdf_add_1d_darray_to_file(1,0,fname,dsetname,descr)
     else if (geometry .eq. duct) then
          call hdf_add_1d_darray_to_file(1,1,fname,dsetname,descr)
     else if (geometry .eq. ellipse) then
          call hdf_add_1d_darray_to_file(1,2,fname,dsetname,descr)
     end if

     dsetname = "Time"
     descr = "Nondimensionalized time, t = a^2/kappa t'"
     call hdf_add_1d_darray_to_file(ntt,t_hist,fname,dsetname,descr)
```

```
dsetname = "Avgd_Mean"
descr = "Cross-section averaged mean for nTot particles, in X direction."
call hdf_add_1d_darray_to_file(ntt,means,fname,dsetname,descr)

dsetname = "Avgd_Variance"
descr = "Cross-section averaged variance for nTot particles, in X direction."
call hdf_add_1d_darray_to_file(ntt,vars,fname,dsetname,descr)

dsetname = "Avgd_Skewness"
descr = "Cross-section averaged skewness for nTot particles, in X direction."
call hdf_add_1d_darray_to_file(ntt,skews,fname,dsetname,descr)

dsetname = "Avgd_Kurtosis"
descr = "Cross-section averaged skewness for nTot particles, in X direction."
call hdf_add_1d_darray_to_file(ntt,kurts,fname,dsetname,descr)

if (.not. (nby .eq. 0)) then

     dsetname = "nBinsY"
     descr = "Number of bins used in Y direction to calculate pointwise statistics."
     call hdf_add_1d_darray_to_file(1,dble(nby),fname,dsetname,descr)

     dsetname = "nBinsZ"
     descr = "Number of bins used in Z direction to calculate pointwise statistics."
     call hdf_add_1d_darray_to_file(1,dble(nbz),fname,dsetname,descr)

     dsetname = "Mean"
     descr = "Pointwise mean in the X direction."
     call hdf_add_3d_darray_to_file(ntt,nby,nbz,means_sl,fname,dsetname,descr)

     dsetname = "Variance"
     descr = "Pointwise variance in the X direction."
     call hdf_add_3d_darray_to_file(ntt,nby,nbz,vars_sl,fname,dsetname,descr)

     dsetname = "Skewness"
     descr = "Pointwise skewness in the X direction."
     call hdf_add_3d_darray_to_file(ntt,nby,nbz,skews_sl,fname,dsetname,descr)

     dsetname = "Kurtosis"
     descr = "Pointwise kurtosis in the X direction."
     call hdf_add_3d_darray_to_file(ntt,nby,nbz,kurts_sl,fname,dsetname,descr)

end if

dsetname = "Hist_centers"
descr = "Bin centers for the cross-sectionally averaged distribution."
call hdf_add_2d_darray_to_file(ntt,nhb,hist_centers,fname,dsetname,descr)
```

```
      dsetname = "Hist_heights"
      descr = "Bin heights for the cross-sectionally averaged distribution."
      call hdf_add_2d_darray_to_file(ntt,nhb,hist_heights,fname,dsetname,descr)

      dsetname = "Peclet"
      descr = "Peclet number."
      call hdf_add_1d_darray_to_file(1,Pe,fname,dsetname,descr)

      dsetname = "aratio"
      descr = "Aspect ratio of the domain."
      call hdf_add_1d_darray_to_file(1,aratio,fname,dsetname,descr)

      dsetname = "q"
      descr = "Shape parameter (only relevant for racetrack)."
      call hdf_add_1d_darray_to_file(1,q,fname,dsetname,descr)

      dsetname = "nTot"
      descr = "Number of particles used."
      call hdf_add_1d_darray_to_file(1,dble(nTot),fname,dsetname,descr)

      dsetname = "mt_seed"
      descr = "Integer seed used in the Mersenne Twister (RNG)."
      call hdf_add_1d_darray_to_file(1,dble(mt_seed),fname,dsetname,descr)

      dsetname = "timesteps"
      descr = "Number of large timesteps."
      call hdf_add_1d_darray_to_file(1,dble(ntt),fname,dsetname,descr)

      dsetname = "dtmax"
      descr = "Maximum internal timestep."
      call hdf_add_1d_darray_to_file(1,dtmax,fname,dsetname,descr)

      dsetname = "t_warmup"
      descr = "Duration initial condition was let sit before turning on the flow."
      call hdf_add_1d_darray_to_file(1,t_warmup,fname,dsetname,descr)

      dsetname = "nhb"
      descr = "Number of bins used for the cross-sectionally averaged distribution."
      call hdf_add_1d_darray_to_file(1,dble(nhb),fname,dsetname,descr)

end subroutine save_the_rest_duct


 ./utils/set_initial_conds_channel_mc.f90

subroutine set_initial_conds_channel_mc(ny,nGates,x0n,nTot,X,Y,y0,a,&
                                        x0width,t_warmup,use_external_ic,ic_file)
! The purpose of the subroutine is in the name.
```

```
!
! y0 refers to point-source initial condition at location y0.

use mtmod
use mod_readbuff

implicit none

     integer, intent(in)                                          :: ny,nGates,x0n,nTot
     double precision, dimension(nTot), intent(inout)             :: X,Y

     double precision, intent(in)                                 :: y0,a,x0width,t_warmup
     logical, intent(in)                                          :: use_external_ic
     character(len=1024), intent(in)                              :: ic_file

     ! Internal
     integer                                                      :: idx,i,j,k,nsteps
     double precision                                             :: xl,yl,dx,dy,dtw
     character(len=1024)                                          :: dsetname

     parameter(nsteps=10)
     ! Advection/diffusion functions!
     external  :: impose_reflective_BC_rect, u_dummy

     ! -------------------------------------------------------------

     if (.not. use_external_ic) then
          ! Construct the initial condition from the parameters specified.
          if (x0n .gt. 1) then
               dx = x0width/(x0n-1)
               xl = -x0width/2
          else
               dx = 0.0d0
               xl = 0.0d0
          end if

          if (nGates .gt. 1) then
               dy = 2.0d0*a/(ny-1)
               yl = -a
          else
               dy = 0.0d0
               yl = y0
          end if

          idx = 0

          do j=0,ny-1
               do k=0,x0n-1
```

```
                    idx = idx + 1
                    X(idx) = xl + k*dx
                    Y(idx) = yl + j*dy

               end do
          end do


     else
          ! Skip all this and read the x,y,z initial data in from the file.
          dsetname = "X"
          call hdf_read_1d_darray(nTot,ic_file,dsetname)
          X = readbuff_double
          deallocate(readbuff_double)

          dsetname = "Y"
          call hdf_read_1d_darray(nTot,ic_file,dsetname)
          Y = readbuff_double
          deallocate(readbuff_double)

     end if

     ! Diffuse the initial condition by calling the advection diffusion operator
     ! with Pe = 0.

     if (t_warmup .gt. 0.0d0) then
          dtw = t_warmup/nsteps

          do i=1,nsteps
               call apply_advdiff1_chan(nTot,X,Y,0.0d0,dtw,a, &
                              u_dummy,impose_reflective_BC_rect)
          end do
     end if

end subroutine set_initial_conds_channel_mc


 ./utils/set_initial_conds_duct_mc.f90

subroutine set_initial_conds_duct_mc(ny,nz,nGates,x0n,nTot,X,Y,Z,y0,z0,a,b,&
                                        x0width,t_warmup,use_external_ic,ic_file)
! The purpose of the subroutine is in the name.
!
! y0,z0 refers to point-source initial condition at location (y0,z0).

use mtmod
use mod_readbuff
```

```
implicit none

     integer, intent(in)                                          :: ny,nz,nGates,x0n,nTot
     double precision, dimension(nTot), intent(inout)             :: X,Y,Z

     double precision, intent(in)                                 :: y0,z0,a,b,x0width,t_warmup
     logical, intent(in)                                          :: use_external_ic
     character(len=1024), intent(in)                              :: ic_file

     ! Internal
     integer                                                      :: idx,i,j,k,nsteps
     double precision                                             :: xl,yl,zl,dx,dy,dz,dtw
     character(len=1024)                                          :: dsetname

     parameter(nsteps=10)

     ! Advection/diffusion functions!
     external  :: impose_reflective_BC_rect, u_dummy

     ! ------------------------------------------------------------

     if (.not. use_external_ic) then
          ! Construct the initial condition from the parameters specified.
          if (x0n .gt. 1) then
               dx = x0width/(x0n-1)
               xl = -x0width/2
          else
               dx = 0.0d0
               xl = 0.0d0
          end if

          if (nGates .gt. 1) then
               dy = 2.0d0*a/(ny-1)
               yl = -a
               dz = 2.0d0*b/(nz-1)
               zl = -b
          else
               dy = 0.0d0
               yl = y0
               dz = 0.0d0
               zl = z0
          end if

          idx = 0
          do i=0,nz-1
               do j=0,ny-1
                    do k=0,x0n-1
                         idx = idx + 1
```

```
                        X(idx) = xl + k*dx
                        Y(idx) = yl + j*dy
                        Z(idx) = zl + i*dz
                    end do
                end do
            end do

        else
            ! Skip all this and read the x,y,z initial data in from the file.
            dsetname = "X"
            call hdf_read_1d_darray(nTot,ic_file,dsetname)
            X = readbuff_double
            deallocate(readbuff_double)

            dsetname = "Y"
            call hdf_read_1d_darray(nTot,ic_file,dsetname)
            Y = readbuff_double
            deallocate(readbuff_double)

            dsetname = "Z"
            call hdf_read_1d_darray(nTot,ic_file,dsetname)
            Z = readbuff_double
            deallocate(readbuff_double)

        end if

        ! Diffuse the initial condition by calling the advection diffusion operator
        ! with Pe = 0.

        if (t_warmup .gt. 0.0d0) then

            dtw = t_warmup/nsteps

            do i=1,nsteps
                call apply_advdiff1_duct(nTot,X,Y,Z,0.0d0,dtw,a,b, &
                                u_dummy,impose_reflective_BC_rect)
            end do
        end if

    end subroutine set_initial_conds_duct_mc


 ./utils/set_initial_conds_ellipse_mc.f90

    subroutine set_initial_conds_ellipse_mc(ny,nz,x0n,a,b,nGates,nTot,X,Y,Z,y0,z0,x0width,t_warmup)
    ! Given all the data, specify the initial conditions in the arrays X,Y,Z.

    implicit none
```

```
double precision, intent(in)                                    :: y0,z0,a,b,x0width,t_warmup
integer, intent(in)                                             :: ny,nz,x0n,nGates,nTot
double precision, dimension(nTot), intent(out)                  :: X,Y,Z

! Internal
integer                                                         :: idx,ix,iy,iz
double precision                                                :: dist,dx,dy,dz,xl,yl,zl

integer                                                         :: nsteps,i
double precision                                                :: dtw

parameter(nsteps=10)

! Advection/diffusion functions!
external  :: impose_reflective_BC_ellipse, u_ellipse

if (x0n .gt. 1) then
     dx = x0width/dble(x0n-1)
     xl = -x0width/2
else
     dx = 0.0d0
     xl = 0.0d0
end if

if (nGates .gt. 1) then
     yl = -a
     dy = 2.0d0*a/(ny-1)
     zl = -b
     dz = 2.0d0*b/(nz-1)
else
     yl = y0
     dy = 0.0d0
     zl = z0
     dz = 0.0d0
end if


idx = 0
do iz=0,nz-1
     do iy=0,ny-1
          dist = ((yl + iy*dy)**2)/(a**2) + ((zl + iz*dz)**2)/(b**2)
          if (dist .le. 1.0d0) then
               do ix=0,x0n-1

                    idx = idx + 1
                    X(idx) = xl + ix*dx
                    Y(idx) = yl + iy*dy
```

```
                    Z(idx) = zl + iz*dz

                end do
            end if
        end do
    end do

    ! Diffuse the initial condition by calling the advection diffusion operator
    ! with Pe = 0.
    if (.true.) then
        if (t_warmup .gt. 0.0d0) then

            dtw = t_warmup/nsteps

            do i=1,nsteps
                call apply_advdiff1_ellipse(nTot,X,Y,Z,0.0d0,dtw,a,b, &
                            u_ellipse,impose_reflective_BC_ellipse,floor(10*dtw))
            end do

        end if
    else
        call apply_advdiff1_ellipse(nTot,X,Y,Z,0.0d0,t_warmup,a,b, &
                        u_ellipse,impose_reflective_BC_ellipse,floor(10*dtw))
    end if

end subroutine set_initial_conds_ellipse_mc
```

./utils/set_initial_conds_racetrack_mc.f90

```
subroutine set_initial_conds_racetrack_mc(x0n,aratio,q,nGates,nTot,X,Y,Z,y0,z0,x0width,t_warmup)
! Given all the data, specify the initial conditions in the arrays X,Y,Z.

use mtmod

implicit none

    double precision, intent(in)                                :: y0,z0,aratio,q,x0width,t_war
    integer, intent(in)                                         :: x0n,nGates,nTot
    double precision, dimension(nTot), intent(out)              :: X,Y,Z

    ! Internal
    integer                                                     :: idx,ix,iy,iz
    double precision                                            :: dist,dx,dy,dz,xl,yl,zl,yw,zw

    integer                                                     :: nsteps,i,j
    double precision                                            :: dtw
```

```
! Advection/diffusion functions!
external  :: impose_reflective_BC_racetrack, u_racetrack

double precision bdistfun_rt

parameter(nsteps=10)

! For the moment, do things differently: Just do random placings with a
! rejection method in the transverse coordinates.
if (x0width .eq. 0.0d0) then
     xl = 0.0d0
     dx = 0.0d0
else
     xl = -x0width/2.0d0
     dx = x0width/(x0n-1)
end if

if (nGates .le. 1) then
     idx = 0
     do i=0,x0n-1
          do j=1,nGates
               idx = idx + 1

               X(idx) = xl + dx*i

               Y(idx) = y0
               Z(idx) = z0

          end do
     end do
else
     yl = -1.3d0
     yw = -2*yl
     zl = -1.3d0/aratio
     zw = -2*zl

     idx = 0
     do i=0,x0n-1
          do j=1,nGates
               idx = idx + 1

               X(idx) = xl + dx*i

               Y(idx) = yl + yw*grnd()
               Z(idx) = zl + zw*grnd()
               do while (bdistfun_rt(Y(idx),Z(idx),aratio,q) .lt. 0.0d0)
                    Y(idx) = yl + yw*grnd()
                    Z(idx) = zl + zw*grnd()
```

```
                end do
            end do
        end do

    end if

    ! Diffuse the initial condition by calling the advection diffusion operator
    ! with Pe = 0.
    if (t_warmup .gt. 0.0d0) then

        dtw = t_warmup/nsteps

        do i=1,nsteps
            call apply_advdiff1_ellipse(nTot,X,Y,Z,0.0d0,dtw,aratio,q, &
                        u_racetrack,impose_reflective_BC_racetrack,floor(10*dtw))
        end do

    end if

end subroutine set_initial_conds_racetrack_mc
```

./utils/set_initial_conds_triangle_mc.f90

```
subroutine set_initial_conds_triangle_mc(ny,nz,x0n,a,nGates,nTot,X,Y,Z,y0,z0,x0width,t_warmup,&
                    use_external_ic,ic_file,nl,lls)
! Given all the data, specify the initial conditions in the arrays X,Y,Z.

!use mtmod
use mod_readbuff

implicit none

    double precision, intent(in)                                :: y0,z0,a,x0width,t_warmup
    integer, intent(in)                                         :: ny,nz,x0n,nGates,nTot,nl
    double precision, dimension(nl,3), intent(in)               :: lls
    logical, intent(in)                                         :: use_external_ic
    character(len=1024), intent(in)                             :: ic_file

    double precision, dimension(nTot), intent(out)              :: X,Y,Z

    ! Internal
    integer                                                     :: idx,ix,iy,iz
    double precision                                            :: dist,dx,dy,dz,xl,yl,zl

    integer                                                     :: nsteps,i
    double precision                                            :: dtw
    double precision, dimension(3)                              :: tempv
```

```
double precision, dimension(nl)                                  :: bvals

character(len=1024)                                              :: dsetname

logical cond
double precision    :: zr,yr

parameter(nsteps=10)

! Advection/diffusion functions!
external  :: impose_reflective_BC_polygon, u_triangle


if (.not. use_external_ic) then
     if (x0n .gt. 1) then
          dx = x0width/dble(x0n-1)
          xl = -x0width/2
     else
          dx = 0.0d0
          xl = 0.0d0
     end if

     if (nGates .gt. 1) then
          yl = -1.0d0
          dy = a*2.0d0*dsqrt(3.0d0)/(ny-1)
          zl = -a*dsqrt(3.0d0)
          dz = a*2.0d0*dsqrt(3.0d0)/(nz-1)

          yr = -1.0d0 + a*2.0d0*dsqrt(3.0d0)
          zr = a*dsqrt(3.0d0)
     else
          yl = y0
          dy = 0.0d0
          zl = z0
          dz = 0.0d0
     end if


     idx = 0
     do iz=0,nz-1
          do iy=0,ny-1

               tempv(1) = 1.0d0
               tempv(2) = yl + iy*dy
               tempv(3) = zl + iz*dz

               call matvec(nl,3,lls,tempv,bvals)
```

```
                    if (all(bvals .ge. 0.0d0)) then

                         do ix=0,x0n-1

                              idx = idx + 1
                              X(idx) = xl + ix*dx
                              Y(idx) = tempv(2)
                              Z(idx) = tempv(3)

                         end do
                    end if
               end do
          end do
     else
          ! Skip all this and read the x,y,z initial data in from the file.
          dsetname = "X"
          call hdf_read_1d_darray(nTot,ic_file,dsetname)
          X = readbuff_double
          deallocate(readbuff_double)

          dsetname = "Y"
          call hdf_read_1d_darray(nTot,ic_file,dsetname)
          Y = readbuff_double
          deallocate(readbuff_double)

          dsetname = "Z"
          call hdf_read_1d_darray(nTot,ic_file,dsetname)
          Z = readbuff_double
          deallocate(readbuff_double)
     end if

     ! Diffuse the initial condition by calling the advection diffusion operator
     ! with Pe = 0.

     if (t_warmup .gt. 0.0d0) then

          dtw = t_warmup/nsteps

          do i=1,nsteps
               call apply_advdiff1_triangle(nTot,X,Y,Z,0.0d0,dtw,a, &
                              u_triangle,impose_reflective_BC_polygon,nl,lls)
          end do

     end if

end subroutine set_initial_conds_triangle_mc
```

./utils/solve_quadratic_eqn.f90

```
subroutine solve_quadratic_eqn(a,b,c,t1,t2)
! Solves the quadratic equation
! a*t**2 + b*t + c .eq. 0.
!
! The two solutions get saved in t1 and t2.
!
! The solutions are assumed real.

implicit none
     double precision, intent(in)  :: a,b,c
     double precision, intent(out) :: t1,t2

     double precision              :: discr

     discr = b**2 - 4.0d0*a*c

     t1 = (-b - dsqrt(discr))/(2.0d0*a)
     t2 = (-b + dsqrt(discr))/(2.0d0*a)

end subroutine solve_quadratic_eqn
```

./utils/sortpairs.f90

```
subroutine sortpairs(nTot,X,Xdup,bin_idxs,nbins)
!
! Given a pair of arrays X, Xdup (double), bin_idxs (integer),
! sorts (buckets) the bin_idxs and carries along the associated
! Xdup value. bin_idxs is known to have integer values from 1 to nbins.
!
! This is a "partial" sorting technically, since we don't need
! the Xdup values to be sorted in any way.
!
! Accomplished by doing a first pass of bin_idxs to see
! how many of each integer there are, then a second pass of
! copying over the contents of X into Xdup in an appropriate order.
!

implicit none
     integer, intent(in)                              :: nTot,nbins
     double precision, dimension(nTot), intent(in)    :: X

     double precision, dimension(nTot), intent(out)   :: Xdup
     integer, dimension(nTot), intent(out)            :: bin_idxs

     integer, dimension(nTot)                         :: bin_idxs2
```

```
      integer, dimension(nbins)                          :: binTots,binCurr
      integer, dimension(nbins+1)                        :: binCum
      integer                                            :: i,idx,ptr

      do i=1,nbins
           binTots(i) = 0
      end do

      !
      ! Do a count of the number of things in each bin.
      !

      do i=1,nTot
           ptr = bin_idxs(i)
           binTots(ptr) = binTots(ptr) + 1
      end do

      !
      ! Generate a cumulative count as pointers.
      ! Makes it easy to reference the start and end elements
      ! of the subset.
      !
      binCum(1) = 1
      do i=2,nbins+1
           binCum(i) = binCum(i-1) + binTots(i-1)
           binCurr(i-1) = binCum(i-1)
      end do

      bin_idxs2 = bin_idxs
      !
      ! Now do a second loop, placing values of X into Xdup
      ! and updating the pointer binCurr along the way.
      !

      do i=1,nTot
           idx = bin_idxs2(i)

           ptr = binCurr(idx)
           binCurr(idx) = binCurr(idx) + 1

           Xdup(ptr) = X(i)
           bin_idxs(ptr) = bin_idxs2(i)
      end do


end subroutine sortpairs
```

./utils/uniform_bins_idx.f90

```
subroutine uniform_bins_idx(nx,X,xmin,xmax,nb,Xidx)
!
! Given a double array X of size nx, and lower and upper
! bounds xmin and xmax and number of bins nb,
! output an integer array Xidx corresponding to
! the bin number assignment, assuming uniformly spaced bins.
!
! In other words, maps the elements 'linearly' to the integers
! 0,1,...,nb-1.

implicit none
     integer, intent(in)                            :: nx,nb
     double precision, intent(in), dimension(nx)    :: X
     double precision, intent(in)                   :: xmin,xmax

     integer, intent(out), dimension(nx)            :: Xidx

     double precision                               :: width

     width = xmax-xmin

     Xidx = floor(((nb-1)/width)*(X - xmin))+1

end subroutine uniform_bins_idx
```

./utils/vector_ops.f90

```
! Scripts for basic operations with vectors and pairs of vectors;
! dot products, lengths,
! applying orthogonal projections, projections orthogonal,
! and reflections. These all MODIFY the first input, so be careful.

double precision function dot_prod(n,u,v)
! Why am I not using BLAS or similar for this? Laziness.
     implicit none
     integer                              :: n,i
     double precision, dimension(n)       :: u,v

     dot_prod=0.0d0
     do i=1,n
          dot_prod = dot_prod + u(i)*v(i)
     end do

end function dot_prod
!
```

```
!
!
double precision function norm(n,u)
     implicit none
     integer                              :: n
     double precision, dimension(n)       :: u

     double precision dot_prod

     norm = dsqrt(dot_prod(n,u,u))
end function norm
!
! ----------------------------------------
!
subroutine normalize(n,u)
! Normalizes 2d vector u.
     implicit none
     integer                              :: n
     double precision, dimension(n)       :: u
     double precision                     :: s

     double precision norm

     s = norm(n,u)
     u = u/s
end subroutine normalize
!
! ----------------------------------------
!
subroutine orth_proj(n,v,u)
! Projects v onto u (u not necessarily unit).
! v is changed on output.
     implicit none
     integer, intent(in)                                 :: n
     double precision, dimension(n), intent(in)          :: u
     double precision, dimension(n), intent(inout)       :: v

     double precision dot_prod

     v = dot_prod(n,u,v)/dot_prod(n,u,u)*u

end subroutine orth_proj
!
! ----------------------------------------
!
subroutine proj_orth(n,v,u)
! The projection of v orthogonal to u,
! u not necessarily unit.
```

```
     implicit none
     integer, intent(in)                 :: n
     double precision, dimension(n), intent(in)        :: u
     double precision, dimension(n), intent(inout)     :: v
     double precision, dimension(n)                    :: temp

     temp = v
     call orth_proj(n,temp,u)
     v = v - temp
end subroutine proj_orth
!
! ----------------------------------------
!
subroutine reflect(n,v,u)
! Reflects v across hyperplane defined by vector u.
! If u is the normal to a surface, it should be the
! _OUTWARD NORMAL_. u does not need to be unit.
     implicit none
     integer, intent(in)                               :: n
     double precision, dimension(n), intent(in)        :: u
     double precision, dimension(n), intent(inout)     :: v
     double precision, dimension(n)                    :: temp

     temp = v
     call orth_proj(n,temp,u)
     v = v - 2.0d0*temp
end subroutine reflect
```

./utils/walkers_in_bin_1d.f90

```
subroutine walkers_in_bin_1d(nGates,X,Y,bin_lo,bin_hi,X_bin,bin_count)
! Takes arrays X,Z, collects all indices bin_lo < Z(i) < bin_hi
! and saves them sequentially in X_bin.
!
! The actual number of relevant values in X_bin
! is unknown a priori, but at most nGates. Hence we keep track of
! the actual number of X positions in a bin with bin_count.
!

implicit none

     integer, intent(in)                               :: nGates
     double precision, dimension(nGates), intent(in)   :: X,Y
     double precision, intent(in)                      :: bin_lo,bin_hi

     double precision, dimension(nGates), intent(out)  :: X_bin
     integer, intent(inout)                            :: bin_count
```

```
     integer                                             :: i

     bin_count=0
     do i=1,nGates
          if ((bin_lo < Y(i)) .and. (Y(i) < bin_hi)) then
               bin_count = bin_count + 1
               X_bin(bin_count) = X(i)
          end if
     end do

end subroutine walkers_in_bin_1d
```

./utils/walkers_in_bin_2d.f90

```
subroutine walkers_in_bin_2d(nTot,X,Y,Z,yl,yh,zl,zh,X_bin,bin_count)

! Takes arrays X,Y,Z, collects all indices satisfying
! yl <= Y(i) < yh and
! zl <= Z(i) < zh,
! and saves them sequentially in X_bin(1:bin_count).
!
! The actual number of relevant values in X_bin
! is unknown a priori, but at most nGates. Hence we keep track of
! the actual number of X positions in a bin with bin_count.
!

implicit none

     integer, intent(in)                                 :: nTot
     double precision, dimension(nTot), intent(in)       :: X,Y,Z
     double precision, intent(in)                        :: yl,yh,zl,zh

     double precision, dimension(nTot), intent(out)      :: X_bin
     integer, intent(out)                                :: bin_count

     ! Internal
     integer                                             :: i

     bin_count=0

     do i=1,nTot
          if ((yl <= Y(i)) .and. (Y(i) < yh) .and. (zl <= Z(i)) .and. (Z(i) < zh)) then
               bin_count = bin_count + 1
               X_bin(bin_count) = X(i)
          end if
     end do
```

```
end subroutine walkers_in_bin_2d
```

./utils/write_outputs_direct.f90

```
subroutine write_outputs(output_combo,nAratios,nTerms,flow_type,output_file)

     implicit none

     ! Internal vars
     integer                                          :: funit,i
     character(len=1024)                              :: output_file

     ! Inputs
     integer                                          :: nAratios,nTerms
     double precision, dimension(1:nAratios,1:2)      :: output_combo
     character(len=1024)                              :: flow_type

     ! Make a filename. Messier than I'd like. Eh.

     funit = 53 ! Arbitrary

     open(funit,file=output_file)

          ! Write solution parameters.
          write(funit,*) nTerms
          write(funit,*) trim(flow_type)
          write(funit,*) nAratios       ! Or, more generally, the number of lines below this one

          do i=1,nAratios
               write(funit,"(ES26.17,ES26.17)") output_combo(i,1),output_combo(i,2)
          end do

     close(funit)

end subroutine write_outputs
```

./utils/write_outputs_mc.f90

```
subroutine write_outputs_mc(Ntrials,Pe,aratio,dt,nArray,nVars,array,filename)
! Not to be confused with the similarly named function when
! doing the exact calculation in inviscid setting.

implicit none
     integer                                          :: Ntrials,nArray,nVars
     double precision                                 :: Pe,aratio,dt
     double precision, dimension(1:nArray,1:nVars)    :: array
```

```
     character(len=1024)                                :: filename

     integer                                      :: funit,k

     funit = 53

     open(funit,file=filename)

          write(funit,*) Ntrials
          write(funit,*) Pe
          write(funit,*) aratio
          write(funit,*) dt
          write(funit,*) nArray

          do k=0,nArray-1
               write(funit,*) k*dt, array(k+1,1:nvars)
          end do

     close(funit)

end subroutine write_outputs_mc


 ./utils/zeroout.f90

subroutine zeroout(n,x)
! Zeroes the elements of x.
implicit none
     integer, intent(in)                             :: n
     double precision, dimension(n), intent(out)  :: x
     integer                                         :: i

     do i=1,n
          x(i) = 0.0d0
     end do
end subroutine zeroout
```

./computation/

./computation/Alpha_eval.f90

```
double precision function Alpha_eval(i,m,p)

     implicit none

     ! Input variables
     integer :: i,m,p

     ! Internal variables
     double precision, parameter   :: half = 0.5d0
     double precision, parameter   :: twooverpi = half/datan(1.0d0)

     Alpha_eval = twooverpi*(i-half)*(m-half)*(p-half)*(-1)**(i+m+p) &
                                        /dble( (i-m-p+half)*(i+m-p-half)*(i-m+p-half)*(i+m+p-3*h
end function Alpha_eval
```

./computation/Beta_tilde.f90

```
double precision function Beta_tilde(k,l,m,aratio)

     implicit none
     integer             :: k,l,m
     double precision    :: aratio

     double precision    :: pi,twooverpi, q, exp_q

     parameter(pi=datan(1.0d0))
     parameter(twooverpi = 2.0d0/pi)

     q = -pi/aratio
     exp_q = dexp(q)


     if ( (k .eq. m-l) .or. (k .eq. l-m) .or. (k .eq. l+m) ) then
          Beta_tilde = 0.0d0
     else
          Beta_tilde = -2.0d0/q * ( &
                              (exp_q**m - exp_q**(k+l))/(k+l-m) &
                              + (exp_q**l - exp_q**(k+m))/(k-l+m) &
                              + (exp_q**(l+m) - exp_q**k)/(k-l-m) &
                              + (1.0d0 - exp_q**(k+l+m))/(k+l+m) &
                              )
     end if
```

```
end function Beta_tilde


 ./computation/accumulate_moments_1d.f90

subroutine accumulate_moments_1d(tt_idx,ntt,nTot,X,Y,a,means,vars,skews,&
               kurts,nby,means_sl,vars_sl,skews_sl,kurts_sl)
! A subroutine to be used in the main loop of the Monte Carlo code.
! Calculates the moments for a specific time.
!
! This version is for 1d (channel) geometry.
!

implicit none

     ! Input arguments
     integer, intent(in)                                        :: tt_idx,ntt,nTot,nby
     double precision, dimension(nTot), intent(in)              :: X,Y
     double precision, intent(in)                               :: a
     double precision, dimension(ntt), intent(inout)            :: means,vars,skews,kurts
     double precision, dimension(ntt,nby), intent(inout)        :: means_sl,vars_sl,&
                                                                     skews_sl,kurts_sl



     ! Internal
     integer                                                    :: kb,bin_count,fi,li,idx,nbins
     double precision                                           :: bin_lo,bin_hi

     !
     ! Array to hold X values located in a certain bin, for a fixed round.
     ! The array size here is really just an upper bound.
     double precision, dimension(nTot)                              :: Xsorted
     integer, dimension(nTot)                                       :: Yidx,bin_idxs

     nbins = nby

     ! Set values to zero just in case they're not already.
     means(tt_idx) = 0.0d0
     vars(tt_idx) = 0.0d0
     skews(tt_idx) = 0.0d0
     kurts(tt_idx) = 0.0d0

     do kb=1,nby
          means_sl(tt_idx,kb) = 0.0d0
          vars_sl(tt_idx,kb) = 0.0d0
          skews_sl(tt_idx,kb) = 0.0d0
          kurts_sl(tt_idx,kb) = 0.0d0
     end do
```

```
!
! Calculate all the moments.
!

call moments(nTot,X,means(tt_idx),vars(tt_idx),skews(tt_idx),kurts(tt_idx))

if (nby .gt. 0) then
     ! Start by binning in Y.
     ! Enumerate the array of bins from 1,...,nby.
     call uniform_bins_idx(nTot,Y,-a,a,nby,Yidx)

     bin_idxs = Yidx

     ! Now sort the list of X positions based on their bin index.

     call sortpairs(nTot,X,Xsorted,bin_idxs,nbins)

     ! Finally, loop over the y and z bin indexes, calculating the moments for
     ! the corresponding subset of X values (which now are grouped together
     ! in the array Xsorted).

     fi = 0    ! First index of active subset
     li = 0    ! Last index of active subset


     do kb=1,nby

          idx = kb

          ! Get the bounds on the active subset,
          ! assuming bin_idxs has been sorted already.
          fi = li+1
          li = fi

          do while ( (bin_idxs(li) .eq. idx) )
               li = li + 1

               if (li .eq. (nTot+1)) then
                    ! If the lower index has passed the size of
                    ! the array, break out. We also should be at the last
                    ! index if we land in here.
                    ! Otherwise something went badly wrong.
                    go to 10
               end if
          end do

10        continue
```

```
                li = li-1 ! Necessary to decrement because of the bookkeeping.

                ! Calculate the moments of this subset of particles!
                call moments(li-fi+1,Xsorted(fi:li),means_sl(tt_idx,kb),vars_sl(tt_idx,kb),&
                             skews_sl(tt_idx,kb),kurts_sl(tt_idx,kb))

                ! If we've exhausted the entries of bin_idxs (i.e., there are more
                ! bins but they're empty), break out of the double loop.
                if (li .eq. nTot) then
                     go to 20
                end if
           end do

     end if


20   continue

end subroutine accumulate_moments_1d
```

./computation/accumulate_moments_2d.f90

```
subroutine accumulate_moments_2d(tt_idx,ntt,nTot,X,Y,Z,yl,yr,zl,zr,means,vars,skews,&
                kurts,nby,nbz,means_sl,vars_sl,skews_sl,kurts_sl)

! A subroutine to be used in the main loop of the Monte Carlo code.
! Calculates the moments for a specific time.
!
! This version is for 2d (duct/ellipse) geometry.
!


implicit none


     ! Input arguments
     integer, intent(in)                                        :: tt_idx,ntt,nTot,nby,nbz
     double precision, dimension(nTot), intent(in)              :: X,Y,Z
     double precision, intent(in)                               :: yl,yr,zl,zr
     double precision, dimension(ntt), intent(inout)            :: means,vars,skews,kurts
     double precision, dimension(ntt,nby,nbz), intent(inout)    :: means_sl,vars_sl,&
                                                                   skews_sl,kurts_sl

     ! Internal
     integer                                                    :: kb,jb,fi,li,idx,nbins,i
     double precision, dimension(nTot)                          :: Xsorted
```

```
integer, dimension(nTot)                                    :: Yidx,Zidx,bin_idxs

double precision bdistfun_rt

nbins = nby*nbz

! Set values to zero just in case they're not already.
! They will also default to in the situation where
! the bins are empty and they aren't handled in the loops below.
means(tt_idx) = 0.0d0
vars(tt_idx) = 0.0d0
skews(tt_idx) = 0.0d0
kurts(tt_idx) = 0.0d0
do kb=1,nby
     do jb=1,nbz
          means_sl(tt_idx,kb,jb) = 0.0d0
          vars_sl(tt_idx,kb,jb) = 0.0d0
          skews_sl(tt_idx,kb,jb) = 0.0d0
          kurts_sl(tt_idx,kb,jb) = 0.0d0
     end do
end do

!
! Calculate all the moments.
!

call moments(nTot,X,means(tt_idx),vars(tt_idx),skews(tt_idx),kurts(tt_idx))

!
! Next, moments across (y,z) bins.
!

if (nby .gt. 0) then
     ! Start by binning independently in Y and Z.
     ! Enumerate the array of bins from 1,...,nby*nbz in the usual way.
     call uniform_bins_idx(nTot,Y,yl,yr,nby,Yidx)
     call uniform_bins_idx(nTot,Z,zl,zr,nbz,Zidx)

     bin_idxs = Yidx + (nby-1)*(Zidx-1) + 1
     do i=1,nTot
          if ((bin_idxs(i) .lt. 1) .or. (bin_idxs(i) .gt. nby*nbz)) then
               write(*,*)
               write(*,*) "DANGER DANGER WILL ROBINSON"
               write(*,*) i,Y(i),Z(i),bdistfun_rt(Y(i),Z(i),0.5d0,0.2d0)
               write(*,*) bin_idxs(i),nby*nbz
               write(*,*) minval(bin_idxs),maxval(bin_idxs)
          end if
     end do
```

```
! Now sort the list of X positions based on their bin index.
call sortpairs(nTot,X,Xsorted,bin_idxs,nbins)


! Finally, loop over the y and z bin indexes, calculating the moments for
! the corresponding subset of X values (which now are grouped together
! in the array Xsorted).

fi = 0    ! First index of active subset
li = 0    ! Last index of active subset

do jb=1,nbz
     do kb=1,nby

          idx = (kb-1) + (nby-1)*(jb-1) + 1

          ! Get the bounds on the active subset,
          ! assuming bin_idxs has been sorted already.
          fi = li+1
          li = fi

          do while ( (bin_idxs(li) .eq. idx) )
               li = li + 1

               if (li .eq. (nTot+1)) then
                    ! If the lower index has passed the size of
                    ! the array, break out. We also should be at the last
                    ! index if we land in here.
                    ! Otherwise something went badly wrong.
                    go to 10
               end if
          end do

10        continue

          li = li-1 ! Necessary to decrement because of the bookkeeping.

          ! Calculate the moments of this subset of particles!
          call moments(li-fi+1,Xsorted(fi:li),means_sl(tt_idx,kb,jb),&
                         vars_sl(tt_idx,kb,jb),&
                         skews_sl(tt_idx,kb,jb),kurts_sl(tt_idx,kb,jb))

          ! If we've exhausted the entries of bin_idxs (i.e., there are more
          ! bins but they're empty), break out of the double loop.
          if (li .eq. nTot) then
               go to 20
          end if
```

```
            end do
        end do

    end if

20  continue



end subroutine accumulate_moments_2d
```

./computation/apply_advdiff1_chan.f90

```
subroutine apply_advdiff1_chan(n,xv,yv,Pe,dt,a,flow,reflector)
! Does the basic advection diffusion operation in the channel.
!
! This is what has been used up to this point (11 May 2016);
! essentially what's been done isan operator splitting where
! advection operator is done first, then diffusion operator.
!
! Arrays X,Y (dimension n)
! scalars Pe, dt, a,
! double precision function flow,
! subroutine reflector.
!

implicit none

    ! Inputs/outputs
    integer, intent(in)                                     :: n
    double precision, dimension(n), intent(inout)           :: xv,yv
    double precision, intent(in)                            :: Pe,dt,a

    ! Internal
    double precision, dimension(2,n)                        :: W
    double precision                                        :: mcvar
    integer                                                 :: i

    ! Interface necessary for the passed function and subroutine;
    ! only specifies the number and type of arguments.
    interface
        double precision function flow(p,q,r)
            double precision :: p,q,r
        end function flow

        subroutine reflector(p,q,r)
```

```
            double precision :: p,q,r
        end subroutine reflector
    end interface


    ! Generate the proper white noise in advance.
    mcvar = 2.0d0*dt
    call my_normal_rng(n,W(1,1:n),0.0d0,mcvar)
    call my_normal_rng(n,W(2,1:n),0.0d0,mcvar)

    do i=1,n
        ! Advection, then diffusion.
        xv(i) = xv(i) + Pe*flow(yv(i),-a,a)*dt + W(1,i)
        yv(i) = yv(i) + W(2,i)
        call reflector(yv(i),-a,a)
    end do

end subroutine apply_advdiff1_chan
```

./computation/apply_advdiff1_duct.f90

```
subroutine apply_advdiff1_duct(n,xv,yv,zv,Pe,dt,a,b, &
                    flow,reflector)
! Does the basic advection diffusion operation in the duct.
!
! This is what has been used up to this point (11 May 2016);
! essentially what's been done is an operator splitting where
! advection operator is done first, then diffusion operator.
!
! Arrays X,Y,Z (dimension n)
! scalars Pe, dt, a, b
! double precision function flow,
! subroutine reflector.
!

use mod_ductflow

implicit none

    ! Inputs/outputs
    integer, intent(in)                                 :: n
    double precision, dimension(n), intent(inout)       :: xv,yv,zv
    double precision, intent(in)                        :: Pe,dt,a,b

    ! Internal
    double precision, dimension(3,n)                    :: W
    double precision                                    :: mcvar
```

```
      integer                                                    :: i

      ! Interface necessary for the passed function and subroutine;
      ! only specifies the number and type of arguments.
      interface
           double precision function flow(p,q)
                double precision :: p,q
           end function flow

           subroutine reflector(p,q,r)
                double precision :: p,q,r
           end subroutine reflector
      end interface


      ! Generate the proper white noise in advance.
      mcvar = 2.0d0*dt
      call my_normal_rng(n,W(1,1:n),0.0d0,mcvar)
      call my_normal_rng(n,W(2,1:n),0.0d0,mcvar)
      call my_normal_rng(n,W(3,1:n),0.0d0,mcvar)

      do i=1,n
           ! Advection, then diffusion.

           xv(i) = xv(i) + Pe*flow(yv(i),zv(i))*dt + W(1,i)
           yv(i) = yv(i) + W(2,i)
           zv(i) = zv(i) + W(3,i)

           ! Call the generic reflection subroutine passed in.
           call reflector(yv(i),-a,a)
           call reflector(zv(i),-b,b)
      end do

end subroutine apply_advdiff1_duct


 ./computation/apply_advdiff1_ellipse.f90

subroutine apply_advdiff1_ellipse(nTot,xv,yv,zv,Pe,dt,a,b, &
                    flow,reflector,maxrefl)

! Does the basic advection diffusion operation in the ellipse.
!
! This is what has been used up to this point (11 May 2016);
! essentially what's been done is an operator splitting where
! advection operator is done first, then diffusion operator.
!
! Arrays X,Y,Z (dimension n)
```

```
! scalars Pe, dt, a, b
!
! double precision function flow,
! subroutine reflector.
!

use mtmod

implicit none

     ! Inputs/outputs
     integer, intent(in)                                          :: nTot,maxrefl
     double precision, dimension(nTot), intent(inout)             :: xv,yv,zv
     double precision, intent(in)                                 :: Pe,dt,a,b

     ! Internal
     double precision, dimension(3,nTot)                          :: W
     double precision                                             :: mcvar,yprev,zprev
     integer                                                      :: i

     ! Interface necessary for the passed function and subroutine;
     ! only specifies the number and type of arguments.
     interface
          double precision function flow(a1,a2,a3,a4)
               implicit none
               double precision :: a1,a2,a3,a4
          end function flow

          subroutine reflector(a1,a2,a3,a4,a5,a6,a7)
               implicit none
               double precision, intent(out) :: a1,a2
               double precision, intent(in)  :: a3,a4,a5,a6
               integer, intent(in)           :: a7
          end subroutine reflector
     end interface


     ! Generate the proper white noise in advance.
     ! Note in the ellipse that the variance of the white noise is the
     ! same in all directions because the nondimensionalization is 'isotropic'.
     mcvar = 2.0d0*dt

     call my_normal_rng(nTot,W(1,1:nTot),0.0d0,mcvar)
     call my_normal_rng(nTot,W(2,1:nTot),0.0d0,mcvar)
     call my_normal_rng(nTot,W(3,1:nTot),0.0d0,mcvar)

     do i=1,nTot
          ! Advection, then diffusion.
```

```
          xv(i) = xv(i) + Pe*flow(yv(i),zv(i),a,b)*dt + W(1,i)

          yprev = yv(i)
          zprev = zv(i)
          yv(i) = yv(i) + W(2,i)
          zv(i) = zv(i) + W(3,i)

          call reflector(yv(i),zv(i),yprev,zprev,a,b,maxrefl)
     end do

end subroutine apply_advdiff1_ellipse
```

./computation/apply_advdiff1_racetrack.f90

```
subroutine apply_advdiff1_racetrack(nTot,xv,yv,zv,Pe,dt,aratio,q, &
                    flow,reflector,maxrefl)

! Does the basic advection diffusion operation in the racetrack.
!
! This is what has been used up to this point (11 May 2016);
! essentially what's been done is an operator splitting where
! advection operator is done first, then diffusion operator.
!
! Arrays X,Y,Z (dimension n)
! scalars Pe, dt, a, b
!
! double precision function flow,
! subroutine reflector.
!

use mtmod

implicit none

     ! Inputs/outputs
     integer, intent(in)                                    :: nTot,maxrefl
     double precision, dimension(nTot), intent(inout)       :: xv,yv,zv
     double precision, intent(in)                           :: Pe,dt,aratio,q

     ! Internal
     double precision, dimension(3,nTot)                    :: W
     double precision                                       :: mcvar,yprev,zprev,y1,z1
     integer                                                :: i

     double precision bdistfun_rt

     ! Interface necessary for the passed function and subroutine;
```

```
     ! only specifies the number and type of arguments.
     interface
          double precision function flow(a1,a2,a3,a4)
               implicit none
               double precision :: a1,a2,a3,a4
          end function flow

          subroutine reflector(a1,a2,a3,a4,a5,a6,a7,a8,a9)
               implicit none
               double precision, intent(out) :: a1,a2
               double precision, intent(in)  :: a3,a4,a5,a6,a7,a8
               integer, intent(in)           :: a9
          end subroutine reflector
     end interface


     ! Generate the proper white noise in advance.
     ! Note in the ellipse that the variance of the white noise is the
     ! same in all directions because the nondimensionalization is 'isotropic'.
     mcvar = 2.0d0*dt

     call my_normal_rng(nTot,W(1,1:nTot),0.0d0,mcvar)
     call my_normal_rng(nTot,W(2,1:nTot),0.0d0,mcvar)
     call my_normal_rng(nTot,W(3,1:nTot),0.0d0,mcvar)

     do i=1,nTot
          ! Advection, then diffusion.
          xv(i) = xv(i) + Pe*flow(yv(i),zv(i),aratio,q)*dt + W(1,i)

          yprev = yv(i)
          zprev = zv(i)
          y1 = yv(i) + W(2,i)
          z1 = zv(i) + W(3,i)

          call reflector(yv(i),zv(i),y1,z1,yprev,zprev,aratio,q,maxrefl)
          if (bdistfun_rt(yv(i),zv(i),aratio,q) .lt. 0.0d0) then
               write(*,*) yv(i),zv(i),bdistfun_rt(yv(i),zv(i),aratio,q)
          end if
     end do

end subroutine apply_advdiff1_racetrack


 ./computation/apply_advdiff1_triangle.f90

subroutine apply_advdiff1_triangle(nTot,xv,yv,zv,Pe,dt,a, &
                   flow,reflector,nl,lls)
```

```
! Does the basic advection diffusion operation in the triangle.
!
! This is what has been used up to this point (11 May 2016);
! essentially what's been done is an operator splitting where
! advection operator is done first, then diffusion operator.
!
! Arrays X,Y,Z (dimension n)
! scalars Pe, dt, a
!
! double precision function flow,
! subroutine reflector.
!

use mtmod

implicit none

     ! Inputs/outputs
     integer, intent(in)                                        :: nTot
     double precision, dimension(nTot), intent(inout)           :: xv,yv,zv
     double precision, intent(in)                               :: Pe,dt,a
     integer, intent(in)                                        :: nl
     double precision, dimension(nl,3), intent(in)              :: lls

     ! Internal
     double precision, dimension(3,nTot)                        :: W
     double precision                                           :: mcvar,yprev,zprev
     integer                                                    :: i

     ! Interface necessary for the passed function and subroutine;
     ! only specifies the number of arguments and their type.
     interface
          double precision function flow(a1,a2,a3)
               implicit none
               double precision    :: a1,a2,a3
          end function flow

          subroutine reflector(a1,a2,a3,a4,a5,a6)
               implicit none
               double precision, intent(inout)               :: a1,a2
               double precision, intent(in)                  :: a3,a4
               integer, intent(in)                           :: a5
               double precision, dimension(a5,3), intent(in) :: a6
          end subroutine reflector
     end interface


     ! Generate the proper white noise in advance.
```

```
     ! Note in the ellipse that the variance of the white noise is the
     ! same in all directions because the nondimensionalization is 'isotropic'.
     mcvar = 2.0d0*dt

     call my_normal_rng(nTot,W(1,1:nTot),0.0d0,mcvar)
     call my_normal_rng(nTot,W(2,1:nTot),0.0d0,mcvar)
     call my_normal_rng(nTot,W(3,1:nTot),0.0d0,mcvar)

     do i=1,nTot
          ! Advection, then diffusion.
          xv(i) = xv(i) + Pe*flow(yv(i),zv(i),a)*dt + W(1,i)

          yprev = yv(i)
          zprev = zv(i)
          yv(i) = yv(i) + W(2,i)
          zv(i) = zv(i) + W(3,i)

          call reflector(yv(i),zv(i),yprev,zprev,nl,lls)
     end do

end subroutine apply_advdiff1_triangle
```

./computation/apply_advdiff2_chan.f90

```
subroutine apply_advdiff2_chan(n,xv,yv,Pe,dt,a,flow,reflector)
! Does the basic advection diffusion operation in the channel.
!
! This is what has been used up to this point (11 May 2016);
! essentially what's been done isan operator splitting where
! advection operator is done first, then diffusion operator.
!
! Arrays X,Y (dimension n)
! scalars Pe, dt, a,
! double precision function flow,
! subroutine reflector.
!

implicit none

     ! Inputs/outputs
     integer, intent(in)                                    :: n
     double precision, dimension(n), intent(inout)          :: xv,yv
     double precision, intent(in)                           :: Pe,dt,a

     ! Internal
     double precision, dimension(2,n)                       :: W
     double precision                                       :: mcvar,dthalf
```

```
     integer                                                    :: i

     ! Interface necessary for the passed function and subroutine;
     ! only specifies the number and type of arguments.
     interface
          double precision function flow(p,q,r)
               double precision :: p,q,r
          end function flow

          subroutine reflector(p,q,r)
               double precision :: p,q,r
          end subroutine reflector
     end interface


     ! Generate the proper white noise in advance.
     mcvar = 2.0d0*dt
     dthalf = dt/2.0d0

     call my_normal_rng(n,W(1,1:n),0.0d0,mcvar)
     call my_normal_rng(n,W(2,1:n),0.0d0,mcvar)

     do i=1,n
          ! Half advection, diffusion, half advection.
          xv(i) = xv(i) + Pe*flow(yv(i),-a,a)*dthalf

          xv(i) = xv(i) + W(1,i)
          yv(i) = yv(i) + W(2,i)
          call reflector(yv(i),-a,a)

          xv(i) = xv(i) + Pe*flow(yv(i),-a,a)*dthalf
     end do

end subroutine apply_advdiff2_chan
```

./computation/asymp_st_channel_moments.f90

```
! Functions for the short-time asymptotics of the moments in the channel.

double precision function asymp_st_channel_m2(Pe,t)
implicit none
! Be careful; we've done a pre-division where the Peclet number
! cancels on numerator and denominator when calculating skewness.
!
! The result here will NOT be correct if we only seek m2.

     double precision              :: Pe,t
```

```
     double precision, parameter   :: rtpi = dsqrt(4.0d0*datan(1.0d0))


     asymp_st_channel_m2 = 2.0d0*t + Pe**2*((4.0d0/45.0d0)*t**2 - (4.0d0/9.0d0)*t**3 &
                    +(128.0d0/(105.0d0*rtpi))*t**3.5d0 -(1.0d0/3.0d0)*t**4)


end function asymp_st_channel_m2

double precision function asymp_st_channel_m3(Pe,t)
implicit none
! Be careful; we've done a pre-step where the Peclet number
! cancels on numerator and denominator when calculating skewness.
!
! The result here will NOT be correct if we only seek m3.

     double precision              :: Pe,t

     double precision, parameter   :: rtpi = dsqrt(4.0d0*datan(1.0d0))


     asymp_st_channel_m3 = Pe**3*((-16.0d0/945.0d0)*t**3 + (16.0d0/45.0d0)*t**4 &
                    -(256.0d0/(105.0d0*rtpi))*t**4.5d0 + (5.0d0/2.0d0)*t**5 &
                    -(14464.0d0/(3465.0d0*rtpi))*t**5.5d0 + (14.0d0/15.0d0)*t**6)

end function asymp_st_channel_m3


 ./computation/impose_reflective_BC_ellipse.f90

subroutine impose_reflective_BC_ellipse(yf,zf,y0,z0,a,b,maxrefl)
! Impose reflective boundaries for the ellipse.
! (yf,zf) is the position after taking a step (corrected on output),
! (y0,z0) is the previous position.

implicit none
     double precision, intent(in)       :: y0,z0,a,b
     double precision, intent(inout)    :: yf,zf
     integer, intent(in)                :: maxrefl
     double precision                   :: distsq
     double precision, dimension(2)     :: v1,nhat

     integer                            :: nrefl,mmr
     logical                            :: goodsol
     double precision                   :: s,yc,zc,yold,zold
```

```
! Idiot-proofing
mmr = max(maxrefl,4)


distsq = (yf/a)**2 + (zf/b)**2
nrefl = 0

yold = y0
zold = z0

do while ( (distsq .gt. 1.0d0) .and. (nrefl .lt. mmr) )
     ! Find the point
     call ell_refl_ssols(yold,yf,zold,zf,a,b,s,goodsol)

     if (goodsol) then
          ! Find the coordinates of intersection.
          yc = yold + s*(yf-yold)
          zc = zold + s*(zf-zold)

          ! Construct the vector going out of the domain.
          v1(1) = yf-yc
          v1(2) = zf-zc

          ! Construct the outward normal vector.
          nhat(1) = 2.0d0*yc/(a**2)
          nhat(2) = 2.0d0*zc/(b**2)
          call normalize(2,nhat)

          call reflect(2,v1,nhat)

          ! Get new point an update the old point if it's needed.
          yold = yc
          zold = zc

          yf = yc + v1(1)
          zf = zc + v1(2)

     else
          ! Send the thing back to where it started. Ugly, but it works.
          yf = y0
          zf = z0
     end if

     ! Update the distance of the new point and the number of reflections.
     distsq = (yf/a)**2 + (zf/b)**2
     nrefl = nrefl + 1
end do
```

```
     ! Last line of defense.
     if ((nrefl .eq. mmr) .and. (distsq .gt. 1.0d0)) then
          yf = yc
          zf = zc
     end if


end subroutine impose_reflective_BC_ellipse

!
! ----------------------------
!

subroutine ell_refl_ssols(y0,yf,z0,zf,a,b,sol,flag)
implicit none
     double precision, intent(in)  :: y0,yf,z0,zf,a,b
     double precision, intent(out) :: sol
     logical, intent(out)          :: flag

     double precision              :: discrim,denom,sol1,sol2,numterm1,numterm2

     double precision ell_refl_discrim

     discrim = ell_refl_discrim(y0,yf,z0,zf,a,b)

     if (discrim .lt. 0) then
          sol1 = 0.0d0
          sol2 = 0.0d0
          flag = .false.
     else
          denom = b**2*(yf-y0)**2 + a**2*(zf-z0)**2
          numterm1 = b**2*y0*(y0-yf) + a**2*z0*(z0-zf)
          numterm2 = a*b*dsqrt(discrim)

          sol1 = (numterm1 + numterm2)/denom
          sol2 = (numterm1 - numterm2)/denom

          flag = .false.

          if ((sol1 .ge. 0.0d0) .and. (sol1 .le. 1.0d0)) then
               sol = sol1
               flag = .true.
          end if

          if ((sol2 .ge. 0.0d0) .and. (sol2 .le. 1.0d0)) then
               if (flag) then
                    ! Both solutions are in [0,1]!
                    ! This seems to happen when the point is on the
```

```
                    ! boundary. Take the larger time.
                    sol = max(sol1,sol2)
                    flag = .true.
               else
                    ! All is well, only one solution.
                    sol = sol2
                    flag = .true.
               end if
          end if

     end if

end subroutine ell_refl_ssols

!
! ----------------------------
!

double precision function ell_refl_discrim(y0,yf,z0,zf,a,b)

implicit none
     double precision, intent(in)  :: y0,yf,z0,zf,a,b

     ell_refl_discrim= b**2*(yf-y0)**2 + a**2*(zf-z0)**2 - (yf*z0 - y0*zf)**2

end function ell_refl_discrim
```

./computation/impose_reflective_BC_polygon.f90

```
subroutine impose_reflective_BC_polygon(p1y,p1z,p0y,p0z,nl,lls)
! Imposes reflective boundary
! conditions for the Monte Carlo simulation
! for a general (convex) polygonal geometry.
!
! The boundaries are specified as a set of
! linear equations; the interior is described
! as when all of them are positive.
!
! Boundary points are where any of them are zero.
!
! Inputs:
!
! double precision,                    :: p0y,p0z
! integer                              :: nl
! double precision, dimension(nl,3)    :: lls
!
! Input/output:
```

```
!
! double precision, dimension(2)        :: p1
!

!use mod_triangle_bdry

implicit none
     double precision, intent(in)                    :: p0y,p0z
     double precision, intent(inout)                 :: p1y,p1z
     integer, intent(in)                             :: nl
     double precision, dimension(nl,3), intent(in)   :: lls

     double precision, dimension(2)                  :: pb,mp,p0,p1
     double precision, dimension(3)                  :: tempv,line
     double precision, dimension(nl)                 :: bvals
     logical, dimension(nl)                          :: bcond
     integer, dimension(nl)                          :: bidxs
!     double precision, dimension(nl)                  :: ctimes
     integer                                         :: nbi,bidx,i,minidx
     double precision                                :: minct,ct

     double precision crosstime

     p0(1) = p0y
     p0(2) = p0z
     p1(1) = p1y
     p1(2) = p1z

     mp(1) = p1y
     mp(2) = p1z

     tempv(1) = 1.0d0
     tempv(2) = mp(1)
     tempv(3) = mp(2)

     call matvec(nl,3,lls,tempv,bvals)
     bcond = (bvals .lt. 0.0d0)


     ! Loop until there are no boundary crossings.
     do while (any(bcond))
          ! Find which boundaries have been crossed.
          call findcond(nl,bcond,nbi,bidxs)

          ! Find the time of crossing on each crossed
          ! boundary. Take the boundary crossed first.

          line = lls(bidxs(1),:)
```

```
            minct = crosstime(pb,mp,line)

            bidx = bidxs(1)
            do i=2,nbi
                 line = lls(bidxs(i),:)
                 ct = crosstime(pb,mp,line)
                 if (ct .lt. minct) then
                      minct = ct
                      bidx = bidxs(i)
                 end if
            end do

            ! Reflect across this boundary.
            line = lls(bidx,:)
            call reflector(pb,mp,line)

            tempv(2) = mp(1)
            tempv(3) = mp(2)

            ! Re-evaluate the new position.
            call matvec(nl,3,lls,tempv,bvals)
            bcond = (bvals .lt. 0.0d0)

       end do

       p1y = tempv(2)
       p1z = tempv(3)

  end subroutine impose_reflective_BC_polygon
  !
  ! ------------------------------------
  !
  double precision function crosstime(p,q,l)
  ! Calculates the time of intersection through line l
  ! in a parameterized path going from point p to q.
  implicit none
       double precision, dimension(2), intent(in)   :: p,q
       double precision, dimension(3), intent(in)   :: l

       crosstime = -(l(1) + l(2)*p(1) + l(3)*p(2))/(l(2)*(q(1)-p(1))+l(3)*(q(2)-p(2)))

  end function crosstime
  !
  ! ------------------------------------
  !
  subroutine reflector(p0,p1,l)
  !
  ! Reflects the particle that would have gone from p0 to p1
```

```
! across the line l. Should not end up in this subroutine
! unless this actually happens.
!
! On output, the points are changed to
!
! p0: point of intersection with l
! p1: position after reflection; p1 = p0+v for a vector v.
!
implicit none
     double precision, dimension(2), intent(inout)     :: p0,p1
     double precision, dimension(3), intent(in)        :: l

     double precision, dimension(2)                    :: gradl,pb,v
     double precision                                  :: s

     double precision crosstime

     gradl(1) = l(2)
     gradl(2) = l(3)

     ! Find the time and location of intersection, take the component
     ! of the vector that's outside the domain
     s = crosstime(p0,p1,l)
     pb = p0 + s*(p1-p0)
     v = p1 - pb

     ! Reflect this vector component across the plane
     call reflect(2,v,gradl)

     ! Update p1 based on this reflection.
     p1 = pb + v
     p0 = pb

end subroutine reflector
```

./computation/impose_reflective_BC_racetrack.f90

```
subroutine impose_reflective_BC_racetrack(yout,zout,y1,z1,y0,z0,aratio,q,maxrefl)
! Impose reflective boundaries for the racetrack.
! (yf,zf) is the position after taking a step,
! (y0,z0) is the previous position.

implicit none
     double precision, intent(in)       :: y0,z0,aratio,q
     double precision, intent(in)       :: y1,z1
     double precision, intent(out)      :: yout,zout
     integer, intent(in)                :: maxrefl
```

```
double precision                    :: bdf,l
double precision, dimension(2)      :: v1,nhat

integer                             :: nrefl,mmr
logical                             :: flag
double precision                    :: s,yc,zc,yold,zold,yf,zf


double precision bdistfun_rt

l = aratio

! Idiot-proofing
mmr = max(maxrefl,4)


bdf = bdistfun_rt(y1,z1,aratio,q)
nrefl = 0

yold = y0
zold = z0
yf = y1
zf = z1

flag = ((bdf .lt. 0.0d0) .and. (nrefl .lt. mmr))

do while ( flag )

     ! Find the point of intersection.
     ! Parameterize the line connecting (y0,z0) to (yf,zf),
     ! calculate a double s indicating point of intersection.
     ! Essentially a 1D calculation, should be able to do
     ! a couple iterations of Newton's method to capture.
     call calc_intersection_pt_rt(yold,zold,yf,zf,aratio,q,s,.false.)

     ! Find the coordinates of intersection.
     yc = yold + s*(yf-yold)
     zc = zold + s*(zf-zold)

     ! Construct the component of the vector going out of the domain.
     v1(1) = yf-yc
     v1(2) = zf-zc

     ! Construct the outward normal vector
     call bdistfun_rt_grad(yc,zc,aratio,q,nhat)


     call reflect(2,v1,nhat)
```

```
          ! Get new point an update the old point if it's needed.
          yold = yc
          zold = zc

          yf = yc + v1(1)
          zf = zc + v1(2)

          ! Update the distance of the new point and the number of reflections.
          bdf = bdistfun_rt(yf,zf,aratio,q)

          nrefl = nrefl + 1


          flag = ((bdf .lt. 0.0d0) .and. (nrefl .lt. mmr))

     end do


     ! Last line of defense.
     if (bdf .lt. 0.0d0) then
          write(*,*) bdf
     end if

     if ((nrefl .eq. mmr) .and. (bdf .lt. 0.0d0)) then
!          write(*,*) "moo"
          yout = y0
          zout = z0
     end if

     yout = yf
     zout = zf


end subroutine impose_reflective_BC_racetrack

!
! ----------------------------
!
subroutine calc_intersection_pt_rt(y0,z0,yf,zf,aratio,q,s,diagnose)
! Calculates the intersection with the boundary
! assuming the starting and ending points are in the interior
! and exterior, appropriately.
!
! Essentially does a few iterations of bisection followed by
! a few iterations of Newton's method.
!
```

```
implicit none
     double precision, intent(in)  :: y0,z0,yf,zf,aratio,q
     double precision, intent(out) :: s
     logical, intent(in)           :: diagnose

     integer, parameter            :: nbi=10 ! Number of bisection iterations
     integer, parameter            :: mnni=5 ! Max number of Newton iterations

     double precision, parameter   :: reltol = 1.0d-4  ! Relative tolerance for cv.
     double precision, parameter   :: abstol = 1.0d-8  ! Absolute tolerance for cv.

     double precision              :: tol
     double precision              :: yl,yr,zl,zr,yc,zc,sl,sr,sc
     double precision              :: vl,vr,vc
     integer                       :: nni,i
     logical                       :: flag

     double precision bdistfun_rt,dderiv_rt

     nni = 0

     sl = 0.0d0
     sr = 1.0d0
     sc = 0.5d0

     yl = y0
     zl = z0
     yr = yf
     zr = zf

     call lininterp_rt(yl,yr,sl,sr,sc,yc)
     call lininterp_rt(zl,zr,sl,sr,sc,zc)


     vl = bdistfun_rt(yl,zl,aratio,q)
     vr = bdistfun_rt(yr,zr,aratio,q)
     vc = bdistfun_rt(yc,zc,aratio,q)

     tol = min(reltol*(dabs(vl)+dabs(vr)),abstol)

     if (diagnose) then
          write(*,*) ""
     end if

     do i=1,nbi
          ! Bisection iteration
          if (vc*vl .lt. 0.0d0) then
               if (diagnose) then
```

```
                write(*,*) "replace right end point"
                write(*,*) sl,sc,sr
                write(*,*) vl,vc,vr
            end if
            vr = vc
            sr = sc
            yr = yc
            zr = zc
            sc = (sl + sr)/2.0d0


        else

            if (diagnose) then
                write(*,*) "replace left end point"
                write(*,*) sl,sc,sr
                write(*,*) vl,vc,vr
            end if
            vl = vc
            sl = sc
            yl = yc
            zl = zc

            sc = (sl + sr)/2.0d0

        end if

        call lininterp_rt(yl,yr,sl,sr,sc,yc)
        call lininterp_rt(zl,zr,sl,sr,sc,zc)
        vc = bdistfun_rt(yc,zc,aratio,q)

!         write(*,*) dabs(vc),tol,yc,zc,sc
        if (diagnose) then
            write(*,*) i,sc,vc,tol
        end if
    end do

    if (diagnose) then
        write(*,*) "end bisection, begin newton"
    end if

    flag = ((dabs(vc) .gt. tol) .and. (nni .le. mnni))
    do while (flag)
        ! Newton iteration.
        sc = sc - bdistfun_rt(yc,zc,aratio,q)/dderiv_rt(y0,yf,z0,zf,aratio,q,sc)

        call lininterp_rt(yl,yr,sl,sr,sc,yc)
        call lininterp_rt(zl,zr,sl,sr,sc,zc)
```

```
          vc = bdistfun_rt(yc,zc,aratio,q)

          nni = nni + 1
          if (diagnose) then
               write(*,*) nni,sc,vc,tol
          end if
!          write(*,*) dabs(vc),tol,yc,zc,sc
          flag = ((dabs(vc) .gt. tol) .and. (nni .le. mnni))
     end do

     if (diagnose) then
          write(*,*) "end newton"
     end if

     ! The rootfinder is having some difficulty with the non-convex domains.
     ! If we get a solution outside of [0,1], hard limit it.
     s = max(sc,0.0d0)
     s = min(s,1.0d0)

end subroutine calc_intersection_pt_rt
!
! ------------------------------------------------
!
subroutine lininterp_rt(zl,zr,sl,sr,sc,zc)
! Linear interpolation.
implicit none
     double precision, intent(in)  :: zl,zr,sl,sr,sc
     double precision, intent(out) :: zc

     double precision    :: m

     m = (zr-zl)/(sr-sl)
     zc = zl + m*(sc-sl)
end subroutine lininterp_rt
!
!
!
double precision function dderiv_rt(y0,yf,z0,zf,aratio,q,s)
! Directional derivative for use in Newton's method above.
! Keep in mind y0,yf,z0,zf are parameters here; the
! derivative is essentially in the direction of the
! vector from (y0,z0) to (yf,zf), evaluated at the
! point s.
!
! The formula was calculated in mathematica.
!

implicit none
```

```
      double precision, intent(in)  :: y0,yf,z0,zf,aratio,q,s
      double precision              :: x,l

      l = aratio

      x = 0.0d0

      x = x + (2*(y0*(-y0 + yf) + q**2*z0*(-z0 + zf) + 2*l**2*q**2* &
      &       (y0**4 - y0**3*yf + 3*y0*yf*z0**2 + z0**3*(z0 - zf) + 3*y0**2*z0*(-2*z0 + zf)) + &
      &      l**4*(-2*y0**4 + 2*y0**3*yf - y0*(yf + 6*yf*z0**2) + z0*(q**2 - 2*z0**2)*(z0 - zf) +
      &         y0**2*(1 + 12*z0**2 - 6*z0*zf))))/(-1 + l**2*q**2)

      x = x + (2*s*((y0 - yf)**2 + q**2*(z0 - zf)**2 - &
      &    6*l**2*q**2*(y0**2 - z0*(yf + z0 - zf) - y0*(yf - 2*z0 + zf))* &
      &       (y0**2 + y0*(-yf - 2*z0 + zf) + z0*(yf - z0 + zf)) + &
      &      l**4*(6*y0**4 - 12*y0**3*yf - yf**2*(1 + 6*z0**2) + 2*y0*yf*(1 + 6*z0*(3*z0 - 2*zf))
      &         (q**2 - 6*z0**2)*(z0 - zf)**2 + y0**2*(-1 + 6*yf**2 - 6*(6*z0**2 - 6*z0*zf + zf**
      & (-1 + l**2*q**2)

      x = x + (12*l**2*(-l + q)*(l + q)*s**2*(y0**4 - 3*y0**3*yf + z0*(-3*yf**2 + (z0 - zf)**2)*(
      &      3*y0**2*(yf**2 - 2*z0**2 + 3*z0*zf - zf**2) - y0*yf*(yf**2 - 3*(3*z0**2 - 4*z0*zf +
      &      (-1 + l**2*q**2)

      x = x + (4*l**2*(l**2 - q**2)*s**3*(y0**4 - 4*y0**3*yf + yf**4 - 4*y0*yf*(yf**2 - 3*(z0 - z
      &      6*yf**2*(z0 - zf)**2 + &
      &      (z0 - zf)**4 + 6*y0**2*(yf + z0 - zf)*(yf - z0 + zf)))/(-1 + l**2*q**2)

      dderiv_rt = x
end function dderiv_rt


 ./computation/impose_reflective_BC_rect.f90

subroutine impose_reflective_BC_rect(z,lower,upper)
! Imposes reflective boundary
! conditions for the Monte Carlo simulation on channel
! geometry (or duct, if applied in each direction)
!
! For now, this assumes there is no double reflecting.
! This relies on a small enough dt that the likelihood
! is outlandishly small.

implicit none
      double precision z,lower,upper,residue

      do while ((z>upper) .or. (z<lower))

           if (z > upper) then
```

```
                residue = z-upper
                z = upper - residue

           else if (z < lower) then

                residue = lower-z
                z = lower + residue

           end if
      end do

end subroutine impose_reflective_BC_rect
!
! ------------------------------------------
!


 ./computation/matvec.f90

subroutine matvec(m,n,A,u,v)
!
! A simple matrix-vector multiplication subroutine.
!
! Inputs:
!
!    m,n: integers
!    A: double precision array, dimension(m,n)
!    u: double precision array, dimension(n)
!
! Outputs:
!
!    v: double precision array, dimension(m)
!

implicit none
     integer, intent(in)                          :: m,n
     double precision, dimension(m,n), intent(in) :: A
     double precision, dimension(n), intent(in)   :: u

     double precision, dimension(m), intent(out)  :: v

     integer                                      :: i,j

     do i=1,m
          v(i) = 0.0d0
          do j=1,n
               v(i) = v(i) + A(i,j)*u(j)
```

```
          end do
     end do

end subroutine matvec
```

./computation/moments.f90

```
subroutine moments(n,x,mean,var,skew,kurt)
! Combination function to compute the mean, variance,
! skewness, kurtosis of an array x.
!
! Adapted from Numerical Recipes.
!
implicit none

     ! In/out
     integer, intent(in)                           :: n
     double precision, dimension(1:n), intent(in) :: x
     double precision, intent(out)                 :: mean,var,skew,kurt

     ! Internal
     integer                             :: i
     double precision                    :: temp

     ! Handle degenerate cases.
     if (n .eq. 0) then
          mean = 0.0d0
          var = 0.0d0
          skew = 0.0d0
          kurt = 0.0d0
     else if (n .eq. 1) then
          mean = x(1)
          var = 0.0d0
          skew = 0.0d0
          kurt = 0.0d0
     else

     ! Usual algorithm, build the mean, then
     ! build the centralized statistics based off of that.
          mean = 0.0d0
          do i=1,n
               mean = mean + x(i)
          end do
          mean = mean/n

          var = 0.0d0
          skew = 0.0d0
```

```
            kurt = 0.0d0

            do i=1,n

                  temp = x(i) - mean
                  var = var + temp**2
                  skew = skew + temp**3
                  kurt = kurt + temp**4
            end do

            var = var/dble(n-1)
            skew = skew/(n*(var**1.5d0))
            kurt = kurt/(n*(var**2)) - 3.0d0

      end if

      if (var .eq. 0.0d0) then
            skew = 0.0d0
            kurt = 0.0d0
      end if

end subroutine moments
```

./computation/precalculate_Alpha.f90

```
subroutine precalculate_Alpha(Alpha,alphaMax)

      implicit none

      ! Input variables
      integer :: alphaMax

      ! Input/output variables
      double precision, dimension(1:alphaMax,1:alphaMax,1:alphaMax) :: Alpha

      ! Internal variables
      integer             :: i,m,p

      ! Functions
      double precision Alpha_eval

      ! This loop could possibly be optimized further, but it will probably never
      ! be a bottleneck, so it's a low priority.
      do i=1,alphaMax
            do m=1,alphaMax
                  do p=1,alphaMax
                        Alpha(i,m,p) = Alpha_eval(i,m,p)
```

```
                end do
            end do
        end do

end subroutine precalculate_Alpha


 ./computation/precompute_uvals.f90

subroutine precompute_uvals(ui,uj,u,ya,za,nTerms,idxlist,uij_vals)
! Precomputing array u, ya, and za, for use in bilinear interpolation
! to optimize function calls in the Monte Carlo iteration.
! Assumes we are working on the square [-1,1]x[-1,1].

implicit none
    ! Input
    integer                                 :: ui,uj,nTerms
    integer, dimension(nTerms)              :: idxlist
    double precision, dimension(nTerms)     :: uij_vals

    ! Input/Output
    double precision, dimension(ui,uj)      :: u
    double precision, dimension(ui)         :: ya
    double precision, dimension(uj)         :: za

    ! Internal
    integer                                 :: i,j

    ! Functions
    double precision u_duct

    ! ------------------------------------

    ! First construct the y,z arrays.
    if (.false.) then
        call uniform_nodes(ui,ya,-1.0d0,1.0d0)
        call uniform_nodes(uj,za,-1.0d0,1.0d0)
    else
        ! Chebyshev nodes. Be aware you need to use the general
        ! index locator for linear interpolation if you use this,
        ! which will slow down function evaluations.
        call padded_cheb_nodes(ui,ya,-1.0d0,1.0d0)
        call padded_cheb_nodes(uj,za,-1.0d0,1.0d0)
    end if


    ! Now compute the corresponding u values.
    do i=1,ui
```

```
          do j=1,uj
               u(i,j) = u_duct(ya(i),za(j),nTerms,idxlist,uij_vals)
          end do
     end do

end subroutine precompute_uvals
```

./computation/precompute_uvals_ss.f90

```
subroutine precompute_uvals_ss(a,b,aratio)
! Precomputing array u, ya, and za, for use in bilinear interpolation
! to optimize function calls in the Monte Carlo iteration.
! Assumes we are working on the square [-1,1]x[-1,1].

use mod_ductflow
implicit none
     ! Input
     double precision, intent(in)                    :: a,b,aratio


     ! Internal
     integer                                         :: i,j
     double precision, dimension(ui)                 :: yatta
     double precision, dimension(uj)                 :: zatta
     double precision, dimension(ui,uj)              :: udumb

     ! Functions
     double precision u_duct_ss

     ! ------------------------------------

     ! First construct the y,z arrays.
     if (.true.) then
          call uniform_nodes(ui,yatta,-a,a)
          call uniform_nodes(uj,zatta,-b,b)
     else
          ! Chebyshev nodes. Be aware you need to use the general
          ! index locator for linear interpolation if you use this,
          ! which will slow down function evaluations.
          call padded_cheb_nodes(ui,ya,-a,a)
          call padded_cheb_nodes(uj,za,-b,b)
     end if


     ya = yatta
     za = zatta
```

```
     ! Now compute the corresponding u values.
     do i=1,ui
          do j=1,uj
               udumb(i,j) = u_duct_ss(ya(i),za(j),nTerms,aratio)
          end do
     end do

     u_precomp = udumb

end subroutine precompute_uvals_ss
```

./computation/racetrack_bdist.f90

```
double precision function bdistfun_rt(y,z,aratio,q)
! Boundary distance function for the racetrack.
! If positive, in the interior, if negative, in
! the exterior, zero on the boundary.
!
! Essentially the Dirichlet flow solution.

implicit none
     double precision    :: y,z,aratio,q,l

     l = aratio

     bdistfun_rt = (y**4 - 6*y**2*z**2 + z**4)*l**2*(l**2-q**2)

     ! The q**2*z**2 here is not a typo
     bdistfun_rt = bdistfun_rt + (y**2 + q**2*z**2)*(1.0d0-l**4)

     bdistfun_rt = bdistfun_rt*(-1.0d0)/(1.0d0-q**2*l**2)

     bdistfun_rt = bdistfun_rt + 1.0d0

end function bdistfun_rt
!
!-------------------------------------------------------
!
subroutine bdistfun_rt_grad(y,z,aratio,q,vec)
! Outward unit normal gradient for the racetrack.
! Partial in y, then partial in z.
!

implicit none
     double precision, intent(in)                :: y,z,aratio,q
     double precision                            :: l
```

```
      double precision, dimension(2), intent(out)  :: vec

      l = aratio

      vec(1) = (-2*y*(-1 + 2*l**2*q**2*(y**2 - 3*z**2) + l**4*(1 - 2*y**2 + 6*z**2))) &
                              &/(-1 + l**2*q**2)

      vec(2) = (2*z*(2*l**4*(-3*y**2 + z**2) + q**2*(1 - l**4 + 6*l**2*y**2 - 2*l**2*z**2))) &
                              &/(-1 + l**2*q**2)

      call normalize(2,vec)

end subroutine bdistfun_rt_grad
```

./computation/u_channel.f90

```
double precision function u_channel(y,a,b)
! Calculate the channel flow velocity; the channel is in the interval
! a,b. This flow is guaranteed to be integral zero on [a,b] and
! second derivative (Laplacian) -1.

implicit none
      double precision y,a,b

      u_channel = 0.5d0*( (y - a)*(b - y) - 1.0d0/6.0d0*(b-a)**2 )

      ! Multiply by a factor of two for Laplacian -2.

      u_channel = 2.0d0*u_channel

end function u_channel
```

./computation/u_duct.f90

```
double precision function u_duct(y,z,nTerms,idxlist,uij_vals)
! Calculate the approximate value of the flow u(y,z), with precalculated
! Fourier coefficients Amn_vals on indices idxlist.
!
! This is the zero-average flow.

implicit none
      double precision                          :: y,z,pi,pisq,half
      integer                                   :: k,i,j,nTerms
      integer, dimension(1:nTerms,1:2)          :: idxlist
      double precision, dimension(1:nTerms)     :: uij_vals

      parameter(pi = 4.0d0*datan(1.0d0))
```

```
      parameter(pisq = pi**2)
      parameter(half = 0.5d0)

      ! Sum over index set.

      u_duct = 0.0d0
      do k=1,nTerms
           i = idxlist(k,1)
           j = idxlist(k,2)

           u_duct = u_duct + uij_vals(k)*( dcos((i-half)*pi*y)*dcos((j-half)*pi*z) &
                                - (-1)**(i+j)/(pisq*(i-half)*(j-half)) )
      end do

      !
      ! Extra factor of two to make the Laplacian -2.
      !

      u_duct = 2.0d0*u_duct

end function u_duct
```

./computation/u_duct_precomp.f90

```
double precision function u_duct_precomp(y,z)
! Calculate the approximate value of the flow u(y,z),
! having already precomputed the flow on a fine grid.
! Essentially this is a wrapper function for linear_interp_2d.
! All the arrays are held in the module mod_ductflow.
!

use mod_ductflow

implicit none

      double precision, intent(in)  :: y,z
      double precision linear_interp_2d

      u_duct_precomp = linear_interp_2d(ui, uj, u_precomp, ya, za, y, z)

end function u_duct_precomp
```

./computation/u_duct_ss.f90

```
double precision function u_duct_ss(y,z,nTerms,aratio)
! Calculate the approximate value of the flow u(y,z),
! using a single-series solution
```

```
! whose Laplacian is guaranteed -2 for any number of terms,
! but boundary conditions at the far walls are only met approximately.
!
! However, the degree of this is relatively insignificant, even
! for a reasonable number of terms, and there is
! great added benefit when later moving to calculate the
! moments of the flow.
!

implicit none
     double precision, intent(in)            :: y,z,aratio
     integer, intent(in)                     :: nTerms

     integer                                 :: k
     double precision                        :: q,yterm,zterm,betak
     double precision                        :: pi,absz

     parameter(pi = 4.0d0*datan(1.0d0))

     u_duct_ss = 0.0d0

     absz = dabs(z)

     do k = 1,nTerms

          q = (k-0.5d0)*pi

          ! The original equation needs to be shuffled around for it
          ! to be numerically stable; cosh(qz)/cosh(q/aratio) is apparently ill-behaved.
          yterm = 4.0d0*(-1)**k/(q**3)*dcos(q*y)
          zterm = (dexp(-q*(1.0d0/aratio + z)) + dexp(q*(-1.0d0/aratio + z)) )
          zterm = zterm/(1.0d0 + dexp(-2.0d0*q/aratio))

          betak = -4.0d0*aratio*dtanh(q/aratio)/(q**5)

          u_duct_ss = u_duct_ss + yterm*zterm - betak

     end do


     u_duct_ss = u_duct_ss + ( 1.0d0/3.0d0 - y**2 )


end function u_duct_ss
```

./computation/u_dummy.f90

```
double precision function u_dummy(y,z)
! Dummy flow to be used when the flow is not important.

implicit none
     double precision    :: y,z

     u_dummy = 0.0d0

end function u_dummy
```

./computation/u_ellipse.f90

```
double precision function u_ellipse(y,z,a,b)
! Calculate the pipe flow velocity.

implicit none
     double precision    :: y,z,a,b
     double precision    :: c,aratio

     aratio = a/b

     c = 0.5d0/(1.0d0 + aratio**2)

     u_ellipse = c*(0.5d0 - (y/a)**2 - (z/b)**2)

     ! Factor of two to make the Laplacian -2.
     u_ellipse = u_ellipse*2.0d0

end function u_ellipse
```

./computation/u_racetrack.f90

```
double precision function u_racetrack(y,z,aratio,q)
! Calculate the racetrack flow velocity.

implicit none
     double precision    :: y,z,aratio,q,l

     l = aratio

     u_racetrack = (y**4 - 6*y**2*z**2 + z**4)*l**2*(l**2-q**2)

     ! The q**2*z**2 here is not a typo.
     u_racetrack = u_racetrack + (y**2 + q**2*z**2)*(1.0d0-l**4)
```

```
      u_racetrack = u_racetrack*(-1.0d0)/(1.0d0-q**2*l**2)

      ! This is actually the lab-frame flow, but the central
      ! statistics are calculated, the sample mean is subtracted
      ! off anyways.
      u_racetrack = u_racetrack + 1.0d0

      ! Appropriate factor to make the Laplacian -2.
      u_racetrack = u_racetrack*(1.0d0+q**2)*(1.0d0-l**4)/(1.0d0-q**2*l**2)

end function u_racetrack
```

./computation/u_triangle.f90

```
double precision function u_triangle(y,z,a)
! Calculate the pipe flow velocity.

implicit none
      double precision    :: y,z,a,rt3

      rt3 = dsqrt(3.0d0)

      u_triangle = 1.0d0/(12.0d0*a)*(a+y)*(2*a+rt3*z-y)*(2*a-rt3*z-y)
      u_triangle = u_triangle - 3.0d0/20.0d0*a**2

      ! Factor of two to make the Laplacian -2.
      u_triangle = u_triangle*2.0d0

end function u_triangle
```

./modules/

./modules/mod_ductflow.f90

```
module mod_ductflow
     ! For arrays relevant to calculate the duct flow.


     integer                                          :: ui,uj,nTerms
     double precision, dimension(:), allocatable      :: ya,za
     double precision, dimension(:,:), allocatable    :: u_precomp


     ! Gridsize for the precalculation of the flow.
     !
     ! 256 in both directions corresponds to spatial step ~0.004,
     ! for uniform grid size.
     ! "Need" 10 points in the boundary layer to be fair,
     ! so this can resolve boundary layer fairly up to aratio = 0.04.
     !
     ! With non-uniform mesh (currently being used) this isn't as
     ! much of a problem, but I haven't attempted any analysis.
     !

     parameter(ui = 256)
     parameter(uj = 256)

     !
     ! Total number of terms to use in the series when
     ! precalculating u. Don't need much using the single series formulation.
     !

     parameter(nTerms = 256)

end module mod_ductflow
```

./modules/mod_duration_estimator.f90

```
module mod_duration_estimator
     ! For estimating how much longer the full program will
     ! take to complete based on history of timesteps.

     integer                                       :: count_rate  ! Count used in system_clock
     integer                                       :: mde_ntt,mde_ntc
     double precision, allocatable, dimension(:)   :: mde_dts
     double precision                              :: mde_pttc,mde_pttc_pretty
     character(len=3)                              :: mde_time_unit
```

```
    integer                                     :: mde_t1,mde_t2

contains
    double precision function predict_completion(mde_ntt,mde_ntc,mde_dts)
    ! Based on the total number of timesteps and number of timesteps
    ! completed, and a filled history mde_dts(1:mde_ntc),
    ! predict how much time is remaining assuming time per timestep is
    ! relatively consistent.

        integer                                 :: mde_ntt,mde_ntc
        double precision, dimension(1:mde_ntt)  :: mde_dts

        predict_completion = sum(mde_dts(1:mde_ntc)) * (mde_ntt-mde_ntc)/mde_ntc

    end function predict_completion

    subroutine mde_pretty_print_time(pttc,pretty_val,time_unit)
    ! Converts pttc into the smallest "standard" time units so that it is bounded by 100. The
    ! input pttc is assumed in units of seconds. The time units going out
    ! are also output ("sec","min","hrs","dys","yrs")

        double precision, intent(in)  :: pttc
        double precision, intent(out) :: pretty_val
        character(len=3)              :: time_unit

        pretty_val = pttc

        ! Seconds
        if (pretty_val .lt. 60.0d0) then
            time_unit = "sec"
            go to 10
        end if

        ! Minutes
        pretty_val = pretty_val/60.0d0
        if (pretty_val .lt. 60.0d0) then
            time_unit = "min"
            go to 10
        end if

        ! Hours
        pretty_val = pretty_val/60.0d0
        if (pretty_val .lt. 24.0d0) then
            time_unit = "hrs"
            go to 10
        end if
```

```
          ! Days
          pretty_val = pretty_val/24.0d0
          if (pretty_val .lt. 365.25d0) then
               time_unit = "day"
               go to 10
          end if

          ! Years
          pretty_val = pretty_val/365.25d0
          time_unit = "yrs"

10        continue
     end subroutine mde_pretty_print_time

end module mod_duration_estimator
```

./modules/mod_readbuff.f90

```
module mod_readbuff

     double precision, dimension(:), allocatable  :: readbuff_double


end module mod_readbuff
```

./modules/mod_time.f90

```
module mod_time
     ! For storing time-related arrays and parameters.

     double precision, dimension(:), allocatable  :: target_times
     integer                                      :: ntt

end module mod_time
```

./modules/mod_triangle_bdry.f90

```
module mod_triangle_bdry
     ! Defining the triangle boundary.

     integer, parameter  :: nl = 3
     double precision, dimension(nl,3), parameter :: lls=reshape( (/ 1.0d0, 1.0d0, 1.0d0, &
                  -0.5d0, -0.5d0, 1.0d0, &
                  dsqrt(3.0d0)/2.0d0, -dsqrt(3.0d0)/2.0d0, 0.0d0  /) , (/nl,3/) )

end module mod_triangle_bdry
```

./modules/mtfort90.f90

```
!-----------------------------------------------------------------------
! A C-program for MT19937: Real number version
!   genrand() generates one pseudorandom real number (double)
! which is uniformly distributed on [0,1]-interval, for each
! call. sgenrand(seed) set initial values to the working area
! of 624 words. Before genrand(), sgenrand(seed) must be
! called once. (seed is any 32-bit integer except for 0).
! Integer generator is obtained by modifying two lines.
!   Coded by Takuji Nishimura, considering the suggestions by
! Topher Cooper and Marc Rieffel in July-Aug. 1997.
!
! This library is free software; you can redistribute it and/or
! modify it under the terms of the GNU Library General Public
! License as published by the Free Software Foundation; either
! version 2 of the License, or (at your option) any later
! version.
! This library is distributed in the hope that it will be useful,
! but WITHOUT ANY WARRANTY; without even the implied warranty of
! MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
! See the GNU Library General Public License for more details.
! You should have received a copy of the GNU Library General
! Public License along with this library; if not, write to the
! Free Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA
! 02111-1307  USA
!
! Copyright (C) 1997 Makoto Matsumoto and Takuji Nishimura.
! When you use this, send an email to: matumoto@math.keio.ac.jp
! with an appropriate reference to your work.
!
!***********************************************************************
! Fortran translation by Hiroshi Takano.  Jan. 13, 1999.
!
!   genrand()      -> double precision function grnd()
!   sgenrand(seed) -> subroutine sgrnd(seed)
!                     integer seed
!
! This program uses the following non-standard intrinsics.
!   ishft(i,n): If n>0, shifts bits in i by n positions to left.
!               If n<0, shifts bits in i by n positions to right.
!   iand (i,j): Performs logical AND on corresponding bits of i and j.
!   ior  (i,j): Performs inclusive OR on corresponding bits of i and j.
!   ieor (i,j): Performs exclusive OR on corresponding bits of i and j.
!
!***********************************************************************
! Fortran version rewritten as an F90 module and mt state saving and getting
! subroutines added by Richard Woloshyn. (rwww@triumf.ca). June 30, 1999
```

```
 module mtmod
! Default seed
    integer(8), parameter :: defaultsd = 4357
! Period parameters
    integer, parameter :: N = 624, N1 = N + 1

! the array for the state vector
    integer(8), save, dimension(0:N-1) :: mt
    integer, save                      :: mti = N1

! Overload procedures for saving and getting mt state
    interface mtsave
      module procedure mtsavef
      module procedure mtsaveu
    end interface
    interface mtget
      module procedure mtgetf
      module procedure mtgetu
    end interface

 contains

!Initialization subroutine
  subroutine sgrnd(seed)
    implicit none
!
!       setting initial seeds to mt[N] using
!       the generator Line 25 of Table 1 in
!       [KNUTH 1981, The Art of Computer Programming
!          Vol. 2 (2nd Ed.), pp102]
!
    integer(8), intent(in) :: seed

    mt(0) = iand(seed,-1)

    do mti=1,N-1
      mt(mti) = iand(69069 * mt(mti-1),-1)
    enddo
!
    return
  end subroutine sgrnd

!Random number generator
  real(8) function grnd()
    implicit integer(a-z)

! Period parameters
```

```
    integer, parameter :: M = 397, MATA  = -1727483681
!                                   constant vector a
    integer, parameter :: LMASK =  2147483647
!                                   least significant r bits
    integer, parameter :: UMASK = -LMASK - 1
!                                   most significant w-r bits
! Tempering parameters
    integer, parameter :: TMASKB= -1658038656, TMASKC= -272236544

    dimension mag01(0:1)
    data mag01/0, MATA/
    save mag01
!                        mag01(x) = x * MATA for x=0,1

    TSHFTU(y)=ishft(y,-11)
    TSHFTS(y)=ishft(y,7)
    TSHFTT(y)=ishft(y,15)
    TSHFTL(y)=ishft(y,-18)

    if(mti.ge.N) then
!                       generate N words at one time
      if(mti.eq.N+1) then
!                            if sgrnd() has not been called,
        call sgrnd( defaultsd )
!                              a default initial seed is used
      endif

      do kk=0,N-M-1
          y=ior(iand(mt(kk),UMASK),iand(mt(kk+1),LMASK))
          mt(kk)=ieor(ieor(mt(kk+M),ishft(y,-1)),mag01(iand(y,1)))
      enddo
      do kk=N-M,N-2
          y=ior(iand(mt(kk),UMASK),iand(mt(kk+1),LMASK))
          mt(kk)=ieor(ieor(mt(kk+(M-N)),ishft(y,-1)),mag01(iand(y,1)))
      enddo
      y=ior(iand(mt(N-1),UMASK),iand(mt(0),LMASK))
      mt(N-1)=ieor(ieor(mt(M-1),ishft(y,-1)),mag01(iand(y,1)))
      mti = 0
    endif

    y=mt(mti)
    mti = mti + 1
    y=ieor(y,TSHFTU(y))
    y=ieor(y,iand(TSHFTS(y),TMASKB))
    y=ieor(y,iand(TSHFTT(y),TMASKC))
    y=ieor(y,TSHFTL(y))

    if(y .lt. 0) then
```

```
      grnd=(dble(y)+2.0d0**32)/(2.0d0**32-1.0d0)
    else
      grnd=dble(y)/(2.0d0**32-1.0d0)
    endif

    return
  end function grnd

!State saving subroutines.
! Usage:  call mtsave( file_name, format_character )
!    or   call mtsave( unit_number, format_character )
! where   format_character = 'u' or 'U' will save in unformatted form, otherwise
!         state information will be written in formatted form.
  subroutine mtsavef( fname, forma )

!NOTE: This subroutine APPENDS to the end of the file "fname".

    character(*), intent(in) :: fname
    character, intent(in)    :: forma

    select case (forma)
      case('u','U')
       open(unit=10,file=trim(fname),status='UNKNOWN',form='UNFORMATTED', &
            position='APPEND')
       write(10)mti
       write(10)mt

      case default
       open(unit=10,file=trim(fname),status='UNKNOWN',form='FORMATTED', &
            position='APPEND')
       write(10,*)mti
       write(10,*)mt

    end select
    close(10)

    return
  end subroutine mtsavef

  subroutine mtsaveu( unum, forma )

    integer, intent(in)    :: unum
    character, intent(in)  :: forma

    select case (forma)
      case('u','U')
       write(unum)mti
       write(unum)mt
```

```
      case default
       write(unum,*)mti
       write(unum,*)mt

      end select

    return
  end subroutine mtsaveu

!State getting subroutines.
! Usage:  call mtget( file_name, format_character )
!    or   call mtget( unit_number, format_character )
! where   format_character = 'u' or 'U' will read in unformatted form, otherwise
!         state information will be read in formatted form.
  subroutine mtgetf( fname, forma )

    character(*), intent(in) :: fname
    character, intent(in)    :: forma

    select case (forma)
      case('u','U')
       open(unit=10,file=trim(fname),status='OLD',form='UNFORMATTED')
       read(10)mti
       read(10)mt

      case default
       open(unit=10,file=trim(fname),status='OLD',form='FORMATTED')
       read(10,*)mti
       read(10,*)mt

    end select
    close(10)

    return
  end subroutine mtgetf

  subroutine mtgetu( unum, forma )

    integer, intent(in)    :: unum
    character, intent(in)  :: forma

    select case (forma)
      case('u','U')
       read(unum)mti
       read(unum)mt

      case default
```

```
      read(unum,*)mti
      read(unum,*)mt

     end select

   return
 end subroutine mtgetu

end module mtmod
```

REFERENCES

[1] G. Taylor, "Dispersion of soluble matter in solvent flowing slowly through a tube," *Proc. R. Soc. Lond. A*, vol. 219, pp. 186–203, 1953.

[2] R. Aris, "On the dispersion of a solute in a fluid flowing through a tube," *Proc. R. Soc. Lond. A*, vol. 235, pp. 67–77, 1956.

[3] P. C. Chatwin, "The approach to normality of the concentration distribution of a solute in a solvent flowing along a straight pipe," *J. Fluid Mech.*, vol. 43, no. 2, pp. 321–352, 1970.

[4] C. V. D. Broeck, "A stochastic description of longitudinal dispersion in uniaxial flows," *Physica*, vol. 112A, pp. 343–352, 1982.

[5] N. Barton, "On the method of moments for solute dispersion," *J. Fluid Mech.*, vol. 126, pp. 205–218, 1983.

[6] P. Chatwin and P. J. Sullivan, "The effect of aspect ratio on longitudinal diffusivity in rectangular channels," *Journal of Fluid Mechanics*, vol. 120, pp. 347–358, 1982.

[7] M. Latini and A. J. Bernoff, "Transient anomalous diffusion in poiseuille flow," *Journal of Fluid Mechanics*, vol. 441, pp. 399–411, 2001.

[8] R. Camassa, Z. Lin, and R. M. McLaughlin, "Exact evolution of the scalar variance in pipe and channel flow," *Commun. Math. Sci.*, vol. 8, no. 2, pp. 601–626, 2010.

[9] S. Vedel and H. Bruus, "Transient taylor-aris dispersion for time-dependent flows in straight channels," *J. Fluid Mech.*, vol. 691, pp. 95–122, 2012.

[10] H. A. Stone, A. D. Stroock, and A. Ajdari, "Engineering flows in small devices: microfluidics toward a lab-on-a-chip," *Annu. Rev. Fluid Mech.*, vol. 36, pp. 381–411, 2004.

[11] D. Dutta, A. Ramachandran, and D. T. Leighton Jr, "Effect of channel geometry on solute dispersion in pressure-driven microfluidic systems," *Microfluidics and Nanofluidics*, vol. 2, no. 4, pp. 275–290, 2006.

[12] A. Ajdari, N. Bontoux, and H. A. Stone, "Hydrodynamic dispersion in shallow microchannels: the effect of cross-sectional shape," *Analytical Chemistry*, vol. 78, no. 2, pp. 387–392, 2006. PMID: 16408918.

[13] N. Bontoux, A. Pepin, Y. Chen, A. Ajdari, and H. A. Stone, "Experimental characterization of hydrodynamic dispersion in shallow microchannels," *Lab Chip*, vol. 6, pp. 930–935, 2006.

[14] F. Gentile, M. Ferrari, and P. Decuzzi, "The transport of nanoparticles in blood vessels: The effect of vessel permeability and blood rheology," *Annals of Biomedical Engineering*, vol. 36, no. 2, pp. 254–261, 2008.

[15] M. Aminian, F. Bernardi, R. Camassa, and R. M. McLaughlin, "Squaring the circle: Geometric skewness and symmetry breaking for passive scalar transport in ducts and pipes," *Phys. Rev. Lett.*, vol. 115, p. 154503, Oct 2015.

[16] L. Grafakos, *Classical Fourier Analysis*, vol. 2. Springer, 2008.

[17] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, 1998.

[18] Nishimura, Takuji, "Mersenne twister in fortran," 1997.

[19] The HDF Group, "Hierarchical data format version 5," 2000-2010.

[20] B. Lapeyre, E. Pardoux, and R. Sentis, *Introduction to Monte Carlo methods for transport and diffusion equations*, vol. 6. Oxford University Press on Demand, 2003.

[21] V. G. Kulkarni, *Modeling and analysis of stochastic systems*. CRC Press, 2009.

[22] P. E. Kloeden, E. Platen, and H. Schurz, *Numerical solution of SDE through computer experiments*. Springer Science & Business Media, 2012.

[23] G. Marsaglia and T. A. Bray, "A convenient method for generating normal variables," *SIAM Review*, vol. 6, no. 3, pp. 260–264, 1964.