1 Table of Contents

1.1 Theory .................................................. 3
1.2 The nekRS Input Files .................................. 11
1.3 Commonly-Used Variables in nekRS .................. 26
1.4 Compiling nekRS ....................................... 29
1.5 Tutorials .............................................. 29
1.6 FAQs .................................................. 29
1.7 Plugins ............................................... 47
1.8 Just-in-time Compilation .............................. 51
1.9 Doxygen .............................................. 52
1.10 Glossary ............................................. 52
1.11 References ........................................... 54

Bibliography .................................................. 55

Index ......................................................... 57
nekRS is a computational fluid dynamics code developed at ANL, UIUC, and PSU. nekRS aims to leverage the present trend in GPU-based HPC systems to perform CFD on GPU-accelerated systems. By using the OCCA library’s unified API, nekRS can run on CPUs and on GPU-accelerated CPUs that support CUDA, HIP, or OpenCL.

This guide is intended to help new users get started with using nekRS, as well as serve as a reference for more advanced users. Because the Nek5000 code is somewhat of a predecessor to nekRS, some aspects of the current nekRS design are selected to enable faster translation of Nek5000 input files into nekRS input files. Throughout this documentation, all such Nek5000-oriented settings will be referred to as “legacy” settings. Because these Nek5000-oriented settings require proficiency in Fortran, structured text formats, and several additional input files, all new users are encouraged to adopt the nekRS-based problem setup.

We recommend working through this user guide in the order below. At the very least, please read The nekRS Input Files page before reading the FAQs page, as some necessary concepts are introduced in this order.

Note: This documentation is a work in progress, and will undergo big changes as more features are added to nekRS. Please open issues to track any missing information at the github repository here.
CHAPTER ONE

TABLE OF CONTENTS

1.1 Theory

This page introduces the governing equations and the numerical discretization at a high level. nekRS includes models for incompressible flow, a partially compressible low-Mach formulation, the Stokes equations, and the \( k-\tau \) RANS equations.

1.1.1 Incompressible Flow Model

The governing equations for conservation of mass, momentum, and energy for an incompressible fluid are

\[
\nabla \cdot \mathbf{u} = 0
\]

\[
\rho \left( \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \right) = -\nabla P + \nabla \cdot \tau + \rho \mathbf{f}
\]

\[
\rho C_p \left( \frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T \right) = \nabla \cdot (k \nabla T) + \dot{q}
\]

In these equations, \( t \) is time, \( \mathbf{u} \) is the velocity vector, \( \rho \) is the density, \( P \) is the pressure, \( \tau \) is the viscous stress tensor, \( \mathbf{f} \) is a force vector, \( C_p \) is the isobaric specific heat capacity, \( T \) is the temperature, \( k \) is the thermal conductivity, and \( \dot{q} \) is a volumetric heat source. If the viscosity is constant, the viscous stress tensor can be contracted to give

\[
\nabla \cdot \tau = \nabla^2 \mathbf{u}
\]

This is referred to as the “no-stress” formulation. In the general case for non-constant viscosity, the viscous stress tensor is given by the Navier-Stokes closure as

\[
\nabla \cdot \tau = \nabla \cdot \left[ \mu \left( \nabla \mathbf{u} + \nabla \mathbf{u}^T \right) \right]
\]

Non-Dimensional Formulation

It is often advantageous to solve these equations in non-dimensional form. Here, we introduce the non-dimensional form in as general a manner as possible, assuming the use of variable material properties for density, viscosity, specific heat capacity, and thermal conductivity that are functions of temperature, \( \rho = \rho(T), \mu = \mu(T), C_p = C_p(T), \) and \( k = k(T) \). For simplicity, the functional notation is omitted throughout.

Introduce the non-dimensional variables \( \mathbf{x} = \frac{x}{L}, \mathbf{u} = \frac{\mathbf{u}}{U}, t = \frac{t}{T}, \) and \( \mathbf{f} = \frac{\mathbf{f} \rho U^2}{\rho_0 U_0^2} \). For the material properties, we non-dimensionalize based on some reference temperature \( T_0 \), such that \( \rho^l = \frac{\rho}{\rho_0}, \mu^l = \frac{\mu}{\mu_0}, k^l = \frac{k}{k_0}, \) and \( C_p^l = \frac{C_p}{C_{p_0}} \). Here, a subscript of 0 is shorthand notation that indicates that the property is evaluated at \( T_0 \), such that \( k_0 \equiv k(T_0) \). Finally, for convection-dominated flows, the pressure is non-dimensionalized in terms of the dynamic pressure as \( P^l = \frac{P}{\rho_0 U_0^2} \).
Inserting these non-dimensional variables into the conservation of mass and momentum equations gives

\[
\frac{\partial u_i^\dagger}{\partial x_i^\dagger} = 0
\]

\[
\rho^\dagger \left( \frac{\partial u_i^\dagger}{\partial t^\dagger} + u_j^\dagger \frac{\partial u_i^\dagger}{\partial x_j^\dagger} \right) = -\frac{\partial P_i^\dagger}{\partial x_i^\dagger} + \frac{1}{Re} \frac{\partial r_{ij}^\dagger}{\partial x_j^\dagger} + \rho^\dagger f_i^\dagger
\]

In these equations, the \( \nabla \) are expanded to explicitly show that all derivatives are taken with respect to the nondimensional space variable \( x^\dagger \). \( Re \) is the Reynolds number

\[
Re \equiv \frac{\rho_0 UL}{\mu_0}
\]

To non-dimensionalize the energy conservation equation, use the previous non-dimensional variables in addition to a non-dimensional temperature, \( T^\dagger = \frac{T - T_0}{\Delta T} \), where \( \Delta T \) is a reference temperature rise relative to a baseline temperature \( T_0 \). The heat source is non-dimensionalized as \( \dot{q}^\dagger = \dot{m} C_p \frac{\Delta T}{L} \), which arises naturally from the simple formulation of bulk energy conservation of \( Q = \dot{m} C_p \Delta T \), where \( Q \) is a heat source (units of Watts) and \( \dot{m} \) is a mass flowrate.

Inserting these non-dimensional variables into the energy conservation equation gives

\[
\rho^\dagger C_p^\dagger \left( \frac{\partial T^\dagger}{\partial t^\dagger} + u_i^\dagger \frac{\partial T^\dagger}{\partial x_i^\dagger} \right) = \frac{1}{Pe} \frac{\partial}{\partial x_i^\dagger} \left( \kappa^\dagger \frac{\partial T^\dagger}{\partial x_i^\dagger} \right) + \dot{q}^\dagger
\]

where \( Pe \) is the Peclet number,

\[
Pe \equiv \frac{LU}{\alpha}
\]

and \( \alpha \) is the thermal diffusivity,

\[
\alpha \equiv \frac{k_0}{\rho_0 C_p 0}
\]

1.1.2 Low-Mach Partially-Compressible Model

1.1.3 Stokes Equations

1.1.4 RANS Models

The \textit{RANS} equations are derived from the conservation of mass, momentum, and energy equations by expressing each term in the equation as the sum of a mean and a fluctuation. Because nekRS is based on the incompressible flow model, all such averages (even for the energy equation) are based on the notion of \textit{Reynolds averaging}, where each field \( f \) is expressed as the sum of a time mean \( \overline{f} \) and a fluctuation, \( f' \),

\[
f(x, t) = \overline{f}(x) + f'(x, t)
\]

where the time averaged is defined as

\[
\overline{f} = \lim_{S \to \infty} \frac{1}{S} \int_0^{t+S} f(x, t) dt
\]

For compressible flows in which energy conservation affects density, the \textit{RANS} equations are instead derived with \textit{Favre averaging}, where each field \( f_i \) is expressed as the sum of a density-weighted time average \( \overline{f_i} \) and a fluctuation \( f_i' \). It is therefore an important distinction here that we only consider \textit{Reynolds averaging}, which leads to a simpler formulation of the \textit{RANS} energy conservation equation that the compressible flow case.
Inserting the above “Reynolds decomposition” for \( u \), \( P \), and \( T \) into the governing equations and averaging in time then gives the RANS equations. For the incompressible flow equations in Incompressible Flow Model, the RANS mass, momentum, and energy equations are

\[
\frac{\partial \overline{\rho u_i}}{\partial x_i} = 0
\]

\[
\rho \left( \frac{\partial \overline{u_i}}{\partial t} + \overline{u_j} \frac{\partial \overline{u_i}}{\partial x_j} \right) = - \frac{\partial \overline{P}}{\partial x_i} + \frac{\partial}{\partial x_j} \left( 2 \mu \overline{S}_{ij} \right) + \rho \overline{f}
\]

\[
\rho C_p \left( \frac{\partial \overline{T}}{\partial t} + \overline{u_i} \frac{\partial \overline{T}}{\partial x_i} \right) = \frac{\partial}{\partial x_i} \left( k \frac{\partial \overline{T}}{\partial x_i} \right) + \overline{q}
\]

where \( \overline{S}_{ij} \) is the mean strain rate tensor,

\[
\overline{S}_{ij} = \frac{1}{2} \left( \frac{\partial \overline{u_i}}{\partial x_j} + \frac{\partial \overline{u_j}}{\partial x_i} \right)
\]

The mass, momentum, and energy conservation equations have the same form as the instantaneous flow equations in Incompressible Flow Model except for the addition of another stress tensor to the momentum equation - \( \rho \overline{u_i' u_j'} \), and the addition of another heat flux vector to the energy equation - \( \rho C_p \overline{u_i T'} \). The stress term in the momentum equation is referred to as the Reynolds stress tensor; \( \rho \frac{\partial \overline{u_i' u_j'}}{\partial x_j} \) represents the time-averaged rate of momentum transfer due to turbulence. The heat flux term in the energy equation is referred to as the turbulent heat flux; \( \rho C_p \frac{\partial \overline{u_i' T'}}{\partial x_i} \) represents the time-averaged rate of energy addition due to turbulence. The objective of RANS models is to provide closures for the Reynolds stress tensor and turbulent heat flux vector in terms of the mean properties such that the time-averaged equations can be solved for the mean flow.

**Boussinesq Approximation**

The RANS models in nekRS are based on the Boussinesq eddy viscosity approximation, which assumes that the momentum flux that induces the Reynolds stresses shares the same functional form as the momentum flux that induces the molecular stresses. In other words, the Navier-Stokes closure that was used to relate the deviatoric stress tensor \( \tau_{ij} \) to the strain rate tensor,

\[
\tau_{ij} = \mu \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) - \frac{2}{3} \mu \frac{\partial u_i}{\partial x_i} \delta_{ij}
\]

is assumed applicable to the Reynolds stress tensor, but with instantaneous velocities replaced by mean velocities and the molecular viscosity replaced by the turbulent eddy viscosity \( \mu_T \),

\[
\rho \overline{u_i' u_j'} = \mu_T \left( \frac{\partial \overline{u_i}}{\partial x_j} + \frac{\partial \overline{u_j}}{\partial x_i} \right) - \frac{2}{3} \mu_T \frac{\partial \overline{u_i}}{\partial x_i} \delta_{ij} - \frac{2}{3} \rho k \delta_{ij}
\]

Here, \( k \) is the turbulent kinetic energy,

\[
k \equiv \frac{1}{2} \left( \overline{u_1^2} + \overline{u_2^2} + \overline{u_3^2} \right)
\]

The final term in the Boussinesq approximation for the Reynolds stress tensor simply ensures that the trace of the Reynolds stress tensor equals \( 2k \), because otherwise, for incompressible flows, the trace of the Reynolds stress tensor would be zero. Inserting the Boussinesq eddy viscosity model for the Reynolds stress tensor into the incompressible flow mean momentum equation then gives

\[
\rho \left( \frac{\partial \overline{u_i}}{\partial t} + u_j \frac{\partial \overline{u_i}}{\partial x_j} \right) = - \frac{\partial \overline{P}}{\partial x_i} + \frac{\partial}{\partial x_j} \left( 2 \left( \mu + \mu_T \right) \overline{S}_{ij} - \frac{2}{3} \rho k \delta_{ij} \right) + \rho \overline{f}
\]
In nekRS, as well as many other RANS codes, it is commonplace to combine the gradient of $P + \frac{2}{3} \rho k$ terms together into a single reduced pressure,

$$P_r \equiv P + \frac{2}{3} \rho k$$

such that the term proportional to $\rho k$ can be bundled into a single pressure gradient kernel,

$$- \frac{\partial P}{\partial x_i} - \frac{\partial}{\partial x_j} \left( \frac{2}{3} \rho k \delta_{ij} \right) \rightarrow - \frac{\partial P_r}{\partial x_i}$$

**Warning:** The pressure solution, available on the nrs->P object and written to output under the name “pressure,” represents this reduced pressure. To obtain $P$, you should subtract $\frac{2}{3} \rho k$ from nrs->P.

**Turbulent Prandtl Number**

Closure for the turbulent heat flux is typically motivated from considerations of the analogy between momentum and energy transfer; while the Boussinesq approximation was used to introduce a relationship between the Reynolds stress tensor $\rho u'_i u'_j$ in terms of the mean strain rate, the turbulent heat flux is assumed proportional to the mean temperature gradient via a gradient diffusion approximation,

$$\rho C_p u'_i T' = k_T \frac{\partial T}{\partial x_i}$$

where $k_T$ is the turbulent conductivity. $k_T$ is related to $\mu_T$, the turbulent momentum diffusivity, by the turbulent Prandtl number $P_{RT}$,

$$P_{RT} \equiv \frac{\nu_T}{\alpha_T}$$

where $\nu_T \equiv \mu_T/\rho \omega$ and $\alpha_T$ is the turbulent thermal diffusivity,

$$\alpha_T \equiv \frac{k_T}{\rho C_p}$$

Inserting this gradient diffusion approximation into the incompressible flow mean energy equation then gives

$$\rho C_p \left( \frac{\partial T}{\partial t} + u'_i \frac{\partial T}{\partial x_i} \right) = \frac{\partial}{\partial x_i} \left[ \left( k + \frac{\mu_T}{P_{RT} C_p} \right) \frac{\partial T}{\partial x_i} \right] + \overline{q}$$

**The $k-\tau$ Model**

nekRS uses the $k-\tau$ turbulence model to close the mean flow equations [Kalitzin]. Because the $k-\epsilon$, $k-\omega$, and $k-\omega$ SST models tend to dominate the RANS space, extra discussion is devoted here to motivating the use of this particular model. Because $P_{RT}$ is typically taken as a constant, often 0.90 [Wilcox], the objective of incompressible flow RANS models is to compute the eddy viscosity and $k$ needed to close the mean momentum and energy equations.

**Note:** Take care not to confuse the inverse of the specific dissipation rate, $\tau$, with the deviatoric molecular stress tensor, which is also represented here as $\tau$ due to convention.

The $k-\tau$ model is a modification of the standard $k-\omega$ turbulence model that bases the second transport equation on the inverse of the specific dissipation rate $\omega$,

$$\tau \equiv \frac{1}{\omega}$$

rather than the on $\omega$. The $k-\tau$ model attempts to retain two important features of the $k-\omega$ model -
1. Good predictions for flows with adverse pressure gradients and separation, and
2. Reasonable prediction of boundary layers and near-wall behavior without wall functions or special low-\(Re_T\) treatments.

These two aspects contribute to better predictions of complex flows with reduced numerical complexity associated with wall functions or damping functions that can cause stiff behavior [Kok] and inaccurate flow predictions. By introducing the definition of \(\tau \equiv 1/\omega\), the \(k-\tau\) model attempts to improve upon the \(k-\omega\) model in two main ways -

1. Simplify wall boundary conditions for the second transport equation, and
2. Bound the source terms in the second transport equation in near-wall regions.

As \(y \to 0\), \(\omega \to y^{-2}\), while \(k \to 0\) [Kok]. Therefore, while \(\omega\) is infinite at walls, \(\tau\) is zero. Traditionally, this singular behavior in \(\omega\) was treated by applying “rough wall” boundary conditions to \(\omega\) with the wall roughness set to a “small enough” value to simulate a hydraulically smooth wall [Kok]. However, this ad hoc approach retains a strong dependence on the near-wall mesh resolution, often requiring prohibitively fine elements to accurately predict boundary layer properties [Kalitzin]. And, such an approach retains near-singular behavior in the first and second derivatives of \(\omega\). Applying a zero boundary condition to \(\tau\) on solid walls is comparatively trivial.

With regards to the second point, the \(\omega\) transport equation contains a source term proportional to \(\omega^2\); because \(\omega \to y^{-2}\) as \(y \to 0\), this source term displays singular behavior as \(y \to 0\). Singular behavior of the source terms can result in large numerical errors and stiffness that negatively affects the convergence of the computational solution. Conversely, all source terms in the \(\tau\) equation are bounded near walls [Kok].

With this motivation, the \(k\) and \(\tau\) equations are described next. A slightly lengthier description is provided for each in order to give greater context to the genesis of this model.

The \(k\) Equation

The \(k\) equation is a model version of the true \(k\) equation. The true \(k\) equation is derived by taking the trace of the Reynolds stress equation, a process that is itself motivated by recognition that the trace of the Reynolds stress tensor is equal to \(2k\),

\[ \overline{u_i' u_i'} = 2k \]

The true \(k\) equation contains terms that depend on the mean flow velocity, the turbulent kinetic energy, and the dissipation, in addition to more exotic terms such as \(\overline{u_i' u_i' u_j' j} \) and \(\overline{P' u_j'}\). These additional fluctuating terms do not bring the true \(k\) equation any closer to a tractable solution, so Prandtl introduced a \(\partial k/\partial x_j\) gradient diffusion approximation for the turbulent transport and pressure diffusion terms \((\frac{1}{2} \rho u_i' u_i' u_j' j + \overline{P' u_j'})\) with a diffusion coefficient of \(\mu_T/\sigma_k\), where \(\sigma_k\) is a constant [Wilcox]. With this gradient diffusion model, the true \(k\) equation is simplified to a tractable model \(k\) equation [Launder],

\[ \frac{\partial (\rho k)}{\partial t} + \nabla \cdot (\rho k \overline{u}) = \nabla \cdot \left[ \left( \mu + \frac{\mu_T}{\sigma_k} \right) \nabla k \right] + P - \rho \epsilon \]

where \(P\) is the production of turbulent kinetic energy by velocity shear,

\[ P \equiv \rho \overline{u_i' u_i'} \frac{\partial \overline{u_j}}{\partial x_j} \]

and \(\epsilon\) is the dissipation per unit mass,

\[ \epsilon \equiv \nu \frac{\partial u_i'}{\partial x_j} \frac{\partial u_j'}{\partial x_j} \]

The production term represents the rate at which energy is transferred from the mean flow to the turbulent flow, while the dissipation term represents the rate at which turbulent kinetic energy is converted to heat. Note that the only difference between this model \(k\) equation and the true \(k\) equation is the introduction of the gradient diffusion approximation for the turbulent transport and pressure diffusion terms.

1.1. Theory
The \( k \) equation used in the \( k-\tau \) model is then obtained as a simple transformation of the standard \( k \) equation by the following relationship [Kok],

\[
\omega \equiv \frac{\epsilon}{\beta^* k}
\]

where \( \beta^* \) is a constant. Inserting \( \omega \beta^* k \) for \( \epsilon \) in the dissipation term \( \rho \epsilon \) gives the \( k \) equation used in nekRS,

\[
\frac{\partial (\rho k)}{\partial t} + \nabla \cdot (\rho k \mathbf{u}) = \nabla \cdot \left[ \left( \mu + \frac{\chi_T}{\sigma_k} \right) \nabla k \right] + P - \rho \beta^* \frac{k}{\tau}
\]

The \( \tau \) Equation

In two-equation \( RANS \) turbulence modeling, the greatest source of uncertainty is the proper choice of the second transport variable. While a \textit{true} \( k \) equation is often used as the starting point for developing the \textit{model} \( k \) equation, it is commonplace to start immediately from an ad hoc, “fabricated,” model equation for the second turbulence variable. Of course, “exact” equations can always be derived for the second turbulence variable through various operations on the mean Navier-Stokes equation or the Reynolds stress equation, but the exact equations for \( \epsilon, \omega \), or other turbulence quantities tend to be far more complex than the exact equation for \( k \) shown earlier.

In 1942, Kolmogorov was the first to propose the \( k-\omega \) model [Wilcox]. His formulation was very heuristic - from the Boussinesq approximation, it is likely that \( \nu_T \propto k \), which requires another variable with dimensions inverse time. Based on the work of Kolmogorov and many subsequent researchers of the \( k-\omega \) model, inserting \( \tau \equiv 1/\omega \) into the \( \omega \) equation gives the \( \tau \) equation - this approach is very similar to that used to obtained the \( k \) equation. The \( \tau \) equation used in nekRS is [Kok]

\[
\frac{\partial (\rho \tau)}{\partial t} + \nabla \cdot (\rho \tau \mathbf{u}) = \nabla \cdot \left[ \left( \mu + \frac{\mu_T}{\sigma_{\tau}} \right) \nabla \tau \right] - \frac{\tau}{k} P + \rho \beta^* - 2\frac{\mu_T}{\tau} \nabla \tau \cdot \nabla \tau
\]

where \( \sigma_{\tau}, \alpha, \) and \( \beta \) are constants. The last term on the right-hand side of the \( \tau \) equation is in practice implemented in the form

\[
\frac{2}{\tau} \nabla \tau \cdot \nabla \tau \rightarrow 8 \nabla \sqrt{\tau} \cdot \nabla \sqrt{\tau}
\]

in order to reduce the discretization error associated with the computation of gradients of a term that scales as \( y^2 \) as \( y \rightarrow 0 \) [Kok].

The Eddy Viscosity

The objective of \( RANS \) models is to estimate the eddy viscosity \( \mu_T \) that appears in the Boussinesq approximation. The particular form for \( \mu_T \) can be understood here in terms of the standard \( k-\epsilon \) model [Launder], for which \( \mu_T \) is given as

\[
\mu_T = C_{\mu} \frac{k^2}{\epsilon}
\]

where \( C_{\mu} \) is a constant. Inserting \( \tau \equiv 1/\omega \) and \( \epsilon = \beta^* \omega k \) gives

\[
\mu_T = \rho k \tau
\]

which presumes that \( C_{\mu} \) and \( \beta^* \) are really the same constant, but with different notation developed separately by the \( k-\epsilon \) researchers and the \( k-\tau \) researchers [Kok].

Closure Coefficients and Other Details

Table \textit{RANS Coefficients} shows the values for the various constants used in nekRS’s \( k-\tau \) model.

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Value</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sigma_k )</td>
<td>5/3</td>
<td></td>
</tr>
<tr>
<td>( \sigma_{\tau} )</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>( P_{TT} )</td>
<td>user-selected</td>
<td>—</td>
</tr>
</tbody>
</table>
A limiter is applied to both $k$ and $\tau$ to prevent negative values of either $k$ or $\tau$.

$$k = \max (k, 0.01|k|)$$

$$\tau = \max (\tau, 0.01|\tau|)$$

**Warning:** nekRS’s $k$-$\tau$ implementation currently requires that the laminar dynamic viscosity and the density are constant, because the setup routines can only accept constant values. See RANS Plugin for more information.

**Note:** Even if the molecular viscosity is constant, you must set `stressFormulation = true` in the input file because the total viscosity (molecular plus turbulent) will not be constant.

**Boundary Conditions**

On walls, because the asymptotic behavior of $\omega$ is $\omega \propto y^{-2}$ as $y \to 0$, and because the instantaneous velocity $u_i = \bar{u}_i + u'_i$ must be zero due to the no-slip condition, both $k$ and $\tau$ should be set to zero on no-slip boundaries.

On turbulent inlets, however, both $k$ and $\tau$ are generally nonzero, and estimates for $k$ and $\tau$ must be provided. Turbulent inflow conditions are usually unknown unless the modeler is fortunate enough to have experimental data - therefore, the specification of inlet conditions on $k$ and $\tau$ tends to be fairly ad hoc. An inlet condition for $k$ can be estimated by prescribing the turbulent intensity at the inlet. The turbulent intensity $I$ is defined as the root mean square fluctuating velocity normalized by the magnitude of the mean velocity, or

$$I \equiv \sqrt{\frac{1}{3} \left( \frac{u'_i u'_j}{u_j} \right)}$$

which can equivalently be written in terms of the turbulent kinetic energy as

$$I \equiv \sqrt{\frac{3}{2} \frac{k}{u_j}}$$

Therefore, if an inlet turbulent intensity can be prescribed, the inlet turbulent kinetic energy is

$$k \equiv \frac{3}{2} I^2 u_j$$

For instance, it is common to assume a uniform turbulent intensity over an inlet of between 1 and 5% for pipe flows [Russo]. Experiments have also quantified the scaling of turbulent intensity with Reynolds number. From simulations and experiments of both incompressible and compressible flows, the turbulent intensities on the axis of a smooth circular pipe are [Russo]:

$$I_{\text{axis}} = \begin{cases} 
0.0853 Re^{-0.0727} & \text{incompressible} \\
0.0550 Re^{-0.0407} & \text{compressible}
\end{cases}$$

With a slightly different definition based on the turbulent intensity averaged over the cross-sectional area of a circular pipe, the turbulent intensity instead scales as [Russo]:

$$I_{\text{area}} = \begin{cases} 
0.140 Re^{-0.0790} & \text{incompressible} \\
0.227 Re^{-0.1} & \text{compressible}
\end{cases}$$

Similar correlations have also been developed for rough pipes [Basse]. It is also commonplace to apply conditions on $k$ in terms of the friction velocity $u_\tau$, defined as

$$u_\tau \equiv \sqrt{\frac{\tau_w}{\rho}}$$
where \( \tau_w \) is the wall shear stress. The wall shear stress can then be estimated using a friction factor correlation for \( f_D \), the Darcy friction factor, which is defined as

\[
\frac{\Delta P}{l} = f_D \frac{\rho u_j u_j}{D}
\]

where \( \Delta P/l \) is a pressure gradient and \( D \) is a hydraulic diameter. For circular pipes, the friction factor is related to \( \tau_w \) as

\[
f_D = 8 \frac{\tau_w}{\rho u_j u_j}
\]

Typically, a simple friction factor correlation such as the Blasius model for pipes, is selected

\[
f_D = 0.316 Re^{-0.25}
\]

Combining \( u_\tau, f_D \), and one of the previous estimates for \( I_{\text{area}} \), such as \( I_{\text{area}} = 0.317 Re^{-0.11} \) \[Basse\] gives the following relationship between \( I_{\text{area}} \) and \( f_D \).

\[
I_{\text{area}} = 0.526 f_D^{0.44}
\]

Then, inserting the relationship between \( u_\tau \) and \( f_D \) gives, after some manipulation,

\[
k = 2.56 u_\tau^2 (u_j u_j)^{0.24}
\]

While this algebraic exercise hasn’t actually introduced any new information or closures beyond what has already been discussed, it is information to see \( k \) expressed in terms of the friction velocity, because some nekRS inputs set \( k \) on inlets by first computing the \( u_\tau \) from a friction correlation and then using an expression similar to above.

In any case, one of these correlations for turbulent intensity, or simply a fixed turbulent intensity of, say, 5%, can be used to prescribe a uniform value of \( k \) on an inlet. However, including some spatial variation in \( k \) on the inlet may reduce Gibbs phenomena if the inlet turbulent intensity enforces the physical zero wall value. Spatial fits, such as those developed for circular pipes \[Russo\], may improve the numerical stability and accuracy of your simulation.

Finally, on outlets, “free-stream” boundary conditions are usually applied to \( k \) and \( \tau \).

### Non-Dimensional Formulation

Now that the \( k-\tau \) model has been presented in its full form, the non-dimensional formulation is provided. Because nekRS’s \( k-\tau \) model is currently limited to constant densities and laminar viscosities, the non-dimensional formulation is slightly simpler than the more general case shown in Non-Dimensional Formulation that derived the non-dimensional form of the instantaneous Navier-Stokes equations.

Introduce the non-dimensional variables \( x^\dagger = \frac{x}{L}, \bar{u}^\dagger = \frac{u}{U}, t^\dagger = \frac{t U}{L} \), and \( f^\dagger = \frac{f L}{U^2} \). For convection-dominated flows, the pressure is non-dimensionalized in terms of the dynamic pressure as \( P^\dagger = \frac{P}{\rho U^2} \). Finally, the turbulent kinetic energy and inverse dissipation rate are non-dimensionalized as \( k^\dagger = \frac{k}{U^2} \) and \( \tau^\dagger = \frac{\tau}{U^2} \).

Inserting these non-dimensional variables into the mean flow mass and momentum conservation equations gives

\[
\frac{\partial \bar{u}_i^\dagger}{\partial x_i^\dagger} = 0
\]

\[
\frac{\partial \bar{u}_i^\dagger}{\partial t^\dagger} + u_j^\dagger \frac{\partial \bar{u}_i^\dagger}{\partial x_j^\dagger} = - \frac{\partial \bar{P}^\dagger}{\partial x_i^\dagger} + \frac{1}{Re} \left( 1 + \frac{\mu_T}{\mu} \right) \frac{\partial \bar{\tau}_{ij}^\dagger}{\partial x_j^\dagger} - \frac{\partial}{\partial x_j^\dagger} \left( \frac{2}{3} k^\dagger \delta_{ij} \right) + f_i^\dagger
\]

To non-dimensionalize the energy conservation equation, use the previous non-dimensional variables in addition to a non-dimensional temperature, \( T^\dagger = \frac{T - T_0}{\Delta T} \), where \( \Delta T \) is a reference temperature rise relative to a baseline temperature \( T_0 \). The heat source is non-dimensionalized as \( \dot{q}^\dagger = \frac{\dot{q}}{\rho C_p U \Delta T L} \), which arises naturally from the simple formulation

\[
\frac{\partial \bar{T}^\dagger}{\partial t^\dagger} = \frac{1}{Re} \frac{\partial}{\partial x_j^\dagger} \left( \frac{2}{3} k^\dagger \delta_{ij} \right) + f_i^\dagger
\]
of bulk energy conservation of \( Q = \dot{m}C_p\Delta T \), where \( Q \) is a heat source (units of Watts) and \( \dot{m} \) is a mass flowrate. Inserting these non-dimensional variables into the energy conservation equation gives

\[
\frac{\partial T^\dagger}{\partial t^\dagger} + \frac{u_i}{x_i} \frac{\partial T^\dagger}{\partial x_i^\dagger} = \frac{1}{Pe} \left( 1 + \frac{\mu_T/Pr_T}{k} C_p \right) \frac{\partial}{\partial x_i^\dagger} \frac{\partial T^\dagger}{\partial x_i^\dagger} + \dot{q}^\dagger
\]

To non-dimensionalize the \( k \) and \( \tau \) equations, define \( P^\dagger = \frac{P}{\mu_T U/L} \) and \( \epsilon^\dagger = \frac{\epsilon}{\mu_T U/L} \). With previous non-dimensional variables already defined, the non-dimensional \( k \) and \( \tau \) equations become

\[
\frac{\partial k^\dagger}{\partial t^\dagger} + \frac{u_i}{x_i} \frac{\partial k^\dagger}{\partial x_i^\dagger} = \frac{1}{Re} \left( 1 + \frac{\mu_T/\sigma_k}{\mu} \right) \frac{\partial}{\partial x_i^\dagger} \frac{\partial k^\dagger}{\partial x_i^\dagger} + P^\dagger - \beta^\dagger \frac{k^\dagger}{\tau^\dagger}
\]

\[
\frac{\partial \tau^\dagger}{\partial t^\dagger} + \frac{u_i}{x_i} \frac{\partial \tau^\dagger}{\partial x_i^\dagger} = \frac{1}{Re} \left( 1 + \frac{\mu_T/\sigma_\epsilon}{\mu} \right) \frac{\partial}{\partial x_i^\dagger} \frac{\partial \tau^\dagger}{\partial x_i^\dagger} - \frac{\sigma_k}{k} P^\dagger + \beta - \frac{2}{Re} \frac{1}{\tau^\dagger} \frac{\partial \tau^\dagger}{\partial x_i^\dagger} \frac{\partial \tau^\dagger}{\partial x_i^\dagger}
\]

Finally, the eddy viscosity is computed as

\[
\mu_T = \rho k \tau
\]

which after inserting the non-dimensional variables becomes

\[
\mu_T^\dagger = \rho U L k^\dagger \tau^\dagger
\]

such that the non-dimensional eddy viscosity can be written as \( \mu_T^\dagger = k^\dagger \tau^\dagger \). Therefore, the overall diffusion coefficients in the mean momentum equation, mean energy equation, \( k \) equation, and \( \tau \) equation are, respectively

\[
\frac{1}{Re} \left( 1 + \frac{\mu_T}{\mu} \right) \rightarrow \frac{1}{Re} + \mu_T^\dagger
\]

\[
\frac{1}{Pe} \left( 1 + \frac{\mu_T/Pr_T}{k} C_p \right) \rightarrow \frac{1}{Pe} + \mu_T^\dagger / Pr_T
\]

\[
\frac{1}{Re} \left( 1 + \frac{\mu_T/\sigma_k}{\mu} \right) \rightarrow \frac{1}{Re} + \mu_T^\dagger / \sigma_k
\]

\[
\frac{1}{Re} \left( 1 + \frac{\mu_T/\sigma_\epsilon}{\mu} \right) \rightarrow \frac{1}{Re} + \mu_T^\dagger / \sigma_\epsilon
\]

### 1.2 The nekRS Input Files

This page describes the input file structure and syntax needed to run a nekRS simulation. A nekRS simulation is referred to as a “case,” and at a minimum requires four files to run:

- Parameter file, with .par extension
- Mesh file, with .re2 extension
- User-defined functions for the host, with .udf extension
- User-defined functions for the device, with .oudf extension

The “case name” is then the common prefix applied to these files - for instance, a complete input description with a case name of “eddy” would be given by the files eddy.par, eddy.re2, eddy.udf, and eddy.oudf. The only restrictions on the case name are:

- It must be used as the prefix on all simulation files, and
- Typical restrictions for naming files for your operating system

### 1.2. The nekRS Input Files
The next four sections describe the structure and syntax for each of these four files for a general case. Because the Nek5000 code is a predecessor to nekRS, some aspects of the current nekRS input file design are selected to enable faster translation of Nek5000 input files into nekRS input files. Because these Nek5000-based approaches require proficiency in Fortran, the inclusion of several additional input files, and in some cases, careful usage of fixed-format text inputs, all Nek5000-based methods for case setup are referred to here as “legacy” approaches. All new users are encouraged to adopt the nekRS-based problem setup.

The scope of this page is merely to introduce the format and purpose of the four files needed to set up a nekRS simulation. Much more detailed instructions are provided on the FAQs page.

### 1.2.1 Parameter File (.par)

Most information about the problem setup is defined in the parameter file. This file is organized in a number of sections, each with a number of keys. Values are assigned to these keys in order to control the simulation settings.

The general structure of the .par file is as follows, where **FOO** and **BAR** are both section names, with a number of (key, value) pairs.

```plaintext
[FOO]
  key = value
  baz = bat

[BAR]
  alpha = beta
  gamma = delta + keyword=value + ...
```

The valid sections for setting up a nekRS simulation are:

- **BOOMERAMG**: settings for the (optional) AMG solver
- **GENERAL**: generic settings for the simulation
- **MESH**: settings for the mesh
- **OCCA**: backend OCCA device settings
- **PRESSURE**: settings for the pressure solution
- **PROBLEMTYPE**: settings for the governing equations
- **SCALARXX**: settings for the XX-th scalar
- **TEMPERATURE**: settings for the temperature solution
- **VELOCITY**: settings for the velocity solution
- **CASEDATA**: custom settings

Each of the keys and value types are now described for these sections. The formatting used here to describe valid key, value combinations is as follows. Take the backend key in the OCCA section as an example:

**backend (CUDA), CPU, HIP, OPENCL, OPENMP, SERIAL [THREAD MODEL]**

Here, backend is the key, and CUDA, CPU, HIP, OPENCL, OPENMP, and SERIAL are all valid values. Defaults are indicated in parentheses - therefore, if you do not explicitly give the backend in the .par file, the CUDA backend is used. Similar conventions are used to describe non-character type values; for instance, (3), <int> indicates that the default value for the indicated key is 3, but any integer value can be provided.

Most of the values associated with the various keys in the .par file are read by nekRS and then saved to various arguments in the options data structure. The argument is indicated in this section within square brackets. For example, the value set by the backend key is stored in the THREAD MODEL argument to options. In other words, if you wanted
to grab the value set by the user for the backend key, and save it in a local variable named user_occa_backend, you can use the getArgs function on the options data structure as follows.

```cpp
std::string user_occa_backend;
options.getArgs("THREAD MODEL", user_occa_backend);
```

In other words, if you have backend = CUDA in the .par file, then user_occa_backend would be set to CUDA in the above code.

Generally, most .par settings are not saved to a data structure, so throughout the code base, whenever information from the .par file is needed, it is simply extracted on-the-fly via the options structure.

nekRS performs validation of the par file. Invalid sections, invalid keys or values, invalid value combinations, missing values etc. will terminate the NekRS run with a clear error message. Deprecated attributes will be highlighted.

**Warning:** This user guide may quickly become out of date unless developers are careful to keep the keys listed here up to date. A list of possible values is also given in `doc/parHelp.txt`

nekRS uses just-in-time compilation to allow the incorporation of user-defined functions into program execution. These functions can be written to allow ultimate flexibility on the part of the user to affect the simulation, such as to define custom fluid properties, specify spatially-dependent boundary and initial conditions, and apply post-processing operations. Some of the parameters in the sections can be overridden through the use of user-defined functions - see, for example, the viscosity key in the VELOCITY section. This parameter is used to set a constant viscosity, whereas for variable-property simulations, a user-defined function will override the viscosity input parameter. A full description of these user-defined functions on the host and device are described in Sections UDF Functions and OUDF Functions. 

So, the description of valid (key, value) pairs here does not necessarily imply that these parameters reflect the full capabilities of nekRS.

**BOOMERAMG section**

This section is used to describe settings for the (optional) AMG solver.

- **coarsenType** [BOOMERAMG COARSEN TYPE]
- **interpolationType** [BOOMERAMG INTERPOLATION TYPE]
- **iterations** <int> [BOOMERAMG ITERATIONS]
- **nonGalerkinTol** [BOOMERAMG NONGALERKIN TOLERANCE]
- **smootherType** [BOOMERAMG SMOOTHER TYPE]
- **strongThreshold** <double> [BOOMERAMG NONGALERKIN TOLERANCE]

**GENERAL section**

This section is used to describe generic settings for the simulation such as time steppers, solution order, and file writing control.

- **constFlowRate** <string> ["CONSTANT FLOW RATE = [value is provided]]

  Set a constant flow rate in a given direction. Either meanVelocity or meanVolumetricFlow must be provided to set the flow rate, and either bid or direction must be provided to set the direction. The following options are valid:
-- meanVelocity <float> [CONSTANT FLOW RATE TYPE = BULK, FLOW RATE]
  Sets the mean velocity.

-- meanVelocity <float> [CONSTANT FLOW RATE TYPE = VOLUMETRIC, FLOW RATE]
  Sets the mean volumetric flow rate.

-- bid <int>, <int> [CONSTANT FLOW FROM BID, CONSTANT FLOW TO BID]
  Sets the flow direction based on two boundary IDs.

-- direction x, y, z [CONSTANT FLOW DIRECTION]
  Sets a flow direction parallel to the global coordinate axis.

• cubaturePolynomialOrder <int> [CUBATURE POLYNOMIAL DEGREE]
  Polynomial order for the cubature. If not specified, this defaults to the integer closest to \( \frac{3}{2}(N + 1) \) minus one, where \( N \) is the polynomial order.

• dealiasing (true), false
  If dealiasing is turned on, [ADVECTION TYPE] is set to CUBATURE+CONVECTIVE, whereas if dealiasing is turned off, [ADVECTION TYPE] is set to CUBATURE.

• dt <string> [DT]
  Time step size. If any of the keyword options targetCFL, max or initial are specified (separated by +), a variable timestep [VARIABLE DT = TRUE] is used. Otherwise, dt is parsed as float and indicates the time step size.

The following keywords may be given:
  -- targetCFL (0.5), <float> [TARGET CFL]: The target CFL is also used to set a default for the subCyclingSteps. If not specified, it is given by \( \max(subcyclingSteps*2, 0.5) \).
  -- max (0), <float> [MAX DT]: Largest allowed timestep. If 0 or unset, the option is ignored.
  -- initial (0), <float> [initially written to DT]: initial timestep.

• elapsedTime <double> [STOP AT ELAPSED TIME]
  Elapsed time at which to end the simulation, if using stopAt = elapsedTime.

• endTime <double> [END TIME]
  Final time at which to end the simulation, if using stopAt = endTime.

• numSteps (0), <int> [NUMBER TIMESTEPS]
  Number of time steps to perform, if using stopAt = numSteps. By default, if not specified, then it is assumed that no time steps are performed.

• oudf [casename].oudf [UDF OKL FILE]
  File name (including extension) of the *.oudf file, relative to the current directory. By default, the stem of the *.par file is used as casename.

• polynomialOrder <int> [POLYNOMIAL DEGREE]
  Polynomial order for the spectral element solution. An order of \( N \) will result in \( N + 1 \) basis functions for each spatial dimension. The polynomial order is currently limited to \( N < 10 \).

• startFrom <string> [RESTART FILE NAME]
  Absolute or relative path to a nekRS output file from which to start the simulation from. When used, the [RESTART FROM FILE] option argument is also set to true. If the solution in the restart file was obtained with a different
polynomial order, interpolation is performed to the current simulation settings. To only read select fields from
the restart file (such as if you wanted to only apply the temperature solution from the restart file to the present
simulation), append +U (to read velocity), +P (to read pressure), or +T (to read temperature) to the end of
the restart file name. For instance, if the restart file is named restart.fld, using restart.fld+T will only read
the temperature solution. If startFrom is omitted, the simulation is assumed to start based on the user-defined
initial conditions at time zero.

- **stopAt (numSteps), elapsedTime, endTime**

  When to stop the simulation, either based on a number of time steps numSteps, a simulated end time endTime,
or a total elapsed wall time elapsedTime. If stopAt = numSteps, the numSteps parameter must be
  provided. If stopAt = endTime, the endTime parameter must be provided. If stopAt = elapsedTime, the
  elapsedTime parameter must be provided.

- **subCyclingSteps (0), <int>, auto [SUBCYCLING STEPS]**

  Number of subcycling steps; if dt: targetCFL is specified, the number of subcycling steps is taken as the
  integer nearest to half the target CFL as given by the dt: targetCFL parameter. In this case, auto ensures
  that an error is raised if dt: targetCFL is not specified.

- **timeStepper (tombo2), bdf1, bdf2, bdf3, tombo1, tombo3 [TIME INTEGRATOR]**

  The method to use for time stepping. Note that if you select any of the BDF options, the time integrator is
  internally set to the TOMBO time integrator of equivalent order.

- **udf [casename].udf [UDF FILE]**

  File name (including extension) of the *.udf file, relative to the current directory. By default, the stem of the
  *.par file is used as casename.

- **usr [casename].usr [NEK USR FILE]**

  File name (including extension) of the *.usr file, relative to the current directory. By default, the stem of the
  *.par file is used as casename.

- **verbose (false), true [VERBOSE]**

  Whether to print the simulation results in verbose format to the screen.

- **writeControl (timeStep), runTime [SOLUTION OUTPUT CONTROL]**

  Method to use for the writing of output files, either based on a time step interval with timeStep (in which
case SOLUTION OUTPUT CONTROL is set to STEPS) or a simulated time interval with runTime (in which case
SOLUTION OUTPUT CONTROL is set to RUNTIME).

- **writeInterval <double> [SOLUTION OUTPUT INTERVAL]**

  Output writing frequency, either in units of time steps for writeControl = timeStep or in units of simulation
time for writeControl = runTime. If a runtime step control is used that does not perfectly align with the time
steps of the simulation, nekRS will write an output file on the timestep that most closely matches the desired
write interval.
Common keys

These parameters may be specified in any of the GENERAL, VELOCITY, TEMPERATURE and SCALARXX sections. If the parameter is not specified in any given VELOCITY, TEMPERATURE or SCALARXX section, its values are usually inherited from the GENERAL section.

The key for the options structure listed here is the GENERAL key; in the other sections, the key is prefixed with the section name.

- **regularization** ("none"), <string> [REGULARIZATION METHOD]
  Filtering settings., options are separated by +. This parameter is mutually exclusive with the (deprecated) filtering parameter. The parameter may be specified in any of the GENERAL, VELOCITY, TEMPERATURE and SCALARXX sections. If the parameter is no specified in any given VELOCITY, TEMPERATURE or SCALARXX section, its values are inherited from the GENERAL section.

  Filtering is analogous to Nek5000; the hpfrt filter is described further in the Nek5000 documentation.

  The following examples for regularization are given in examples:

  ```
  # examples/turbPipePeriodic
  regularization = hpfrt + nModes=1 + scalingCoeff=10
  # examples/double_shear
  regularization=avm+c0+highestModalDecay+scalingCoeff=0.5+rampconstant=1
  ```

  - none: regularization is disabled.
  - hpfrt: High-pass filter. The following settings apply to this mode:
    * **nmodes** (<int>) [HPFRT MODES]
      Number of filtered modes \((N - N')\), where \((N)\) is the polynomial degree and \((N')\) the number of fully resolved modes.
    * **cutoffratio** <float>
      Alternatively, the number of filtered modes can be given by the cutoff ratio, where \(N' + 1 = \text{filterCutoffRatio} \)
    * **scalingcoeff** (1.0), <expression> (required) [HPFRT STRENGTH]
      filter weight
  
  - avm+hpfResidual: use HPF Residual AVM, or avm+highestModalDecay: use Persson’s highest modal decay AVM. The AVM is described in [Persson], and only allowed for scalars. If specified in GENERAL, the regularization parameter must be overwritten in the VELOCITY section. The following settings apply to these modes:
    * **scalingcoeff** (1.0), <expression> (required) [REGULARIZATION SCALING COEFF]
      filter weight
    * the nmodes, cutoffratio and scalingcoeff parameters described above. With HighestModalDecay mode, scalingcoeff is interpreted (and overwrites) as vismaxcoeff.
    * **vismaxcoeff** (0.5), <float> [REGULARIZATION VISMAX COEFF]:
      controls maximum artificial viscosity
    * **c0** [REGULARIZATION AVM C0]:
      if provided, make viscosity C0 continous across elements
REGULARIZATION RAMP CONSTANT:
controls ramp to maximum artificial viscosity

**MESH section**

This section is used to describe mesh settings and set up various mesh solvers for mesh motion.

**partitioner** [MESH PARTITIONER]

**solver** elasticity, none, user

If solver = none, the mesh does not move and MOVING MESH is set to false. Otherwise, the solver is stored in MESH SOLVER. When solver = user, the mesh moves according to a user-specified velocity. Alternatively, if solver = elasticity, then the mesh motion is solved with an ALE formulation.

**OCCA section**

This section is used to specify the OCCA backend for parallelization.

**backend** (CUDA), CPU, HIP, OPENCL, OPENMP, SERIAL [THREAD MODEL]

OCCA backend; CPU is the same as SERIAL, and means that parallelism is achieved with MPI.

**deviceNumber** (LOCAL-RANK), <int> [DEVICE NUMBER]

**PRESSURE section**

The PRESSURE section describes solve settings for the pressure equation. Note that this block is only read if the VELOCITY block is also present.

**downwardSmother** ASM, jacobi, RAS [PRESSURE MULTIGRID DOWNWARD SMOOTHER]

**galerkinCoarseOperator** <bool> [GALERKIN COARSE OPERATOR]

**maxIterations** <int> [PRESSURE MAXIMUM ITERATIONS]

**pmultigridCoarsening** [PRESSURE MULTIGRID COARSENING]

**preconditioner** jacobi, multigrid, none, semfem, semg [PRESSURE PRECONDITIONER]

The pressure preconditioner to use; semg and multigrid both result in a multigrid preconditioner.

**residualProj** (true), false [PRESSURE RESIDUAL PROJECTION]

**residualProjectionStart** <int> [PRESSURE RESIDUAL PROJECTION START]

**residualProjectionVectors** <int> [PRESSURE RESIDUAL PROJECTION VECTORS]

**residualTol** <double> [PRESSURE SOLVER TOLERANCE]

Absolute residual tolerance for the pressure solution

**smootherType** additive, asm, chebyshev, chebyshev+ras, chebyshev+asm, ras [PRESSURE MULTIGRID SMOOTHER]

**solver**

**upwardSmother** ASM, JACOBI, RAS [PRESSURE MULTIGRID UPWARD SMOOTHER]
**PROBLEMTYPE section**

This section is used to control the form of the governing equations used in nekRS. While individual equations can be turned on/off in the VELOCITY, TEMPERATURE, and SCALAR sections, this block is used for higher-level control of the forms of those equations themselves.

**equation stokes**

Whether to omit the advection term in the conservation of momentum equation, therefore solving for the Stokes equations. If `equation = stokes`, then [ADVECTION] is set to false.

**stressFormulation (false), true [STRESSFORMULATION]**

Whether the viscosity (molecular plus turbulent) is not constant, therefore requiring use of the full form of the viscous stress tensor $\tau$. By setting `stressFormulation = false`, $\nabla \cdot \tau$ is represented as $\nabla \cdot \tau = \mu \nabla^2 u$. Even if the molecular viscosity is constant, this parameter must be set to `true` when using a RANS model because the turbulent viscosity portion of the overall viscosity is not constant.

**SCALAR section**

This section is used to define the transport parameters and solver settings for each passive scalar. For instance, in a simulation with two passive scalars, you would have two sections - SCALAR01 and SCALAR02, each of which represents a passive scalar.

**boundaryTypeMap <string[]>**

Array of strings describing the boundary condition to be applied to each sideset, ordered by sideset ID. The valid characters/strings are shown in Table Passive Scalar Boundary Conditions.

**diffusivity <double>**

Although this is named `diffusivity`, this parameter doubly represents the conductivity governing diffusion of the passive scalar. In other words, the analogue from the TEMPERATURE section (a passive scalar in its internal representation) is the `conductivity` parameter. If a negative value is provided, the conductivity is internally set to $1/|k|$, where $k$ is the value of the `conductivity` key. If not specified, this defaults to 1.0.

**residualProjection <bool>**

**residualProjectionStart <int>**

**residualProjectionVectors <int>**

**residualTol <double>**

Absolute residual tolerance for the passive scalar solution

**rho <double>**

Although this is name `rho`, this parameter doubly represents the coefficient on the total derivative of the passive scalar. In other words, the analogue from the TEMPERATURE section (a passive scalar in its internal representation) is the `rhoCp` parameter. If not specified, this defaults to 1.0.
**TEMPERATURE section**

This section is used to define the transport parameters and solver settings for the temperature passive scalar.

- **boundaryTypeMap <string[]>**
  
  Array of strings describing the boundary condition to be applied to each sideset, ordered by sideset ID. The valid characters/strings are shown in Table Passive Scalar Boundary Conditions.

- **conductivity <double> [SCALAR00 DIFFUSIVITY]**
  
  Constant thermal conductivity; if a negative value is provided, the thermal conductivity is internally set to $1/|k|$, where $k$ is the value of the conductivity key. If not specified, this defaults to 1.0.

- **residualProj <bool> [SCALAR00 RESIDUAL PROJECTION]**

- **residualProjectionStart <int> [SCALAR00 RESIDUAL PROJECTION START]**

- **residualProjectionVectors <int> [SCALAR00 RESIDUAL PROJECTION VECTORS]**

- **residualTol <double> [SCALAR00 SOLVER TOLERANCE]**

- **rhoCp <double> [SCALAR00 DENSITY]**
  
  Constant volumetric isobaric specific heat. If not specified, this defaults to 1.0.

- **solver none**
  
  You can turn off the solution of temperature by setting the solver to none.

**VELOCITY section**

This section is used to define the transport properties and solver settings for the velocity.

- **boundaryTypeMap <string[]>**
  
  Array of strings describing the boundary condition to be applied to each sideset, ordered by sideset ID. The valid characters/strings are shown in Table Flow Boundary Conditions. Note that no boundary conditions need to be specified in the PRESSURE section, since the form of the pressure conditions are specified in tandem with the velocity conditions with this parameter.

- **density <double> [DENSITY]**
  
  Constant fluid density. If not specified, this defaults to 1.0.

- **maxIterations (200), <int> [VELOCITY MAXIMUM ITERATIONS]**
  
  Maximum number of iterations for the velocity solve

- **residualProj <bool> [VELOCITY RESIDUAL PROJECTION]**

- **residualProjectionStart <int> [VELOCITY RESIDUAL PROJECTION START]**

- **residualProjectionVectors <int> [VELOCITY RESIDUAL PROJECTION VECTORS]**

- **residualTol <double> [VELOCITY SOLVER TOLERANCE]**

  Absolute tolerance used for the velocity solve.

- **solver none [VELOCITY SOLVER]**

  You can turn off the solution of the flow (velocity and pressure) by setting the solver to none. Otherwise, if you omit solver entirely, the velocity solve will be turned on. If you turn the velocity solve off, then you automatically also turn off the pressure solve.

- **viscosity <double> [VISCOSITY]**
Constant dynamic viscosity; if a negative value is provided, the dynamic viscosity is internally set to $1/|\mu|$, where $\mu$ is the value of the viscosity key. If not specified, this defaults to 1.0.

**CASEDATA section**

This section may be used to provide custom parameters in the .par file that are to be read in the .udf file. For example, you may specify

```
[CASEDATA]
  Re_tau = 550
```

in the .par file; the parameters should be read in the *UDF_Setup0* function, e.g.

```
static dfloat Re_tau;
platform->par->extract("casedata", "re_tau", Re_tau);
```

NekRS does not check the contents of the CASEDATA section; such checks may be added in the *UDF_Setup0* function as well.

**Deprecated parameters**

**GENERAL section**

- **filterCutoffRatio <double>** [deprecated, see *regularization*]
- **filtering hpfrt** [deprecated, see *regularization*]
  
  If filtering = hpfrt, [FILTER STABILIZATION] is set to RELAXATION, and filterWeight must be specified. If filtering is not specified, [FILTER STABILIZATION] is set to NONE by default.
- **filterModes <int>** [HPFRT MODES] [deprecated, see *regularization*]
  
  Number of filter modes; minimum value is 1. If not specified, the number of modes is set by default to the nearest integer to $(N + 1)(1 - f_c)$, where $f_c$ is the filter cutoff ratio.
- **filterWeight <double>** [HPFRT STRENGTH] [deprecated, see *regularization*]

**Legacy Option (.rea)**

An alternative to the use of the .par file is to use the legacy Nek5000-based .rea file to set up the case parameters. See the *Mesh File (.re2)* section of the *Nek5000 documentation*\(^1\) for further details on the format for the .rea file.

The .rea file contains both simulation parameters (now covered by the .par file) as well as mesh information (now covered by the .re2 file). This section here only describes the legacy approach to setting simulation parameters via the .rea file.

\(^1\) While the heading for Mesh File (.re2) seems to suggest that the contents refer only to the .re2 format, the actual text description still points to the legacy .rea format.
1.2.2 Mesh File (.re2)

The nekRS mesh file is provided in a binary format with a nekRS-specific .re2 extension. This format can be produced by either:

- Converting a mesh made with commercial meshing software to .re2 format, or
- Directly creating an .re2-format mesh with nekRS-specific scripts

There are three main limitations for the nekRS mesh:

- nekRS is restricted to 3-D hexahedral meshes.
- The numeric IDs for the mesh boundaries must be ordered contiguously beginning from 1.
- The .re2 format only supports HEX8 and HEX 20 (eight- and twenty-node) hexahedral elements.

Lower-dimensional problems can be accommodated on these 3-D meshes by applying zero gradient boundary conditions to all solution variables in directions perpendicular to the simulation plane or line, respectively. All source terms and material properties in the governing equations must therefore also be fixed in the off-interest directions.

For cases with conjugate heat transfer, nekRS uses an archaic process for differentiating between fluid and solid regions. Rather than block-restricting variables to particular regions of the same mesh, nekRS retains two independent mesh representations for the same problem. One of these meshes represents the flow domain, while the other represents the heat transfer domain. The nrs_t struct, which encapsulates all of the nekRS simulation data related to the flow solution, represents the flow mesh as nrs_t.mesh. Similarly, the cds_t struct, which encapsulates all of the nekRS simulation data related to the convection-diffusion passive scalar solution, has one mesh for each passive scalar. That is, cds_t.mesh[0] is the mesh for the first passive scalar, cds_t.mesh[1] is the mesh for the second passive scalar, and so on. Note that only the temperature passive scalar uses the conjugate heat transfer mesh, even though the cds_t struct encapsulates information related to all other passive scalars (such as chemical concentration, or turbulent kinetic energy). All non-temperature scalars are only solved on the flow mesh.

**Warning:** When writing user-defined functions that rely on mesh information (such as boundary IDs and spatial coordinates), you must take care to use the correct mesh representation for your problem. For instance, to apply initial conditions to a flow variable, you would need to loop over the number of quadrature points known on the nrs_t meshes, rather than the cds_t meshes for the passive scalars (unless the meshes are the same, such as if you have heat transfer in a fluid-only domain). Also note that the cds_t * cds object will not exist if your problem does not have any passive scalars.

nekRS requires that the flow mesh be a subset of the heat transfer mesh. In other words, the flow mesh always has less than (or equal to, for cases without conjugate heat transfer) the number of elements in the heat transfer mesh. Creating a mesh for conjugate heat transfer problems requires additional pre-processing steps that are described in the *Creating a Mesh for Conjugate Heat Transfer* section. The remainder of this section describes how to generate a mesh in .re2 format, assuming any pre-processing steps have been done for the special cases of conjugate heat transfer.

**Converting an Existing Commercial Mesh**

The most general and flexible approach for creating a mesh is to use commercial meshing software such as Cubit or Gmsh. After creating the mesh, it must be converted to the .re2 binary format. Depending on the mesh format (such as Exodus II format or Gmsh format), a conversion script is used to convert the mesh to .re2 format. See the *Converting a Mesh to .re2 Format* section for examples demonstrating conversion of Exodus and Gmsh meshes into .re2 format.
Nek5000 Script-Based Meshing

A number of meshing scripts ship with the Nek5000 dependency, which allow you to directly create .re2 format meshes without the need of commercial meshing tools. These scripts, such as genbox, take user input related to the desired grid spacing to generate meshes for fairly simple geometries. Please consult the Nek5000 documentation for more information on the use of these scripts.

Legacy Option (.rea)

An alternative to the use of the .re2 mesh file is to use the legacy Nek5000-based .rea file to set up the mesh. See the Mesh File (.re2) section of the Nek5000 documentation Page 20 for further details on the format for the .rea file.

The .rea file contains both simulation parameters (now covered by the .par file) as well as mesh information (now covered by the .re2 file). This section here only describes the legacy approach to setting mesh information via the .rea file.

The mesh section of the .rea file can be generated in two different manners - either by specifying all the element nodes by hand, or with the Nek5000 mesh generation scripts introduced in Section Nek5000 Script-Based Meshing. Because the binary .re2 format is preferred for very large meshes where memory may be a concern, the .rea file approach is considered to be a legacy option. The mesh portion of the legacy .rea file can be converted to the .re2 format with the reatore2 script, which also ships with the Nek5000 dependency.

1.2.3 User-Defined Host Functions (.udf)

User-defined functions for the host are specified in the .udf file. These functions can be used to perform virtually any action that can be programmed in C++. Some of the more common examples are setting initial conditions, querying the solution at regular intervals, and defining custom material properties and source terms. The available functions that you may define in the .udf file are as follows. From the examples shown on the Detailed Usage page, you will see that usage of these functions requires some proficiency in the C++ language as well as some knowledge of the nekRS source code internals.

**UDF_ExecuteStep(nrs_t* nrs, dfloat time, int tstep)**

This user-defined function is probably the most flexible of the nekRS user-defined functions. This function is called once at the start of the simulation just before beginning the time stepping, and then once per time step after running each step.

**UDF_LoadKernels(nrs_t* nrs)**

This user-defined function is used to load case-specific device kernels that are used in other UDF functions. For instance, if you add a custom forcing term to the momentum equations, you need to tell nekRS to compile that kernel by loading it in this function. The custom material property example shown in the Setting Custom Properties with UDF_Setup section demonstrates how to load kernels with this function. The process is quite simple, and only involves:

- Declaring all kernels as static occa::kernel at the top of the .udf file
- Loading those kernels in UDF_LoadKernels
- Defining those kernels in the device user file (the .oudf file)

The only kernels in the .oudf file that don’t need to be explicitly loaded are the boundary condition kernels that ship with nekRS. During the .oudf just-in-time compilation, nekRS will search the .oudf file for any functions that match the nekRS boundary condition functions, and automatically create and load a kernel based on the function internals.
set by the user. For instance, in the setUDF function in the nekRS source code, the .oudf file is scanned for a string matching scalarDirichletConditions (one of the boundary condition functions in Table Passive Scalar Boundary Conditions). If this string is found, then the function internals written by the user are cast into a generic OCCA kernel that is then written into a just-in-time compiled OKL-language file at .cache/udf/udf.okl.

```cpp
found = buffer.str().find("void scalarDirichletConditions");
if (found == std::string::npos)
  out << "void scalarDirichletConditions(bcData *bc){}\n"

out <<
  "@kernel void __dummy__(int N) {
  " for (int i = 0; i < N; ++i; @tile(16, @outer, @inner)) {
  "}
  "}"
```

The UDF_LoadKernels function is passed the nekRS simulation object nrs to provide optional access to the occa::properties object on the nrs->kernelInfo object. In addition to loading kernels, this function can also be used to propagate user-defined variables to the kernels. See the Defining Variables to Access in Device Kernels section for a description of this feature.

**UDF_Setup0(MPI_Comm comm, setupAide & options)**

This user-defined function is passed the nekRS MPI communicator comm and a data structure containing all of the user-specified simulation options, options. This function is called once at the beginning of the simulation before initializing the nekRS internals such as the mesh, solvers, and solution data arrays. Because virtually no aspects of the nekRS simulation have been initialized at the point when this function is called, this function is primarily used to modify the user settings. For the typical user, all relevant settings are already exposed through the .par file; any desired changes to settings should therefore be performed by modifying the .par file.

This function is intended for developers or advanced users to overwrite any user settings that may not be exposed to the .par file. For instance, setting timeStepper = tombo2 in the GENERAL section triggers a number of other internal settings in nekRS that do not need to be exposed to the typical user, but that perhaps a developer may want to modify for testing purposes.

**UDF_Setup(nrs_t* nrs)**

This user-defined function is passed the nekRS simulation object nrs. This function is called once at the beginning of the simulation after initializing the mesh, solution arrays, material property arrays, and boundary field mappings. This function is most commonly used to:

- Apply initial conditions to the solution
- Assign function pointers to user-defined source terms and material properties

Any other additional setup actions that depend on initialization of the solution arrays and mesh can of course also be placed in this function.

1.2. The nekRS Input Files
Other Functions for Custom Sources on the udf Structure

In addition to the UDF_Setup0, UDF_Setup, UDF_ExecuteStep, and UDF_LoadKernels, there are other user-defined functions. These functions are handled in a slightly different manner - rather than be tied to a specific function name like UDF_Setup0, these functions are provided in terms of generic function pointers to any function (provided the function parameters match those of the pointer). The four function pointers are named as follows in nekRS:

<table>
<thead>
<tr>
<th>Function pointer</th>
<th>Function signature</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>udf.converged</td>
<td>f(nrs_t* nrs, int stage)</td>
<td></td>
</tr>
<tr>
<td>udf.uEqnSource</td>
<td>f(nrs_t* nrs, float t, m o_U, m o_FU)</td>
<td>momentum source</td>
</tr>
<tr>
<td>udf.sEqnSource</td>
<td>f(nrs_t* nrs, float t, m o_S, m o_SU)</td>
<td>scalar source</td>
</tr>
<tr>
<td>udf.properties</td>
<td>f(nrs_t* nrs, float t, m o_U, m o_S, m o_Up, m o_Sp)</td>
<td>material properties</td>
</tr>
<tr>
<td>udf.div</td>
<td>f(nrs_t* nrs, float t, m o_div)</td>
<td>thermal divergence</td>
</tr>
</tbody>
</table>

To shorten the syntax above, the type m is shorthand for `occa::memory`, and f is the name of the function, which can be any user-defined name. Other parameters that appear in the function signatures are as follows:

- nrs is a pointer to the nekRS simulation object
- stage
- t is the current simulation time
- o_U is the velocity solution on the device
- o_S is the scalar solution on the device
- o_FU is the forcing term in the momentum equation
- o_SU is the forcing term in the scalar equation(s)
- o_Up is the material properties (\(\mu\) and \(\rho\)) for the momentum equation
- o_Sp is the material properties (\(k\) and \(\rho C_p\)) for the scalar equation(s)
- o_div

The udf.uEqnSource allows specification of a momentum source, such as a gravitational force, or a friction form loss. The udf.sEqnSource allows specification of a source term for the passive scalars. For a temperature passive scalar, this source term might represent a volumetric heat source, while for a chemical concentration passive scalar, this source term could represent a mass source. See the Setting Custom Source Terms section for an example of setting custom source terms.

The udf.properties allows specification of custom material properties for the flow and passive scalar equations, which can be a function of the solution as well as position and time. See the Setting Custom Properties section for an example of setting custom properties.

Finally, udf.div allows specification of the thermal divergence term needed for the low Mach formulation.

Legacy Option (.usr)

The legacy alternative to user-defined functions in the .udf file is to write Fortran routines in a .usr file based on Nek5000 code internals.
1.2.4 User-Defined Device Functions (.oudf)

This file contains all user-defined functions that are to run on the device. These functions include all functions used to apply boundary conditions that are built in to nekRS, as well as any other problem-specific device functions.

**Boundary Condition Functions**

The type of condition to apply for each boundary is specified by the `boundaryTypeMap` parameter in the `.par` file. A character or longer-form word is used to indicate each boundary condition, where the entries in `boundaryTypeMap` are listed in increasing boundary ID order. However, this single line only specifies the type of boundary condition. If that boundary condition requires additional information, such as a value to impose for a Dirichlet velocity condition, or a flux to impose for a Neumann temperature condition, then a device function must be provided in the `.oudf` file. A list of all possible boundary conditions is as follows. For boundary conditions that require additional input from the user, a device function is also listed. For other boundary conditions that are fully specified simply by the type of condition (such as a wall boundary condition for velocity, which sets all three components of velocity to zero without additional user input), no device function is needed to apply that condition.

### Table 1.2: Flow Boundary Conditions

<table>
<thead>
<tr>
<th>Function</th>
<th>Character Map</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>pressureDirichletConditions(bcData* bc)</td>
<td></td>
<td>Dirichlet pressure condition</td>
</tr>
<tr>
<td>velocityDirichletConditions(bcData* bc)</td>
<td>v.inlet</td>
<td>Dirichlet velocity condition</td>
</tr>
<tr>
<td>velocityNeumannConditions(bcData* bc)</td>
<td>p</td>
<td>Neumann velocity condition</td>
</tr>
<tr>
<td>N/A</td>
<td>w, wall</td>
<td>No-slip wall for velocity</td>
</tr>
<tr>
<td>N/A</td>
<td>o, outlet, outflow</td>
<td>Zero-gradient velocity</td>
</tr>
<tr>
<td>N/A</td>
<td>slipx</td>
<td></td>
</tr>
<tr>
<td>N/A</td>
<td>slipy</td>
<td></td>
</tr>
<tr>
<td>N/A</td>
<td>slipz</td>
<td></td>
</tr>
<tr>
<td>N/A</td>
<td>symx</td>
<td></td>
</tr>
<tr>
<td>N/A</td>
<td>symy</td>
<td></td>
</tr>
<tr>
<td>N/A</td>
<td>symz</td>
<td></td>
</tr>
</tbody>
</table>

### Table 1.3: Passive Scalar Boundary Conditions

<table>
<thead>
<tr>
<th>Function</th>
<th>Character Map</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalarDirichletConditions(bcData* bc)</td>
<td>t.inlet</td>
<td>Dirichlet condition</td>
</tr>
<tr>
<td>scalarNeumannConditions(bcData* bc)</td>
<td>f.flux</td>
<td>Neumann condition</td>
</tr>
<tr>
<td>N/A</td>
<td>p</td>
<td>Periodic</td>
</tr>
<tr>
<td>N/A</td>
<td>i.zeroflux</td>
<td>Zero-gradient</td>
</tr>
<tr>
<td>N/A</td>
<td>o.outlet, outflow</td>
<td>Zero-gradient</td>
</tr>
</tbody>
</table>

Each function has the same signature, and takes as input the bc object. This object contains all information needed to apply a boundary condition - the position, unit normals, and solution components. The “character map” refers to the character in the `boundaryTypeMap` key in the `.par` file that will trigger this boundary condition. The character map can either be a single letter, or a more verbose (and equivalent) string.

The `scalar`-type boundary conditions are called for boundary conditions on passive scalars, while the `pressure`- and `velocity`-type conditions are called for the boundary conditions on the flow.

Each of these functions is only called on boundaries that contain that boundary. For instance, if only boundaries 3 and 4 are primitive conditions on velocity, then `velocityDirichletConditions` is only called on boundaries 3 and 4. See the Setting Boundary Conditions section for several examples on how to set boundary conditions with device functions.
1.3 Commonly-Used Variables in nekRS

To become a proficient user of nekRS requires some knowledge of the data structures used to store the mesh, solution fields, and simulation settings. While many commercial CFD codes have developed user interfaces that allow most user code interactions to occur through a GUI or even a text-based format, nekRS very much remains a research tool. As such, even “routine” actions such as setting boundary and initial conditions requires an understanding of the source code structure in nekRS. This requirement is advantageous from a flexibility perspective, however, because almost any user action that can be written in C++ .udf or OKL in .oudf files can be incorporated into a nekRS simulation.

This page contains a summary of some of the most commonly-used variables and structures used to interact with nekRS. For array-type variables, the size of the array is also listed in terms of the length of each dimension of that array. For instance, if the size of an array is \( N \) elements * \( Np \), then the data is stored first by each element, and second by each quadrature point. If the variable is not an array type, the size is shown as 1.

Some variables have an equivalent form that is stored on the device that can be accessed in device kernels. All such device variables and arrays that live on the device by convention are prefixed with \( o_\cdot \). That is, \( \text{mesh->x} \) represents all the \( x \)-coordinates of the quadrature points, and is stored on the host. The same data, but accessible on the device, is \( \text{mesh->o_x} \). Not all variables and arrays are automatically available on both the host and device, but those that are available are indicated with a \( \checkmark \) in the “Device?” table column.

1.3.1 Mesh

This section describes commonly-used variables related to the mesh, which are all stored on data structures of type \( \text{mesh}_t \). nekRS uses an archaic approach for conjugate heat transfer applications, i.e. problems with separate fluid and solid domains. For problems without conjugate heat transfer, all mesh information is stored on the \( \text{nrs->mesh} \) object, while for problems with conjugate heat transfer, all mesh information is stored on the \( \text{nrs->cds->mesh} \) object. More information is available in the Creating a Mesh for Conjugate Heat Transfer section. To keep the following summary table general, the variable names are referred to simply as living on the \( \text{mesh} \) object, without any differentiation between whether that \( \text{mesh} \) object is the object on \( \text{nrs} \) or \( \text{nrs->cds} \).

Some notable points of interest that require additional comment:

- The MPI communicator is stored on the mesh, since domain decomposition is used to divide the mesh among processes. Most information stored on the \( \text{mesh} \) object strictly refers to the portion of the mesh “owned” by the current process. For instance, \( \text{mesh->Nelements} \) only refers to the number of elements “owned” by the current process \( \text{(mesh->rank)} \), not the total number of elements in the simulation mesh. Any exceptions to this process-local information is noted as applicable.
1.3.2 Flow Solution Fields and Simulation Settings

This section describes the members on the nrs object, which consist of user settings as well as the flow solution. Some of this information is simply assigned a value also stored on the nrs->mesh object. Some notable points that require additional comment:

- Like the mesh object, the solution fields are stored only on a per-rank basis. That is, nrs->U only contains the velocity solution for the elements “owned” by the current process.

- Solution arrays with more than one component (such as velocity, in nrs->U) are indexed according to a fieldOffset. This offset is chosen to be larger than the actual length of the velocity solution (which is the total number of quadrature points on that rank, or nrs->Nlocal) due to performance reasons. That is, you should use the fieldOffset to index between components, but within a single component, you should not attempt to access entries with indices between i * (fieldOffset - Nlocal), where i is the component number, because those values are not actually used to store the solution (they are the end of a storage buffer).

Some members only exist on the device - in this case, the variable name shown in the first column explicitly shows the o_ prefix to differentiate that this member is not available in this form on the host. For instance, the o_mue member is only available on the device - there is no corresponding array nrs->mue member.

variable name | size | device? | meaning
--- | --- | --- | ---
comm | 1 | | MPI communicator
device | 1 | | backend device
dim | 1 | | spatial dimension of mesh
elementInfo | Nelements | | phase of element (0 = fluid, 1 = solid)
EToB | Nelements * Nfaces | ✓ | boundary ID for each face
N | 1 | | polynomial order for each dimension
NboundaryFaces | 1 | | total number of faces on a boundary (rank sum)
Nelements | 1 | | number of elements
Nfaces | 1 | | number of faces per element
Nfp | 1 | | number of quadrature points per face
Np | 1 | | number of quadrature points per element
rank | 1 | | parallel process rank
size | 1 | | size of MPI communicator
vmapM | Nelements * Nfaces * Nfp | ✓ | quadrature point index for faces on boundaries
x | Nelements * Np | ✓ | x-coordinates of quadrature points
y | Nelements * Np | ✓ | y-coordinates of quadrature points
z | Nelements * Np | ✓ | z-coordinates of quadrature points
<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Size</th>
<th>Device?</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>cds</td>
<td>1</td>
<td></td>
<td>convection-diffusion solution object</td>
</tr>
<tr>
<td>cht</td>
<td>1</td>
<td></td>
<td>whether the problem contains conjugate heat transfer</td>
</tr>
<tr>
<td>dim</td>
<td>1</td>
<td></td>
<td>spatial dimension of nrs-&gt;mesh</td>
</tr>
<tr>
<td>dt</td>
<td>3</td>
<td></td>
<td>time step for previous 3 time steps</td>
</tr>
<tr>
<td>fieldOffset</td>
<td></td>
<td></td>
<td>offset in flow solution arrays to access new component</td>
</tr>
<tr>
<td>FU</td>
<td>NVfields * nEXT * fieldOffset</td>
<td>✓</td>
<td>source term for each momentum equation for each step in the time stencil</td>
</tr>
<tr>
<td>isOutputStep</td>
<td></td>
<td></td>
<td>if an output file is written on this time step</td>
</tr>
<tr>
<td>lastStep</td>
<td>1</td>
<td></td>
<td>if this time step is the last time step of the run</td>
</tr>
<tr>
<td>mesh</td>
<td>1</td>
<td></td>
<td>mesh used for the flow simulation</td>
</tr>
<tr>
<td>nEXT</td>
<td>1</td>
<td></td>
<td>number of time steps in the time derivative stencil</td>
</tr>
<tr>
<td>NiterU</td>
<td>1</td>
<td></td>
<td>number of iterations taken in last velocity solve</td>
</tr>
<tr>
<td>NiterP</td>
<td>1</td>
<td></td>
<td>number of iterations taken in last pressure solve</td>
</tr>
<tr>
<td>NLocal</td>
<td>1</td>
<td></td>
<td>number of quadrature points local to this process</td>
</tr>
<tr>
<td>Nscalar</td>
<td>1</td>
<td></td>
<td>number of passive scalars to solve for</td>
</tr>
<tr>
<td>NTfields</td>
<td>1</td>
<td></td>
<td>number of flow-related fields to solve for ( (\vec{V} + T) )</td>
</tr>
<tr>
<td>NVfields</td>
<td>1</td>
<td></td>
<td>number of velocity fields to solve for</td>
</tr>
<tr>
<td>o_mue</td>
<td>fieldOffset</td>
<td>✓</td>
<td>total dynamic viscosity (laminar plus turbulent) for the momentum equation</td>
</tr>
<tr>
<td>options</td>
<td>1</td>
<td></td>
<td>object containing user settings from .par file</td>
</tr>
<tr>
<td>o_rho</td>
<td>fieldOffset</td>
<td>✓</td>
<td>density for the momentum equation</td>
</tr>
<tr>
<td>P</td>
<td>fieldOffset</td>
<td>✓</td>
<td>pressure solution for most recent time step</td>
</tr>
<tr>
<td>prop</td>
<td>2 * fieldOffset</td>
<td>✓</td>
<td>total dynamic viscosity (laminar plus turbulent) and density (in this order) for the momentum equation</td>
</tr>
<tr>
<td>U</td>
<td>NVfields * fieldOffset</td>
<td>✓</td>
<td>velocity solution for all components for most recent time step</td>
</tr>
</tbody>
</table>

### 1.3.3 Passive Scalar Solution Fields and Simulation Settings

This section describes the members on the cds object, which consist of user settings as well as the passive scalar solution. Note that, from Flow Solution Fields and Simulation Settings, the cds object is itself stored on the nrs flow solution object. Many of these members are copied from the analogous variable on the nrs object. For instance, cds->fieldOffset is simply set equal to nrs->fieldOffset. In a few cases, however, the names on the cds object differ from the analogous names on the nrs object, such as for cds->NSfields and nrs->Nscalar, which contain identical information.
1.4 Compiling nekRS

This page describes how to build nekRS in a general sense, as well as providing specific instructions for some of the more common HPC systems used by the nekRS development team.

First, clone the repository from github. Next, set the NEKRS_HOME environment variable to a location in your file system where you would like to place the executables and other build files. For example, this can be:

```bash
user$ export NEKRS_HOME=$HOME/.local/nekrs
```

Then, be sure to add this directory to your path:

```bash
user$ export PATH=${NEKRS_HOME}:${PATH}
```

To avoid repeating these steps for every new shell, you may want to add these environment variable settings in a .bashrc.

Next, run the `makenrs` script in the main level of the repository as

```bash
user$ ./makenrs
```

You will be prompted as to whether you want to build the CUDA, HIP, and OpenCL backends. To avoid passing these settings from the standard input, you can also run `makenrs` with boolean values following the `makenrs` command that indicate the backend settings for each of these three backends. For example, to build without CUDA, HIP, and OpenCL, you can run:

```bash
user$ ./makenrs 0 0 0
```

1.4.1 Summit

1.4.2 Sawtooth

1.5 Tutorials

1.6 FAQs

This page describes how to perform a wide variety of user interactions with nekRS for setting boundary conditions, converting between mesh formats, defining and running device kernels, writing output files, and much more. Please first consult the Input File Syntax page for an overview of the purpose of each nekRS input file to provide context on where the following instructions fit into the overall code structure. Throughout this section, variables and data structures in the nekRS source code are referenced - a list defining these variables and structures is available on the Commonly Used Variables page for reference.
1.6.1 Building the Nek5000 Tool Scripts

Some user actions in nekRS require the use of scripts available with Nek5000. To build these scripts, you will need to separately clone Nek5000, and then navigate to the tools directory and run the makefile to compile all the scripts.

```
user$ git clone https://github.com/Nek5000/Nek5000.git
user$ cd Nek5000/tools
user$ ./maketools all
```

This should create binary executables in the Nek5000/bin directory. You may want to add this to your path in order to quickly access those scripts.

1.6.2 Scripts That Ship with nekRS

In addition to the scripts that ship with Nek5000 described in Building the Nek5000 Tool Scripts, a number of scripts ship with nekRS itself. These scripts are located in the $NEKRS_HOME/bin directory, where NEKRS_HOME is an environment variable set as part of the build process. A brief summary of these scripts and their usage is as follows.

- **nrsmpi <casename> <processes>:** run nekRS in parallel with <processes> parallel processes for the case files that are prefixed with casename.
- **nrsbmpi <casename> <processes>:** same as nrsmpi, except that nekRS runs in the background
- **nrspre <casename> <target GPUs>:** precompile nekRS case (see Just-in-time Compilation)
- **nrsqsub_lassen <casename> <nodes> <wall time>:** submission script for Lassenhttps://computing.llnl.gov/computers/lassen, a supercomputer at Lawrence Livermore National Laboratory. A number of other settings are specified within the script itself.
- **nrsqsub_summit <casename> <nodes> <wall time>:** submission script for Summithttps://www.olcf.ornl.gov/summit/, a supercomputer at Oak Ridge National Laboratory. A number of other settings are specified within the script itself.
- **nrsvis <casename>:** postprocess fld-type nekRS output files into a form readable by Paraview or Visit.

1.6.3 Converting a Mesh to .re2 Format

The most general and flexible approach for creating a mesh is to use commercial meshing software such as Cubit or Gmsh. After creating the mesh, it must be converted to the .re2 binary format. The following two sections describe how to convert Exodus and Gmsh meshes into .re2 binary format with scripts that ship with the Nek5000 dependency. First build these scripts following the instructions in the Building the Nek5000 Tool Scripts section.

Converting an Exodus mesh

To convert from an Exodus format mesh (for this case, named my_mesh.exo) to the .re2 format, use the exo2nek script:

```
user$ exo2nek
```

Then, follow the on-screen prompts associated with the exo2nek script. exo2nek will convert all elements in the Exodus mesh (TET6, WEDGE6, HEX8, HEX20) to HEX20 elements and dump into .re2 format.
Converting a Gmsh mesh

To convert from a Gmsh format mesh (for this case, named my_mesh.msh) to the .re2 format, use the gmsh2nek script:

```
user$ gmsh2nek
Enter mesh dimension: 3
Input (.msh) file name: my_mesh
```

All your mesh should be hexahedral elements. Before exporting from Gmsh, you will need to set the mesh order to 2. The Gmsh mesh format must also be version 2, ASCII/binary format. If your Gmsh version shows a pop-up box when exporting the mesh, do not click “Save all elements” or “Save parametric elements”.

1.6.4 Creating a Mesh for Conjugate Heat Transfer

Mesh generation for conjugate heat transfer requires an additional pre-processing step before performing other steps of the mesh generation process such as those described in the Converting a Mesh to .re2 Format section. The nekRS approach for conjugate heat transfer is still dependent on legacy limitations from Nek5000. Unfortunately, you cannot simply use a standard commercial meshing tool and define fluid and solid regions according to block IDs - you must individually create the mesh for the fluid and the solid, and then merge them with the pretex script.

1.6.5 Setting Initial Conditions with UDF_Setup

This section provides an example for setting initial conditions with the UDF_Setup user-defined function that was introduced on the Input Files page. The following code snippet sets initial conditions for all three components of velocity, the pressure, and two passive scalars. You may not necessarily have all of these variables in your model - this example is just intended to cover all possibilities.

For this example, the initial conditions are $V_x = \sin(x)\cos(y)\cos(z)$, $V_y = -\cos(x)\sin(y)\cos(z)$, and $V_z = 0$ for the three components of velocity; $P = 101325$ for the pressure; and $\phi_0 = 573$ and $\phi_1 = 100 + z$ for the two passive scalars indicated generically as $\phi_0$ and $\phi_1$.

**Note:** If present, the temperature variable is represented internally in nekRS as a passive scalar, since the form of the equation is the same as those solver for other passive scalars such as chemical concentration.

Because these initial conditions will be a function of space, we must first obtain the mesh information, for which we use the nrs->mesh pointer. All solution fields are stored in nekRS in terms of the quadrature points (also referred to as the GLL points). So, we will apply the initial conditions by looping over all of these quadrature points, which for the current MPI process is equal to mesh->Np, or the number of quadrature points per element, and mesh->Nelements, the number of elements on this process.

Next, we can get the $x$, $y$, and $z$ coordinates for the current quadrature point with the x, y, and z pointers on the mesh object. Finally, we programatically set initial conditions for the solution fields. nrs->U is a single array that holds all three components of velocity; the nrs->fieldOffset variable is used to shift between components in this array. nrs->P represents the pressure. Finally, nrs->S is a single array that holds all of the passive scalars. Similar to the offset performed to index into the velocity array, the nrs->cds->fieldOffset variable is used to shift between components in the nrs->S array.

```c
void UDF_Setup(nrs_t* nrs)
{
  mesh_t* mesh = nrs->mesh;
```

(continues on next page)
int num_quadrature_points = mesh->Np * mesh->Nelements;

for (int n = 0; n < num_quadrature_points; n++) {
    float x = mesh->x[n];
    float y = mesh->y[n];
    float z = mesh->z[n];

    nrs->U[n + 0 * nrs->fieldOffset] = sin(x) * cos(y) * cos(z);
    nrs->U[n + 1 * nrs->fieldOffset] = -cos(x) * sin(y) * cos(z);
    nrs->U[n + 2 * nrs->fieldOffset] = 0;
    nrs->P[n] = 101325.0;
    nrs->S[n + 0 * nrs->cds->fieldOffset] = 573.0;
    nrs->S[n + 1 * nrs->cds->fieldOffset] = 100.0 + z;
}
}

1.6.6 Grabbing User .par Settings

nekRS conveniently allows the user to define their own parameters in the .par file that can then be accessed in the .udf functions. This is useful for programmatically setting boundary conditions, forcing terms, and many other simulation settings. For instance, suppose that the initial condition for velocity will vary from run to run and is possibly used in several places in the .udf functions. Rather than continually edit the .udf file (which will require repeated just-in-time compilation), these settings can be set with user-defined parameters in the .par file.

As an example, we will define a parameter named initialVelocity in the VELOCITY block.

```
[VELOCITY]
residualTol = 1e-6
density = 1.5
viscosity = 2.4e-4
boundaryTypeMap = inlet, wall, wall, wall, wall, outlet
initialVelocity = 1.5
```

To access this value in the .udf functions, call the extract(String key, String value, T & destination) function on nrs->par as follows.

```
void UDF_Setup(nrs_t* nrs)
{
    double initial_Vz;
    nrs->par->extract("velocity", "initialvelocity", initial_Vz);

    mesh_t* mesh = nrs->mesh;
    int num_quadrature_points = mesh->Np * mesh->Nelements;

    for (int n = 0; n < num_quadrature_points; n++) {
        nrs->U[n + 0 * nrs->fieldOffset] = 0;
        nrs->U[n + 1 * nrs->fieldOffset] = 0;
        nrs->U[n + 2 * nrs->fieldOffset] = initial_Vz;
    }
}
```
The extracted user parameter can then be used throughout the .udf functions, as well as propagated to the device kernels as described in Section Defining Variables to Access in Device Kernels.

1.6.7 Defining Variables to Access in Device Kernels

The customization of a nekRS problem to a specific case is one with both the host-side user functions in the .udf file, as well as device-side user functions in the .oudf file. For convenience purposes, nekRS supports setting non-pointer-type variables in the .udf file that are accessible in the device kernels in the .oudf file. This section shows an example of this usage.

Suppose that a device kernel requires a parameter representing a pressure gradient, which is then used to determine a forcing kernel. One option would be to pass that pressure gradient to the device kernel through its function parameters. The kernel in the .oudf file would look something like the following.

```c
@kernel void myForcingKernel(const dfloat dp_dx, /* more parameters */) {
    double foo = 2 * dp_dx;
    // do something
}
```

Alternatively, we can define a variable, p_dp_dx, that we set from the .udf file. While this variable propagation can be done in any of the user-defined functions that has nrs as an input parameter, for consistency purposes we will use the UDF_LoadKernels function for this purpose.

To set p_dp_dx to 5.5 from the .udf file, write to the kernelInfo object on the nrs object. The defines/<p_name> syntax indicates that a variable on the device is being declared with a name p_name that will be accessible simply as p_name in the device kernels.

```c
void UDF_LoadKernels(nrs_t * nrs) {
    occa::properties & kernelInfo = *nrs->kernelInfo;

    kernelInfo["defines/p_dp_dx"] = 5.5;
    // other stuff related to loading the kernels
}
```

Then, the kernel would be simplified to the following. You will note that nothing needs to be passed through the kernel function arguments - p_dp_dx is simply available as if it were a local variable to the function.

```c
@kernel void myForcingKernel(/* more parameters */) {
    double foo = 2 * p_dp_dx;
    // do something
}
```

If you grep for kernelInfo["defines in the nekRS source code, you will see that this variable propagation features is also used extensively throughout a normal problem setup. For instance, the number of velocity fields to solve for is propagated to the device in the nrsSetup function.

```c
nrs_t* nrsSetup(MPI_Comm comm, occa::device device, setupAide &options, int buildOnly) {
    (continues on next page)
}
Again, the convention is to precede all such propagated variables with the `p_` prefix. No list of all such variables propagated automatically within a nekRS simulation is maintained, so always check if the information you’d like to propagate is perhaps already automatically propagated.

### 1.6.8 Setting Boundary Conditions with Device Kernels

Because all nekRS solves are performed on the device, boundary conditions on the solution (which may change from time step to time step and be arbitrary functions of the solution itself) are also applied on the device. The types of boundary conditions on each solution field are specified in the `.par` file with the `boundaryTypeMap` key.

### 1.6.9 Setting Custom Properties

Custom material properties can be set for the flow and passive scalar equations by assigning the `udf.properties` function pointer to a function with a signature that takes the `nrs` pointer to the nekRS solution object, the simulation time `time`, the velocity solution on the device `o_U`, the passive scalar solution on the device `o_S`, the flow material properties on the device `o_UProp`, and the passive scalar material properties on the device `o_SProp`.

This section provides an example of setting \( \mu \) and \( \rho \) for the flow equations and \( k \) and \( \rho C_p \) for two passive scalars. Suppose our problem contains velocity, pressure, temperature, and two passive scalars. The `[VELOCITY]`, `[PRESSURE]`, `[TEMPERATURE]`, `[SCALAR01]`, and `[SCALAR02]` sections of the `.par` file would be as follows. Because we will be setting custom properties for the pressure, velocity, and first two passive scalars (temperature and `SCALAR01`), we can let nekRS assign the default values of unity to all properties for those governing equations until we override them in our custom property function. We still need to define the material properties for `SCALAR02`, however, because we will not be overriding those properties in our function.

```
[PRESSURE]
residualTol = 1e-6

[VELOCITY]
boundaryTypeMap = v, 0, W
residualTol = 1e-8

[TEMPERATURE]
boundaryTypeMap = t, 0, I
residualTol = 1e-8

[SCALAR01]
boundaryTypeMap = t, 0, I
residualTol = 1e-8

[SCALAR02]
boundaryTypeMap = t, 0, t
residualTol = 1e-7
```
Also suppose that our problem contains conjugate heat transfer, such that some of the mesh is fluid while some of the mesh is solid.

In UDF_Setup, we next need to assign an address to the udf.properties function pointer to a function with the correct signature where we eventually assign our custom properties. Our UDF_Setup function would be as follows.

```c
void UDF_Setup(nrs_t* nrs)
{
    udf.properties = &material_props;
}
```

Here, `material_props` is our name for a function in the .udf file that sets the material properties. Its name is arbitrary, but it must have the following signature.

```c
void material_props(nrs_t* nrs, dfloat time, occa::memory o_U, occa::memory o_S,
                     occa::memory o_UProp, occa::memory o_SProp)
{
    // set the material properties
}
```

This function is called after the solve has been performed on each time step, so the material properties are lagged by one time step with respect to the simulation.

**Note:** You must place the `material_props` function before UDF_Setup (and before any other function that uses `material_props`) in the .udf file in order for the just-in-time compilation to succeed.

Suppose we would like to set \( \rho = 1000.0 \) and \( \mu = 2.1e-5e^{-\phi_0/500}(1+z) \) for the flow equations; because only the fluid domain has flow, we do not need to set these properties on the solid part of the domain. For the first passive scalar \( \phi_0 \), we would like to set \( (\rho C_p)_f = 2e3(1000 + PV_x) \) and \( k_f = 2.5 \) in the fluid domain, and \( (\rho C_p)_s = 2e3(1000 + PV_x) \) and \( k_s = 3.5 \) in the solid domain. Here, \( P \) is the thermodynamic pressure and \( V_x \) is the \( x \)-component velocity. For the second passive scalar \( \phi_1 \), we would like to set \( \rho C_p = 0 \) and \( k = 5 + \phi_0 \) in both the fluid and solid domains. Our material property function would be as follows. Note that these boundary conditions are selected just to be comprehensive and show all possible options for setting constant and non-constant properties with dependencies on properties - they do not necessarily represent any realistic physical case.

// declare all the kernels we will be writing
static occa::kernel viscosityKernel;
static occa::kernel constantFillKernel;
static occa::kernel heatCapacityKernel;
static occa::kernel conductivityKernel;

void material_props(nrs_t* nrs, dfloat time, occa::memory o_U, occa::memory o_S,
                     occa::memory o_UProp, occa::memory o_SProp)
{
    mesh_t* mesh = nrs->mesh;

    // viscosity and density for the flow equations
    const occa::memory o_mue = o_UProp.slice(0 * nrs->fieldOffset * sizeof(dfloat));
    const occa::memory first_scalar = o_S.slice(0 * cds->fieldOffset * sizeof(dfloat));
```
The o_UProp and o_SProp arrays hold all material property information for the flow equations and passive scalar equations, respectively. In this function, you see six "slice" operations performed on o_UProp and o_SProp in order to access the two individual properties (diffusive constant and time derivative constant) for the three equations (momentum, scalar 0, and scalar 1). The diffusive constant ($\mu$ for the momentum equations and $k$ for the passive scalar equations) is always listed first in these arrays, while the coefficient on the time derivative ($\rho C_p$ for the momentum equations and $\rho C_p$ for the passive scalar equations) is always listed second in these arrays.

To further elaborate, $\mu$ and $\rho$ are accessed as slices on o_UProp. Because viscosity is listed before density, the offset in the o_UProp array to get the viscosity is zero, while the offset to get the density is nrs->fieldOffset * sizeof(dfloat)). $k$ and $\rho C_p$ are accessed as slices in o_SProp. Because the passive scalars are listed in order and the conductivity is listed first for each user, the offset in the o_SProp array to get the conductivity for the first passive scalar is zero, while the offset to get the heat capacity for the first passive scalar is cds->fieldOffset. Finally, the offset in the o_SProp array to get the conductivity for the second passive scalar is 2 * cds->fieldOffset, while the offset to get the heat capacity for the second passive scalar is 3 * cds->fieldOffset.

The viscosityKernel, constantFillKernel, heatCapacityKernel, and conductivityKernel functions are all user-defined device kernels. These functions must be defined in the .oudf file, and the names are arbitrary. For each of these kernels, we declare them at the top of the .udf file. In order to link against our device kernels, we must instruct nekRS to use its just-in-time compilation to build those kernels. We do this in UDF_LoadKernels by calling the udfBuildKernel function for each kernel. The second argument to the udfBuildKernel function is the name of the kernel, which appears as the actual function name of the desired kernel in the .oudf file.

```c
void UDF_LoadKernels(nrs_t* nrs)
{
    viscosityKernel = udfBuildKernel(nrs, "viscosity");
    constantFillKernel = udfBuildKernel(nrs, "constantFill");
}
```
In order to write these device kernels, you will need some background in programming with OCCA. Please consult the OCCA documentation before proceeding\(^1\).

First, let’s look at the `constantFill` kernel. Here, we want to write a device kernel that assigns a constant value to a material property. So that we can have a general function, we will write this such that it can be used to set constant (but potentially different) properties in the fluid and solid phases for conjugate heat transfer applications.

**Note:** Material properties for the flow equations (i.e. viscosity and density) do not need to be specified in the solid phase. If you define flow properties in solid regions, they are simply not used.

The `constantFill` kernel is defined in the `.oudf` file as follows\(^2\). OCCA kernels operate on the device. As input parameters, they can take non-pointer objects on the host (such as `Nelements`, `fluid_val`, and `solid_val` in this example), as well as pointers to objects of type `occa::memory`, or device-side memory. The device-side objects are indicated with the `@restrict` tag.

**Note:** Device-side memory in nekRS is by convention preceded with a `o_` prefix in order to differentiate from the host-side objects. In the initialization of nekRS, most of the simulation data is copied over to the device. All calculations are done on the device. The device-side solution is then only copied back onto the host for the purpose of writing output files.

**Warning:** Because nekRS by default only copies the device-side solution back to the host for the purpose of writing output files, if you touch any host-side objects in your user-defined functions, such as in `UDF_ExecuteStep`, you must ensure that you only use the host-side objects after they have been copied from device back to the host. Otherwise, they would not be “up to date.” You can ensure that the host-side objects reflect the real-time nekRS solution by either (a) only touching the host-side solution on output writing steps (which can be determined based on the `nrs->isOutputStep` variable), or (b) calling the appropriate routines in nekRS to force data to be copied from the device back to the host. For the latter option, please refer to the Copying From Device to Host section.

For this example, we loop over all the elements. The `eInfo` parameter represents a mask, and takes a value of zero for solid elements and a value of unity for fluid elements. Next, we loop over all of the GLL points on the element, or `p_Np`. This variable is set within nekRS to be the same as `mesh->Np` using the device variable feature described in the Defining Variables to Access in Device Kernels section. This particular variable is always available, and you do not need to pass it explicitly into device functions. Finally, we set the value of the `property` to the value specified in the function parameters.

\[\text{heatCapacityKernel} = \text{udfBuildKernel}(\text{nrs}, \text{"heatCapacity"});\]
\[\text{conductivityKernel} = \text{udfBuildKernel}(\text{nrs}, \text{"conductivity"});\]

---

\(^1\) There are many different ways to write OCCA kernels. The examples shown here are by no means the most optimal form, and are only intended for illustration.

\(^2\) OCCA kernels are programmed in OKL, a thin extension to C++. Unfortunately, the pygmentize Python syntax highlighter does not recognize OKL syntax, so these examples here lack syntax highlighting.
const bool is_solid = eInfo[e];

for (int n = 0; n < p_Np; ++n ; @inner(0))
{
    const int id = e * p_Np + n;
    property[id] = fluid_val;
    if (is_solid)
        property[id] = solid_val;
}

Now, let’s look at the slightly more complex conductivity kernel. Here, our function signature is very different from that of the constantFill kernel. While we still pass the number of elements, we no longer need to check whether we are in a fluid element or a solid element, since the conductivity for the second passive scalar is going to be the same in both phases. All that we need to pass in is the coupled scalar scalar, or $\phi_0$ in our material property correlation $k = 5 + \phi_0$ that we listed earlier. The property passed in then should represent the conductivity we are setting.

```cpp
@kernel void conductivity(const dlong Nelements, @restrict const dfloat* scalar,
                          @restrict dfloat* property)
{
    for (dlong e = 0; e < Nelements; ++e ; @outer(0))
    {
        for (int n = 0; n < p_Np; ++n ; @inner(0))
        {
            const int id = e * p_Np + n;
            const dfloat scalar = scalar[id];
            property[id] = 5.0 + scalar;
        }
    }
}
```

A key aspect of writing device kernels is that the device kernel can only operate on non-pointer objects or pointers to device memory. Whatever the form of your material properties, you just need to be sure to pass in all necessary information. Now, let’s look at the even more complex viscosity kernel. Here, we need to pass in the scalar $\phi_0$ and the $z$-coordinate that appear in the viscosity model.

```cpp
@kernel void viscosity(const dlong Nelements, @restrict const dfloat* scalar,
                        @restrict const dfloat* z, @restrict dfloat* property)
{
    for (dlong e = 0; e < Nelements; ++e ; @outer(0))
    {
        for (int n = 0; n < p_Np; ++n ; @inner(0))
        {
            const int id = e * p_Np + n;
            const dfloat scalar = scalar[id];
            const dfloat z = z[id];
            property[id] = 2.1E-5 * exp(-scalar / 500.0) * (1.0 + z);
        }
    }
}
```
The final kernel that wraps up this example is the heatCapacity kernel.

### 1.6.10 Setting Custom Source Terms

Custom source terms can be added to the momentum conservation equation and/or the energy conservation equation by assigning the `udf.uEqnSource` and `udf.sEqnSource` function pointers, respectively, to functions with the appropriate signature. Each of these cases are described separately next. The process is conceptually very similar to the process for declaring custom properties in Setting Custom Properties, so you may find it useful to first review that section.

To set a custom source term for the momentum equation, you must assign the `udf.uEqnSource` function pointer to a function with a signature that takes the `nrs` pointer to the nekRS solution object, the simulation time `time`, the velocity solution on the device `o_U`, and the momentum source term on the device `o_FU`. In UDF_Setup, we need to assign an address to the `udf.uEqnSource` function pointer to a function with the correct signature where we will eventually compute a momentum source. Our UDF_Setup function would be as follows.

```c
void UDF_Setup(nrs_t * nrs)
{
    udf.uEqnSource = &custom_source;
}
```

Here, `custom_source` is our name for a function in the `.udf` file that computes the momentum source. Its name is arbitrary, but it must have the following signature.

```c
void custom_source(nrs_t * nrs, dfloat time, occa::memory o_U, occa::memory o_FU)
{
    // compute the momentum source
}
```

**Note:** You must place the `custom_source` function _before_ `UDF_Setup` (and before any other function that uses `custom_source`) in the `.udf` file in order for the just-in-time compilation to success.

Suppose we would like to add a gravitational force to the $z$ momentum equation, of form $-\rho f g$. For the momentum equation, the source term is defined on a per-mass basis; in other words, we must provide the vector $f$ for a source with strong form $\rho f$. Our custom source function would be as follows.

```c
// declare all kernels we will be writing
static occa::kernel constantFillKernel;

void custom_source(nrs_t * nrs, dfloat time, occa::memory o_U, occa::memory o_FU)
{
    mesh_t * mesh = nrs->mesh;

    // what momentum equation we want to add gravity to
    int component = 2;

    constantFillKernel(nrs->mesh->Nelements, -9.81, component * nrs->fieldOffset, o_FU);
}
```
The constantFillKernel is a user-defined device kernel. This function must now be defined in the .oudf file; the name is arbitrary. In order to link against our device kernels, we must also instruct nekRS to use its just-in-time compilation to build those kernels. We do this in UDF_LoadKernels by calling the udfBuildKernel function for the kernel. The second argument to the udfBuildKernel function is the name of the kernel, which appears as the actual function name of the desired kernel in the .oudf file.

```c
void UDF_LoadKernels(nrs_t * nrs)
{
    constantFillKernel = udfBuildKernel(nrs, "constantFill");
}
```

The constantFill kernel is now defined in the .oudf file as follows.

```c
@kernel void constantFill(const dlong Nelements, const dfloat value,
    const int offset, @restrict dfloat * source)
{
    for (dlong e = 0; e < Nelements; ++e ; @outer(0))
    {
        for (int n = 0; n < p_Np; ++n ; @inner(0))
        {
            const int id = e * p_Np + n + offset;
            source[id] = value;
        }
    }
}
```

### 1.6.11 Solving in Non-Dimensional Form

nekRS can solve its governing equations in either dimensional or non-dimensional form with careful attention to the specification of the material properties. To solve in dimensional form, the density, viscosity, rhoCp, conductivity, and diffusivity parameters in the .par file simply take dimensional forms. Solving in non-dimensional form requires only small changes from the dimensional approach. For the case of constant properties, the transformation to non-dimensional form is trivial, but slightly more care is required to solve in non-dimensional form with variable properties. These two approaches are described next with reference to the incompressible Navier-Stokes model described in *Incompressible Flow Model*.

#### Constant Properties

For the case of constant properties for $\rho, \mu, C_p$, and $k$, solution in non-dimensional form is achieved by simply specifying the non-dimensionalized version of these properties in the .par file. To be explicit, for the momentum and energy conservation equations, the input parameters should be specified as:

- $\rho \rightarrow \rho^\dagger \equiv \frac{\rho}{\rho_0}$
- viscosity $\rightarrow \frac{\mu}{\rho_0 U_L} \mu^\dagger \equiv \frac{\mu_0}{\rho_0 U_L} \frac{\mu}{\rho_0 U_L}$
- rhoCp $\rightarrow \rho^\dagger C_p \equiv \frac{\rho C_p}{\rho_0 C_{p,0}}$
- conductivity $\rightarrow \frac{k}{\rho_0 C_{p,0} U_L} \kappa^\dagger \equiv \frac{k_0}{\rho_0 C_{p,0} U_L} \frac{k}{\rho_0 C_{p,0} U_L}$

For the $k$ and $\tau$ equations, if present, the input parameters for both the $k$ equation should be specified as:

- $\rho \rightarrow 1.0$
- diffusivity $\rightarrow \frac{1}{\rho_0}$
Notice that these non-dimensional forms for the \( k \) and \( \tau \) equations are slightly simpler than the forms for the mean momentum and energy equations - this occurs because nekRS’s \( k-\tau \) model is restricted to constant-property flows, so we do not need to consider \( \rho \neq 1 \) or \( \mu \neq 1 \).

If a volumetric heat source is present, it must also be specified in non-dimensional form as

\[
\dot{q}^\dagger = \frac{\dot{q}}{\rho_0 C_{p,0} U \Delta T/L}
\]

If a source term is present in the momentum conservation equation, that source term must also be specified in non-dimensional form as

\[
s^\dagger = \frac{s}{\rho_0 U^2/L}
\]

where \( s \) is the source term in the dimensional equation, with dimensions of mass / square length / square time.

In addition, all boundary conditions must also be non-dimensionalized appropriately. Some of the more common boundary conditions and their non-dimensionalizations are:

- fixed velocity: \( u_i^\dagger = \frac{u_i}{U} \), i.e. divide all dimensional velocity boundary values by \( U \)
- fixed temperature: \( T^\dagger = \frac{T - T_0}{\Delta T} \), i.e. from all dimensional temperature boundary values, first subtract \( T_0 \) and then divide by \( \Delta T \)
- fixed pressure: \( P^\dagger = \frac{P}{\rho_0 U^2} \), i.e. divide all dimensional pressure boundary values by \( \rho_0 U^2 \)
- heat flux: \( q^\dagger = \frac{q}{\rho_0 C_{p,0} U \Delta T} \), i.e. divide all dimensional heat flux boundary values by \( \rho_0 C_{p,0} U \Delta T \)
- turbulent kinetic energy: \( k^\dagger = \frac{k}{U^2} \), i.e. divide the dimensional turbulent kinetic energy by \( U^2 \)
- inverse specific dissipation rate: \( \tau^\dagger = \frac{\tau}{L/U} \), i.e. divide the dimensional inverse specific dissipation rate by \( L/U \)

If the Prandtl number is unity, then because \( Pe \equiv Re Pr \), the coefficient on the diffusion kernel in both the momentum and energy conservation equations will be the same (for the case of constant properties).

Note: Several of the nekRS input files use syntax inherited from Nek5000 that allows shorthand expressions that are often convenient for the Reynolds and Peclet numbers, which appear as inverses in the non-dimensional equations. Specifying \( \text{conductivity} = -1000 \) is shorthand for \( \text{conductivity} = 1/1000 \).

### Variable Properties

For the case of variable properties, the procedure is similar to the case for constant properties, except that the properties must be specified in the \.oudf\ kernels. It is best practice to simply omit the \rho, viscosity, rhoCp,\ and \conductivity\ fields from the \.par\ file entirely. Then, in the \.oudf\ kernels, you must include kernels that apply the variable properties in the same manner as in \Constant Properties\. See \Setting Custom Properties\ for more information on the kernel setup.

### 1.6.12 Copying From Device to Host

All solutions take place on the host, and data transfer of the solution back to the host must be manually performed by the user if you would like to access \nrs->U, nrs->p, nrs->cds->S,\ or other solution objects, in host-side functions. To copy the solution from the device to the host, use the \nek_ocopyFrom(double time, int tstep)\ routine in the \nekInterfaceAdapter.cpp\ file. This function performs the following actions:

1. Copy the nekRS solution from the nekRS device arrays to the nekRS host arrays - that is, \nrs->o_U\ is copied to \nrs->U,\ and so on. This allows you to access the solution on the host as \nrs->U, nrs->p, nrs->S,\ etc.
2. Copy the nekRS solution from the nekRS host arrays to the Nek5000 backend arrays.

If you only want to access the nekRS host side arrays such as \texttt{nrs->U}, you can skip the second part by directly using \texttt{OCCA} memory copy functions like the following, which copies from the device array \texttt{nrs->o_U} to the host array \texttt{nrs->U}.

\begin{verbatim}
\texttt{nrs->o_U.copyTo(nrs->U);}
\end{verbatim}

### 1.6.13 Writing an Output File

nekRS will automatically write output files according to the \texttt{writeControl} criterion set in the \texttt{.par} file. However, it may be desirable to have finer-grained control of output writing, such as if you want the solution at a specific time step, but that time step is not an integer multiple of \texttt{writeInterval}. In this case, you can force the output file writing to occur by calling the \texttt{outfld(double time, double outputTime)} function in the \texttt{nekrs} namespace. This function performs the following actions:

1. Copy the nekRS solution from the nekRS device arrays directly to the backend Nek5000 arrays.
2. Write an output file.

Note that this function is slightly different from the \texttt{nek_ocopyFrom} function described in the \texttt{Copying Device to Host} section. This function is solely intended for writing output, so no effort is expended in copying the device solution into the nekRS host arrays - that step is bypassed, and the device solution is copied straight into the Nek5000 backend arrays. The \texttt{nek_ocopyFrom} routine should really only be used if you require access to the nekRS solution arrays on the host, while the \texttt{outfld} routine should be used strictly for writing output files.

By default, nekRS will only write the velocity, pressure, and temperature to an output file. However, you may have problem-specific fields that you want to view, such as \( y^+ \). To write other fields to files, nekRS re-uses the functions that are used to write the velocity, pressure, and temperature to write other fields. Note that this imposes limitations on both the dimensionality of fields that can be output, as well as how they are named in the output files. For example, suppose you would like to write three fields to a file:

- \texttt{o_yPlus}, a device array that holds \( y^+ \) values, and
- \texttt{o_Uavg}, a device array that holds a time-averaged velocity field, and
- \texttt{o_rst}, a device array that holds the one component of the Reynolds stress tensor.

To write these three fields to an output file, use the \texttt{writeFld} function as follows. The \texttt{writeFld} function takes eight arguments, and has a signature \texttt{void writeFld(const char* suf, dfloat t, int coords, int FP64, void* o_u, void* o_p, void* o_s, int NSf)}. In this example, the first parameter, "usr", is a three-character prefix that will determine how the new output file is written. While the velocity, pressure, and temperatures are written to files named \texttt{case0.f<time_step>}, where \texttt{case} is the case name and \texttt{<time_step>} is a six-digit number indicating the time step, any additional fields we will write are written to separate files. So for this example, we will write three fields to files named \texttt{usrcase0.f<time_step>}. The next three parameters simply indicate the time step that is being written, whether coordinates are written, and if the results should be written in double precision. Next, the three fields that are to be output are provided. The order is very important - the first of these fields must be of length \texttt{nrs->fieldOffset * nrs->NVfields} because it represents a component vector field (this is how velocity is written in the usual output file). The second of these fields must be of length \texttt{nrs->fieldOffset}, because it represents a non-component field (this is how pressure is written in the usual output file). Finally, the third of these fields must be of length \texttt{nrs->cds->fieldOffset * Nscalar}, because it represents a passive scalar field (this is how the passive scalars are written in the usual output file).

```
void UDF_ExecuteStep(nrs_t* nrs, dfloat time, int tstep)
{
    // get o_yPlus, o_Uavg, and o_rst in the scope of this function
```

(continues on next page)
bool coords = true;
bool FP64 = true;
int Nscalar = nrs->cds->Nscalar;
writeFld("usr", time, coords, FP64, &o_Uavg, &o_rst, &o_yPlus, Nscalar);
}

Warning: writeFld can only write data of type dfloat. So, if you want to write an integer field to a field, you must first convert that data to dfloat.

nekRS’s output system does not have any means by which to understand what these fields represent. Therefore, the names of these fields in the output file will be velocity, pressure, and temperature, even if those names have no relationship to what is being output. Therefore, for this example, the usrcase0.f<time_step> files will contain the following:

- o_Uavg is written to a field named velocity
- o_rst is written to a field named pressure
- o_yPlus is written to a field named temperature

nekRS’s output system requires additional maneuvering if you wish to output more than one of each of these three categories of fields. For instance, suppose you want to output three different fields, o_field1, o_field2, and o_field3, each of size nrs->fieldOffset. Because only one input argument to writeFld can have these dimensions, three separate output files need to be written, and in each of these files, our field of interest is named pressure. To fill the other two field arguments of the writeFld function, a void pointer is passed in to indicate that neither of the other two fields are written.

void UDF_ExecuteStep(nrs_t* nrs, dfloat time, int tstep)
{
    // get o_field1, o_field2, o_field3 in the scope of this function

    bool coords = true;
    bool FP64 = true;
    int Nscalar = nrs->cds->Nscalar;
    occa::memory o_null;
    writeFld("fl1", time, coords, FP64, &o_null, &o_field1, &o_null, Nscalar);
    writeFld("fl2", time, coords, FP64, &o_null, &o_field2, &o_null, Nscalar);
    writeFld("fl3", time, coords, FP64, &o_null, &o_field3, &o_null, Nscalar);
}

This will write three output files, which contain the following.

- fl1case0.f<time_step> contains o_field1, but named pressure
- fl2case0.f<time_step> contains o_field2, but named pressure
- fl3case0.f<time_step> contains o_field3, but named pressure
1.6.14 Visualizing Output Files

nekRS output files all have the form `<case0>.fld<n>`, where `<case>` is the case name and `<n>` is a five-digit number indicating the number of the output file (each output file represents a single time step that is output according to the settings for `writeControl` and `writeInterval` in the `.par` file). These output files are in a custom binary format that requires an additional postprocessing step in order to visualize in Paraview. In the directory where the case files are located, run the `visnek` script:

```
user$ visnek case
```

which will create a `case.nek5000` file that is viewable in Paraview. See *Building the Nek5000 Tool Scripts* for instructions on compiling the `visnek` program.

1.6.15 Calculating the Distance to a Wall

nekRS allows users to access many Nek5000 “backends” through the (optional) `<case>.usr` file. A common use case is to calculate the distance from each GLL point to a boundary, such as for setting initial conditions for turbulent quantities or other closures. The procedure to compute and then use these values is as follows.

First, in the `usrdat2` subroutine, make sure that all boundaries for which you want to compute the distance for are marked as “wall” boundaries in the `cbc` array. In the example shown below, we assume that the mesh already has sidesets defined in it (assigned through Cubit/gmsh/however else the mesh was created). We then loop over all the GLL points and determine if the point is on the boundary of interest by checking if the boundary ID is equal to the sideset of interest. This is done by checking the absolute difference between the bc array and the sideset value of interest (in this example, the sideset is 7). If the boundary ID matches the sideset of interest, then we set the `cbc` array to `W`, or the character that indicates a no-slip wall boundary.

```
subroutine usrdat2
  include 'SIZE'
  include 'TOTAL'
  integer e,f

  n = lx1*ly1*lz1*nelv
  nxz = nx1*nz1
  nface = 2*ldim

  do iel=1,nelv
    do ifc=1,2*ndim
      if (abs((bc(5,ifc,iel,1)-7.0)).lt.1e-4) cbc(ifc,iel,1)= 'W'
    enddo
  enddo

  return
end
```

In other words, if your wall boundaries were instead boundaries 3 and 4, the `if (abs...)` lines in the above example would become:

```
if (abs((bc(5,ifc,iel,1)-3.0)).lt.1e-4) cbc(ifc,iel,1)= 'W'
if (abs((bc(5,ifc,iel,1)-4.0)).lt.1e-4) cbc(ifc,iel,1)= 'W'
```

Next, in the `usrdat3` subroutine, you simply need to call the `dist` function, which loops over all boundaries with `W` type and determines the distance of all GLL points to those boundaries. The result of the calculation should be stored into the `nrs_scptr(1)` pointer, which is then what we will access in the `.udf` file.
subroutine usrdat3
include 'SIZE'
include 'TOTAL'

common /scrach_o1/
w1(lx1*ly1*lz1*lelv)
w2(lx1*ly1*lz1*lelv)
w3(lx1*ly1*lz1*lelv)
w4(lx1*ly1*lz1*lelv)
w5(lx1*ly1*lz1*lelv)

common /scrach_o2/
ywd(lx1,ly1,lz1,lelv)

COMMON /NRSSCPTR/ nrs_scptr(1)
integer*8 nrs_scptr

call distf(ywd,7,'W',w1,w2,w3,w4,w5)
nrs_scptr(1) = loc(ywd)
return
end

In other words, if your wall boundaries were instead boundaries 3 and 4, the call distf... lines in the above example would become:

call distf(ywd,3,'W',w1,w2,w3,w4,w5)
call distf(ywd,4,'W',w1,w2,w3,w4,w5)

Then, you can access the results of the distance-to-wall calculation in the .udf by assigning a pointer to the nek::scPtr(1) array. Note that this call must be within UDF_ExecuteStep so that the Nek5000 backend will have been called first.

void UDF_ExecuteStep(nrs_t * nrs, dfloat time, int tstep)
{
    double * wall_distance = (double *) nek::scPtr(1);
    // then, you can copy it into some device-side memory so you can use it in BCs if you want
    auto mesh = nrs->meshV;
    int n_gll_points = mesh->Np * mesh->Nelements;
    int write_location = 2; // "slice" into which you want to write, in case nrs->o_usrwrk␣→holds other info
    nrs->o_usrwrk.copyFrom(wall_distance, n_gll_points * sizeof(dfloat), write_location *␣→nrs->fieldOffset * sizeof(dfloat));
}
1.6.16 Periodic Boundary Conditions

NekRS supports periodic boundary conditions. To set up a periodic case, first you need to run `exo2nek` to establish the pairings between the periodic sidesets. All this information will be prompted on the screen by `exo2nek`; you will provide the sideset IDs of the periodic boundaries, a search tolerance for identifying paired sides, and a translation vector that points from one of the paired sidesets to the other. For example, if you want to have one periodic surface that is a $z$-plane at $z = -1.0$ that is paired to another $z$-plane at $z = 1.0$, the translation vector would be $(0.0, 0.0, 2.0)$.

After generating the mesh, you then need to modify the sideset IDs inside the `usrdat2` function. Any boundary that is now periodic, you need to set `boundaryID(ifc,iel)` to 0. For all non-periodic boundaries, you need to “renormalize” those boundaries to “begin counting” from 1. For example, consider an original (non-periodic) mesh with sidesets 1, 2, 3, and 4. You run `exo2nek` and set up sidesets 2 and 3 as periodic. Then, in the code snippet below, you would reset sidesets 2 and 3 in `boundaryID` to zero. For the remaining two boundaries (originally 1 and 4), you need to renormalized those to boundaries 1 and 2 (because NekRS wants the boundaries to be ordered sequentially beginning from 1).

```fortran
subroutine usrdat2
include 'SIZE'
include 'TOTAL'
integer e,f

n = lx1*ly1*lz1*nelv
nxz = nx1*nz1
nface = 2*ldim

do iel=1,nelt
do ifc=1,2*ndim
   if (boundaryID(ifc,iel).eq. 1) then
      boundaryID(ifc,iel) = 1
   else if (boundaryID(ifc,iel).eq. 2) then
      boundaryID(ifc,iel) = 0
   else if (boundaryID(ifc,iel) .eq. 3) then
      boundaryID(ifc,iel) = 0
   else if (boundaryID(ifc,iel) .eq. 4) then
      boundaryID(ifc,iel) = 2
   endif
endo
endo
return
end
```

Then, in the other case files, you do not need any boundary conditions for the periodic boundaries - for instance, in the `<case>.par` file for this example, the boundary conditions set in `boundaryTypeMap` would only display the boundary conditions for the non-periodic boundaries (and similarly in the `<case>.oudf` file). Finally, in order to enforce periodic flow with a constant flow rate, specify the `constFlowRate` parameter in the `<case>.par` file, such as

```
[GENERAL]
constFlowRate = meanVelocity=1.0 + direction=Z
```
1.6.17 Stamping Initial Conditions

For many periodic flows, you can save significant computing time by solving the flow equations on a shorter-height mesh, and then “stamping” that solution onto a full-height mesh (where you might then be solving for passive scalar transport). NekRS allows you to “stamp” a partial-height solution onto a full-height mesh using the gfldr utility. To do so, you simply need to call the gfldr function in a loop inside of userchk(). Below, nd represents the number of times you want to stamp a short-height solution to obtain the full-height case and delta represents the height of one short-height domain. So, the example below would represent a previous solution (short.fld) on a short-height domain of height 62.42, that you want to stamp five times onto a new mesh that has a height of 312.1.

```fortran
subroutine userchk()
  include 'SIZE'
  include 'TOTAL'

  ntot = lx1*ly1*lz1*nelv

  do nd = 0,5
    delta = 62.421731741003335
    do i = 1,ntot
      zm1(i,1,1,1) = zm1(i,1,1,1) - delta*nd
    enddo
    call gfldr('short.fld')
    do i = 1,ntot
      zm1(i,1,1,1) = zm1(i,1,1,1) + delta*nd
    enddo
  enddo
return
end
```

1.7 Plugins

NekRS contains several “plugins” that provide both physics models and postprocessing capabilities. NekRS’s RANS and low-Mach models, for instance, are provided as plugins. While significant attention is not provided to most of the inner source code structure of NekRS, these plugins require more in-depth explanation because their usage requires non-trivial modifications to the .udf files. Before reading this page, first consult User-Defined Host Functions (.udf) so that you have the necessary background on each of the .udf functions that will be discussed.
1.7.1 RANS $k-\tau$ Plugin

The RANS $k-\tau$ plugin is available in the src/plugins/RANSktau.hpp header file. In order to add the $k-\tau$ model to your case, you need to include this file in your .udf file and manually add all the requisite parts of the $k-\tau$ methodology. Unless otherwise noted, all code snippets in this section are placed in the .udf file.

First, add the necessary include file at the top of your .udf file:

```cpp
#include "udf.hpp"
#include "plugins/RANSktau.hpp"
```

This is required in order to access the methods in the RANS plugin. The following sections then each describe a step in the RANS model setup using the plugin.

Add the Physics Kernels

The calculations performed to add contributions to the residuals occur within OCCA kernels. In order to add the RANS equations, the corresponding physics kernels must first be included. The RANS kernels are added to by calling the RANSktau::buildKernel function within UDF_LoadKernels:

```cpp
void UDF_LoadKernels(nrs_t * nrs)
{
    RANSktau::buildKernel(nrs);
}
```

The RANSktau::buildKernel function performs two main actions:

1. Propagate the constants in the $k-\tau$ model in RANS Coefficients to become available in OCCA kernels with the approach as described in Defining Variables to Access in Device Kernels.

2. Add the OCCA kernels that will perform the calculations needed to apply the $k-\tau$ model.

Add the Closure Properties Calculation

Next, add the function that will update the properties used in the governing equations. An example is shown in Setting Custom Properties for setting custom user-defined properties for a laminar flow scenario. The necessary steps to add the material properties for the RANS model is much simpler, however, but consists of the same essential steps:

1. Set the udf.properties function pointer to a function local to the .udf file that actually computes the properties

2. Add that property function to the .udf

For the first step, assign the udf.properties function pointer to a function in the .udf with signature void (nrs_t* nrs, dfloat time, occa::memory o_U, occa::memory o_S, occa::memory o_UProp, occa::memory o_SProp). Based on the example shown in Setting Custom Properties, for illustration purposes we will name this function material_properties:

```cpp
void UDF_Setup(nrs_t * nrs)
{
    // other stuff unrelated to properties

    udf.properties = &material_properties;
}
```

Then, for the second step, we need to add the following material_properties function in the .udf file:
void material_props(nrs_t* nrs, dfloat time, occa::memory o_U, occa::memory o_S, occa::memory o_UProp, occa::memory o_SProp)
{
    RANSktau::updateProperties();
}

Warning: nekRS’s $k$-$\tau$ implementation currently requires that the laminar dynamic viscosity and the density are constant. Therefore, you should not have any other material properties being set in this function like there were in Setting Custom Properties.

The RANSktau::updateProperties function performs two main actions:

1. Apply a limiter to $k$ and $\tau$ as described in RANS Models.
2. Compute the turbulent viscosity as $\mu_T \equiv p k \tau$ and then set the diffusion coefficients in the momentum, $k$, and $\tau$ equations to be $\mu + \mu_T$, $\mu + \mu_T / \sigma_k$, and $\mu + \mu_T / \sigma_\tau$, respectively.

Add the Source Terms Calculation

The same passive scalar infrastructure that is used to solve the energy conservation equation is used to solve the $k$ and $\tau$ passive scalar equations. However, these equations clearly have different forms - therefore, we need to explicitly add these unique source terms to the $k$ and $\tau$ equations. While we loaded the RANS kernels in Add Physics Kernels, we still need to add those kernels to the governing equations. An example was provided in Setting Custom Source Terms, but the necessary steps to add the RANS source terms is much simpler, but consists of the same essential steps:

1. Set the udf.sEqnSource function pointer to a function local to the .udf file that actually computes the source terms
2. Add that source term function to the .udf

For the first step, assign the udf.sEqnSource function pointer to a function in the .udf with signature void (nrs_t *nrs, dfloat time, occa::memory o_S, occa::memory o_FS). Based on the example shown in Setting Custom Source Terms, for illustration purposes we will name this function user_q:

```c
void UDF_Setup(nrs_t * nrs)
{
    // other stuff unrelated to the source terms
    udf.sEqnSource = &user_q;
}
```

Then, for the second step, we need to add the following material_properties function in the .udf file:

```c
void user_q(nrs_t *nrs, dfloat time, occa::memory o_S, occa::memory o_FS)
{
    RANSktau::updateSourceTerms();
}
```
**Add the Turbulent Prandtl Number**

For cases with passive scalar equations, you must manually add the additional component to the diffusivity, $\mu_T/Pr_T$. This is done in the function pointer to be the `udf.properties` function pointer after updating the the closure properties for the momentum equation as described in **Add the Closure Properties Calculation**. Building on the closure property example, this section shows an example for applying the additional turbulent contribution to the diffusivity for a case with one passive scalar that represents temperature.

**Note:** Manual adjustment to the conductivity is only required for the passive scalar equations that represent mean flow properties - that is, you do not need to manually adjust the conductivity for other passive scalars that represent turbulence quantities, such as $k$ or $\tau$. But if your case has both temperature and chemical concentration passive scalars, for instance, you will need to perform similar adjustments to the diffusivity in the chemical concentration equation as to the adjustments shown in this example for the temperature passive scalar equation.

The following adjustment to the energy equation diffusion coefficient should be performed in our `material_properties` function:

```cpp
void material_props(nrs_t* nrs, dfloat time, occa::memory o_U, occa::memory o_S,
                    occa::memory o_UProp, occa::memory o_SProp)
{
    // update the momentum equation properties, as described earlier
    RANSktau::updateProperties();

    // fetch the laminar thermal conductivity
    dfloat k_laminar;
    nrs->options.getArgs("SCALAR00 DIFFUSIVITY", k_laminar);

    // manually update the energy equation diffusivity
    const dfloat Pr_T = 0.9;
    occa::memory o_mu_T = RANSktau::o_mue_t();
    occa::memory o_mu = nrs->cds->o_diff + 0 * nrs->cds->fieldOffset * sizeof(dfloat);
    nrs->scalarScaledAddKernel(nrs->Nlocal, k_laminar, 1.0 / Pr_T, o_mu_T, o_mu);
}
```

The `scalarScaledAddKernel` is an OCCA kernel that scales an input by a scalar and then adds a constant scalar to the multiplication. That is, this kernel computes

$$y = a + bx$$

where $a$ is the kernel’s second input parameter, $b$ the third input parameter, and $x$ the fourth input parameter. First, we fetch the laminar thermal conductivity that was set in the `.par` file and save it locally in `k_laminar`. Then, we define the turbulent Prandtl number - for this case, we set it to 0.9. Next, we grab the turbulent viscosity just computed in `RANSktau::updateProperties()` by calling `RANSktau::o_mue_t()`, which simply returns the turbulent viscosity. We will save the turbulent conductivity in the first passive scalar “slot” (because we are adjusting the conductivity for the temperature equation, i.e. the first passive scalar) in `cds->o_diff`, which stores the conductivity (laminar plus turbulent) for all passive scalars. To summarize, the `scalarScaledAddKernel` kernel is adjusting the diffusion coefficient in the temperature passive scalar equation to be

$$\frac{1}{Pe} + \frac{\mu_T^i}{Pr_T}$$

where $Pe$ is the Peclet number. Note that this particular example applies to a non-dimensional case. As described at length in *The k-tau Model*, a dimensional formulation of the $k$-$\tau$ model would instead compute the diffusion coefficient
in the temperature passive scalar equation as
\[ k + \frac{\mu_T}{P_T} C_p \]

**Initialize the RANS Solve**

Finally, the last step to initialize the RANS solve is to call the RANSktau::setup function. This function has signature
\[
\text{void setup(nrs_t * nrs, dfloat mu, dfloat rho, int ifld)} - \text{nrs is the flow simulation object, } \mu \text{ is the constant laminar viscosity, } \rho \text{ is the constant density, and } \text{ifld is the integer corresponding to the } k \text{ scalar. This function should be called in UDF_Setup as follows:}
\]

```cpp
void UDF_Setup(nrs_t * nrs)
{
    // other stuff unrelated to calling RANSktau::setup
    const int scalarFieldStart = 1;
    dfloat mu_laminar, rho;
    nrs->options.getArgs("VISCOSITY", mu_laminar);
    nrs->options.getArgs("DENSITY", rho);
    RANSktau::setup(nrs, mu_laminar, rho, scalarFieldStart);
}
```

As mentioned previously, nekRS’s \( k-\tau \) model is currently restricted to constant laminar dynamic viscosity and constant density, and the values passed into this setup function define those properties.

**Warning:** For consistency, be sure that the viscosity and density passed in to RANSktau::setup are the same as the properties used in the mean flow equations. In the example above, this is ensured by grabbing the VISCOSITY and DENSITY input parameters from the .par file.

Finally, ifld simply indicates where in the sequence of passive scalars the \( k \) scalar is positioned. For instance, if your problem has a temperature passive scalar (scalar 0 by definition) and a chemical concentration passive scalar (which you have indicated as SCALAR01 in the .par file), then the \( k \) scalar should be positioned as the second scalar, and ifld = 2.

**Warning:** It is assumed that in the passive scalar list that ifld corresponds to the \( k \) passive scalar and ifld + 1 corresponds to the \( \tau \) passive scalar. Be sure to order the scalars in the input file to respect this assumption.

### 1.7.2 Low-Mach Plugin

### 1.7.3 Turbulence Statistics Plugin

### 1.7.4 Velocity Recycling Plugin

### 1.8 Just-in-time Compilation

nekRS uses just-in-time compilation to build the functions in the .udf and .oudf case files, as well as for compiling certain fixed-size arrays based on the order of the polynomial approximation or other problem settings.
For most cases, no special actions need to be taken by the user for this process to work correctly. However, a high-level understanding of the just-in-time compilation is useful to know what steps need to be taken to fully clear the cached build files, as well as how to perform the pre-compilation separately from a full run to obtain more accurate runtime measurements.

When nekRS performs just-in-time compilation, object files are created in the .cache directory within the current case directory. To completely clear the cached state of the case, simply delete the .cache directory:

```bash
user$ rm -rf .cache/
```

**Tip:** If you experience strange behavior when running your case during the precompilation step (such as failures to build in COMMON blocks or other parts of the code that you are not touching in the .udf and .oudf files), try deleting the .cache directory and trying again. It is not uncommon for the precompilation process to miss the need to build new versions of object files if you are making frequent changes to the nekRS source. This is also sometimes encountered if you are using multiple nekRS versions in different projects (such as standalone nekRS or nekRS wrapped within a multiphysics coupling application such as ENRICO), but don’t have your environment completely self-consistent.

The precompilation process usually takes on the order of one minute. Depending on the use case, it may be advantageous to force the precompilation separately from the run itself. To precompile the case, use the nrspre script. See the Scripts That Ship with nekRS section for where to find this script.

As an example, to precompile a case with name my_case for a later run with less than or equal to 4 GPUs, the nrspre script would be used as follows. After the precompilation, run as normal with the nrsmpi script, which then skips the precompilation step since the case has already been compiled.

```bash
user$ nrspre my_case 4
user$ nrsmpi my_case 32
```

### 1.9 Doxygen

The Doxygen pages can be found here.

### 1.10 Glossary

**ALE**
Arbitrary Lagrange Eulerian

**AMD**
Advanced Micro Devices

**AMG**
Algebraic Multi Grid

**ANL**
Argonne National Laboratory

**API**
Application Programming Interface

**AVM**
Artificial Viscosity Method described in [Persson]
BDF
Backward Differentiation Formula

CFD
Computational Fluid Dynamics

CFL
Courant-Friedrichs-Lewy

CUDA
Compute Unified Device Architecture, or Nvidia’s API for programming with GPUs

CPU
Central Processing Unit

ENRICO
Exascale Nuclear Reactor Investigative COde that couples nekRS to neutronics tools, see it on github.

FPGA
Field-Programmable Gate Array

GLL
Gauss-Lobatto-Legendre

GPU
Graphics Processing Unit

GUI
Graphical User Interface

HIP
AMD’s API for programming with GPUs

HPC
High Performance Computing

MPI
Message Passing Interface

Nek5000
A high performance CFD spectral element code; see it on github.

OCCA
Open Concurrent Compute Abstraction, an API for interacting with various GPU backends; see it on github.

OKL
OCCA Kernel Language

OpenCL
A standard maintained by the Khronos Group for programming with CPUs, GPUs, FPGAs, and other hardware accelerators.

PSU
Pennsylvania State University

RANS
Reynolds-Averaged Navier-Stokes

SST
Shear Stress Transport

UIUC
The University of Illinois at Urbana-Champaign
1.11 References
BIBLIOGRAPHY


INDEX

A
ALE, 52
AMD, 52
AMG, 52
ANL, 52
API, 52
AVM, 52

B
BDF, 53

C
CFD, 53
CFL, 53
CPU, 53
CUDA, 53

E
ENRICO, 53

F
FPGA, 53

G
GLL, 53
GPU, 53
GUI, 53

H
HIP, 53
HPC, 53

M
MPI, 53

N
Nek5000, 53

O
OCCA, 53
OKL, 53

OpenCL, 53
P
PSU, 53
R
RANS, 53
S
SST, 53
U
UIUC, 53