5/19/2022

CATS User Guide

Manual for Users and Developers

Austin Ladshaw
OAK RIDGE NATIONAL LABORATORY

Contents

Mass Balances 10
Separated Phase Mass Balances 10
Combined Phase Mass Balances 12
Energy Balances 14
Separated Phase Energy Balances 14
Combined Phase Energy Balances 16
Navier-Stokes 18
Incompressible Flow - Built-in MOOSE Module 18
Compressible Flow - CATS Methods 19
Electrochemical Catalysis 20
Butler-Volmer Kinetics 20
Electric Potentials and Current 21
Electrode Phase 21
Electrolyte Phase 22
Equations for Current through System 22
Coupling Butler-Volmer Kinetics to Mass Balance 23
Alternative Unit Basis for Volume Averaging 23
DG vs CG Methods 25
Best Solver Options 26
Executioner 26
Time Schemes 26
Solver Type 26
PETSc Options 26
LU and ILU Methods 29
GASM and ASM Methods 30
GMRES (and GCR) Methods 31
Debugging Issues with Linear Solvers and Preconditioners 32
Line Search 32
Tolerances 33
Time Stepper 33
Preconditioning 34
Finite Difference Preconditioner 34

Single Matrix Preconditioner 34
Example Options 35
Example 1: GMRES with ASM preconditioner and BJacobi sub-preconditioner 36
Example 2: FMGRES with GMRES preconditioner and LU sub-preconditioner 36
Example 3: GMRES with ILU preconditioner and increased factor levels 37
Example 4: Setting up all options 38
Model Outputs 40
Output Options 40
Postprocessors 41
ElementVariableValue 41
SideAverageValue 41
ElementAverageValue 42
AreaPostprocessor 42
VolumePostprocessor 42
Exodus Files 43
Input Files 44
GlobalParams 44
Mesh 45
GeneratedMeshGenerator 45
FileMeshGenerator 46
Problem 46
Functions 46
Variables 47
AuxVariables 49
Kernels 49
DGKernels 49
AuxKernels 49
BCs 50
InterfaceKernels 50
Materials 50
Postprocessors 51
Preconditioning 51
Executioner 51

Outputs 52
Simulation Examples 53
Conservation of Mass 53
The Input File 54
Conservation of Energy 60
Coupling Mass and Energy 62
Adding Gas Properties 62
Coupling Mass and Energy with Simple Gas Properties 64
MOOSE CG Incompressible Navier-Stokes 65
INS Cartesian - Fully Bounded 65
INS Cartesian - Semi-Bounded 68
INS RZ 70
CATS DG Navier-Stokes 70
Variables 70
Kernels/DGKernels 71
Boundary Conditions 73
Other Information 75
Coupling Navier-Stokes, Mass, and Energy 76
Simulations with Numerous Reactions 77
Example 1: Multi-Reaction Adsorption 77
Example 2: Multi-Reaction Adsorption with Bulk Mass Balance Coupling 79
Simulations with Subdomains 80
Simulations with Hybrid FD/FE Microscale Diffusion 83
Packed-bed Adsorption 84
Packed-bed Adsorption (same problem, without Microscale) 86
Monolith CO Light-off 87
Monolith CO Light-off with True Wall Thickness 88
Doing Everything: Fully Coupled Mass-Energy with Microscale Physics 88
Electrochemical Cell Example: Vanadium Flow Battery 92
Running the Code 94
Unit Tests 94
Single Core Command Line 95
Multi-Core Command Line 95

To Do List 96
Specific Kernels 96
High Advection Stabilization 96
Finite Volumes CFD 96
Kernels for Ohmic Heating 96
Kernels 97
ActivityConstraint 97
ArrheniusEquilibriumReaction 98
ArrheniusEquilibriumReactionEnergyTransfer 99
ArrheniusReaction 100
ArrheniusReactionEnergyTransfer 101
ButlerVolmerCurrentDensity 102
ConstMassTransfer 103
ConstReaction 104
CoupledCoeffTimeDerivative 105
CoupledPorePhaseTransfer 106
CoupledSumFunction 107
DivergenceFreeCondition 108
ElectrodeCurrentFromPotentialGradient 108
ElectrodePotentialConductivity 110
ElectrolyteCurrentFromPotentialGradient 111
ElectrolyteCurrentFromIonGradient 113
ElectrolytePotentialConductivity 115
ElectrolyteIonConductivity 117
EquilibriumReaction 119
ExtendedLangmuirFunction 120
ExtendedLangmuirModel 121
FilmMassTransfer 122
GAdvection (DGAdvection) 123
GAnisotropicDiffusion (DGAnisotropicDiffusion) 125
GConcentrationAdvection (DGConcentrationAdvection) 127
GNernstPlanckDiffusion (DGNernstPlanckDiffusion) 128
GNSMomentumAdvection (DGNSMomentumAdvection) 130

GNSViscousVelocityDivergence (DGNSViscousVelocityDivergence) 133
GPhaseThermalConductivity (DGPhaseThermalConductivity) 136
GPoreConcAdvection (DGPoreConcAdvection) 138
GThermalConductivity (DGThermalConductivity) 139
GVarPoreDiffusion (DGVarPoreDiffusion) 140
GVariableDiffusion (DGVariableDiffusion) 142
InhibitedArrheniusReaction 143
InhibitedArrheniusReactionEnergyTransfer 144
InhibitionProducts 145
LangmuirInhibition 146
MaterialBalance 147
MicroscaleCoefTimeDerivative 148
MicroscaleCoupledCoefTimeDerivative 150
MicroscaleCoupledVariableCoefTimeDerivative 152
MicroscaleDiffusion 153
MicroscaleDiffusionInnerBC 154
MicroscaleDiffusionOuterBC 155
MicroscaleScaledWeightedCoupledSumFunction 157
MicroscaleVariableCoefTimeDerivative 158
MicroscaleVariableDiffusion 159
MicroscaleVariableDiffusionInnerBC 160
MicroscaleVariableDiffusionOuterBC 161
ModifiedButlerVolmerReaction 162
PairedLangmuirInhibition 164
PhaseEnergyTransfer 166
PhaseTemperature 167
Reaction 168
ScaledWeightedCoupledSumFunction 169
TimeDerivative 170
VarSiteDensityExtLangModel 171
VariableCoefTimeDerivative 171
VariableCoupledCoeffTimeDerivative 172
VectorCoupledGradient 173

VariableLaplacian 175
VariableVectorCoupledGradient 176
WeightedCoupledSumFunction 177
Boundary Conditions 180
CoupledDirichletBC 180
CoupledNeumannBC 181
CoupledVariableFluxBC 181
CoupledVariableGradientFluxBC 182
DGConcFluxLimitedStepwiseBC 183
DGConcFluxStepwiseBC 184
DGConcentrationFluxBC 185
DGConcentrationFluxLimitedBC 185
DGDiffuseFlowMassFluxBC 186
DGDiffusionFluxBC 187
DGFlowEnergyFluxBC 188
DGFlowMassFluxBC 190
DGFluxBC 190
DGFluxLimitedBC 191
DGFluxLimitedStepwiseBC 193
DGFluxStepwiseBC 193
DGNSMomentumOutflowBC 194
DGPoreConcFluxBC_ppm 195
DGPoreConcFluxBC 197
DGPoreConcFluxStepwiseBC 198
DGPoreDiffFluxLimitedBC 198
DGPoreDiffFluxLimitedStepwiseBC 200
DGVarVelDiffFluxLimitedBC 201
DGVarVelDiffFluxLimitedStepwiseBC 201
DGWallEnergyFluxBC 202
DirichletBC 203
INSNormalFlowBC 204
PenaltyDirichletBC 206
Interface Kernels 207

InterfaceConstReaction 208
InterfaceEnergyTransfer 209
InterfaceMassTransfer 210
Initial Conditions 212
InitialActivity 212
InitialButlerVolmerCurrentDensity 213
InitialDaviesActivityCoeff 214
InitialInhibitionProducts 215
InitialIonicStrength 215
InitialLangmuirInhibition 216
InitialModifiedButlerVolmerReaction 217
InitialPotentialDifference 219
InitialPhaseEnergy 220
Auxiliary Kernels 222
AuxAvgLinearVelocity 222
AuxElectrodeCurrent 223
AuxElectrolyteCurrent 224
AuxErgunPressure 225
AuxFirstOrderRecycleBC 227
AuxPostprocessorValue 227
DarcyWeisbachCoefficient 228
DaviesActivityCoeff 229
ElectrolyteConductivity 230
ErgunCoefficient 232
GasDensity 233
GasEffectiveThermalConductivity 234
GasPropertiesBase 235
GasSolidHeatTransferCoef 236
GasSpecHeat 237
GasSpeciesAxialDispersion 238
GasSpeciesDiffusion 239
GasSpeciesEffectiveTransferCoef 240
GasSpeciesKnudsenDiffusionCorrection 242

GasSpeciesMassTransCoef 242
GasSpeciesPoreDiffusion 244
GasThermalConductivity 244
GasVelocityCylindricalReactor 245
GasViscosity 247
GasVolSpecHeat 248
IonicStrength 249
KozenyCarmanDarcyCoefficient 249
LinearChangeInTime 251
MicroscaleIntegralAvg 251
MicroscaleIntegralTotal 252
MicroscalePoreVolumePerTotalVolume 253
MonolithAreaVolumeRatio 254
MonolithMicroscaleTotalThickness 255
MonolithHydraulicDiameter 256
SchloeglDarcyCoefficient 257
SchloeglElectrokineticCoefficient 258
SimpleFluidPropertiesBase 261
SimpleFluidDensity 261
SimpleFluidDispersion 262
SimpleFluidViscosity 264
SimpleFluidElectrolyteViscosity 265
SimpleFluidFlatSurfaceMassTransCoef 266
SimpleFluidMonolithMassTransCoef 268
SimpleFluidSphericalMassTransCoef 269
SimpleGasPropertiesBase 270
SimpleGasFlatSurfaceMassTransCoef 272
SimpleGasMonolithMassTransCoef 273
SimpleGasMonolithHeatTransCoef 274
SimpleGasSphericalMassTransCoef 275
SimpleGasSphericalHeatTransCoef 277
SimpleGasCylinderWallHeatTransCoef 278
SimpleGasEffectivePoreDiffusivity 279

SimpleGasPoreDiffusivity 280
SimpleGasEffectiveKnudsenDiffusivity 281
SimpleGasKnudsenDiffusivity 282
SimpleGasDispersion 284
SimpleGasDensity 285
SimpleGasViscosity 286
SimpleGasVolumeFractionToConcentration 287
SimpleGasIsobaricHeatCapacity 288
SimpleGasThermalConductivity 288
SolidsVolumeFraction 289
SphericalAreaVolumeRatio 290
TemporalStepFunction 291
VectorMagnitude 292
VoidsVolumeFraction 292
Materials 294
INSFluid 294
Utilities 295
Egret 295
Error 295
Macaw 295

Mass Balances

In CATS, mass is balanced on a moles of a chemical species per total volume per time basis. In principle, you can use any units of mass, volume, and time you want. However, once you couple with the energy balances, your units will begin to matter more significantly. Generally, you will need to make sure that the units for things like reaction rates are consistent in the mass and energy balances when the rate terms are coupled to each respective balance equation. You can apply conversion factors manually to the coupling between the balances (by using different 'weight' or 'scale' factor arguments), however, the software does not implicitly understand what units you work with. This is why it is very important that you …

PAY ATTENTION TO YOUR UNITS AND DERIVE YOUR OWN BALANCES!

This message will be repeated very often in the discussions below. For the rest of this discussion, we will assume the mass balances are to be done in mol/m^3/s for consistency.

Separated Phase Mass Balances

If your domain does not contain a second phase (such as solid particles or air bubbles), or if there are two phases separated by boundaries and separated subdomains defined in your mesh, then the mass balances will not involve any porosity or void-volume correction terms.

For instance, consider the mass balance of a chemical species (C_i in mol/m^3) in an open-channel of a monolith catalyst.

$$\frac{\partial C_i}{\partial t} + \nabla \cdot (\mathbf{v} \cdot C_i) = \nabla \cdot (D \cdot \nabla C_i) + \sum_{\forall j} r_j$$

In this formulation, we also represent a series of reactions (r_j) that involve C_i and occur in the bulk fluid phase. The sign of the specific reaction (r_j) will be negative for a loss of C_i and positive for a gain in C_i. Units of rj are presumed here to be in mol/m^3/s, same as C_i. The velocity vector (**v** in m/s) and diffusion coefficient (D in m^2/s) simulate the transport of the material through the channel.

Additionally, we can include the transfer of mass to another phase (i.e., walls of the channel) through an Interface Kernel (which behaves similar to a boundary condition). Here, $C_{w,i}$ (in mol/m^3) represents the concentration of the ith species in the pore spaces of the wall and k (in m/s) represents the mass transfer rate. Note that this kernel yields combined units of mol/m^2/s, instead of mol/m^3/s. This is because interface kernels (and Boundary Conditions in DG methods) are integrated over the area of the elements to produce a total mass flux.

Mass Flux @ the channel walls: $k(C_i - C_{w,i})$

The other boundary conditions for the mass balances in the open-channel are based on advective flux into and out of the simulation domain. The normal vector (n) is known internally and is even used to determine whether the given boundary is the outlet or inlet boundary (see DGFluxBC). This is sufficient to describe the physics of mass balance in the open-channel.

Inlet Boundary: $(\mathbf{v} \cdot \mathbf{n}) \cdot C_{inlet,i}$

Outlet Boundary: $(\mathbf{v} \cdot \mathbf{n}) \cdot C_i$

For a mass balances in the separate subdomain for the channel walls, we can derive a mass balance in terms of the pore-concentration and pore-diffusion through the washcoat matrix.

$$\varepsilon_w \frac{\partial C_{w,i}}{\partial t} = \nabla \cdot \left(\varepsilon_w D_p \cdot \nabla C_{w,i}\right) + \varepsilon_w \sum_{\forall j} r_{p,j} + (1 - \varepsilon_w) \sum_{\forall k} r_{s,k}$$

In this formulation, ε_w represents the volume of washcoat-pores per total washcoat volume, $C_{w,i}$ is the pore-space concentration (which is specifically moles per volume of washcoat-pores in mol/m^3), $r_{p,j}$ represent reactions that can take place in the pore-spaces (thus, is assumed to have same units as $C_{w,i}$ and need the ε_w conversion), and the $r_{s,k}$ represents surface/adsorption reactions in units of moles per volume of only the solids, i.e., mol/m^3/s (which is why there is the (1-ε_w) unit conversion).

For the washcoat, there is no advective flux boundary condition, but there is the mass flux from the channel wall. This is implemented as an Interface Kernel and does not need to be repeated here or in the input file. There is no other boundary condition to consider for the washcoat since the natural boundary condition is the so-called No Flux boundary.

Lastly, for the immobile species (i.e., surface/adsorbed species) in the system, their own mass balances are based solely on the surface reactions taking place and a site/material balance to account for the losses in reaction site availability as the reactions progress.

$$\frac{dq_i}{dt} = \sum_{\forall k} r_{s,k}$$

$$q_{max} = \sum_{\forall j} n_j q_j + q_e$$

In this representation, the adsorbed concentration qi (in mol/m^3) is based on moles of surface species per volume of only solids, i.e., the same unit basis as $r_{s,k}$. The site balance is closed by invoking a kernel like MaterialBalance to ensure that the total number of occupied and unoccupied sites matches the theoretical total site density. In the above balance, the q_{max} represents the maximum sites (in mol/m^3), n_j represents the number of sites occupied by q_j, and q_j is representative adsorbed species (in mol/m^3) at those sites, and q_e represents empty sites (in mol/m^3).

NOTE: The above is only an example of mass balances. Your problem may vary. For instance, $r_{s,k}$ may be represented in units of mol/m^2/s for adsorption or catalytic reactions. In that case, there is an additional unit conversion term that goes in front (A_s) to represent the ratio of the total reactive surface area per unit of solids volume (in m^{-1}).

PAY ATTENTION TO YOUR UNITS AND DERIVE YOUR OWN BALANCES!

Combined Phase Mass Balances

In some situations, it may be more computationally efficient or more convenient to keep your simulation domain as one continuous mesh, rather than dividing into subdomains for different phases. For instance, if simulating adsorption or catalysis in a packed bed, it is nearly impossible to fully separate the solids from the gases in your grid because the packing of the particles would make the mesh far too complex to construct and divided into all the subdomains for each individual particle. Thus, you would instead have a single simulation domain and "separate" the variables that are in solids or gases by using bulk unit conversion factors that represent a volume average of particle count or void space in each finite mesh element.

For starters, consider the transport of mass through a packed column. We will start by considering the mass balance on the gas species concentration. Just like in the Separated Phase Mass Balances, we will represent the gas-phase concentration with C_i in units of moles per gas volume (mol/m^3). However, for the mass balance on this species, we need to now add a unit conversion factor (ε) that represents the average gas volume per total volume in the mesh.

$$\varepsilon\frac{\partial C_i}{\partial t}+\nabla\cdot(\varepsilon\mathbf{v}\cdot C_i)=\nabla\cdot(\varepsilon D\cdot\nabla C_i)+\varepsilon\sum_{\forall j}r_j-(1-\varepsilon)\cdot k\cdot A_o\left(C_i-C_{w,i}\right)$$

Just like before, we have our velocity vector (**v** in m/s), a diffusion parameter (D in m^2/s), and reactions that can occur in the gas-phase (r_j) with same units as C_i (moles per volume of gas). The concentration term $C_{w,i}$ represents the concentration in the pore-spaces of the solid particles.

NOTE: Often times you will see this type of porous flow use a superficial gas velocity ($\mathbf{v_s}$). Here, the velocity vector **v** is representative of the average velocity of fluid elements in the domain. By definition, the product of average velocity and porosity (ε) is the superficial velocity: $\varepsilon\mathbf{v}=\mathbf{v_s}$.

The new feature in the mass balance is the addition of the mass transfer into the pore-spaces of the solids. Previously, this was introduced as an integrated mass transfer boundary, however, here it must be included in the mass balance since there is no interface boundary on the domain. In addition, since it is not an integrated mass flux term, the regular mass transfer also includes 2 unit conversions: (i) A_o, which represents the outer shell surface area of particles per hard shell volume of the particles in the domain and (ii) (1-ε), which represents the volume of solids per total volume. The culmination of these conversions ensure that each term in the balance has units of moles of gas species per total volume per time.

The boundary conditions for the gas-phase concentrations are essentially the same as before, but we replace the actual velocity with the superficial velocity.

Inlet Boundary: $(\varepsilon\mathbf{v}\cdot\mathbf{n})\cdot C_{inlet,i}$

Outlet Boundary: $(\varepsilon\mathbf{v}\cdot\mathbf{n})\cdot C_i$

Although there is no longer a specified subdomain for the pore-space concentration ($C_{w,i}$), we still need to provide a mass balance for this species. The major difference here is that we would not include diffusion through the pore-space because there is not defined grid or mesh subdomain for it to diffuse through.

$$\varepsilon_w \frac{\partial C_{w,i}}{\partial t} = -k \cdot A_o\left(C_{w,i} - C_i\right) + \varepsilon_w \sum_{\forall j} r_{p,j} + (1 - \varepsilon_w) \sum_{\forall k} r_{s,k}$$

In this formulation, the mass balance is done in units of moles of gas species per total volume of solids per time. Unit conversion factor ε_w is the solid pore volume per bulk solid volume. To convert further to moles per total volume, we would multiple all terms by (1-ε), i.e., bulk solids volume per total volume, thus we can forgo that step since it would factor out of each. The $r_{s,k}$ factor has units of moles per volume of solids (not bulk volume of solids), which is why it gets a factor of (1-ε_w).

The immobile species in this example (q_i) is formulated exactly the same as in the Separated Phase Mass Balances.

$$\frac{dq_i}{dt} = \sum_{\forall k} r_{s,k}$$

$$q_{max} = \sum_{\forall j} n_j q_j + q_e$$

In this representation, the adsorbed concentration qi (in mol/m^3) is based on moles of surface species per volume of only solids, i.e., the same unit basis as $r_{s,k}$. The site balance is closed by invoking a kernel like MaterialBalance to ensure that the total number of occupied and unoccupied sites matches the theoretical total site density. In the above balance, the q_{max} represents the maximum sites (in mol/m^3), n_j represents the number of sites occupied by q_j, and q_j is representative adsorbed species (in mol/m^3) at those sites, and q_e represents empty sites (in mol/m^3).

NOTE: Microscale diffusion simulations for Combined Phase Mass Balances are being developed using a hybridization of finite differences for the microscale regime and finite elements for the macroscale mesh. See MicroscaleDiffusion for computation details. Still under construction!

AGAIN: NOTE: The above is only an example of mass balances. Your problem may vary. For instance, $r_{s,k}$ may be represented in units of mol/m^2/s for adsorption or catalytic reactions. In that case, there is an additional unit conversion term that goes in front (A_s) to represent the ratio of the total reactive surface area per unit of solids volume (in m^{-1}).

AGAIN: PAY ATTENTION TO YOUR UNITS AND DERIVE YOUR OWN BALANCES!

Energy Balances

In CATS, energy balances are done in terms of energy per unit volume per time, specifically with each term having the units of J/m^3/s (or W/m^3) [see note below]. The variables being balances are those of internal energy density (E) and NOT the temperature of a given phase (T). This is the most conservative form of the energy balance as it will not only capture how temperature variations impact system energy, but also how density and specific heat variations impact that energy density as well. In the below sections, we outline and detail the strong forms of energy balances for Separated Phase Energy Balances and Combined Phase Energy Balances.

NOTE: If you use the new SimpleGasProperties system, then you are ***NO LONGER*** forced to use this unit basis. Users can use any length, time, energy, and mass units (within the limits outlined in SimpleGasProperties). An example of this can be found here.

Again: PAY ATTENTION TO YOUR UNITS AND DERIVE YOUR OWN BALANCES!

Separated Phase Energy Balances

By separated phase, we mean that these are energy balances that are on a clearly divided subdomain of a mesh that represents a gas, liquid, or solid phase energy balance. These energy balances would not apply to a domain that does not have a clear subdivision between solids and fluids (such as a packed bed reactor), but would apply to a monolith catalyst wherein the empty channels of the monolith are 1 subdomain and the washcoat is another, separated subdomain.

For instance, consider an energy balance for the fluid phase in the monolith channels.

$$\frac{\partial E_f}{\partial t} + \nabla \cdot \left(\mathbf{v} \cdot E_f\right) = \nabla \cdot \left(K_f \cdot \nabla T_f\right) + \sum_{\forall j} \left(-\Delta H_j\right) r_j$$

In this model equation, E_f (J/m^3) is the energy density variable for the fluid phase, **v** (m/s) is the velocity vector for the fluid elements, K_f (W/m/K) is the fluid thermal conductivity, T_f (K) is the temperature of the fluid phase, r_j (mol/m^3/s) are a series of chemical reactions that may occur in the fluid phase, and ΔH_j (J/mol) are the enthalpies of those reactions. Note that if you need to couple the energy balance with a mass balance, your concentrations in each respective phase should have units of mol/m^3. Otherwise, you will need additional unit conversions to include the heats of reaction.

The energy balance equation for the fluid phase must be coupled to the temperature variable of the fluid phase (and the concentrations/reactions in the fluid phase if they are present). The temperature of the fluid phase must be included in your problem as its own equation and residual (see PhaseTemperature). That temperature will be a function of the energy density, fluid density (ρ_f in kg/m^3), and fluid heat capacity (c_{pf} in J/kg/K) as follows:

$$E_f = \rho_f c_{pf} T_f$$

The boundary conditions for the energy balance of the fluid phase are similar to those of the mass balances (see Separated Phase Mass Balances). Since the fluid phase is in contact with walls of the channel, energy can be transferred from one phase to another (i.e., from wall to fluid or from washcoat to fluid depending on your specific solution domain). That energy transfer will have a very similar form

to the mass transfer at the washcoat walls. Here, h is the heat transfer coefficient (W/m^2/K) and T_w (in K) is the temperature of the washcoat subdomain (or the reactor walls depending on implementation).

$$\text{Energy Transfer @ washcoat/walls:} \qquad h\left(T_f - T_w\right)$$

The above expression would be implemented as an InterfaceKernel (in the case of the washcoat) or as a BoundaryCondition (in the case of the system/reactor walls). Recall that here that this residual yields combined units of J/m^2/s, instead of J/m^3/s. This is because interface kernels (and Boundary Conditions in DG methods) are integrated over the area of the elements to produce a total energy flux.

For the other boundary conditions, we need to consider the flux of energy carried into and out of our domain by the advective fluid velocities. Most commonly, we would know what the fluid temperature, density, and specific heat are at the inlet boundary, but not necessarily the outlet boundary. There for, the mathematical representation for each will be different. However, they are both implemented in the same boundary condition kernel (DGFlowEnergyFluxBC) in CATS. This is because we would implicitly know whether or not to consider the boundary an exit or entrance based solely on the inner product between the velocity vector and the normal vector to the mesh boundary surface.

$$\text{Inlet Boundary:} \qquad (\mathbf{v} \cdot \mathbf{n}) \cdot \rho_f c_{pf} T_{f,inlet}$$

$$\text{Outlet Boundary:} \qquad (\mathbf{v} \cdot \mathbf{n}) \cdot E_f$$

In this monolith channel energy balance example, we also must consider the energy balance in the washcoat areas of the monolith. For simplification, we will assume the gases in the pore spaces of the washcoat are the same temperature as the solids of the washcoat, thus we need to only consider the bulk energy density of the washcoat (E_w in J/m^3) as our variable.

$$\frac{\partial E_w}{\partial t} = \nabla \cdot (K_w \cdot \nabla T_w) + \varepsilon_w \sum_{\forall j} \left(-\Delta H_j\right) r_{p,j} + (1 - \varepsilon_w) \sum_{\forall k} (-\Delta H_k) r_{s,k}$$

In this formulation, K_w (in W/m/K) is either the solids thermal conductivity or an effective washcoat conductivity (see GasEffectiveThermalConductivity), $r_{p,j}$ (in mol/m^3/s) are the gas reactions taking place in the pore-spaces with enthalpy of ΔH_j (J/mol), $r_{s,k}$ (in mol/m^3/s) are the surface/solids reactions taking place with enthalpy of ΔH_k (J/mol), and ε_w is the porosity of the washcoat. Note that each term here has units of energy per washcoat volume per time. The ε_w factors are used because the concentrations in the pore-space are units of mass per pore-volume and/or mass per only solid-volume. If your reaction or concentration terms have different units, then this form of the energy balance will be slightly different.

To recover the bulk temperature of the washcoat (T_w in K), you again use the PhaseTemperature kernel, but with bulk washcoat density (ρ_w in kg/m^3) and bulk washcoat specific heat (c_{pw} in J/kg/K).

$$E_w = \rho_w c_{pw} T_w$$

NOTE: The above is only an example of energy balances. Your problem may vary. For instance, $r_{s,k}$ may be represented in units of mol/m^2/s for adsorption or catalytic reactions. In that case, there is an additional unit conversion term that goes in front (A_s) to represent the ratio of the total reactive surface area per unit of solids volume (in m^{-1}).

PAY ATTENTION TO YOUR UNITS AND DERIVE YOUR OWN BALANCES!

Combined Phase Energy Balances

Following along with the derivation from Combined Phase Mass Balances, there are situations in which your domain has two distinct phases (i.e., solids and fluids), but your mesh may be incapable of clearly and efficiently subdividing those phases into separated subdomains. In this case, we need to derive energy balances for the fluids and solids on an averaged basis using parameters that represent the average volume of solids or fluids within each element of the domain.

For starters, consider the transport of energy through a packed column. We will start by considering the energy balance on the fluid phase. Just like in the Separated Phase Energy Balances, we will represent the fluid energy density with E_f in units of energy per fluid volume (J/m^3). However, for the energy balance on this variable, we need to now add a unit conversion factor (ε) that represents the average gas volume per total volume in the mesh.

$$\varepsilon\frac{\partial E_f}{\partial t}+\nabla\cdot\left(\varepsilon\mathbf{v}\cdot E_f\right)=\nabla\cdot\left(\varepsilon K_f\cdot\nabla T_f\right)+\varepsilon\sum_{\forall j}\left(-\Delta H_j\right)r_j-(1-\varepsilon)\cdot h\cdot A_o\left(T_f-T_s\right)$$

Just like before, we have our velocity vector (**v** in m/s), a fluid conductivity parameter (K_f in W/m/K), and reactions that can occur in the fluid (r_j) with their respective enthalpies ΔH_j in J/mol. The new coupled variable is for the temperature of the solid particles (T_s in K).

NOTE: Often times you will see this type of porous flow use a superficial gas velocity ($\mathbf{v_s}$). Here, the velocity vector **v** is representative of the average velocity of fluid elements in the domain. By definition, the product of average velocity and porosity (ε) is the superficial velocity: $\varepsilon\mathbf{v}=\mathbf{v_s}$.

The new feature in the energy balance is the addition of the energy transfer with the solid particles. Previously, this was introduced as an integrated energy transfer boundary, however, here it must be included in the energy balance since there is no interface boundary on the domain. In addition, since it is not an integrated energy flux term, the regular energy transfer also includes 2 unit conversions: (i) A_o, which represents the outer shell surface area of particles per hard shell volume of the particles in the domain and (ii) (1-ε), which represents the volume of solids per total volume. The culmination of these conversions ensure that each term in the balance has units of energy of the fluid per total volume per time.

The temperature of the fluid phase can be recovered using the PhaseTemperature kernel, with a slightly different equation, which includes the ε factor for.

$$E_f=\rho_f c_{pf}T_f$$

The boundary conditions for the fluid energy flux at the open boundaries are essentially the same as before, but we replace the actual velocity with the superficial velocity.

Inlet Boundary: $\left(\varepsilon\mathbf{v}\cdot\mathbf{n}\right)\cdot\rho_f c_{pf}T_{f,inlet}$

Outlet Boundary: $\left(\varepsilon\mathbf{v}\cdot\mathbf{n}\right)\cdot E_f$

In addition to the energy flow boundary conditions, you will also have energy flux boundary conditions at the walls of the domain. We represent that boundary as a DGWallEnergyFluxBC where T_w is the temperature of the wall (in K), h_w is the heat transfer coefficient (in W/m^2/K), and ε accounts for the fraction of the fluid phase that is actually in contact with the wall surface area.

Wall Boundary: $$h_w \cdot \varepsilon \cdot (T_f - T_w)$$

For the energy balance in the solid particles, we include the energy transfer from the fluid phase along with the solids thermal conductivity or *effective* solids conductivity (K_s in W/m/K). This conductivity term is included such that we can simulate the impact of neighboring particles exchanging energy with each other. We also include heat of reactions for reactions in the pore-spaces ($r_{p,j}$ in mol/m^3/s) and reactions on the solids/surfaces ($r_{s,k}$ in mol/m^3/s), along with the respective enthalpies for each. The ε_s parameter is the porosity of the solid particles themselves. Note that each term in this energy balance has units of energy density per unit volume of bulk solids per time.

$$\frac{\partial E_s}{\partial t} = \nabla \cdot (K_s \cdot \nabla T_s) - h \cdot A_o(T_s - T_f) + \varepsilon_s \sum_{\forall j} (-\Delta H_j) r_{p,j} + (1 - \varepsilon_s) \sum_{\forall k} (-\Delta H_k) r_{s,k}$$

There is not energy exchange boundary condition needed at the open ends of the channel, since the solid particles are assumed stationary in the domain and we already account for energy exchange with the fluid phase volumetrically. However, we do need to include a boundary condition for energy exchange at the walls of the reactor system since the particles are in direct contact with the walls due to the tight packing of the particles. Just like with the fluid phase wall boundary, the solid phase wall boundary will involve a contact fraction (1-ε) to represent the fraction of the area at the wall that the solids are in contact with.

Wall Boundary: $$h_w \cdot (1 - \varepsilon) \cdot (T_s - T_w)$$

To recover the bulk temperature of the solids (T_s in K), you again use the PhaseTemperature kernel, but with bulk particle density (ρ_s in kg/m^3) and bulk particle specific heat (c_{ps} in J/kg/K).

$$E_s = \rho_s c_{ps} T_s$$

NOTE: The above is only an example of energy balances. Your problem may vary. For instance, $r_{s,k}$ may be represented in units of mol/m^2/s for adsorption or catalytic reactions. In that case, there is an additional unit conversion term that goes in front (A_s) to represent the ratio of the total reactive surface area per unit of solids volume (in m^{-1}).

PAY ATTENTION TO YOUR UNITS AND DERIVE YOUR OWN BALANCES!

Navier-Stokes

The CATS module does contain some kernels and methods (DGNSMomentumAdvection, DGNSMomentumOutflowBC, and VectorCoupledGradient) to create a DG implementation of Navier-Stokes equations. However, it can also be linked and/or coupled with the Navier-Stokes module in MOOSE built-in with the MOOSE framework (mooseframework.org/modules/navier_stokes/). Provided below are some brief details and instructions for invoking the pre-built Navier-Stokes solvers. Example input files to run simulations are provided under the Examples section. An example specifically showing the utilization of our own custom DG implementation of incompressible and compressible Navier-Stokes can be seen here.

Incompressible Flow - Built-in MOOSE Module

For systems in which the density variations are very small (< %10) and/or systems involving Mach numbers well below 0.3 (i.e., velocities less than 100 m/s), it is generally acceptable to assume that the fluid phase is incompressible. This assumption greatly simplifies and stabilizes the simulation of the conservation of momentum. The incompressible, convective form of the Navier-Stokes equations are represented by the following 2 equations:

$$\text{Momentum:} \qquad \rho_f \frac{\partial \mathbf{v}}{\partial t} + \rho_f (\mathbf{v} \cdot \nabla)\mathbf{v} = -\nabla \bar{p} + \mu_f \nabla^2 \mathbf{v} + \rho_f \mathbf{g} + \mathbf{f}$$

$$\text{Continuity:} \qquad (\nabla \cdot \mathbf{v}) = 0$$

In the above equations, ρ_f is the density of the fluid (in kg/m^3), μ_f is the viscosity of the fluid (in kg/m/s), **v** is the velocity field vector (in m/s), **g** is a gravitational constant (in m/s^2), **f** is a body force vector function (in N/m^3), and $\bar{p}$ is the dynamic pressure (in Pa) caused by the fluid motion. Note that since this representation only involves dynamic pressure, recovering the total pressure in a domain would require some other kernels or auxiliary system (see AuxErgunPressure for a pressure calculation in packed columns or monolith structures.)

Solving the incompressible Navier-Stokes equations requires both the momentum and continuity equations, as well as a set of variables for each velocity component (x, y, and z) and the dynamic pressure. Dynamic pressure is resolved through the continuity equation. However, the continuity equation itself is insufficient for determining the total pressure variations in the domain. It is only sufficient for resolving the gradients of dynamic pressure needed in the Navier-Stokes incompressible momentum equation.

In addition to the above equations for the interior of the domain, you must provide boundary conditions for the velocity components. For the incompressible Navier-Stokes module in MOOSE, you will need to provide some form of inlet/outlet flow condition for open boundaries and a “no slip” condition for closed boundaries or walls. For details on boundary condition options, see PenaltyDirichletBC, DirichletBC, and/or INSNormalFlowBC. In general, you will use PenaltyDirichletBC or DirichletBC to force 0 velocity for all velocity components at walls (i.e., the “no slip” condition) and you will use INSNormalFlowBC, PenaltyDirichletBC, or DirichletBC to specify a specific velocity flux or velocity profile at the inlet or outlet boundaries.

NOTE: You can also use INSNormalFlowBC to impose a ‘No Penetration’ condition, rather than ‘No Slip’.

Compressible Flow - CATS Methods

Also in CATS is an implementation of compressible Navier-Stokes flow using DG methods, which were developed in such that they would technically be applicable to both incompressible and compressible simulation conditions. Thus, it is generally advised to always assume your simulation involves a compressible fluid. In this way, your fluid flow field will adjust to changes in fluid density during the simulation.

The formulation for compressible Navier-Stokes flow is as follows:

$$\frac{\partial(\rho\mathbf{u})}{\partial t} + \nabla\cdot(\rho\mathbf{u}\otimes\mathbf{u}) = -\nabla p + \mu\nabla^2\mathbf{u} + \frac{1}{3}\mu\nabla(\nabla\cdot\mathbf{u})$$

$$\frac{\partial\rho}{\partial t} + \nabla\cdot(\rho\mathbf{u}) = 0$$

In most situation, you can assume that the fluid mass exists at a pseudo-steady-state. In such a case, the time derivative of density becomes zero and the equations simplify to:

$$\rho\frac{\partial(\mathbf{u})}{\partial t} + \nabla\cdot(\rho\mathbf{u}\otimes\mathbf{u}) = -\nabla p + \mu\nabla^2\mathbf{u} + \frac{1}{3}\mu\nabla(\nabla\cdot\mathbf{u})$$

$$\nabla\cdot(\rho\mathbf{u}) = 0$$

In CATS, we implement these equations in a piecewise manner, such that each component of velocity is its own variable and has its own kernels. This would then transform our conservation of momentum equation into the following:

$$\rho\frac{\partial u_i}{\partial t} + \nabla\cdot(\rho\mathbf{u}\cdot u_i) = -\mathbf{n}_i\cdot\nabla p + \mu\nabla^2 u_i + \frac{1}{3}\mu\nabla\left(\mathbf{D}_{ix}\cdot\nabla u_x + \mathbf{D}_{iy}\cdot\nabla u_y + \mathbf{D}_{iz}\cdot\nabla u_z\right)$$

Where *i* represents the *i*-th velocity component (x, y, or z). As such, $\mathbf{n}_i$ is a unit vector in the *i*-th direction, and $\mathbf{D}_{ix}$, $\mathbf{D}_{iy}$, and $\mathbf{D}_{iz}$ are tensor matrices whose entries are all 0, except for entry at the *i*-th row and x (or y or z) column, which has a 1.

The continuity equation is used to resolve the pressure gradients in the flow field. To solve this equation, you must enforce a pressure value at the exits/outlets of the domain. The standard is to specify that the exits/outlets of the domain have 0 pressure. In this way, the pressure calculated inside the domain represents only the dynamic pressure (and not absolute pressure).

The BCs for the velocity variables should be setup to give a 'No Slip' condition at the walls of the domain and an outflow of momentum flux at each exit/outlet of the domain. Additional external forces, such as gravity, can be added to this problem as needed through other kernels acting on each component of velocity. An example of the usage of the CATS Navier-Stokes kernels can be found here.

NOTE: Because we implement Navier-Stokes in a piece-wise fashion, this formulation is valid ONLY in the Cartesian coordinate system (i.e., <x, y, z> coordinates).

Electrochemical Catalysis

In addition to standard catalysis systems, CATS also has developed kernels for electrochemical catalysis for batteries, flow batteries, and/or electrochemical conversion systems. To work out these types of problems, there are a number of additional kernels to append to a standard mass balance, as well as a new set of kernels and variables to work out the electrical potential and current through the system. Each set of kernels are discussed briefly below.

Butler-Volmer Kinetics

The standard method by which the kinetics of charging and discharging take place is based on redox reactions between electron storing ions in the system. Those redox reactions are mostly commonly described through Butler-Volmer relationships, which can relate the redox kinetics of a reaction to the current density produced from that reaction. In CATS, we separate out the reaction rates from the current density produced, and then relate them together through separate kernels. In doing so, we can change at will how we choose to represent the rates of reaction without needing to change how that rate is coupled to produce current.

In CATS, redox reaction rates are formulated by the ModifiedButlerVolmerReaction kernel. We refer to this formulation as modified because (i) this term accounts only for the rate of reaction and (ii) this term reformulates the standard Butler-Volmer reaction rate to use the actual potential difference without the 'overpotential' term, which in itself include natural log transformation of the reaction at steady-state. This formulation is mathematically equivalent to the original Butler-Volmer reaction rate but is numerically more stable to implement.

We can represent this reaction term as follows. Consider a redox reaction of the following form.

$$\sum_{jR} s_{jR} c_{jR} \leftrightarrow \sum_{iO} s_{iO} c_{iO} + ne^-$$

Here, s_{jR} and s_{iO} are the stoichiometry for the reduced state and oxidized state, respectively, and n is the number of electrons transferred. Given that the activate energy of a redox reaction is directly proportional to the potential difference in the system, we can now write a rate of reaction as follows.

$$(-r) = k_a C_R exp\left(\frac{[1-\alpha]nF[\Delta\phi]}{RT}\right) - k_c C_O exp\left(\frac{-\alpha nF[\Delta\phi]}{RT}\right)$$

Where C_R and C_O are defined as…

$$C_O = \prod_{iO} c_{iO}^{s_{iO}} \qquad C_R = \prod_{jR} c_{jR}^{s_{jR}}$$

and $\Delta\phi$ is defined as… $\Delta\phi = \phi_s - \phi_e$. Here, ϕ_s and ϕ_e are electrode and electrolyte potentials, respectively.

In these formulations, our concentrations should always have units of moles per volume, where the volume is generally in a cubic length (not liters). Because these reactions generally take place on the surface of a solid phase (i.e., electrode), the units for k_a and k_c are typically found in length per time. The F parameter represents Faraday's constant (96485.3 C/mole). The α parameter is the electron transfer coefficient, which is usually taken to be 0.5 for a symmetric electron transfer.

The rate parameters k_a and k_c are related to the equilibrium potential of the redox reaction and would be calculated as follows.

$$k_c = k \cdot exp\left(\frac{\alpha E_0' nF}{RT}\right)$$

$$k_a = k \cdot exp\left(-\frac{(1-\alpha) E_0' nF}{RT}\right)$$

Where E_0' is the equilibrium potential (in V) and k is the standard rate for the redox reaction (usually in length per time). These 2 parameters are the most common kinetic parameters you would find in literature.

The rate of the redox reaction is related to current density (J) through the following relationship.

$$J = nA_s F(-r)$$

Where n is the number of electrons exchanged in the reaction, A_s is the specific surface area that the reaction takes place on (in solid surface area per solid volume, or total volume **[NOTE: in the case shown here, it is per total volume]**), and ($-r$) is the reaction rate term. The kernel in CATS to couple reaction rates to current density is ButlerVolmerCurrentDensity.

NOTE on Units: Every literature source has slightly different variations on how to couple kinetics with current density. The most important thing to remember is that current density (J) must ALWAYS have units of Coulombs per volume per time (e.g., $C/m^3/s$). PAY CLOSE ATTENTION to whether or not it needs to be per TOTAL VOLUME or per SOLID VOLUME. This may change how specific area (A_s) is defined for your problem. Additionally, this definition means that your reaction rate ($-r$) must ALWAYS have units in moles per area per time (e.g., $mol/m^2/s$). Thus, depending on how the reaction rate term is defined, the k rate parameter may have different units to facilitate the need to always end up with this set of units for the overall rate.

Electric Potentials and Current

Electrochemical systems are inherently a multi-phase problem, therefore, at a minimum there needs to be variables for the potential in the electrode (or solid phase) and electrolyte (or fluid phase). We will denote these variables as ϕ_s and ϕ_e for electrode and electrolyte potentials, respectively. Each of these can be resolved with Poisson-like equations using continuous Galerkin shape functions for greater efficiency. Details for the equations of each are provided below. Both are based on a divergence-free condition on the current.

Electrode Phase

The potential in the electrode is the simplest equation and is defined as follows.

$$\nabla \cdot \left((1-\varepsilon)\sigma_s \nabla \phi_s\right) - \sum_{\forall k} J_k = 0$$

Where ε is the bulk porosity of the electrode, σ_s is the electric conductivity of the electrode (in C/V/length/time), and J_k is the Butler-Volume current density from the k^{th} redox reaction occurring in the electrodes (often there is only 1 redox reaction occurring in the electrode to drive the current, but here we generically represent the possibility of more). The CATS kernel for the divergence of the electrode potential gradients is ElectrodePotentialConductivity.

NOTE: The exact form of this equation may change depending on the units for J_k. In the above formulation, it assumes J_k is on a TOTAL VOLUME basis. If we use a SOLID VOLUME basis on J_k, then we do not need to include the (1 - ε) term.

Electrolyte Phase

The potentials in the electrolyte are much more complex. This is because there is some current created by gradients of ions in solution and also the fact that the conductivity of the electrolyte is itself a function of the local ion concentrations. Mathematically, we can define the potentials in the electrolyte phase as follows.

$$\nabla \cdot \left(K_{eff} \nabla \phi_e\right) + \nabla \cdot \left(F\varepsilon \left[\sum_{j \in \text{ions}} z_j D_j \nabla c_j \right] \right) + \sum_{\forall k} J_k = 0$$

Where the effective electrolyte conductivity (K_{eff}) is defined as…

$$K_{eff} = \varepsilon \frac{F^2}{RT} \left[\sum_{j \in \text{ions}} {z_j}^2 D_j c_j \right]$$

In these equations, we only need to sum over the ions in solution because only ions will contribute to electric potential and current in the system. The parameters are (i) z_j the valence of the ion, (ii) D_j the diffusivity of the ions, (iii) c_j the concentration of ions, (iv) and all other parameters are same as previous defined.

Each of the added terms above are separate kernels in CATS. The effective conductivity term comes from ElectrolytePotentialConductivity and the ion gradients term comes from ElectrolyteIonConductivity. Each can be used independently, but you generally use all together.

NOTE: To couple a set of redox current density terms to both phases, you would use the CATS kernel ScaledWeightedCoupledSumFunction for each. The sign of the weights will be opposite for the electrode and electrolyte phases.

Equations for Current through System

The current is solved separately, in the system using standard kernels, however, it is still recommended to use Discontinuous Galerkin shape functions for greater stability. The current in the electrolyte ($\mathbf{i_e}$) and electrode ($\mathbf{i_s}$) would be as follows.

$$\mathbf{i_e} = -\left(K_{eff}\nabla\phi_e\right) - \left(F\varepsilon\left[\sum_{j\in\text{ions}} z_j D_j \nabla c_j\right]\right)$$

$$\mathbf{i_s} = -\left((1-\varepsilon)\sigma_s\nabla\phi_s\right)$$

Where all parameters are as previously defined. The CATS kernels for this are ElectrolyteCurrentFromPotentialGradient and ElectrolyteCurrentFromIonGradient, for the electrolyte current, and ElectrodeCurrentFromPotentialGradient for the electrode current. Each of these are done piecewise, so each direction of current is determined by a separate variable (for each direction) and separate sets of kernels. See the above kernels for more information.

Coupling Butler-Volmer Kinetics to Mass Balance

Since we have separated the specifics of the redox reaction rates from the definition of the Butler-Volmer current density, this allows us to couple the redox reactions to mass balances using the same original set of kernels for standard mass balances (i.e., ScaledWeightedCoupledSumFunction). However, we also need to include the impact of potential on the transport of the ions in the system. This can be done through the Nernst-Planck relationship (for which we have a set of DG kernels for this portion of ion transport, GNernstPlanckDiffusion and DGNernstPlanckDiffusion). Mathematically, we can represent the mass balance as follows.

$$\varepsilon\frac{\partial c_i}{\partial t} + \nabla\cdot(\mathbf{v}\cdot c_i) = \nabla\cdot(\varepsilon D_i\cdot\nabla c_i) + \nabla\cdot\left(\frac{z_i F}{RT}D_i\cdot\varepsilon\cdot c_i\nabla\phi_e\right) + A_s\sum_k u_{k,i}(-r_k)$$

Where $u_{k,i}$ represents the molar impact of redox reaction r_k on the ion concentration c_i.

NOTE: You can also include other reactions into this material balance that are NOT redox specific reactions. This would be done by adding another set of reaction rate variables in the summation and not just summing over the redox reactions in the system.

Alternative Unit Basis for Volume Averaging

In the above discussion, the current density term (J) had units of Coulombs per total volume per time (i.e., the same units as the other terms in the electric potential equations). This is a unit basis based on the fact that the specific area (A_s) had units of solids surface area per total volume. As a user, you need to be careful with what the units of A_s are given as. If A_s is given instead as solids surface area per solids volume, then some additional conversion factors would need to be applied for coupling the reactions with the potentials AND the concentrations.

For example, if we assume A_s is solids area per solids volume, then the balance equations become as follows.

$$\nabla \cdot \left(K_{eff} \nabla \phi_e\right) + \nabla \cdot \left(F\varepsilon \left[\sum_{j \in \text{ions}} z_j D_j \nabla c_j \right] \right) + (1-\varepsilon) \sum_{\forall k} J_k = 0$$

$$\nabla \cdot \left((1-\varepsilon)\sigma_s \nabla \phi_s\right) - (1-\varepsilon) \sum_{\forall k} J_k = 0$$

$$\varepsilon \frac{\partial c_i}{\partial t} + \nabla \cdot (\mathbf{v} \cdot c_i) = \nabla \cdot (\varepsilon D_i \cdot \nabla c_i) + \nabla \cdot \left(\frac{z_i F}{RT} D_i \cdot \varepsilon \cdot c_i \nabla \phi_e \right) + (1-\varepsilon) A_s \sum_k u_{k,i} (-r_k)$$

NOTE: This does also depend on the units of the rate term (r), however, this is usually always in moles exchanged per solids area per time.

PAY ATTENTION TO YOUR UNITS AND DERIVE YOUR OWN BALANCES!

DG vs CG Methods

MOOSE has support for both Discontinuous Galerkin (DG) and Continuous Galerkin (CG) Finite Element (FE) analysis. In CATS, we have specifically expanded upon the DGFE methods for mass, energy, and momentum balances for chemical catalysis. Most of the kernels developed in this library were created such that they could be specifically used in the DGKernel system. However, they were also created such that a user may choose to ignore the DG kernels and methods and instead run simulations with the standard CGFE methods. In general, if you want to use DG methods, you will create your Variables as 'MONOMIAL' shape functions and invoke DGKernels that pair with a standard kernel of the same name. Additionally, since DG methods cannot impose Dirichlet type boundary conditions, and suite of DG specific boundary conditions have been provided for a variety of different domains, fluxes, and/or boundary types. Those are any boundary condition that are prefixed with 'DG'. Examples of how the DG methods are invoked can be found here.

NOTE: The 'DG' specific boundary conditions are also valid for CGFE methods.

The tradeoffs between DGFE and CGFE methods come from differences between stability and efficiency of the resulting numerical problem. DG methods are generally more numerically stable and can often better capture sharp gradients in a system. They are also easier to stabilize further through the use of flux limiters (not yet implemented) and/or diffusion penalty terms ('sigma' parameter discussed here). DG methods can also more easily use higher order shape functions that CG methods to improve accuracy. This is because the order of a CG shape function is tied to the mesh elements, due to CG methods requiring a piecewise continuous solution in the domain. DG methods are piecewise discontinuous, so each individual element can be any order regardless of the neighbor elements.

While it may seem that DG methods are superior to CG methods due to potential for higher stability and accuracy, CG methods significantly outmatch DG methods in computational efficiency. This is for main 2 reasons: (i) 'FIRST' order DG methods vs 'FIRST' order CG methods have approximately twice the degrees of freedom that must be solved for and (ii) DG methods require the use of the DG kernels which must reconstruct non-linear residuals at and around each element by reconstructing the fluxes into and out of that element as a function of all of its neighbor's elements fluxes. Thus, not only does the non-linear problem size grow larger for DG methods, but the computational time to formulate the residuals (and Jacobians) also increases. For 1D and 2D simulations, these differences are usually fairly minor, however, they can become problematic for large 3D simulations. Users should bear this in mind when planning out their run cases.

Best Solver Options

MOOSE allows the users to specify a plethora of solver options for (i) the time integration scheme, (ii) the non-linear solver type, (iii) the linear solver type, (iv) the type of preconditioner, (v) the solver tolerances, etc. Given these abundant options it can be overwhelming to determine what works best or what solvers should be invoked. This section talks generally about which solvers and options have had the most efficient and accurate results in the testing of CATS.

Executioner

The 'Executioner' block of the input file is where you would specify most of your solver options except for your preconditioner. The most import options you will want to customize and control are provided below.

Time Schemes

The 'scheme' option is where you direct MOOSE to use a specific time integration scheme for the simulation. MOOSE supports numerous integration schemes for both implicit and explicit methods. However, for CATS it is highly recommended that you only use implicit methods for stability and efficiency. While implicit methods are more complex than explicit methods, with proper preconditioning they can be more computationally efficient by allowing for larger time steps to be taken resulting in fewer overall steps needed to complete a simulation.

Most commonly, CATS uses the standard 'implicit-euler' method since it is the most efficient and most stable scheme. However, if you desire higher levels of computational accuracy, you can specify either the 'bdf2' or 'crank-nicolson' methods. Of those two schemes, 'bdf2' is usually the best, especially for any simulations utilizing the Navier-Stokes module or simulations with advectively dominant physics.

Solver Type

In MOOSE, the 'solve_type' option is used to denote the non-linear solver method the user wants to deploy. Again, there are many different options available, but for CATS it is recommended you use 1 of 2 possible methods: (i) 'newton' or (ii) 'pjfnk'. If you are running a somewhat small simulation on a single core or personal computer, often times the 'newton' method will be faster. However, for large scale simulations or simulations on a computer cluster, it is highly recommended you use 'pjfnk'. If you use the 'newton' method you will not need to provide any preconditioner to the solver, but your solve will not scale well as you request more computer cores or resources. Using 'pjfnk' will allow the simulation to scale better on multiple cores, but will REQUIRE a preconditioner to be specified in order for the solve to be efficient (see Preconditioning for details).

PETSc Options

The linear solvers in MOOSE are directed and handled by the PETSc library. Thus, you will need to specify a set of options for the linear solvers to be used in conjunction with your simulation. These options are specified in 3 tags: (i) 'petsc_options', (ii) 'petsc_options_iname', and (iii) 'petsc_options_value'. Each tag accepts a list of arguments. The number of arguments in 'petsc_options_iname' must be the same number of arguments in 'petsc_options_value' because the

'iname' represents the name of the PETSc argument and the corresponding 'value' represents what you are setting that argument to.

Generally, it is a good idea to always set 'petsc_options' to '-snes_converged_reason' as this will print out to the console window the reason why a particular time step converged or failed to converge. You can use this information to try and determine whether or not other solver options need to be changed or modified to improve convergence. For instance, if the program reports that a time step did not converge due to DIVERGED_LINESEARCH or something similar, then you may need to change line searching options. There are other arguments you can place here as well, which will be discussed more in later sections.

The other tags ('petsc_options_iname' and 'petsc_options_value') are used to specific the linear solver you want to use, the preconditioning algorithm to apply to your preconditioning matrix, and some other special options specific to your linear solver. Due to the complexity and breadth of these options, we cannot go over everything in this document. Instead, we will just outline some of the more popular options you can invoke.

petsc_options_iname → -ksp_type : petsc_options_value → gmres / fgmres / gcr / cgs / bcgs

This is the name of the option for the Krylov subspace method to be used as the iterative linear solver. Many linear solvers may work for this option, but we have outlined 4 options that seem to give the best result: (i) gmres, (ii) fgmres, (iii) gcr, (iv) cgs, and (v) bcgs. The methods 'gmres', 'fgmres', and 'gcr' are the most stable solvers, but are computationally more expensive. Note that 'gmres' is actually the default solver is no option is specified. They should be used if you notice that the linear iterations struggle to converge, as they are almost guaranteed to converge (with proper preconditioning). Both cgs and bcgs are much more efficient solvers, but do not guarantee convergence, even with preconditioning.

NOTE: If you are planning to apply 'ksp' as the '-pc_type' (see below), then you ***NEED*** to use 'fgmres' as the '-ksp_type' here. Otherwise, convergence may not occur.

petsc_options_iname → -pc_type : petsc_options_value → asm / gasm / bjacobi

The '-pc_type' tag is used to specify the name of the primary preconditioning operator to use with the given linear solver. From the range of options tested in CATS, it was found that (i) asm, (ii) gasm, and (iii) bjacobi provide the best efficiencies and residual reduction. Both 'asm' and 'gasm' use the additive Schwarz method. The difference between 'asm' and 'gasm' is whether or not a single matrix block can be shared across multiple processors (as in 'gasm') or whether a processor is assigned a block or blocks in whole (as in 'asm'). The 'bjacobi' method applies a Jabobi preconditioning operation to each block the problem is divided into. As such, 'bjacobi' is much more parallelizable than either 'asm' or 'gasm' and may scale better with many CPUs and/or GPUs. However, 'bjacobi' is generally a worse preconditioner and should only be used if you need the simulation to be massively parallel or when wanting to utilize GPU acceleration.

In addition to providing a standard preconditioning type, users can also give 'ksp' for the argument '-pc_type'. When doing so, you are specifying that you want to precondition with another 'ksp' method. The outer 'ksp' method is dictated by '-ksp_type' (see above), then the inner 'ksp' method is set by a new argument '-ksp_ksp_type'. This argument can be any of the '-ksp_type' arguments from before (see below for options).

If '-pc_type' == 'ksp', then…

petsc_options_iname → -ksp_ksp_type: **petsc_options_value → gmres / fgmres / gcr / cgs / bcgs**

If no '-ksp_ksp_type' is given, then it will default to 'gmres'. Users will also need to provide an argument for '-ksp_pc_type' for the terminal preconditioner instead of '-sub_pc_type' as shown below. An example of this can be found in this example.

NOTE: If you are planning to apply 'redundant' as the '-ksp_pc_type' (see below), then you ***NEED*** to use 'fgmres' as the '-ksp_ksp_type' here. Otherwise, convergence may not occur.

(**NOTE**: Currently, CATS does not support GPU acceleration, but that may change.)

petsc_options_iname → -sub_pc_type : **petsc_options_value → lu / ilu / bjacobi / asm**

The 'sub_pc_type' tag is used to specify the terminal preconditioning operation to be applied to individual blocks of the matrix when divided into subdomain regions by the primary preconditioner. The best option in terms of convergence is the 'lu' solver, which essentially applies a direct solve to the sub-block of the matrix. For greater scalability and parallelization, you can use either 'ilu', 'bjacobi', or 'asm' for the terminal preconditioner. If you are running on a personal computer, just use 'lu' for best performance.

(**NOTE**: If your simulation involves the MOOSE Navier-Stokes module, you MUST use 'lu' for this option)

If the '-pc_type' was set to 'ksp', then this argument does nothing. You must instead set an argument for '-ksp_pc_type' as shown below.

petsc_options_iname → -ksp_pc_type : **petsc_options_value → lu / ilu / bjacobi / asm gasm / pbjacobi / redundant**

Generally, 'lu' gives the best convergence behavior, but does not scale well. For better scalability, it is recommended to use 'ilu', 'asm', 'gasm', 'bjacobi', 'pbjacobi', or 'redundant'. The 'redundant' method will run an additional 'ksp' method with 'gmres' as the terminal preconditioner and applies 'lu' on a series of sub-blocks divided among a number of MPI processes. This might be overkill for most problems, but is a method that scales relatively well and has good convergence.

More information on PETSc solvers and options can be found at https://www.mcs.anl.gov/petsc/.

LU and ILU Methods

In the above section, we discussed some of the most common PETSc arguments you can use with a focus on the manner in which specific types of solvers are invoked. However, each of those solver methods have their own options to play with as well. For instance, the LU and ILU preconditioners have numerous options available to help stabilize the solve. If you begin to see error message such as 'SUBPC_ERROR', then maybe you need to change these options.

On thing you should try to always have active, is a 'NONZERO' shift type for LU and ILU methods. You can specify this option as follows:

petsc_options_iname → -pc_factor_shift_type : petsc_options_value → NONZERO

petsc_options_iname → -sub_pc_factor_shift_type : petsc_options_value → NONZERO

petsc_options_iname → -ksp_pc_factor_shift_type : petsc_options_value → NONZERO

NOTE: You MUST supply this argument corresponding to the correct usage of LU/ILU in your PETSc options.

For instance, if you set LU/ILU as the '-sub_pc_type', then you must set the '-sub_pc_factor_shift_type' as the 'NONZERO' method. Pay close attention to the prefixes on these options

Specifically for ILU, you can also set the factorization levels to augment the amount of 'fill' in the incomplete factorization procedure. By default, the 'fill' level is ILU(1), which allows for an amount of matrix sparsity equal to A^2, where A is the original matrix. Making this value higher will increase the amount of 'fill' and make the factorization more complete (e.g., ILU(2) corresponds to a sparsity of A^3). You can set this value with the following arguments:

petsc_options_iname → -pc_factor_levels : petsc_options_value → (# > 1)

petsc_options_iname → -sub_pc_factor_levels : petsc_options_value → (# > 1)

petsc_options_iname → -ksp_pc_factor_levels : petsc_options_value → (# > 1)

NOTE: You MUST supply this argument corresponding to the correct usage of ILU in your PETSc options.

For instance, if you set ILU as the '-sub_pc_type', then you must set the '-sub_pc_factor_levels' to the specified value you desire. Pay close attention to the prefixes on these options

The default library used for LU and ILU factorization is the MUMPS solver library. MUMPS has a significant set of options (https://petsc.org/main/docs/manualpages/Mat/MATSOLVERMUMPS.html) that the user can also control through the 'petsc_options_*' arguments in the input files. Most of the

time you will not need to update any of these options, however, occasionally you may get 'SUBPC_ERROR', 'DIVERGED_PC', and/or 'FACTOR_OUT_OF_MEMORY' errors reported for very large and highly coupled systems of equations. If these errors do occur, you may want to make changes to the default MUMPS options.

The three most important options that CATS has used are as follows:

petsc_options_iname → -mat_mumps_cntl_1 : petsc_options_value → (# < 1 & > 0)

petsc_options_iname → - mat_mumps_cntl_3 : petsc_options_value → (# < 1 & > 0)

petsc_options_iname → - mat_mumps_icntl_23 : petsc_options_value → (# > 200)

where '-mat_mumps_cntl_1' controls the relative tolerance used in pivoting, '-mat_mumps_cntl_3' controls the absolute tolerance used in pivoting, and '-mat_mumps_icntl_23' controls the maximum allowable additional memory (in MB) that each processor is allowed to allocate during pivoting. The tolerances represent the value at which a pivot point would be accepted. The lower this value is, the more pivoting is allowed (which requires more memory). The default values are 0.01 for relative pivot tolerance and 0 for absolute pivot tolerance. In general, by allowing more memory through the '-mat_mumps_icntl' option allows the solver to find solutions more easily, but may cost significant memory.

GASM and ASM Methods

The difference between 'gasm' and 'asm' is that 'gasm' can share blocks/subdomains among a set of processors/cores, where as in 'asm' each processor/core has its own set of blocks/subdomains to work on and does not share that information with any other processor. As a result, 'asm' is more memory efficient than 'gasm', but 'gasm' is often a *better* preconditioner.

Each of these solvers can change the number of blocks/subdomains that each processor works on, as well as an amount of 'overlap' allowed between those blocks/subdomains. In general, the more overlap you have, the more memory is needed, but better convergence can be achieved. Thus, with each of these methods you are constantly balancing convergence vs memory efficiency.

In general, I have found that the default number of blocks/subdomains per processor (which defaults to 1) is usually sufficient, and improvements can be made by increasing the levels of overlap between 10 to 100. However, this may be very problem specific. Below shows how to set these options.

For ASM:

petsc_options_iname → -pc_asm_blocks : petsc_options_value → (# > 1)

petsc_options_iname → -sub_pc_ asm_blocks : petsc_options_value → (# > 1)

petsc_options_iname → -ksp_pc_ asm_blocks : petsc_options_value → (# > 1)

petsc_options_iname → -pc_asm_overlap : petsc_options_value → (# 10 - 100)

petsc_options_iname → -sub_pc_asm_overlap: petsc_options_value → (# > 10 - 100)

petsc_options_iname → -ksp_pc_asm_overlap: petsc_options_value → (# > 10 - 100)

NOTE: You MUST supply this argument corresponding to the correct usage of ASM in your PETSc options. Pay close attention to the prefixes in each argument.

For instance, if you set ASM as the '-sub_pc_type', then you must set the '-sub_pc_asm_overlap' (etc) to the specified value you desire.

For GASM:

petsc_options_iname → -pc_gasm_subdomains : petsc_options_value → (# > 1)

petsc_options_iname → -sub_pc_gasm_subdomains : petsc_options_value → (# > 1)

petsc_options_iname → -ksp_pc_gasm_subdomains : petsc_options_value → (# > 1)

petsc_options_iname → -pc_gasm_overlap : petsc_options_value → (# 10 - 100)

petsc_options_iname → -sub_pc_gasm_overlap: petsc_options_value → (# > 10 - 100)

petsc_options_iname → -ksp_pc_gasm_overlap: petsc_options_value → (# > 10 - 100)

NOTE: You MUST supply this argument corresponding to the correct usage of GASM in your PETSc options. Pay close attention to the prefixes in each argument.

For instance, if you set GASM as the '-sub_pc_type', then you must set the '-sub_pc_gasm_overlap' (etc) to the specified value you desire.

GMRES (and GCR) Methods

Both 'gmres' (and 'fgmres') and 'gcr' are "restarted" subspace methods. These methods build a growing subspace of search vectors and hold each vector in memory to build the next step. Because of this, in order for them to be practical the vector space has to be periodically flushed to free up memory. By default, PETSc will only hold 30 vectors in the subspace before flushing and restarting. For very large problems with many mesh elements, 30 vectors may not be enough to hold on to. To change this options you must also have either '-ksp_gmres_restart' or '-ksp_gcr_restart' as a 'petsc_options_iname' tag to be changed. Ideally, you would choose a large number that represents a significant fraction of your 'degrees of freedom' in the problem, which is related to the number of elements in the mesh and the number of non-linear variables being solved for. If the supplied number is too large, then it may tax your computer memory. You invoke these arguments as shown below.

petsc_options_iname → -ksp_gmres_restart : petsc_options_value → (large #)

petsc_options_iname → -ksp_gcr_restart : **petsc_options_value → (large #)**

NOTE: Whether you are using 'fgmres' or 'gmres', the restart argument is always by the same name, so the options shown above is valid for either method.

In addition to setting the restart value, you may also need to specify an orthogonalization procedure for 'fgmres' or 'gmres'. By default, both methods use a 'classical Gram-Schmidt' orthogonalization procedure, which is a fast algorithm, but can be unstable. You can change this to the 'modified Gram-Schmidt' method by adding the following argument to 'petsc_options'.

petsc_options → -ksp_gmres_modifiedgramschmidt

Debugging Issues with Linear Solvers and Preconditioners

Whenever you are having solver issues, you can view what the solvers are doing, how they are invoked, and the options they were invoked with by adding a couple of additional arguments to the 'petsc_options' list. By default, I always recommend leaving the '-snes_converged_reason' option active. This gives minimal reporting and will simply tell you why the linear (and non-linear) solvers have stopped. This is done by reporting the PETSc flags, which will report why it converged or why it didn't converge.

For a more advanced view of the linear solver actions, you can also add '-ksp_view' option to the 'petsc_options' argument. This will report a significant amount of information to the console including what linear solvers, preconditions, and sub-preconditioners are being called, where they are called in the solve, and the options they were called with.

petsc_options → -ksp_view : **All linear solver information**

petsc_options → -snes_converged_reason : **Reason the solver stopped**

Line Search

Line searching is a method that applies to the non-linear iterations in order to smooth out the residuals at each iteration. By default, the 'line_search' option is set to 'none' (i.e., to not apply line searching). Other options include: (i) 'bt', (ii) 'l2', and (iii) 'basic'. Generally, adding line search to the simulation will very slightly increase the computation time at each non-linear step. Thus, it is recommended to not use line searching if it is not needed. However, some very difficult simulations with poor initial conditions or rapidly changing non-linear variables may struggle to converge without it. In these situations, the 'l2' line search method has been shown to be very effective and is the most stable line search method.

NOTE: Line searching will not always help a simulation converge and can occasionally have the opposite effect if the initial condition or prior time step is nowhere near the current solution. Sometimes a better option is to just take smaller time steps with no line searching.

Tolerances

In MOOSE, you have the ability to change and adjust all the tolerances you want. In some cases, you may need to adjust the solution tolerances from their default values, especially if you are simulating a transient problem to the steady-state solution. The following list of tolerances and iteration limits have been found to work well.

NOTE: tolerances can also be directly setup in the PETSc options (see examples below).

nl_rel_tol → from 1e-6 to 1e-8

nl_abs_tol → from 1e-6 to 1e-8

nl_rel_step_tol → from 1e-10 to 1e-16

nl_abs_step_tol → from 1e-10 to 1e-16

l_tol → from 1e-4 to 1e-6

nl_max_its → from 10 to 30

l_max_its → from 100 to 500 (really depends on the number of elements in the mesh)

Time Stepper

In addition to specifying a time scheme, the user needs to direct MOOSE how to choose its next time step, when to start and stop the simulation, and specify a cap to the time step size. The start time, end time, and maximum step size are given under the Executioner block, while the time step method is given in a TimeStepper subblock. In the TimeStepper subblock, the user specifies the 'type' of stepper algorithm to use, as well as the initial time step to take. Below is an example of how the times and time stepper are specified in the Executioner block.

```
start_time = 0

end_time = 10

dtmax = 0.5

[./TimeStepper]

    type = ConstantDT

    dt = 0.1

[../]
```

The TimeStepper 'type' will usually be either 'ConstantDT' to specify a constant time step to take at each solve or 'SolutionTimeAdaptiveDT' which will adjust the time step size up or down after each solve based on how well or how quickly the previous solve went. In general, the method for 'SolutionTimeAdaptiveDT' should be used with a small initial time step and will increase the time step

size to a maximum to accelerate the simulation to complete in a more timely manner. If you are so inclined, you can also create custom Time Steppers to use in CATS.

Preconditioning

In order for the linear iterations to converge efficiently, the user is REQUIRED to provide some form of preconditioner. Otherwise, convergence is extraordinarily slow or sometimes impossible. However, if the 'solve_type' was set to 'newton', then there is no need for a preconditioner. For any large scale simulations on multiple processors or computer clusters, you should be using 'pjfnk' as your 'solve_type' and MUST provide a preconditioner.

Each residual contribution in CATS is fully developed with appropriate Jacobian elements and Off-Diagonal Jacobian elements to make preconditioning very simple. This relieves the user from having to employ some of the more complex preconditioning options, such as the physics-based MOOSE preconditioner, which are extremely tedious to setup in the simulations. Instead, you will only ever need to use the Single Matrix Preconditioner method. This will be by far the most efficient method to use in CATS. However, we will also discuss the Finite Difference Preconditioner, which may be useful for debugging issues with developed residuals or solvers.

Finite Difference Preconditioner

This preconditioning option is highly inefficient, but is useful for occasionally tracking down problems that may persist in the developed modules and kernels. Essentially, the Finite Difference Preconditioner is a matrix preconditioner constructed from a full finite differences sweep of all residuals invoked in a simulation. Thus, it creates a "perfect" preconditioner that should theoretically solve the linear subproblem in a single iteration. If you use this option and see that the linear steps do NOT converge within 1 to 2 steps, then there is likely a problem in the residuals. To invoke the Finite Difference Preconditioner, you add the following blocks to your input file:

```
[Preconditioning]
    [./FDP_method]
        type = FDP
        full = true
    [../]
[]
```

Single Matrix Preconditioner

This is the preconditioner option that every simulation case in CATS should have for any realistic problem you want to simulate. It should be extremely efficient due to the numerous Jacobian and Off-Diagonal Jacobian functions that have been manually coded into each and every customized CATS kernels. When used correctly, it should create a nearly perfect preconditioning matrix. You invoke the use of this option by including the following blocks in your input file.

Example: Full Jacobian

```
[Preconditioning]
    [./SMP_method]
        type = SMP
        full = true
    [../]
[]
```

Using the keyword argument 'full = true' (as shown above) is a very simple way to create a nearly perfect preconditioning matrix to apply to any model in CATS. However, if you are running simulations are very large, complex 3D geometries, it may be better to be more selective of the Jacobian elements that actually get constructed in your matrix. This can be accomplished by using the keyword argument 'coupled_groups' followed by a list of variable pairs that you want to construct the Jacobians for.

For example, let's consider that you are running a basic simulation of mass, momentum, and energy balances all coupled together. In the full Jacobian, all the partials of each variable with respect to each other variable will be initialized and computed. However, we may not need all these partials to be computed or may want to build a more sparse matrix. We can use the 'coupled_groups' to tell MOOSE which set of partials we most care about. From the example below, our Jacobian will be constructed from partial derivatives between concentration (C) and temperature (T), as well as pressure (P) and each component direction of velocity (vel_x, and vel_y).

Example: Selective Sparse Jacobian

```
[Preconditioning]
    [./SMP_method]
        type = SMP
        coupled_groups = 'C,T  P,vel_x  P,vel_y'
    [../]
[]
```

Example Options

This section provides a sampling of how these solver options are structured in the CATS input files. Again, this section and example is not exhaustive, but demonstrates how to invoke some of the most common solver options that have resulted in the most accurate and efficient simulations.

For problems that are 'difficult' to converge, it is generally recommended to use 'LU' or 'ILU' are your terminal preconditioner at some point in the solve. This is because 'LU' and 'ILU' are currently the best preconditioning methods.

For problems that are VERY 'difficult', I recommend using 'fgmres' with a 'ksp' preconditioner that uses either 'fgmres' or 'gmres' followed by a sub-preconditioner of 'lu' or 'ilu'. When using both 'lu'

and ‘ilu’, be sure to always set the shift to ‘NONZERO’. For ‘ilu’, you may need to increase the ‘factor levels’ to make a good preconditioner.

Example 1: GMRES with ASM preconditioner and BJacobi sub-preconditioner

```
[Executioner]
        type = Transient
        scheme = bdf2
        solve_type = pjfnk
        petsc_options = ‘-snes_converged_reason’
        petsc_options_iname = ‘-ksp_type   -ksp_gmres_restart  -pc_type   -sub_pc_type’
        petsc_options_value =  ‘gmres        100                    asm          bjacobi’
        line_search = none
        nl_rel_tol = 1e-8
        nl_abs_tol = 1e-6
        nl_rel_step_tol = 1e-10
        nl_abs_step_tol = 1e-10
        l_tol = 1e-6
        nl_max_its = 10
        l_max_its = 100
        start_time = 0
        end_time = 10
        dtmax = 0.5
        [./TimeStepper]
                type = SolutionTimeAdaptiveDT
                dt = 0.1
        [../]
[]
```

Example 2: FMGRES with GMRES preconditioner and LU sub-preconditioner

```
[Executioner]
        type = Transient
        scheme = implicit_euler
        solve_type = pjfnk
        petsc_options = ‘-snes_converged_reason
                        -ksp_gmres_modifiedgramschmidt’

        petsc_options_iname = ‘-ksp_type
                              -ksp_gmres_restart
                              -pc_type
                              -ksp_ksp_type
                              -ksp_pc_type
                              -ksp_pc_factor_shift_type’
```

```
        petsc_options_value =  'fgmres
                               100
                               ksp
                               gmres
                               lu
                               NONZERO'
        line_search = none
        nl_rel_tol = 1e-8
        nl_abs_tol = 1e-6
        nl_rel_step_tol = 1e-10
        nl_abs_step_tol = 1e-10
        l_tol = 1e-6
        nl_max_its = 10
        l_max_its = 100
        start_time = 0
        end_time = 10
        dtmax = 0.5
        [./TimeStepper]
                type = ConstantDT
                dt = 0.5
        [../]
[]
```

Example 3: GMRES with ILU preconditioner and increased factor levels

```
[Executioner]
        type = Steady
        solve_type = pjfnk
        petsc_options = '-snes_converged_reason
                        -ksp_gmres_modifiedgramschmidt'

        petsc_options_iname = '-ksp_type
                               -ksp_gmres_restart
                               -pc_type
                               -pc_factor_shift_type
                               -pc_factor_levels'

        petsc_options_value =  'gmres
                               100
                               ilu
                               NONZERO
                               3'
        line_search = none
        nl_rel_tol = 1e-8
```

```
    nl_abs_tol = 1e-6
    nl_rel_step_tol = 1e-10
    nl_abs_step_tol = 1e-10
    l_tol = 1e-6
    nl_max_its = 10
    l_max_its = 100
[]
```

Example 4: Setting up all options

When options are passed to PETSc from your input file, PETSc will only use options that it needs. As a result, it may be to your benefit to set ALL available options in your input files. This way, you can simply toggle a few values at the input to change the solvers, rather than have to write a completely new set of options.

Additionally, you can directly set some of the tolerances, maximum iterations, or other options directly in PETSc. Those options are also demonstrated below. When these options are set in the PETSc options, they will override any options given below them in the MOOSE Executioner block.

NOTE: Not a FULLY exhaustive list, but the most comprehensive you will likely need.

```
[Executioner]
    type = Steady
    solve_type = pjfnk
    petsc_options = '-snes_converged_reason
                    -ksp_gmres_modifiedgramschmidt
                    -ksp_view'

    petsc_options_iname = '-ksp_type
                          -ksp_gmres_restart      -ksp_gcr_restart

                          -pc_type                -sub_pc_type
                          -ksp_ksp_type           -ksp_pc_type

                          -pc_factor_shift_type       -pc_factor_levels
                          -sub_pc_factor_shift_type   -sub_pc_factor_levels
                          -ksp_pc_factor_shift_type   -ksp_pc_factor_levels

                          -pc_asm_blocks          -pc_asm_overlap
                          -sub_pc_asm_blocks      -sub_pc_asm_overlap
                          -ksp_pc_asm_blocks      -ksp_pc_asm_overlap

                          -pc_gasm_subdomains         -pc_gasm_overlap
                          -sub_pc_gasm_subdomains     -sub_pc_gasm_overlap
                          -ksp_pc_gasm_subdomains     -ksp_pc_gasm_overlap
```

```
                                -snes_max_it    -snes_atol     -snes_rtol   -snes_stol
                                -ksp_max_it     -ksp_atol       -ksp_rtol'



        petsc_options_value =  'fgmres
                                100      100

                                ksp       lu
                                gmres    lu

                                NONZERO   3
                                NONZERO   3
                                NONZERO   3

                                3    50
                                3    50
                                3    50

                                3    50
                                3    50
                                3    50

                                20    1e-8    1e-10   1e-10
                                100   1e-6    1e-6'
        line_search = none
[]
```

Model Outputs

The standard output for the MOOSE model includes (i) CSV files and (ii) Exodus files, however, no outputs are setup by default. You must turn on output options from the 'Outputs' block in the input file. This section discusses some of the basic output, how to request specific types of outputs, and how to interpret results of the output.

Output Options

For CATS, there are 4 primary output flags you should know about: (i) 'exodus', (ii) 'csv', (iii) 'interval', and (iv) 'print_linear_residuals'. Each option is as follows:

exodus : true / false

The 'exodus' tag is used to determine whether or not you wish to have an Exodus file produced during the simulation. An Exodus file is a binary file that contains all information about the simulation domain and the variables on that domain. It is the primary file you will want to use for visualizing the results from CATS. However, it may be unnecessary if your domain is 0D or 1D. For more information on Exodus files, see Exodus Files below.

csv : true / false

The 'csv' tag is used to determine whether or not you wish to have a .csv file created during the simulation of the values of the computed Postprocessors. You can import or open .csv files directly into MS Excel (or another spreadsheet software) to create plots of specific data or variables as a function of simulation time. The Postprocessors are MOOSE calculation objects that are used to estimate variable values at specific locations in the mesh, integrals of variable values over the domain, or average values of variables at a boundary or interface in the domain. For more information on Postprocessors, see the next section below.

interval : (some #)

The 'interval' tag is used to tell MOOSE how often you want output from the model to be created or displayed. By default, this value is set to 1, which means that at the end of every time step, solution values are recorded to the console window and/or to the various output files. For very long simulations with many time steps, it may be helpful to set this to a larger number so that some output gets suppressed and the output files don't become to large. For instance, if you set this value to 10, then MOOSE will only record the output of every 10^{th} time step.

print_linear_residuals : true / false

The 'print_linear_residuals' tag is used to determine whether or not you want MOOSE to report the residuals at each linear iteration. It is usually not necessary to set this to 'true', but may be useful for

debugging or testing the code. When you see that the linear residuals are not reducing very quickly, that can be an indication that the preconditioning or the chosen linear solver is inappropriate.

Postprocessors

Postprocessors are a set of built-in calculation features in MOOSE that allow the user to: (i) probe the values of variables in specific elements, (ii) obtain average values for variables in the domain or at a boundary, and (iii) obtain integral values for variables in the domain or at a boundary. Below are some of the most common Postprocessors and how they are invoked inside the 'Postprocessors' block of the input file.

ElementVariableValue

This postprocessor is used to probe the value of a variable at a specific element in the domain. We generally do not use this postprocessor simply because it is relatively difficult to use as you have to provide the actual element id you want to probe and not just a location in special dimensions. You can invoke this option as shown below.

```
# Probing the value of variable 'u' at element '1'
[./u_at_1]
        type = ElementVariableValue
        variable = u
        elementid = 1
        execute_on = 'initial timestep_end'
[../]
```

SideAverageValue

This postprocessor is used to compute the average value of a variable at a boundary or interface of the domain. We commonly use this postprocessor to observe changes in outlet conditions of a domain for a particular variable. For example, probing the average value of the value of a concentration variable at the 'outlet' of the domain gives us the "breakthrough" data from the simulation. Note that you must provide the name of the boundary for which this postprocessor applies, as well as the name of the variable to probe.

```
# Probing the value of variable 'C' a boundary named 'exit'
[./C_exit]
        type = SideAverageValue
        variable = C
        boundary = 'exit'
        execute_on = 'initial timestep_end'
[../]
```

ElementAverageValue

This postprocessor is used to formulate an average value of a variable in the entire domain or subdomain. If your mesh involves subdomains, then you are required to provide the name of the subdomain for which this average will apply. Each subdomain will either be named by the user or automatically given a number by MOOSE depending on how the mesh was constructed. This postprocessor is commonly used to provide estimates for average values of non-mobile variables, like adsorption capacities, in a domain.

```
# Computing the average value of adsorption variable 'q' in subdomain 'washcoat'
[./q_avg]
        type = ElementAverageValue
        variable = q
        block = 'washcoat'
        execute_on = 'initial timestep_end'
[../]
```

AreaPostprocessor

This postprocessor is used to compute the actual surface area (on a 3D mesh), or length (on a 2D mesh) of a particular boundary or interface. In most cases, you will not need to use this postprocessor, but it is occasionally useful if you are checking mass/energy balances and need an accurate account of the size of the inlet/outlet of the domain. No variable needs to be given, just the name of the boundary.

```
# Calculating the area of the 'exit' of the domain
[./area_exit]
        type = AreaPostprocessor
        boundary = 'exit'
        execute_on = 'initial timestep_end'
[../]
```

VolumePostprocessor

This postprocessor is used to compute the actual volume (on a 3D mesh), or area (on a 2D mesh), of a particular subdomain or the entire domain. Much like the AreaPostprocessor, this is not necessarily needed, but is occasionally useful for checking mass/energy balances when you need an accurate account of the size of each subdomain.

```
# Calculating the volume of the 'washcoat' of subdomain
[./vol_wash]
        type = VolumePostprocessor
        block = 'washcoat'
        execute_on = 'initial timestep_end'
[../]
```

Exodus Files

The Postprocessors can only yield specific information at a given location or some average values, but all of the simulate results are available in the Exodus file. Since the Exodus file is a binary file, you will need a specific application to read the data. Cubit is a common application that can read Exodus files, but it is only available for Government work. You can also use an open source software such as ParaView to read Exodus files.

More information on Cubit can be found at https://cubit.sandia.gov/.

More information on ParaView can be found at https://www.paraview.org/.

Input Files

In this section, we generically discuss the structure of MOOSE/CATS input files. Each input file is structured in a manner reminiscent of html or xml. It is a "hierarchical input text format" where each block of the input starts with the block name in square brackets (e.g., [Variables]) and that block ends when an empty set of square brackets is encountered (e.g., []). Inside each block there may be subblocks. All subblocks are started by [./subblock_name] and ended by [../]. Inside the blocks and subblocks are where all parametric, variable, function, and other arguments are placed. Any line in the input file preceded by a '#' is a comment and will not be read in by MOOSE. The example below just demonstrates this structure. The arguments with '' marks can denote string arguments or a list of arguments, depending on the context.

```
[Block_1]
        param = 13
        bool_arg = true
        # Comment about Subblock_1
        [./Subblock_1]
                another_arg = 'something'
                list_of_args = '1 2 3'
        [../]
[]
```

Provided below in the next few sections are examples and discussion on the many types of blocks and subblocks commonly used in CATS. Note that this list is not exhaustive for the MOOSE framework. For more information on input files, visit https://mooseframework.org/.

GlobalParams

The GlobalParams block is a place where you can put any arguments you want to be used by all other blocks, unless specifically overridden within that particular block. It is very useful for reducing some redundancy in the input file structure. Often times, multiple kernels will need the same arguments to be declared each time they are invoked. Thus, if you put those arguments into GlobalParams, then those arguments will remain consistent throughout all the kernels that use them.

NOTE: When MOOSE searches for input arguments, it will ALWAYS check the local kernel blocks FIRST for the needed arguments. If they are not found locally, then it checks GlobalParams.

A great example of when and why to use GlobalParams comes from how we invoke the Navier-Stokes module in MOOSE. Each Navier-Stokes kernel requires a specific set of stabilization and other arguments to be shared among each invocation of the kernels. Therefore, instead of repeating this information in each Navier-Stokes specific kernel block, we can put them all into GlobalParams as follows.

```
[GlobalParams]
        gravity = '0 0 0'
        integrate_p_by_parts = true
```

```
        supg = true
        pspg = true
        alpha = 0.5
        laplace = true
        convective_term = true
        transient_term = true
    []
```

For more detailed information on these arguments, see Incompressible Navier-Stokes example.

Mesh

The Mesh block is where you direct MOOSE to either: (i) construct a mesh domain and subdomains to simulate on or (ii) read in a mesh file that represents your simulation domain. There are many different ways in which MOOSE can generate and use meshes, so we will only discuss some of the basics here. For more information, go to https://www.mooseframework.org/syntax/index.html.

GeneratedMeshGenerator

This subblock allows the user to very quickly and easily create lines, rectangles, and cubic domains to simulate on. In concert with the RZ-Cylindrical option in the Problem block, we can also create cylindrical domains with this subblock. The users provide the minimum and maximum x, y, and z points for the domain and the number of elements to divide each dimension into. By default, the domain 'block' ID number is set to 0 and the boundaries of the mesh get set to 'left right' for the x-direction ('left' → minimum x and 'right' → maximum x), 'bottom top' for the y-direction ('bottom' → minimum y and 'top' → maximum y), and 'front back' for the z-direction ('front' → z minimum and 'back' → z maximum).

In the example below, we use the GeneratedMeshGenerator to create a 1-by-5 sized 2D mesh in the xy plane with 10 subdivisions in each the x and y directions.

```
[Mesh]
    [./my_mesh]
        type = GeneratedMeshGenerator
        dim = 2
        xmin = 0
        xmax = 1
        ymin = 0
        ymax = 5
        nx = 10
        ny = 10
    [../]
[]
```

FileMeshGenerator

For more complex mesh structures, you can direct MOOSE to simply read in a mesh file. MOOSE supports numerous mesh file types. For a complete list of supported mesh formats, check out https://www.mooseframework.org/source/meshgenerators/FileMeshGenerator.html. In CATS, we most commonly use the '.unv' and '.msh' type meshes. One major advantage of these mesh formats is they allow for the naming of blocks, boundaries, and interfaces. Thus, this allows us to create complex meshes with various subdomains and unusual boundaries that other kernels can identify by name.

In the example below, we use FileMeshGenerator to read in a 2D mesh representing a monolith channel. That mesh has 2 blocks named 'washcoat' and 'channel', as well as having boundaries identified as 'inlet' and 'outlet' (i.e., the open flow boundaries) and 'interface' the interface boundary between the 'washcoat' and 'channel' subdomains. Note that here we do not need to declare any other arguments to set these names. The names come from the mesh file and can be referenced later in the input file.

```
[Mesh]
    [./2D_monolith]
        type = FileMeshGenerator
        file = 2DChannel.msh
    [../]
[]
```

Problem

In CATS, there is only a single purpose to the Problem block that is applicable which is to direct a coordinate system transformation from XYZ-Cartesian to RZ-Cylindrical. We use the RZ-Cylindrical domain when we want to simulate a 3D column in a 2D domain. It provides a convenient way to simulate gas flow through a cylindrical shape without needing to create and mesh an actual column. If your problem is not RZ-Cylindrical, then you do not need to include this block in your input file.

When we want a 2D cylindrical domain, we only need to add the Problem block and specify the coordinate system as 'RZ' as shown below. Note that in this transformation of the coordinate system, the x-direction becomes the radial coordinate R and the y-direction becomes the axial coordinate Z. Thus, if this block were included in concert with the GeneratedMeshGenerator object from before, we would create a 2 diameter cylinder (e.g., x length = 1 → radius of 1) with a 5 unit length.

```
[Problem]
    coord_type = RZ
[]
```

Functions

The Functions block on its own doesn't actually add anything to the simulation, but can be used in coordination with other blocks to setup custom initial conditions or boundary conditions. There are many different built-in functions in MOOSE (https://www.mooseframework.org/syntax/index.html), but

the most commonly used function is the ParsedFunction object. It is very easy to use this object to create custom initial conditions for variables if we want that variable to initially have some spatial variation in our domain.

The ParsedFunction object reads in a user argument that represents a mathematical expression using parameters for 'x', 'y', 'z', and 't' to update the value of the function based on the specific location in the domain and/or a given point in time. The operators that are recognized by the MOOSE parser include, but are not limited to: (i) '+', (ii) '-', (iii) '*', (iv) '/', and (v) 'exp(..)'. More options can be found at https://www.mooseframework.org/source/functions/MooseParsedFunction.html.

Provided below is an example of a parsed function we could use to set a custom initial condition for a variable in our domain as a function of the z-location in the domain. Use '' marks around the function to encapsulate the whole expression.

```
[Functions]
    [./q_ic]
        type = ParsedFunction
        value = '4.01E-4 * -1.8779 * exp( z / 10) / (1 – exp(10) )'
    [../]
[]
```

Variables

The Variables block is where you put down all the non-linear variables that you plan to use the CATS physics/chemistry kernels to solve. Each variable name is given by the user and is interpreted as the name given as the subblock in the input file for that variable. Within each variable subblock, the user must specify: (i) the 'order' of the variable, (ii) the 'family' of the variable, and (iii) an initial condition, which can be done as a constant value, a user function, or some other custom kernel. Additionally, if your mesh involves subdomains, then you need to provide the 'block' name or ID that the variable exists on. Providing nothing will assign the variable to all blocks.

The 'family' of the variable is where the type of shape or test function is defined. MOOSE supports a number of shape functions and the comprehensive list of shape function options can be found here (https://mooseframework.inl.gov/source/variables/MooseVariable.html). By default, the 'family' is set to 'LAGRANGE' if the user does not provide any argument here. If you are going to use Discontinuous Galerkin (DG) methods and kernels, we recommend using the 'MONOMIAL' shape functions. In principle, you can use any shape functions that your mesh and methods can support. In CATS, there are methods to support mass, energy, and momentum balances with 'MONOMIAL' shape functions.

The 'order' of the variable can be either 'CONSTANT', 'FIRST', or 'SECOND' (or higher depending on whether your mesh and shape function supports these orders). You can only use 'CONSTANT' order if you are using a 'MONOMIAL' shape function. In general, you should make all variables 'FIRST' order. For DG methods specifically, you can lower the order to 'CONSTANT' to maximize simulation stability.

To set a constant initial condition, you just add the 'initial_condition' argument to the subblock and give a value to set the variable to. For any other initial condition, you must add another subblock

named ‘InitialCondition’ and any arguments that that particular custom initial condition argument may need.

In the below examples, we create variables for concentration (C), adsorption (q), energy (E), and velocity in the x-direction (vel_x). Each variable is on block 0. We show how to set the variable on the block, but this line is unnecessary if there is only 1 block. Only the velocity needs to be made a ‘LAGRANGE’ variable, while the others are ‘MONOMIAL’. Variables C and vel_x get set to a constant initial condition, while q and E get set to a function and custom CATS initial condition kernel (InitialPhaseEnergy), respectively. The function for the initial condition of q comes from the Functions block (see above). The purpose is just to provide a demonstration of the various options for setting up non-linear variables in CATS.

```
[Variables]
    [./C]
        order = FIRST
        family = MONOMIAL
        initial_condition = 1e-9
        block = 0
    [../]

    [./q]
        order = FIRST
        family = MONOMIAL
        block = 0
        [./InitialCondition]
            type = FunctionIC
            #below sets the name of the function object in Functions block
            function = q_ic
        [../]
    [../]

    [./E]
        order = FIRST
        family = MONOMIAL
        block = 0
        [./InitialCondition]
            type = InitialPhaseEnergy
            specific_heat = 1000
            density = 1.2
            temperature = 298
        [../]
    [../]

    [./vel_x]
        order = FIRST
```

```
        family = LAGRANGE
        initial_condition = 0
        block = 0
    [../]
[]
```

AuxVariables

The AuxVariables block is formulated identically to that of the Variables block. Each subblock is the name of an auxiliary variable, each auxiliary variable must have an 'order' and a 'family', and each can have constant initial conditions or custom/function initial conditions. The only difference is that an auxiliary variable does not have any associated kernels, however, they can have an associated AuxKernel. Often times, CATS will use auxiliary variable even without an AuxKernel just so that parametric constants can be referenced to by name in other blocks, rather than repeating the values in those blocks. See Variables for examples on how to structure this block.

Kernels

The Kernels block is where different pieces of physics and chemistry are invoked. This will likely be the largest block in the input file since every variable needs multiple kernels to describe all the physics. Each kernel will have its own subblock and each subblock must at least give (i) the 'type' of kernel, (ii) the 'variable' the kernel acts on, and (iii) the 'block' or subdomain name/ID that the kernel applies to. Note that the 'block' argument is only needed if there are multiple subdomains in the mesh. Examples of how to invoke each kernel in CATS is provided in significant detail throughout the main Kernels section of this guide.

DGKernels

One of the major features in CATS is in how the transport equations for mass and energy are handled. For these physics, we use Discontinuous Galerkin (DG) methods, which are significantly more stable and more accurate from problems involving conservation laws compared to the standard Galerkin methods. In CATS, we have developed a host of custom advective and diffusive transport kernels to take advantage of this methodology. Whenever your physics involve these types of transport mechanism, you must use the corresponding DGKernels. In addition, each DG kernel must also be accompanied by a corresponding standard Galerkin kernel in the MOOSE system to complete the residual formation. More discussion of this can be found here and examples of how to invoke the DG kernels is provided in all the Simulation Examples.

AuxKernels

The AuxKernels block are for providing a calculation for a specific AuxVariable. Each auxiliary variable is only allowed to have one auxiliary kernel. In CATS, the auxiliary kernels are most commonly used to set specific gas properties, rate parameters, and/or transport parameters. However, you can use the auxiliary system for a wide range of calculations. It should be noted though that you should not use

auxiliary kernels for variables such as concentration/mass or energy densities. These should be treated as non-linear variables and coupled fully and implicitly. The auxiliary kernels and variables are only loosely coupled with the non-linear variables. Usage examples and all CATS auxiliary kernels can be found in more detail in the main AuxKernels section.

BCs

The BCs block is used to set boundary conditions for any variable that needs them. By default, if no boundary condition is provided, then MOOSE applies a so-called "natural" boundary condition, which essentially would enforce a zero slope at a boundary. This is sufficient for any "closed" boundaries on the domain or for any variable that represents an immobile phase, such as an adsorption variable. For mobile variables, you will need to provide some specific boundary conditions.

For the conservation of mass and energy problems, you cannot actually provide any true Dirichlet type boundary conditions because solutions are not defined at nodes, they are defined at elements. Thus, in CATS, there are a host of integrated boundary conditions you can apply for phenomena such as inflow/outflow and wall energy transfer, as well as an emulated or penalty based pseudo-Dirichlet boundary condition. For the Navier-Stokes module, you can use either the DirichletBC, PenaltyDirichletBC, or INSNormalFlowBC options. More information on all boundary conditions can be found in the main Boundary Conditions section.

InterfaceKernels

The InterfaceKernels are essentially boundary conditions on mesh elements that represent the interface between 2 subdomains in the global mesh. You will only ever need to invoke these kernels when your mesh contains more than 1 'block' ID. We use these kernels to facilitate the transfer of mass and energy from one domain to another. When invoked, you must provide the name of the variable on the primary mesh/subdomain, as well as the 'neighbor' variable that the other subdomain. Examples of this can be found in the main Interface Kernels section.

Materials

The Materials block is a section of the input file where certain properties can be set for the various subdomains of a mesh. When you declare any materials in your input file, all blocks must invoke a material, regardless of whether or not those properties are needed on each subdomain. An individual material may be invoked on multiple blocks if necessary. Often times this is the simplest way to resolve any MOOSE errors stemming from missing materials.

In CATS, we don't really use the Materials system for handling parameters. Typically, all of our parameters may be set as constants or may be specified as AuxVariables. The reasoning for this distinction in our framework boils down to code modularity and ease-of-use. When a particular property or parameter is declared in MOOSE as a "Material Property", the kernels that expect that property will only function correctly if the specific material object, which calculates or declares that property by name, is invoked. On the other hand, if that property is declared a variable, then MOOSE will accept any

argument in the kernel that is a variable, an auxiliary variable, or a specified constant. Thus, making the same bit of code significantly more modular. In addition, since the CATS kernels are designed specifically to anticipate parameters as variables, all the necessary off-diagonal Jacobian elements that the MOOSE system would need to resolve the coupling have already been pre-programmed into those kernels. If we instead used the Materials system, then we would have to update the entire kernel base if we wanted to include those material properties as actual variables at a later point in time.

Although CATS does not make use of the Materials system, the built-in MOOSE Navier-Stokes module does. Therefore, we must discuss here how a user wanting to invoke Navier-Stokes in their simulation can do so in CATS. We have created a custom Material object called INSFluid, which provides a simple way for CATS properties to be linked to the incompressible Navier-Stokes module while still taking advantage of the CATS method of coupling variables and parameters. Thus, in any simulation where you use the incompressible Navier-Stokes module, you should create a Materials block in your input file as shown below. Note that the expected arguments for 'density' and 'viscosity' are allowed to be the names of CATS variables (or auxiliary variables) or may be given constant values. See INSFluid for more details on usage and unit conventions.

```
[Materials]
        [./ins_mat]
                type = INSFluid
                block = 0
                density = 1.225
                viscosity = 1.81E-5
        [../]
[]
```

Postprocessors

The Postprocessors block of the input file is used for calculating specific integrated variable values, boundary values, and/or probing variables in the mesh at specific elements. See the main Postprocessors section above for details on commonly used types of postprocessors. To invoke any of the above calculations, simply add the subblocks detailed in the subsections of Postprocessors into a '[Postprocessors]' block in your input file.

Preconditioning

Unless you are using the 'Newton' solver in the Executioner block, you will need to provide some form of preconditioning matrix. The Preconditioning block of the input file tells MOOSE how to construct a preconditioning matrix. For more details, see the main Preconditioning section above.

Executioner

The Executioner block tells MOOSE how to solve each time step, advance each time step, the linear and non-linear solvers to use, and the tolerances of those solvers. Without an Executioner block,

your simulation will not run. The Best Solver Options section above talks at length about what all needs to be in the Executioner block and provides a simple example.

Outputs

The last block of the input file is the Outputs block, which directs how often MOOSE should produce an output and what that output should be. Realistically, there are only 4 output options that you will invoke in CATS. Those options are detailed in the main Outputs section above. Below provides a brief example of how those options are invoked in the input file.

```
[Outputs]
    print_linear_residuals = true
    exodus = true
    csv = true
    interval = 10
[]
```

Simulation Examples

This section will provide simple demonstrations of how to use CATS to solve some a variety of problems. You can use the examples provided here to help you learn your way around the modeling framework and test out developing your own simulation cases. All simulations provided below are for demonstration purposes only. Your parameters, domains, variables, units, and objects may vary.

Conservation of Mass

The most common type of problem you may seek to solve involves a conservation of mass, wherein you track the fluid phase concentration evolution in a domain over time, as well as the mass transfer of the fluid phase to a washcoat or particle pore-space, then track adsorption or other reactions as well. For the purpose of this first demonstration to familiarize you with the input file syntax and basics of invoking kernels, let's first consider a simple, generic material balance as follows.

Consider the adsorption of a gas species in a column packed with solid particles. This model requires: (i) a mass balance on the gas species in the bulk spaces, (ii) a mass balance on the gas species in the pore-spaces of the particle, (iii) an adsorption reaction inside the particle, and (iv) a mass balance on the available surface sites. Thus, it will involve 4 primary equations each to solve for a different non-linear variable: (i) C for concentration in bulk, (ii) C_p for concentration in pore-spaces, (iii) q for adsorption, and (iv) S for surface sites. We also need to think about the units for each variable, as that will dictate how the equations should be formed. Most commonly: (i) C will have units of moles per cubic meter of gas, (ii) C_p will have moles per cubic meter of gas, (iii) q will have units of moles per kg of particle, and (iv) S will have units of moles per kg of particle. Thus, our equations might be as follows:

C: $$\varepsilon\frac{\partial C}{\partial t}+\nabla\cdot(\varepsilon\mathbf{v}\cdot C)=\nabla\cdot(\varepsilon D\cdot\nabla C)-(1-\varepsilon)\cdot k\cdot A_o\left(C-C_p\right)$$

C_p: $$\varepsilon_p\frac{\partial C_p}{\partial t}=-k\cdot A_o\left(C_p-C\right)-\rho_p\frac{dq}{dt}$$

q: $$\frac{dq}{dt}=k_f C_p S-k_r q$$

S: $$S=S_{max}-q$$

In this representation, ε is the bulk porosity, **v** is the linear velocity, D is the dispersion coefficient, k is the mass transfer rate, A_o is the ratio of particle outer surface area to particle volume (for spherical particles it would be 3/r where r is particle radius), ε_p is the porosity of the particles, ρ_p is the particle bulk density, k_f and k_r are forward and reverse reaction rates, and S_{max} is the maximum available adsorption sites.

To solve this problem, there are actually many different combinations of kernels that you can use to invoke each piece of physics for each equation. It largely depends on what is most convenient for your usage, but here we will using the following kernels:

VariableCoefTimeDerivative: $\varepsilon\frac{\partial C}{\partial t}$ and $\varepsilon_p\frac{\partial C_p}{\partial t}$

TimeDerivative: $\frac{dq}{dt}$

CoupledCoeffTimeDerivative: $-\rho_p \frac{dq}{dt}$

GPoreConcAdvection (DGPoreConcAdvection): $\nabla \cdot (\varepsilon \mathbf{v} \cdot C)$

GVarPoreDiffusion (DGVarPoreDiffusion): $\nabla \cdot (\varepsilon D \cdot \nabla C)$

FilmMassTransfer: $-(1-\varepsilon) \cdot k \cdot A_o(C - C_p)$ and $-k \cdot A_o(C_p - C)$

ConstReaction: $k_f C_p S - k_r q$

MaterialBalance: $S = S_{max} - q$

For more information on each of these kernels, click on the respective kernel names. It is worth noting that some of these kernels accept variable arguments for their parameters, while others expect given constant values. In addition, the (1-ε) term for the first FilmMassTransfer kernel is not actually explicitly available in that given kernel. Thus, it may be convenient to create a new, compound rate parameter that is the product of (1-ε) and k and pass that product to the kernel as the k value. As this project continues to grow and evolve, more and more specific kernels will be added as necessary in order to prevent these kinds of special treatment you may need to apply to your problems.

In addition to the kernels needed to simulate each piece of physics in the 4 equations, we need to include boundary conditions for the first equation, since it is a PDE. For this problem, we will have a boundary condition kernel at the inlet and a boundary condition at the outlet. Because our bulk concentration variable C involves a porosity term, we need to use the DGPoreConcFluxBC kernel. This kernel can be applied to the inlet and outlet boundaries simultaneously or separately because it uses the velocity vector and normal vectors at the boundary to naturally distinguish between inputs and outputs. At the inlet boundary, we need to specify a value for the concentration entering the domain.

DGPoreConcFluxBC: $(\varepsilon \mathbf{v} \cdot \mathbf{n}) \cdot C_{inlet}$ and $(\varepsilon \mathbf{v} \cdot \mathbf{n}) \cdot C$

Now that we have identified all the kernels and boundary conditions, we can begin to construct the input file to simulate this case. For detailed information on each block of the input file and input file structure in general, review the Input Files section. In the below section, we discuss each block of the input file for this particular example and provide our reasoning for each specific parameter choice and/or kernel usage. The full input file will be available in the CATS code repository (https://github.com/aladshaw3/cats/tree/master/user_examples).

The Input File

First, we will add blocks to the input file to represent the physical domain we want to simulate on. In this example, we want to simulate adsorption in a packed column and thus our domain will be made cylindrical. To make an axis-symmetric 2D domain, we must specify a '[Problem]' block with 'coord_type' set to 'RZ' and then use the GeneratedMeshGenerator in the mesh block as shown below.

```
[Problem]
        coor_type = RZ
[]
```

```
[Mesh]
    [./my_mesh]
        type = GeneratedMeshGenerator
        dim = 2
        nx = 5
        ny = 20
        xmin = 0
        xmax = 0.05
        ymin = 0
        ymax = 0.1
    [../]
[]
```

The sections above create a cylindrical mesh with a radius of 0.05 m (x-direction → radius) and length of 0.1 m (y-direction → length). The domain is meshed into 5x20 elements and identified as a 2 dimensional structure.

After setting up the simulation domain, we need to declare what variable and auxiliary variables are involved with this simulation. The variables are the values that the model is implicitly solving for: (i) C, (ii) C_p, (iii) q, and (iv) S. Auxiliary variables can be specific parameters that are solved for independently or are just place holders for specific constant values we give them. In this example, we will just use constants as our auxiliary variables and declare their values to be their initial conditions.

```
[Variables]
    [./C]
        order = FIRST
        family = MONOMIAL
        initial_condition = 0
    [../]
    [./Cp]
        order = FIRST
        family = MONOMIAL
        initial_condition = 0
    [../]
    [./q]
        order = FIRST
        family = MONOMIAL
        initial_condition = 0
    [../]
    [./S]
        order = FIRST
        family = MONOMIAL
        initial_condition = 0
    [../]
[]
```

```
[AuxVariables]
        # porosity
        [./eps]
                order = FIRST
                family = LAGRANGE
                initial_condition = 0.5
        [../]
        [./vel_x]
                order = FIRST
                family = LAGRANGE
                initial_condition = 0
        [../]
        [./vel_y]
                order = FIRST
                family = LAGRANGE
                initial_condition = 3
        [../]
        [./vel_z]
                order = FIRST
                family = LAGRANGE
                initial_condition = 0
        [../]
        [./D]
                order = FIRST
                family = LAGRANGE
                initial_condition = 2.5E-5
        [../]
        [./k]
                order = FIRST
                family = LAGRANGE
                initial_condition = 1
        [../]
        [./k_eps]
                # Represents (1 – eps)*k
                order = FIRST
                family = LAGRANGE
                initial_condition = 0.5
        [../]
        [./eps_p]
                order = FIRST
                family = LAGRANGE
                initial_condition = 0.25
        [../]
```

```
        [./S_max]
            order = FIRST
            family = LAGRANGE
            initial_condition = 1
        [../]
[]
```

Note that the AuxVariables do not cover all the parameters. Some kernels we use are looking for parameters in their subblocks as specific constant values, so we do not place those here as an AuxVariable. In future, newer kernels may have more flexibility for accepting both variable arguments and constant arguments.

Next, we need to invoke the Kernels, DGKernels, and BCs to invoke all the chemistry and physics that governs our model. We then pass to those kernels the variables, constant values, and other arguments that those kernels need to build the total model. Also note that every 'G' or 'DG' prefixed kernel or DG kernel must be paired with the same input arguments. The combinations of these two sets of kernels are needed to fully describe transport physics with DG methods. See here for more information.

```
[Kernels]
    #Conservation of mass on C
    [./C_dot]
        type = VariableCoefTimeDerivative
        variable = C
        coupled_coef = eps
    [../]
    [./C_gadv]
        type = GPoreConcAdvection
        variable = C
        porosity = eps
        ux = vel_x
        uy = vel_y
        uz = vel_z
    [../]
    [./C_gdiff]
        type = GVarPoreDiffusion
        variable = C
        porosity = eps
        Dx = D
        Dy = D
        Dz = D
    [../]
    [./C_trans_Cp]
        type = FilmMassTransfer
        variable = C
```

```
        coupled = Cp
        rate_variable = k_eps
        # Area to volume ratio: Ao
        av_ratio = 5000
[../]

#Conservation of mass on Cp
[./Cp_dot]
        type = VariableCoefTimeDerivative
        variable = Cp
        coupled_coef = eps_p
[../]
[./Cp_trans_C]
        type = FilmMassTransfer
        variable = Cp
        coupled = C
        rate_variable = k
        # Area to volume ratio: Ao
        av_ratio = 5000
[../]
[./ads_q]
        type = CoupledCoeffTimeDerivative
        variable = Cp
        coupled = q
        time_coeff = 1500
[../]

# Conservation of mass for q
[./q_dot]
        type = TimeDerivative
        variable = q
[../]
[./q_rxn]
        type = ConstReaction
        variable = q
        this_variable = q
        forward_rate = 2
        reverse_rate = 0.5
        scale = 1
        reactants = 'Cp S'
        reactant_stoich = '1 1'
        products = 'q'
        product_stoich = '1'
[../]
```

```
        # Conservation of mass for S
        [./S_bal]
                type = MaterialBalance
                variable = S
                this_variable = S
                coupled_list = 'S q'
                weights = '1 1'
                total_material = S_max
        [../]
[]

[DGKernels]
        [./C_dgadv]
                type = DGPoreConcAdvection
                variable = C
                porosity = eps
                ux = vel_x
                uy = vel_y
                uz = vel_z
        [../]
        [./C_dgdiff]
                type = DGVarPoreDiffusion
                variable = C
                porosity = eps
                Dx = D
                Dy = D
                Dz = D
        [../]

[]

[BCs]
        [./C_FluxIn]
                type = DGPoreConcFluxBC
                variable = C
                boundary = 'bottom'
                u_input = 1
                porosity = eps
                ux = vel_x
                uy = vel_y
                uz = vel_z
        [../]
```

```
        [./C_FluxOut]
            type = DGPoreConcFluxBC
            variable = C
            boundary = 'top'
            porosity = eps
            ux = vel_x
            uy = vel_y
            uz = vel_z
        [../]
[]
```

Next, for this demonstration, we want to monitor some specific simulation results as Postprocessors. Of particular interest is (i) the exit concentration of C to yield a breakthrough curve and (ii) the average adsorption (q) in the column. Since breakthrough occurs at a boundary, we want to use the SideAverageValue object at the 'top' boundary. This will yield and integral average of C in the top-most set of elements at that boundary.

```
[Postprocessors]
    [./C_exit]
        type = SideAverageValue
        boundary = 'top'
        variable = C
        execute_on = 'initial timestep_end'
    [../]
    [./q_avg]
        type = ElementAverageValue
        variable = q
        execute_on = 'initial timestep_end'
    [../]
[]
```

Lastly, we need to setup the Preconditioning, Executioner, and Outputs. Examples for how to setup these blocks were provided earlier (Preconditioning and Executioner and Outputs). To view all of these put together, check out the actual input file for this example in the CATS code repository at (https://github.com/aladshaw3/cats/tree/master/user_examples).

Conservation of Energy

This example will continue from where the last one left off. Here, we will consider an energy balance inside a packed column. In this scenario, we have the energy density of the fluid phase (E_f) and solid phases (E_s) to consider. Those energy densities are related to the temperatures of each phase (T_f and T_s) through their respective densities and heat capacities. In the absence of reactions and adsorption, those energy balances and temperature relationships could be represented as follows.

E_f: $\varepsilon \frac{\partial E_f}{\partial t} + \nabla \cdot \left(\varepsilon \mathbf{v} \cdot E_f\right) = \nabla \cdot \left(\varepsilon K_f \cdot \nabla T_f\right) - (1-\varepsilon) \cdot h \cdot A_o \left(T_f - T_s\right)$

E_s: $\frac{\partial E_s}{\partial t} = \nabla \cdot (K_s \cdot \nabla T_s) - h \cdot A_o \left(T_s - T_f\right)$

T_f: $E_f = \rho_f c_{pf} T_f$

T_s: $E_s = \rho_p c_{ps} T_s$

Note that the units from above are as follows: E_f and E_s (J/m^3), K_f and K_s (W/m/K), h (W/m^2/K), A_o (m^{-1}), T_f and T_s (K), ρ_f and ρ_p (kg/m^3), and c_{pf} and c_{ps} (J/kg/K). The energy density of the fluid is per volume of fluid and the energy density of the solid is per bulk volume of solids. As a consequence, the solid properties must also be per bulk volume (at least for this example).

The kernels that correspond to the above physics are shown below:

VariableCoefTimeDerivative: $\varepsilon \frac{\partial E_f}{\partial t}$

TimeDerivative: $\frac{\partial E_s}{\partial t}$

GPoreConcAdvection (DGPoreConcAdvection): $\nabla \cdot \left(\varepsilon \mathbf{v} \cdot E_f\right)$

GPhaseThermalConductivity (DGPhaseThermalConductivity): $\nabla \cdot \left(\varepsilon K_f \cdot \nabla T_f\right)$ and $\nabla \cdot (K_s \cdot \nabla T_s)$

PhaseEnergyTransfer: $-(1-\varepsilon) \cdot h \cdot A_o \left(T_f - T_s\right)$ and $-h \cdot A_o \left(T_s - T_f\right)$

PhaseTemperature: $E_f = \rho_f c_{pf} T_f$ and $E_s = \rho_p c_{ps} T_s$

In addition to the above system of equations for the interior of the domain, we must include boundary conditions for E_f and Es, since these are the variables that are represented by PDEs. As with the conservation of mass, the mobile fluid phase will have boundary conditions at the inlet and outlet of the column where the gas flows into and out of the system. Those boundary condition kernels are provided below.

DGFlowEnergyFluxBC: $(\varepsilon \mathbf{v} \cdot \mathbf{n}) \cdot \rho_f c_{pf} T_{f,inlet}$ and $(\varepsilon \mathbf{v} \cdot \mathbf{n}) \cdot E_f$

However, since energy can also be exchanged across the walls of the domain, we must also include a boundary condition for that exchange of energy for both the fluid and solid phases. The reasoning behind including this energy term for both phases is due to the fact that in a packed column, but phases will be in contact with the wall. Each may have a different representative area for contact with the wall represented by the ε and $(1-\varepsilon)$ factors, respectively.

DGWallEnergyFluxBC: $h_w \cdot \varepsilon \cdot \left(T_f - T_w\right)$ and $h_w \cdot (1-\varepsilon) \cdot (T_s - T_w)$

Since we have already covered the input file structure in great detail in the above example, as well as in the main section on input files, we will not rework through all those details here. Instead, just see the sample input file provided (https://github.com/aladshaw3/cats/tree/master/user_examples).

Coupling Mass and Energy

Up to this point, we have demonstrated both a mass and energy balance separately, however, realistic simulations will involve the coupling of mass and energy together. To accomplish this, we need to make a minor kernel change to the mass balances and add a new kernel to the energy balances. For the mass balance, we need to update the ConstReaction kernel to the ArrheniusReaction or ArrheniusEquilibriumReaction kernel to couple the forward and reverse rates with temperature. Since the reaction in the mass balance is an adsorption/surface reaction, it should be coupled with the solid temperature and not the gas temperature. The new kernel also has different parameters to set and those parameters will be common with the corresponding ArrheniusReactionEnergyTransfer or ArrheniusEquilibriumReactionEnergyTransfer kernels. Thus, it is advantageous to place those common parameters under the GlobalParams block of the input file as shown below.

```
[GlobalParams]
    forward_activation_energy = 0
    forward_pre_exponential = 2
    reverse_activation_energy = 2305
    reverse_pre_exponential = 1
    enthalpy = -2305
[]
```

In the Kernels block of the input file, the ConstReaction kernel for variable q is replaced with ArrheniusReaction. Since the forward and reverse rates are calculated as a function of temperature, we can replace the input arguments for the rates with just the coupled temperature of the solid (Ts). For the energy balance of the solid phase, we need to add a new kernel to account for the energy transfer caused by the reaction. Thus, the new energy balance would be as shown below.

E_s: $$\frac{\partial E_s}{\partial t} = \nabla \cdot (K_s \cdot \nabla T_s) - h \cdot A_o\left(T_s - T_f\right) + \rho_p(-\Delta H)\left[k_f C_p S - k_r q\right]$$

The new term involves the enthalpy of the reaction, the reaction rate expression, and a unit conversion factor, which in this case is the particle bulk density. This ensures that the units of the new term are in energy per bulk volume of solids. In the ArrheniusReactionEnergyTransfer kernel, the density of the particles is passed as the 'volume_frac' argument. The enthalpy was already declared in the GlobalParams block, so it does not need to be repeated here. To look at the actual input file for this coupled simulation case, check out (https://github.com/aladshaw3/cats/tree/master/user_examples).

Adding Gas Properties

Up to this point, we have simulated processes with constant coefficients. However, rate parameters such as the diffusion/dispersion coefficients and film mass transfer rates are themselves functions of temperature and other factors based on kinetic theory of gases. If we want properties such as these to be calculate automatically, we can invoke the various Auxiliary Kernels for those parameter calculations to use in our simulation.

In this demo, we will use the auxiliary system to calculate (i) pressure drop across the column, (ii) dispersion coefficients, (iii) film mass transfer rates, (iv) gas density, (v) gas viscosity, (vi) gas thermal

conductivity, and (vii) gas specific heat. To help accommodate this, we will also add another auxiliary variable for the 'carrier' gas, since our concentration variable C is too dilute to account for total gas density. In addition, previously we had introduced an auxiliary variable 'k_eps' to represent a conversion factor of the true k value by the $(1 - \varepsilon)$ factor. Now, we need to move that unit conversion into the kernel add apply it to the 'av_ratio' to correct the kernel so that the 'k' value can come exclusively from the auxiliary kernels.

Below are the auxiliary kernels we need to invoke and what they apply to:

AuxErgunPressure:	New auxiliary variable P (in Pa)
GasSpeciesAxialDispersion:	Parameter D (in m^2/s)
GasSpeciesMassTransCoef:	Parameter k (in m/s)
GasDensity:	Parameter ρ_f (in kg/m^3)
GasViscosity:	New auxiliary variable μ_f (in kg/m/s)
GasThermalConductivity:	Parameter K_f (in W/m/K)
GasSpecHeat:	Parameter c_{pf} (in J/kg/K)

For this demo, our carrier gas variable will be 'N2' (i.e., nitrogen gas) and we will give that a concentration of 40 mol/m^3, which will roughly correspond to density of air at standard temperature and pressure. Then, in the GlobalParams block, we add a set of input arguments that are common to all the auxiliary kernels we are invoking in this simulation. This includes the list of gases, their corresponding molecular weights, and their Sutherland's constants. In this example, our gas concentration variable C will represent the concentration of water vapor, so we will use the molecular weight and Sutherland information for water. Therefore, we add the following lines to the GlobalParams block. For more information on these input arguments, see GasPropertiesBase.

```
gases = 'N2 C'
molar_weights = '28 18'
sutherland_temp = '300.55 298.16'
sutherland_const = '111 784.72'
spec_heat = '1.04 1.97'
is_ideal_gas = false
execute_on = 'initial timestep_end'
```

Lastly, add the auxiliary kernels list above to the AuxKernels block of the input file. Each auxiliary kernel may require some addition variable arguments, such as temperature and pressure. For details on each input argument, click on the links/names of the auxiliary kernels named above. To view the input file for this simulation, check out (https://github.com/aladshaw3/cats/tree/master/user_examples).

NOTE: You will need to change some of the auxiliary variables from LAGRANGE to MONOMIAL families. In general, only velocities and temperatures should be LAGRANGE.

NOTE: Simulation results for temperature will be very different, because of the new properties.

Coupling Mass and Energy with Simple Gas Properties

In the discussion above, we talked about using the GasProperties system to calculate properties associated with mass and energy balances. In this older properties system, a user was forced into using a specific unit basis for their problems. This can be very inconvenient for systems that have widely varied length and time scales. In addition, these more involved property calculations requires very specific information on individual gas species properties that may be difficult to find. To get around both these issues, a newer SimpleGasProperties system was developed that makes a few simplifying assumptions and relationships to calculate system properties with limited parameter knowledge. This newer system also allows the user to specify parameter units on input and desired units on output, thus allowing a user to perform balances on any unit basis for both mass and energy balances.

An example of using this system can be found in the CATS repository located under 'user_examples/simple_coupling_mass_and_energy.i'. In this example, we solve a fully coupled mass and energy system for the light-off of CO and H_2 in a three-way catalyst. We can represent the problem mathematically as follows…

MASS BALANCES

Fluid: $$\varepsilon_b \frac{\partial C_i}{\partial t} + \nabla \cdot (\varepsilon_b \mathbf{v} \cdot C_i) = \nabla \cdot (\varepsilon_b D \cdot \nabla C_i) - (1-\varepsilon_b) \cdot A_o \cdot k_m \left(C_i - C_{w,i}\right)$$

Solid: $$\varepsilon_w (1-\varepsilon_b) \frac{\partial C_{w,i}}{\partial t} = (1-\varepsilon_b) \cdot A_o \cdot k_m \left(C_i - C_{w,i}\right) + (1-\varepsilon_b) \cdot \sum u_i r_j$$

Reactions:

$CO + 0.5\ O_2 \rightarrow CO_2$	$r_1 = k_1 C_{w,CO} C_{w,O2}$
$H_2 + 0.5\ O_2 \rightarrow H_2O$	$r_2 = k_2 C_{w,H2} C_{w,O2}$
$CO + H2O \leftrightarrow CO2 + H2$	$r_3 = k_3 C_{w,CO} C_{w,H2O} - k_{-3} C_{w,CO2} C_{w,H2}$

ENERGY BALANCES

Fluid: $$\varepsilon_b \frac{\partial E_f}{\partial t} + \nabla \cdot \left(\varepsilon_b \mathbf{v} \cdot E_f\right) = \nabla \cdot \left(\varepsilon_b K_f \cdot \nabla T_f\right) - (1-\varepsilon_b) \cdot h_s \cdot A_o \left(T_f - T_s\right)$$

Solid: $$(1-\varepsilon_b) \frac{\partial E_s}{\partial t} = \nabla \cdot \left((1-\varepsilon_b) K_s \cdot \nabla T_s\right) - (1-\varepsilon_b) \cdot h_s \cdot A_o \left(T_s - T_f\right) + (1-\varepsilon_b) \cdot \sum (-\Delta H_i) r_j$$

Fluid Temperature: $$E_f = \rho_f c_{pf} T_f$$

Solid Temperature: $$E_s = \rho_p c_{ps} T_s$$

In this example, the spatial dimensions are in cm, the time dimension is in min, our concentrations (C_i and $C_{w,i}$) are in units of mol/L, and our energy densities (E_f and E_s) are in J/cm^3. Because of our unit convention for concentrations, the reaction variables (r_i) have units of mol/L/min, where the volume is on a solids volume basis. Thus, when those reaction kernels are put into both the mass and energy balances, we need the scale factor of (1 - ε_b) to put the balances on the basis of total volume (rather than just solids volume). In addition, this means that the units of reaction enthalpies (ΔH_i) must include a conversion from volume as L to volume as cm^3 to account for the fact that out

volume units on energy density are different from out volume units on concentration. As a result, the units for ΔH_i are J*L/mol/cm^3 (which is equivalent to kJ/mol).

Also note, that the sign of ΔH_i used in the kernels is the opposite sign of the actual ΔH_i values. This is because of the implicit negative sign in the energy balance. Thus, if ΔH_i is -283 kJ/mol, you supply the value as +283.

The kernels invoked are mostly the same sets of kernels discussed above for mass and energy balances independently, with some minor differences for coupling reactions. Both the reaction coupling in the mass and reaction coupling in the energy balance use the same kernel (see below).

ScaledWeightedCoupledSumFunction: $(1-\varepsilon_b)\cdot\sum u_i r_j$ and $(1-\varepsilon_b)\cdot\sum(-\Delta H_i) r_j$

In this example, we also use the recursive Krylov subspace methods to precondition linear steps with other Krylov subspace methods. Details of this, and other example details are written in the comments of the simulation run file ('user_examples/simple_coupling_mass_and_energy.i'). See that input file for additional details.

MOOSE CG Incompressible Navier-Stokes

The incompressible Navier-Stokes module is not developed in CATS, it is a part of the MOOSE framework. However, we can utilize this module inside of CATS to resolve approximations to the flow field in a domain. In this section, we will discuss 2 methods to invoke the Navier-Stokes module: (i) in cartesian coordinates and (ii) in RZ-cylindrical coordinates. The simulations in cartesian coordinates will be further subdivided into fully bounded and semi-bounded domains. Information on each case is provided in the sections below.

INS Cartesian - Fully Bounded

In a fully bounded domain, there will be walls to constrain and constrict the flow such that only the truly open boundaries will allow for flux. At the walls of the system, we will apply a "no slip" boundary condition, which is a standard wall condition for Navier-Stokes problems. Then, we only need to define either the inlet flux or outlet flux for the open boundaries. The reason we only need to specify one of the two open boundaries is because the continuity equation will inherently enforce a total flow balance such that the flow leaving the domain will equal the flow entering the domain.

For the purpose of this demonstration, we will use a 2D flow domain of effectively the same size as the mesh from the previous examples. However, the previous mesh was in 'RZ' cylindrical coordinates. Here, we are in cartesian coordinates, so we will remove the 'RZ' designation in the Problem block of the input file. In addition, we will modify the mesh such that the total x-direction length would span the 'diameter' of the column from the previous example. The reason for this is so we can get an approximate flow cross section of the column as if the column were a cubic block rather than a cylinder. Since we are extended the length of the x-direction, we also need to increase the number of elements in the x-direction. The new mesh block can be assembled as follows.

```
[Mesh]
    [./my_mesh]
        type = GeneratedMeshGenerator
        dim = 2
        nx = 10
        ny = 20
        xmin = -0.05
        xmax = 0.05
        ymin = 0
        ymax = 0.1
    [../]
[]
```

We also need to add several input arguments into the GlobalParams block for some stabilization options for the incompressible Navier-Stokes module. It is highly recommended that you always use the same options provided below, except for the 'gravity' and 'alpha' arguments. These can be changed to fit your specific problem or needed level of stabilization. The 'gravity' argument is a vector of acceleration constants you can provide for x, y, and z, respectively. In this case, we specify a gravitational acceleration constant of 9.8 m/s^2 in the negative y-direction. The 'alpha' option can be any number between 0.1 and 2, where 1 seems to give the best results. See the example file for more details (https://github.com/aladshaw3/cats/tree/master/user_examples).

```
[GlobalParams]
    gravity = '0 -9.8 0'
    alpha = 1
    integrate_p_by_parts = true
    supg = true
    pspg = true
    laplace = true
    convective_term = true
    transient_term = true
[]
```

To use the incompressible Navier-Stokes module, we are required to provide a Materials block that gives the fluid phase density (in kg/m^3) and viscosity (in kg/m/s). For this, there is a custom Materials object in CATS called INSFluid that we will invoke. The input arguments for density and viscosity may be given as constants or as auxiliary variables. For the purpose of this demo, we will give them constants that emulate the approximate values of density and viscosity of air under our simulation conditions from before. The Materials block for this input file is structured as below.

```
[Materials]
    [./ins_material]
        type = INSFluid
        density = 1.13
```

```
        viscosity = 2.25E-5
    [../]
[]
```

In simulations of incompressible Navier-Stokes, the velocity variable is introduced as it's vector components (i.e., v_x, v_y, and v_z) and a variable for dynamic pressure (p) must be included for the continuity equation. For 2D simulation, we still include v_z, but only as an auxiliary variable whose value is set to 0. All of the variables for Navier-Stokes must be of the LAGRANGE family. See the example files for samples of inputs (https://github.com/aladshaw3/cats/tree/master/user_examples).

The kernels that need to be invoked for the incompressible Navier-Stokes module are formulated as follows. Note that there are no DG kernels to invoke here because the MOOSE Navier-Stokes module uses a different stabilization technique for transport.

```
[Kernels]
    #Continuity Equation
    [./mass]
        type = INSMass
        variable = p
        u = vel_x
        v = vel_y
        w = vel_z
        p = p
    [../]

    # Conservation of x-momentum
    [./x_dot]
        type = INSMomentumTimeDerivative
        variable = vel_x
    [../]
    [./x_ins]
        type = INSMomentumLaplaceForm
        variable = vel_x
        u = vel_x
        v = vel_y
        w = vel_z
        p = p
        component = 0
    [../]

    # Conservation of y-momentum
    [./y_dot]
        type = INSMomentumTimeDerivative
        variable = vel_y
    [../]
```

```
        [./y_ins]
            type = INSMomentumLaplaceForm
            variable = vel_y
            u = vel_x
            v = vel_y
            w = vel_z
            p = p
            component = 1
        [../]

    []
```

The most important components for the incompressible Navier-Stokes simulations are the boundary conditions. Due to how the module was constructed in MOOSE, the kernels will always be the same. Thus, the various boundary conditions you apply are really what controls the flow profile. Review the incompressible Navier-Stokes section for a listing of the most common boundaries. In this example, we have 2 wall boundaries (i.e., 'left' and 'right' that correspond to the 'xmin' and 'xmax' values in the mesh) and 2 open boundaries (i.e., 'bottom' and 'top' that correspond to 'ymin' and 'ymax' values in the mesh). At the 'bottom' boundary we define our inflow conditions using the INSNormalFlowBC, which dictates the flux normal to the boundary for a given variable. We do not need to specify anything for the 'top' boundary as the continuity equation will inherently enforce the continuity of the flow. For the 'left' and 'right' boundaries, we need to apply the PenaltyDirichletBC for both 'vel_x' and 'vel_y' to enforce the "no slip" boundary conditions at the walls. The BCs block of the input file can be viewed from the example files (https://github.com/aladshaw3/cats/tree/master/user_examples).

Lastly, we may want to use some custom Postprocessors as a way to check and make sure the continuity equation and our boundary conditions are conserving the flow. To check this, we can use the VolumetricFlowRate postprocessor that is built into the MOOSE module to compute the flow rate at the 2 open boundaries. If the flow is conserved, then the exit flow rate will approximately equal to the exit flow rate. Note that depending on your tolerances for the solver, these values may be slightly different, but they should be very close to each other. Be sure to check out the actual example input files for these simulation cases (https://github.com/aladshaw3/cats/tree/master/user_examples).

NOTE: The incompressible Navier-Stokes module has very poor convergence if you do not use 'lu' as one of your preconditioning options. See PETSc options for more details.

INS Cartesian - Semi-Bounded

Occasionally, your simulation domain may not be fully bounded (i.e., does not fully confine the flow or have walls at all boundaries) or you may want to simulate a case where there is expected to be symmetry of the flow profile at a boundary. In these cases, everything from the above example will be the same, but we must change the boundary conditions that we specify. In this example, let's consider the same domain as above, but because the flow profile will be symmetric about x=0, we will only mesh half the domain and invoke some custom boundaries to create that flow symmetry.

The new mesh would be as follows:

```
[Mesh]
    [./my_mesh]
        type = GeneratedMeshGenerator
        dim = 2
        nx = 5
        ny = 20
        xmin = 0
        xmax = 0.05
        ymin = 0
        ymax = 0.1
    [../]
[]
```

Now, the 'left' boundary (corresponding to 'xmin') of the domain is our axis of symmetry. We will have the same inflow and 'right' boundary conditions (i.e., "no slip" at the walls) as the previous example, but the 'left' boundary will be different for 'vel_x' and 'vel_y'. For 'vel_x' at the 'left' boundary, we must enforce a zero velocity condition, which is essentially the same as the "no slip" boundary. The reason for this is we must not allow velocities to "flux" across a boundary normal to our axis of symmetry. If flux is allowed, then we essentially have a "vacuum" condition at that boundary that would allow flow to be lost to outside our domain. So we use the PenaltyDirichletBC to ensure there is no flux in the x-direction at the axis of symmetry.

```
    [./x_center]
        type = PenaltyDirichletBC
        variable = vel_x
        boundary = 'left'
        value = 0
        penalty = 1000
    [../]
```

For the 'vel_y' at the 'left' boundary, we must use the NeumannBC to enforce the slope of the velocity in the y-direction is zero. Note that you can also invoke this boundary condition by doing nothing in the input file as this is the "natural" boundary condition for standard finite elements. The physical interpretation of this boundary condition is that we are enforcing that the velocity in elements just outside the domain must be the same as the velocity for elements just inside the domain (i.e., the velocity profile is symmetric about the domain).

```
    [./y_center]
        type = NeumannBC
        variable = vel_y
        boundary = 'left'
        value = 0
    [../]
```

For additional details, check out (https://github.com/aladshaw3/cats/tree/master/user_examples). Note that the simulation results from the fully bounded and semi-bounded domains will produce the same velocity profiles from x=0 to x='xmax', thus demonstrating that we can reduce complexity of simulations by taking advantage of symmetry where possible.

INS RZ

The last incompressible Navier-Stokes example is for building on prior mass and energy examples in our previously cylindrical domain. What we did above (fully bounded and semi-bounded) was to try to estimate a flow profile for a 2D slice of our column, but in cartesian coordinates. All dimensions were the same as the cylindrical column, but the spatial representation was not accurate. As a result, the cartesian examples provided an effective flow profile for a cubic conduit and not a cylinder. To get the flow profile in a cylinder, we must use some custom 'RZ' Navier-Stokes kernels that apply coordinate system transformations necessary to simulate a more accurate flow profile.

NOTE: These kernels are significantly less efficient and may change in future.

There are only 3 changes that need to be made to our previous example simulations to account for the coordinate changes: (i) the Problem block needs to define the 'coord_type' as 'RZ', (ii) the 'INS' kernels, except for the time derivatives, need to be suffixed with 'RZ' (i.e., INSMass → INSMassRZ), and (iii) the boundary conditions need to be updated. Boundary condition at the inflow boundary for the 'vel_y' variable is the same as before, but now the "no slip" conditions need to be applied to 'vel_x' at both the 'left' and 'right' boundaries (i.e., center line and the wall) while the "no slip" condition for 'vel_y' is only applied at the 'right' boundary (i.e., the wall). Check out the example files for all Navier-Stokes cases at (https://github.com/aladshaw3/cats/tree/master/user_examples).

CATS DG Navier-Stokes

Within CATS, we now have a working implementation of both incompressible and compressible Navier-Stokes using DG methods. This implementation solves the continuity and momentum conservation equations (outlined here) by performing integration-by-parts for each velocity component and by applying penalty-based pseudo-Dirichlet boundary conditions to enforce a 'No Slip' condition at the walls or obstructions in a mesh.

NOTE: A consequence of integrating-by-parts is that this formulation is valid ONLY in Cartesian coordinates. Thus, you cannot use these methods currently for 'RZ' coordinate systems.

Variables

To use our Navier-Stokes module, you will need to define a 'pressure' variable and a 'velocity' variable for each direction (i.e., x, y, and z directions). NOTE: Any velocity component is technically optional, as you can replace one velocity term with a 0 if you don't need that dimension.

In this example, we will create variables for 'pressure' and 'vel_x' and 'vel_y' because we will do a 2D Navier-Stokes simulation. To do this, we MUST define 'pressure' as a 'FIRST' order 'LAGRANGE'

variable and each velocity as a 'SECOND' order 'MONOMIAL' variable. For velocity in the z-direction, we create an 'AuxVariable' and default its value to 0.

NOTE: In general, you should provide variables for all velocity components, even if the simulation is only 2D. This is because how some of the 'DG' kernels are implemented, you are required to supply velocity terms as variables and cannot just use default coupling.

```
[Variables]
        [./pressure]
                order = FIRST
                family = LAGRANGE
                initial_condition = 0.0
        [../]
        [./vel_x]
                order = SECOND
                family = MONOMIAL
                initial_condition = 0.0
        [../]

        [./vel_y]
                order = SECOND
                family = MONOMIAL
                initial_condition = 0.0
        [../]
[]

[AuxVariables]
        [./vel_z]
                order = SECOND
                family = MONOMIAL
                initial_condition = 0.0
        [../]
[]
```

Kernels/DGKernels

Next, we need to add kernels to describe the Navier-Stokes equations and integrate the solution by parts. This is accomplished by using a combination of some of the existing DG kernels discussed in other examples, as well as combining those kernels with new kernels designed specifically for the Navier-Stokes equations. Below gives a brief example of how to invoke the kernels we need, and what equations they solve.

```
[Kernels]
  ####  Enforce Divergence of velocity = 0 ###
  ### NOTE: If 'rho' is constant, then this will be incompressible
  ###          but if 'rho' is not constant, this will account for pressure accordingly
  [./continuity]
```

```
            type = DivergenceFreeCondition
            variable = pressure
            ux = vel_x
            uy = vel_y
            uz = vel_z
            coupled_scalar = rho
[../]


### Conservation of x-momentum ###
#       This example only include 'internal' forces
#       (i.e., the viscous forces and pressure forces)

# rho* d(vel_x)/dt
[./x_dot]
            type = VariableCoefTimeDerivative
            variable = vel_x
            coupled_coef = rho
[../]
# -grad(P)_x
[./x_press]
            type = VectorCoupledGradient
            variable = vel_x
            coupled = pressure
            vx = 1
[../]
# Div*(mu*grad(vel_x))
[./x_gdiff]
            type = GVariableDiffusion
            variable = vel_x
            Dx = mu
            Dy = mu
            Dz = mu
[../]
# Div*(rho*vel*vel_x)
[./x_gadv]
             type = GNSMomentumAdvection
             variable = vel_x
             this_variable = vel_x
             density = rho
             ux = vel_x
             uy = vel_y
             uz = vel_z
[../]
# (1/3)*mu* (grad(Div*vel)_x)
[./x_gvisdiv]
             type = GNSViscousVelocityDivergence
             variable = vel_x
```

```
                this_variable = vel_x
                viscosity = mu
                ux = vel_x
                uy = vel_y
                uz = vel_z
  [../]

[]

# Recall:  ANY 'G' prefixed kernel MUST have an equivalent 'DG' kernel active
[DGKernels]
 # Div*(mu*grad(vel_x))
 [./x_dgdiff]
            type = DGVariableDiffusion
            variable = vel_x
            Dx = mu
            Dy = mu
            Dz = mu
 [../]
 # Div*(rho*vel*vel_x)
 [./x_dgadv]
              type = DGNSMomentumAdvection
              variable = vel_x
              this_variable = vel_x
              density = rho
              ux = vel_x
              uy = vel_y
              uz = vel_z
 [../]
 # (1/3)*mu* (grad(Div*vel)_x)
 [./x_gvisdiv]
               type = DGNSViscousVelocityDivergence
               variable = vel_x
               this_variable = vel_x
               viscosity = mu
               ux = vel_x
               uy = vel_y
               uz = vel_z
  [../]

[]
```

Boundary Conditions

Lastly, we need to enforce proper boundary conditions on the system. For the pressure variable, the simplest boundary condition is to enforce that the pressure at any outlet to the system is 0. In this way, you are actually solving for the 'dynamic' or 'gage' pressure in the system rather than the absolute pressure. However, in principle, you can supply any Dirichlet condition you want for pressure at the

outlet. **It should also be noted, the units of pressure that get solved for are entirely dependent on the units you give for density and viscosity of the fluid.**

```
[BCs]
        # Zero pressure at exit
        [./press_at_exit]
                type = DirichletBC
                variable = pressure
                boundary = 'outlet'
                value = 0.0
        [../]
[]
```

The boundary conditions for velocity should come in 3 parts: (i) some inlet velocity for flux into the domain, (ii) some conservation of momentum statement leaving the domain, and (iii) some form of 'No Slip' condition at the walls or at any obstructions in the domain. For BCs (i) and (iii), we will make use of the 'PenaltyDirichletBC' system in MOOSE. In most cases tested, a 'penalty' value of about 300 seems to be the best compromise between accuracy and efficiency. There are several flavors of these types of boundary conditions, so have a look at this link for more options (https://mooseframework.inl.gov/syntax/index.html).

For BC of type (ii), we have developed a conservation of momentum equation for the outflow boundary of any domain (DGNSMomentumOutflowBC). This would be applied to all velocity components at all outlets of a domain.

```
[BCs]
        # Inlet velocity at boundary
        [./vel_x_inlet]
                type = FunctionPenaltyDirichletBC
                variable = vel_x
                boundary = 'inlet'
                penalty = 3e2
                function = '0.1*t'
        [../]

        ### Momentum Flux Out of Domain ###
        # in x-direction
        [./vel_x_outlet]
                type = DGNSMomentumOutflowBC
                variable = vel_x
                this_variable = vel_x
                boundary = 'outlet'
                density = rho
                ux = vel_x
                uy = vel_y
```

```
                uz = vel_z
        [../]
        # in y-direction
        [./vel_y_outlet]
                type = DGNSMomentumOutflowBC
                variable = vel_y
                this_variable = vel_y
                boundary = 'outlet'
                density = rho
                ux = vel_x
                uy = vel_y
                uz = vel_z
        [../]

        ### No Slip Conditions at the Walls ###
        # in x-direction
        [./vel_x_obj]
                type = PenaltyDirichletBC
                variable = vel_x
                boundary = 'top bottom object'
                value = 0.0
                penalty = 300
        [../]

        # in y-direction
        [./vel_y_obj]
                type = PenaltyDirichletBC
                variable = vel_y
                boundary = 'top bottom object'
                value = 0.0
                penalty = 300
        [../]
[]
```

Other Information

The simulation run file for the above example also includes a conservative 'tracer' to view how the simulated flow field carries a mass through the domain and around an obstruction. The file to run is found in 'user_examples' under the file name 'dg_incompressible_navier_stokes_implementation.i'. The compressible version is 'dg_compressible_navier_stokes_implementation.i'. When you run the incompressible file and simulate a flow field, you will get a flow field that looks something Figure 1 below (NOTE: The compressible example involves a linearly decreasing density in the x-direction, and so velocity increases to ensure conservation of fluid mass when this occurs). This image shows the relative magnitudes of the velocity vector, where blue indicates a velocity of 0 (i.e., no slip at the walls or the object) and the darker red indicates high velocities.

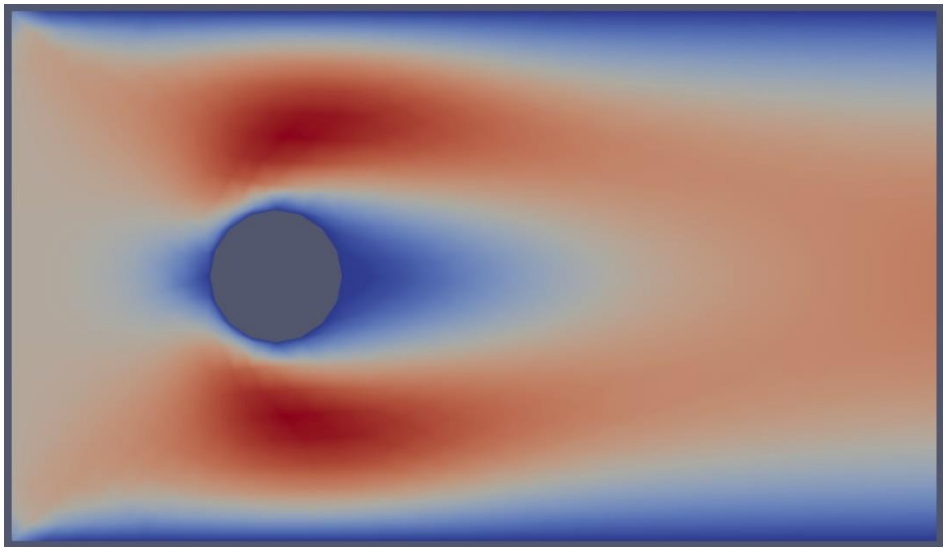

Figure 1: Incompressible flow field around an obstruction. Velocity at the inlet and exit are roughly the same because the flow is incompressible and the aperture for the inlet and outlet are the same size.

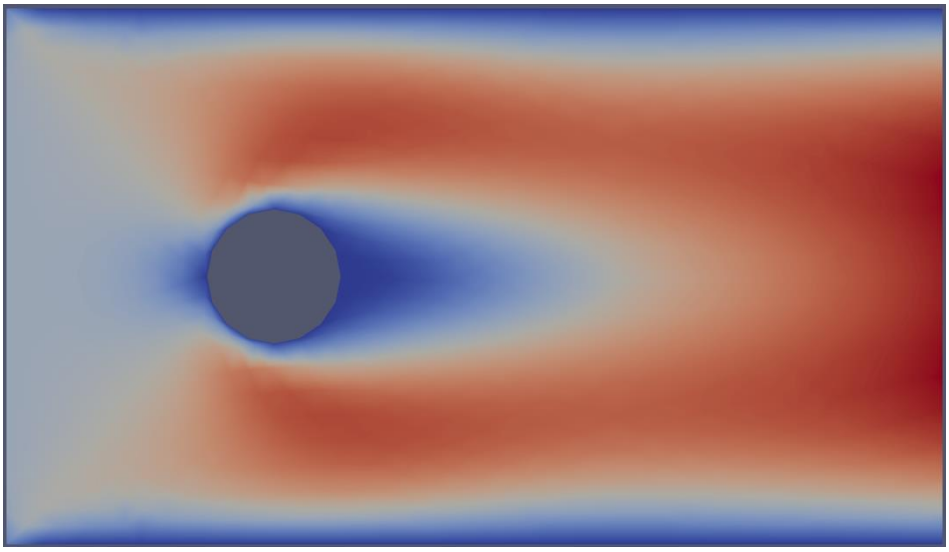

Figure 2: Compressible flow field around an obstruction. The density in this flow field varies linearly in the x-direction from 1 to 0.5 (left to right). As such, the velocities get higher towards the exit of the domain to maintain the total fluid mass flux.

Coupling Navier-Stokes, Mass, and Energy

The culmination of all prior examples would be to couple together all physics of flow fields, mass balances, and energy balances. To keep things simple, we will build upon the input file developed for the "Adding Gas Properties" example and simply add in the incompressible Navier-Stokes example for RZ coordinates. Note that the RZ formulation of the incompressible Navier-Stokes is not very computationally efficient, so in this input file many of the solver options where changed to gain greater efficiency and stability. The end result from this simulation is remarkably similar that that of the "Adding Gas Properties" example results. This is mainly because in a packed column the flow profiles are fairly uniform, so adding Navier-Stokes to this simulation did not add much more unique information. In

general, for packed columns in RZ coordinates, it is usually unnecessary to invoke Navier-Stokes kernels. Instead, you can produce a reasonable approximation by just using an average linear velocity based on the total volumetric flow rate and the cross-sectional void area of the column. Have a look at the sample input file for more details (https://github.com/aladshaw3/cats/tree/master/user_examples).

Simulations with Numerous Reactions

So far we have discussed how to setup relatively simple problems that involve only a handful of reactions. However, real chemical systems with be extraordinarily complex and might involve many, many reactions. While you can use everything we have discussed previously to solve these problems, it may quickly become cumbersome to redefine a reaction again and again when it shows up in multiple mass balances or equations.

To eliminate redundancy around this issue, you can define reactions and their associated rates as non-linear variable themselves, then couple those variables representing those rate functions into other non-linear variables using the kernels for WeightedCoupledSumFunction and ScaledWeightedCoupledSumFunction.

Example 1: Multi-Reaction Adsorption

Consider the following set of reactions:

(1) $S + A \leftrightarrow q_1$

(2) $q_1 + A \leftrightarrow q_2$

In this example, we want to define accumulation terms (or time derivatives) to solve for the adsorbed amounts of each surface species (q_1 and q_2). Mathematically, if we can define each reaction rate as:

$$r_1 = k_{f,1}*S*A - k_{r,1}*q_1 \quad \& \quad r_2 = k_{f,2}*q_1*A - k_{r,2}*q_2$$

Then, we can define each rate of accumulation of q_1 and q_2 as:

$$dq_1/dt = r_1 - r_2 \quad \& \quad dq_2/dt = r_2$$

While you could redefine these in terms of the variables for S, A, and each q_i variable, it is most convenient to use a summation of rate terms, i.e., define them by r_i.

To do this, we introduce variables for each standard components (A, S, q1, and q2) in the Variables block, but also define variables for r1 and r2. Then, in the Kernels block, we create residuals to reflect functions denoting the representation of those rates (as seen below). Once those rate variables are defined, we then can reference them in the WeightedCoupledSumFunction kernel and use that in conjunction with the TimeDerivative kernel to define the adsorption for q1 and q2 (as seen below).

```
[Kernels]
    #r1 residual contributions
    [./r1_val]
        type = Reaction
```

```
        variable = r1
[../]
[./r1_rate]
        type = ConstReaction
        variable = r1
        this_variable = r1
        forward_rate = 1
        reactants = 'S A'
        reactant_stoich = '1 1'
        reverse_rate = 0.1
        products = 'q1'
        product_stoich = '1'
[../]

#r2 residual contributions
[./r2_val]
        type = Reaction
        variable = r2
[../]
[./r2_rate]
        type = ConstReaction
        variable = r2
        this_variable = r2
        forward_rate = 2
        reactants = 'q1 A'
        reactant_stoich = '1 1'
        reverse_rate = 0.01
        products = 'q2'
        product_stoich = '1'
[../]

#q1 residual contributions:  dq1/dt = r1 - r2
[./q1_dot]
        type = TimeDerivative
        variable = q1
[../]
[./q1_rate_sum]
        type = WeightedCoupledSumFunction
        variable = q1
        coupled_list = 'r1 r2'
        weights = '1 -1'
[../]

#q2 residual contributions:  dq2/dt = r2
```

```
        [./q2_dot]
            type = TimeDerivative
            variable = q2
        [../]
        [./q2_rate_sum]
            type = WeightedCoupledSumFunction
            variable = q2
            coupled_list = 'r2'
            weights = '1'
        [../]

    [../]
```

Check out WeightedCoupledSumFunction for more details on this method.

Example 2: Multi-Reaction Adsorption with Bulk Mass Balance Coupling

In the previous example, we only demonstrated how to use reaction rate variables to evaluate some basic time derivative functions. However, the process will likely need to be coupled with the bulk phase variable 'A' since as these reactions occur they will pull 'A' from the bulk phase. While you can use the WeightedCoupledSumFunction as we did for 'q', you can use ScaledWeightedCoupledSumFunction instead, which gives you a bit more flexibility for cross-coupling between phases by introducing a scaling factor that is often used as a unit conversion from solid to bulk concentrations.

As a demonstration, let's consider that the disappearance of 'A' from the bulk phase due to the reactions from the prior example would take the following functional form:

$$\varepsilon_b \frac{\partial A}{\partial t} = (1-\varepsilon_b)\sum_{\forall j} w_j r_j \quad ==> \quad \varepsilon_b \frac{\partial A}{\partial t} = (1-\varepsilon_b)[-r_1 - r_2]$$

The weights (w_j) for each reaction would be negative for 'A' because 'A' is being consumed in each reaction. The 'scaling factor' for this example would be represented by ($1 - \varepsilon_b$), which could be given as a constant or another non-linear variable. In this case, ε_b represents the pore-to-volume ratio and ($1 - \varepsilon_b$) would represent the solids-to-volume ratio for the system.

Setting up this problem to solve would involve first creating the variables and kernels for r1 and r2 (see previous example), then creating the kernels for the bulk concentration of A using the VariableCoefTimeDerivative and ScaledWeightedCoupledSumFunction kernels (see below).

```
    [Kernels]
        #A residual contributions:  0.25*dA/dt = 0.75*(-r1 - r2)
        [./A_dot]
            type = VariableCoefTimeDerivative
            variable = A
            coupled_coef = 0.25
        [../]
```

```
        [./A_rate_sum]
            type = ScaledWeightedCoupledSumFunction
            variable = A
            coupled_list = 'r1 r2'
            weights = '-1 -1'
            scale = 0.75
        [../]

    [../]
```

Checkout ScaledWeightedCoupledSumFunction for more details on this method.

Simulations with Subdomains

All of the examples above were created on meshes that were not divisible into subdomains. This is common for packed columns, since there are far too many particles to individually mesh within the global domain. However, for more structured domains such as a monolith catalyst, we may want to actually mesh the open channels of the monolith separate from the porous-washcoat materials. In this case, we will need to construct a mesh with a 3rd party meshing software first or using some mesh breaking functions in MOOSE. For simplicity, these examples here will use meshes created in Gmsh (http://gmsh.info/), an open source meshing program. In addition, we will only develop these examples for a mass balance, however, these instructions will transfer over to energy balances and only the specific kernels you invoke will be different.

In this example, we will investigate simulating the transfer of mass through a single monolith channel and the mass transfer and adsorption that occurs at the boundaries of the washcoat and in the washcoat itself. The formulation of the mass balance follows the Separated Phase Mass Balances equation derivations. For the concentration of mass in the channel (C), we track that concentration based on a standard advection-diffusion equation. Mass transfer (MT) into the washcoat is a function applied only at the physical interface that exists between the channel-space variable (C) and the washcoat-space variable (C_w). This function essentially replaces what was the FilmMassTransfer kernel from the first mass balance example. Once mass transfers across that boundary, it can then travel through the washcoat via diffusion and can undergo adsorption (q). In this example, the units of adsorption are in moles adsorbed per volume of washcoat, thus there is no additional unit conversion term needed on the time derivative. The adsorption process itself will then follow the same Langmuir like reaction we specified in the first example. For your particular problem, the units may vary and thus you would include some conversion factors.

C (in mol/m^3): $\frac{\partial C}{\partial t} + \nabla \cdot (\mathbf{v} \cdot C) = \nabla \cdot (D \cdot \nabla C)$

MT @ interface: $k(C - C_w)$

C_w (in mol/m^3): $\varepsilon_w \frac{\partial C_w}{\partial t} = \nabla \cdot (\varepsilon_w D_w \cdot \nabla C_w) - \frac{dq}{dt}$

q (in mol/m^3): $\frac{dq}{dt} = k_f C_p S - k_r q$

S (in mol/m^3): $S = S_{max} - q$

The kernels we need to invoke for each piece of physics are as shown below. You can click on the links to get additional details on what each kernel is doing and what the input arguments are.

VariableCoefTimeDerivative: $\varepsilon_w \frac{\partial C_w}{\partial t}$

TimeDerivative: $\frac{dq}{dt}$ and $\frac{\partial C}{\partial t}$

CoupledCoeffTimeDerivative: $-\frac{dq}{dt}$

GConcentrationAdvection (DGConcentrationAdvection): $\nabla \cdot (\mathbf{v} \cdot C)$

GVariableDiffusion (DGVariableDiffusion): $\nabla \cdot (D \cdot \nabla C)$

GVarPoreDiffusion (DGVarPoreDiffusion): $\nabla \cdot (\varepsilon_w D_w \cdot \nabla C_w)$

ConstReaction: $k_f C_p S - k_r q$

MaterialBalance: $S = S_{max} - q$

Just like before, we need to include boundary conditions for the mobile phases. For the channel space concentration (C), the boundary conditions are essentially the same as they were before, but without the porosity term. The boundary condition for the washcoat concentration (C_w) actually comes from the interface kernel to facilitate transfer from the channel to the washcoat. Note that whenever an interface kernel is invoked, it only needs to be invoked once and will be automatically applied to both domains that the interface acts on.

DGConcentrationFluxBC: $(\mathbf{v} \cdot \mathbf{n}) \cdot C_{inlet}$ and $(\mathbf{v} \cdot \mathbf{n}) \cdot C$

InterfaceMassTransfer: $k(C - C_w)$

The mesh we use for this simulation is a 2D representation of a single monolith channel that has a gap in the middle between each layer of washcoat. This gap will help us simulate what would occur if the washcoat were not applied in a single, continuous layer of even thickness. The length of the channel is 5 cm and the total channel diameter is 0.127 cm with a washcoat layer thickness of about 0.027 cm. For your reference, both the .geo script file and .msh file for the mesh are provided (2DChannel.geo and 2DChannel.msh) in (https://github.com/aladshaw3/cats/tree/master/user_examples). We direct MOOSE/CATS to read in this object in the Mesh block as follows. Note that this mesh file identifies each subdomain as 'channel' and 'washcoat' and also identifies the boundaries as 'inlet', 'outlet', and 'inner_walls' for the interface between the washcoat and channel.

```
[Mesh]
    [./mesh_file]
        type = FileMeshGenerator
        file = 2DChannel.msh
    [../]
[]
```

Because our simulation involves subdomains, each variable and kernel must not identify the 'block' (i.e., subdomain) that that variable or kernel acts on. We will follow the same variable declarations from the first example, but will now include a 'block' designation for each variable as follows.

```
[Variables]
    [./C]
        order = FIRST
        family = MONOMIAL
        initial_condition = 0
        block = 'channel'
    [../]
    [./Cw]
        order = FIRST
        family = MONOMIAL
        initial_condition = 0
        block = 'washcoat'
    [../]
    [./q]
        order = FIRST
        family = MONOMIAL
        initial_condition = 0
        block = 'washcoat'
    [../]
    [./S]
        order = FIRST
        family = MONOMIAL
        initial_condition = 0
        block = 'washcoat'
    [../]
[]
```

As with the variables block, any auxiliary variables we want to use as parameters must also declare what 'block' (i.e., subdomain) those auxiliary variables apply to, and the same goes for any auxiliary kernels that calculate the values for those auxiliary variables. In this example, we will use auxiliary variables to just hold parametric information and will not calculate those value as was done in the "Adding Gas Properties" example.

When invoking the physics/chemistry kernels for our variables, we again must include that 'block' (i.e., subdomain) designation for each kernel. Below, we show an example of the input syntax for doing so. For the full input file, see (https://github.com/aladshaw3/cats/tree/master/user_examples).

```
#Calling the material balance in the washcoat for S
```

```
[./mat_bal]
        type = MaterialBalance
        variable = S
        this_variable = S
        coupled_list = 'S q'
        weights = '1 1'
        total_material = S_max
        block = 'washcoat'
[../]
```

The boundary conditions and interface kernels do not need the 'block' argument set. This is because the 'boundary' argument is sufficient for instructing MOOSE when and where these kernels apply to the full domain. Interface kernels only need to be invoked once for each boundary/physics they apply to and they will automatically apply residuals for both variables that get coupled. Below shows an example of how to invoke the interface kernel for this simulation. For the full input file, see the example file in (https://github.com/aladshaw3/cats/tree/master/user_examples).

```
[InterfaceKernels]
        type = InterfaceMassTransfer
        variable = C
        neighbor_var = Cw
        boundary = 'inner_walls'
        transfer_rate = 2
[]
```

NOTE: In this simulation example, the tolerances for the solver option had to be significantly tightened and we changed to a direct linear solver (i.e., 'solve_type = newton'). This was done to improve convergence for a very fine mesh with very small elements. The mesh used here was done in units of meters, but given the size of the channel, it may be more appropriate to use centimeters. If you choose to change unit basis on the mesh, make sure all your parameters align with the new units.

Simulations with Hybrid FD/FE Microscale Diffusion

CATS has developed within it a hybrid Finite-Difference/Finite-Element method for simulating intraparticle or intralayer diffusion of a microscale and couple those physics fully with the macroscale physics. The advantage of using this approach over using sub-domains is to simplify the mesh in the model and reduce problem dimensionality. Doing so will lead to a reduction in computation resources needed to resolve the full problem, without making significant concessions in model applicability. To see details on the microscale diffusion kernels developed in CATS, check out all kernels prefixed with 'Microscale' starting from here.

Packed-bed Adsorption

The hybrid FD/FE method was originally developed for resolving the microscale diffusion physics for columns packed with particles. This is because a domain such as that cannot actually use a sub-domain approach to resolve the microscale (at least not in any efficient way). Thus, if we discretize the microscale using finite differences, then formulate a "weak form" of that discretization on the macroscale grid, we can then still solve for intraparticle diffusion in a column packed with thousands or millions of particles.

This example seeks to simulate the adsorption and desorption behavior of water vapor on spherical zeolite pellets. The equations we will solve in this example are as follows:

Macroscale: $\varepsilon_b \frac{\partial C_{H2O}}{\partial t} + \nabla \cdot (\varepsilon_b \mathbf{v} \cdot C_{H2O}) = \nabla \cdot (\varepsilon_b D \cdot \nabla C_{H2O}) - (1-\varepsilon_b) \cdot G_a \cdot k_m \left(C_{H2O} - C_{H2O,p}\right)$

Microscale: $r^2 \varepsilon_p (1-\varepsilon_b) \frac{\partial C_{H2O,p}}{\partial t} = \frac{\partial}{\partial r}\left(r^2 \varepsilon_p (1-\varepsilon_b) D_p \frac{\partial C_{H2O,p}}{\partial r}\right) + r^2 (1-\varepsilon_b) \cdot (-1) \cdot r_{H2O}$

Adsorption Reaction: $H_2O + S \leftrightarrow q$ $\quad r_{H2O} = k_f C_{H2O,p} S - k_r q$

Microscale Mass Transfer BC: $\varepsilon_p (1-\varepsilon_b) D_p \frac{\partial C_{H2O,p}}{\partial r} = k_m \left(C_{H2O} - C_{H2O,p}\right)$

Microscale Interior BC: $\frac{\partial C_{H2O,p}}{\partial r} = 0$

Surface ODEs: $\frac{dS}{dt} = -r_{H2O}$ $\quad \frac{dq}{dt} = r_{H2O}$

Both the macroscale mass balance and microscale mass balance are on a per total volume basis, that is what all the scale factors with the various ε_i values are doing. The reaction is introduced via a non-linear variable (r_{H2O}), which can be applied to the microscale balance and equations for available sites (S) and adsorbed water (q). This makes the coupling simpler from the user's perspective. All of the parameters are for this example will be automatically calculated using the SimpleGasProperties interface through as series of auxiliary kernels. Temperature will be assumed isothermal for simplicity, but we include a linear temperature ramp to create a Temperature Programmed Desorption (TPD) curve towards the end of the simulation.

The kernels we will be invoking are as follows:

VariableCoefTimeDerivative: $\varepsilon_b \frac{\partial C_{H2O}}{\partial t}$

GPoreConcAdvection (DGPoreConcAdvection): $\nabla \cdot (\varepsilon_b \mathbf{v} \cdot C_{H2O})$

GVarPoreDiffusion (DGVarPoreDiffusion): $\nabla \cdot (\varepsilon_b D \cdot \nabla C_{H2O})$

FilmMassTransfer: $(1-\varepsilon_b) \cdot G_a \cdot k_m \left(C_{H2O} - C_{H2O,p}\right)$

MicroscaleVariableCoefTimeDerivative: $r^2 \varepsilon_p (1-\varepsilon_b) \frac{\partial C_{H2O,p}}{\partial t}$

MicroscaleVariableDiffusion: $\frac{\partial}{\partial r}\left(r^2 \varepsilon_p (1-\varepsilon_b) D_p \frac{\partial C_{H2O,p}}{\partial r}\right)$

MicroscaleScaledWeightedCoupledSumFunction: $r^2(1-\varepsilon_b)\cdot(-1)\cdot r_{H2O}$

ArrheniusEquilibriumReaction (with Reaction): $r_{H2O} = k_f C_{H2O,p} S - k_r q$

MicroscaleVariableDiffusionOuterBC: $\varepsilon_p(1-\varepsilon_b)D_p \frac{\partial C_{H2O,p}}{\partial r} = k_m\left(C_{H2O} - C_{H2O,p}\right)$

MicroscaleVariableDiffusionInnerBC: $\frac{\partial C_{H2O,p}}{\partial r} = 0$

WeightedCoupledSumFunction (with TimeDerivative): $\frac{dS}{dt} = -r_{H2O}$ $\frac{dq}{dt} = r_{H2O}$

In order to resolve the microscale domain, for this example we will divide the microscale into 5 nodes (or 5 discrete interior pellet volumes). We will denote those nodes as (0, 1, 2, 3, 4), where 0 represents the inner most position (i.e., center of the particle) and 4 represents the outer most position (i.e., where particle is in contact with bulk gases). When we do this, we must also create variable IDs for each interior variable at each position. Thus, instead of adding a variable for interior pellet concentration as 'H2Op', we create a series of interior concentration variables as 'H2Op_0', 'H2Op_1', 'H2Op_2', 'H2Op_3' and 'H2Op_4'. The same is applied for each reaction variable (r_0 through r_4) and each surface species (S_0 through S_4 and q_0 through q_4).

Kernels for the microscale physics must be repeated for each nodal position. The only difference will be at the BC nodes of the microscale, which replace the MicroscaleVariableDiffusion kernel with the MicroscaleVariableDiffusionOuterBC kernel (for node 4) and the MicroscaleVariableDiffusionInnerBC kernel (for node 0). This process is somewhat tedious, but is necessary to create the full "weak form" of the microscale on the macroscale mesh domain. To see details on the microscale diffusion kernels developed in CATS, check out all kernels prefixed with 'Microscale' starting from here.

The parameters for each coefficient in this model will be automatically calculated using the following auxiliary kernels:

VoidsVolumeFraction: ε_b

GasVelocityCylindricalReactor: $\mathbf{v}$ ➔ here, we are only calculating the y-component of velocity to get an average linear velocity in the packed column.

SphericalAreaVolumeRatio: G_a

SolidsVolumeFraction: $(1-\varepsilon_b)$

MicroscalePoreVolumePerTotalVolume: $\varepsilon_p(1-\varepsilon_b)$

SimpleGasSphericalMassTransCoef: k_m

SimpleGasEffectivePoreDiffusivity: $\varepsilon_p(1-\varepsilon_b)D_p$

SimpleGasDispersion: D

Additionally, there are a number of other auxiliary kernels we are using:

LinearChangeInTime: To automatically raise temperature linearly as a function of time, give some starting time, ending time, and temperature target. This is used to emulate a TPD.

MicroscaleIntegralAvg: To automatically calculate average interior concentrations of $C_{H2O,p}$, S, and q. This will integrate over the values of the individual nodal variables in the microscale domain and produce average amounts of adsorption, free sites, or other variables in that space of interest.

A full run down of the input file to run these simulations, as well as some develop notes and comments, can be found under the 'user_examples' folder filename 'simulations_with_spherical_microscale_diffusion.i' (https://github.com/aladshaw3/cats/blob/master/user_examples/simulations_with_spherical_microscale_diffusion.i).

Packed-bed Adsorption (same problem, without Microscale)

This example shows how to setup a simulation to model the same problem as the example above, but not include the microscale physics inside the particles. The purpose of this example is to show how much of a difference may be when you include microscale diffusion or not. In this particular case, the inclusion of microscale diffusion does not have a significant impact. Whether or not it is needed depends on how small the diffusion coefficient is relative to the particle sizes.

Even if we do not explicitly discretize the microscale, we can still have variables represent what is happening in the microscale, we just won't be able to model diffusion through that microscale. Instead, all microscale variables will be representative of the average levels found in that subdomain. Thus, our mathematical description will be as follows:

Macroscale: $$\varepsilon_b \frac{\partial C_{H2O}}{\partial t} + \nabla \cdot (\varepsilon_b \mathbf{v} \cdot C_{H2O}) = \nabla \cdot (\varepsilon_b D \cdot \nabla C_{H2O}) - (1-\varepsilon_b) \cdot G_a \cdot k_m \left(C_{H2O} - C_{H2O,p}\right)$$

Microscale: $$\varepsilon_p (1-\varepsilon_b) \frac{\partial C_{H2O,p}}{\partial t} = (1-\varepsilon_b) \cdot G_a \cdot k_m \left(C_{H2O} - C_{H2O,p}\right) + (1-\varepsilon_b) \cdot (-1) \cdot r_{H2O}$$

Adsorption Reaction: $H_2O + S \leftrightarrow q$ $\quad r_{H2O} = k_f C_{H2O,p} S - k_r q$

Surface ODEs: $\frac{dS}{dt} = -r_{H2O}$ $\quad \frac{dq}{dt} = r_{H2O}$

As you can see, the only difference is that the intraparticle diffusion (as associated outer boundary condition) is essentially just replaced with the mass-transfer statement (same statement from the macroscale balance). Since we have already gone through all the kernels in the previous example, we will not do a detailed run down here. A full run down of the input file to run these simulations, as well as some develop notes and comments, can be found under the 'user_examples' folder filename 'equivalent_spherical_particle_adsorption_without_microscale.i ' (https://github.com/aladshaw3/cats/blob/master/user_examples/equivalent_spherical_particle_adsorption_without_microscale.i).

Monolith CO Light-off

In addition to using the hybrid FD/FE method for resolving microscale diffusion in particles, you can use this method to also simulate the microscale diffusion inside of monolith walls/washcoating. This can be accomplished in the same way it was done in the example above. However, in a monolith, we have diffusion through a wall and there may be many walls within a given channel that we diffuse through. For each wall you have, you will need to identify new microscale variables for that wall. For instance, if you have square channels, then you would need a variable that represents the concentration of a species in each wall at a given nodal position (e.g., variable C_w0_n4 could be a designation for concentration in wall 0 at nodal position 4 of the microscale). See next example for details.

This would be even more tedious than the prior example, but is a way to also explore the possibility of different coatings on different walls in a given channel. Instead of showing an example of this, we will show a simplified example of microscale diffusion in washcoated monoliths using a calculation of an 'effective total monolith thickness' for the microscale. More information on how this thickness is calculated can be found here (MonolithMicroscaleTotalThickness). Note that using this thickness will only yield an approximate intralayer diffusion effect. It is more accurate to use a true thickness and a set of microscale physics for each wall. This can be done with this same modeling paradigm simply by expanding the microscale variables to include wall IDs and node IDs, duplicating the microscale kernels for each wall, and providing true wall/washcoat thickness.

The physics for this problem are represented mathematically as follows:

Macroscale: $\varepsilon_b \frac{\partial C_i}{\partial t} + \nabla \cdot (\varepsilon_b \mathbf{v} \cdot C_i) = \nabla \cdot (\varepsilon_b D \cdot \nabla C_i) - (1-\varepsilon_b) \cdot G_a \cdot k_m \left(C_i - C_{w,i}\right)$

Microscale: $\varepsilon_w (1-\varepsilon_b) \frac{\partial C_{w,i}}{\partial t} = \frac{\partial}{\partial r}\left(\varepsilon_w (1-\varepsilon_b) D_p \frac{\partial C_{w,i}}{\partial r}\right) + (u_i) \cdot r_{lightoff}$

Lightoff Reaction: CO + 0.5 O2 → CO2 $\quad r_{lightoff} = k_f C_{w,CO} C_{w,O2}$

Microscale Mass Transfer BC: $\varepsilon_w (1-\varepsilon_b) D_p \frac{\partial C_{w,i}}{\partial r} = k_m \left(C_i - C_{w,i}\right)$

Microscale Interior BC: $\frac{\partial C_{w,i}}{\partial r} = 0$

In this formulation, each index i represents either CO, O_2, or CO_2. The parameter u_i is the molar impact that the lightoff reaction would have on a given gas species. This value is -1 for i = CO, -0.5 for i = O_2, and +1 for i = CO_2.

This example will use all the same kernels from the previous example, so we will not repeat that information here. Just like before, all the parameters are calculated using the SimpleGasProperties auxiliary system. Most of those calculations are the same as the prior example, so we will just summarize the ones below that are different.

MonolithAreaVolumeRatio: G_a

MonolithHydraulicDiameter: d_h → not directly used in the above kernels, but is used in the calculation of other auxiliary properties.

SimpleGasMonolithMassTransCoef: k_m

MonolithMicroscaleTotalThickness: This for the calculation of the 'effective total thickness' for the microscale dimension of the monolith. This should be calculated ahead of time and its value should be used as the 'micro_length' parameter for the microscale kernels.

Additionally, in this example we use the auxiliary system to setup the lightoff temperatures based on a set of data (rather than simulating those temperatures for simplicity). We can do this by making use of the 'PiecewiseMultilinear' function built into MOOSE (see https://mooseframework.inl.gov/source/functions/PiecewiseMultilinear.html). Combined with the auxiliary kernel 'FunctionAux' these features will take a data file with time-position temperature information and setup a simulation that contains those exact temperature gradients (with some linear interpolations for time-position data that are missing).

A full run down of the input file to run these simulations, as well as some develop notes and comments, can be found under the 'user_examples' folder filename 'simulations_with_monolith_microscale_diffusion.i' (https://github.com/aladshaw3/cats/blob/master/user_examples/simulations_with_monolith_microscale_diffusion.i).

Monolith CO Light-off with True Wall Thickness

This example does the same calculations from above, but separates out the diffusion into the individual walls of the monolith channel (denoted with *_w0, *_w1, *_w2, and *_w3 for each wall). Since this example is solving the same problem, we will not go through everything here. Instead, a full run down of the input file to run these simulations, as well as some develop notes and comments, can be found under the 'user_examples' folder filename 'simulations_with_monolith_microscale_diffusion_multiwalls.i' (https://github.com/aladshaw3/cats/blob/master/user_examples/simulations_with_monolith_microscale_diffusion_multiwalls.i).

Doing Everything: Fully Coupled Mass-Energy with Microscale Physics

In this final example, we use the hybrid Finite-Difference/Finite-Element methods for simulating intraparticle diffusion in packed columns and build upon all prior examples to simulate a more realistic and relevant simulation case.

For this example, we will simulate the coupling of mass and energy in a packed cylindrical column with particles containing 'coke' (i.e., unburned carbon). Then, we introduce oxygen, in a nitrogen carrier gas, into the column to oxidize the carbon inside of the particles creating CO_2 and a significant increase in system heat due to the exothermic nature of the reaction. The mass and energy balances for the macroscale are as follows:

O_2: $$\varepsilon \frac{\partial C_{O2}}{\partial t} + \nabla \cdot (\varepsilon \mathbf{v} \cdot C_{O2}) = \nabla \cdot \left(\varepsilon D_{e,O2} \cdot \nabla C_{O2}\right) - (1-\varepsilon) \cdot k_{O2} \cdot A_o \left(C_{O2} - C_{p,O2}\right)$$

CO_2: $$\varepsilon \frac{\partial C_{CO2}}{\partial t} + \nabla \cdot (\varepsilon \mathbf{v} \cdot C_{CO2}) = \nabla \cdot \left(\varepsilon D_{e,CO2} \cdot \nabla C_{CO2}\right) - (1-\varepsilon) \cdot k_{CO2} \cdot A_o \left(C_{CO2} - C_{p,CO2}\right)$$

E_f: $$\varepsilon \frac{\partial E_f}{\partial t} + \nabla \cdot \left(\varepsilon \mathbf{v} \cdot E_f\right) = \nabla \cdot \left(\varepsilon K_f \cdot \nabla T_f\right) - (1-\varepsilon) \cdot h_s \cdot A_o \left(T_f - T_s\right)$$

E_s: $$(1-\varepsilon)\frac{\partial E_s}{\partial t} = \nabla \cdot \left((1-\varepsilon) \cdot K_s \cdot \nabla T_s\right) - (1-\varepsilon) \cdot h_s \cdot A_o \left(T_s - T_f\right)$$
$$+(1-\varepsilon) \cdot \sum_{\forall j} \left(-\Delta H_j\right) A_{s,j} r_{s,j}$$

T_f: $$E_f = \rho_f c_{pf} T_f$$

T_s: $$E_s = \rho_p c_{ps} T_s$$

Since nitrogen gas is assumed 'inert' as the carrier gas, we do not include physics for this species and instead just treat it as an auxiliary variable held to a constant value. All the parameters for this model are resolved using the auxiliary system. Since our focus here is on the microscale physics, we do not discuss all aspects of the variable, kernel, and auxiliary blocks of the input file. Check out the full input file at (https://github.com/aladshaw3/cats/tree/master/user_examples) for additional details.

Many of the above equations should be familiar right now, but let's take a moment to discuss some of the key unique features. Each energy balance is done in terms of energy per total volume, which is why each has a different unit conversion factor associated with it to convert energy per phase volume to energy per total volume. The energy balance for the solids (E_s) involves a summation over a set of surface reactions ($r_{s,j}$) that take place at specific locations (j) within the particles. Since this is a surface based reaction, the factor $A_{s,j}$ represents the approximate reactive surface area at location j inside the particles. Thus, the sum of all representative areas $A_{s,j}$ would yield the total reactive surface area of the particles.

The reaction rate ($r_{s,j}$) then represents the actual carbon oxidation reaction that occurs at the j location inside of a particle. We would represent this reaction as shown below:

$r_{s,j}$: $C + O_2 \rightarrow CO_2$ $$r_{s,j} = k_f q_{c,j} C_{p,O2,j}$$

Where $q_{c,j}$ represents the surface concentration of active carbon at the j location in the particle and $C_{p,O2,j}$ represents the pore-space concentration of oxygen at the j location in the particle. Note that here we represent this reaction as reversible, so the pore-space concentration of carbon-dioxide ($C_{p,CO2,j}$) does not show up in the rate expression. However, the formation of carbon-dioxide will be a function of the above rate expression.

To resolve the intraparticle diffusion of oxygen and carbon-dioxide, we need to introduce 3 more mass balances that all act on the microscale domain of the individual particles. Those base equations would be as follows:

$C_{p,O2}$: $$\varepsilon_p \frac{\partial C_{p,O2}}{\partial t} = \nabla \cdot \left(\varepsilon_p D_{p,O2} \cdot \nabla C_{p,O2}\right) - A_s r_s$$

$C_{p,CO2}$: $$\varepsilon_p \frac{\partial C_{p,CO2}}{\partial t} = \nabla \cdot \left(\varepsilon_p D_{p,CO2} \cdot \nabla C_{p,CO2}\right) + A_s r_s$$

q_c: $$\frac{dq_c}{dt} = -r_s$$

These equations are written in their respective "strong forms" and as such they are not currently discretized into the j intraparticle positions, as we showed for the energy balance. The r_s term is the same rate expression that we showed earlier, but without the j subscripts. For details on how these equations get transformed and semi-discretized into the "weak forms", take a look at the 'Microscale' kernels starting here.

In addition to the above expressions, the intraparticle mobile phase also requires boundary conditions. These boundary conditions are how we couple together the macro- and micro-scales of the simulation. Currently, CATS only offers a Cauchy type boundary condition for this coupling. That coupled expression, in its strong form, would be as follows:

$C_{p,O2}$: $\varepsilon_p D_{p,O2} \frac{\partial C_{p,O2}}{\partial r} = k_{O2}\left(C_{O2} - C_{p,O2}\right)$

$C_{p,CO2}$: $\varepsilon_p D_{p,CO2} \frac{\partial C_{p,CO2}}{\partial r} = k_{CO2}\left(C_{CO2} - C_{p,CO2}\right)$

The interior boundary condition enforces a "no flux" or "r-symmetric" condition for the interior most location of the particles (i.e., at the 0 radius of the particles). More information on this type of boundary condition can be found here.

For this example, the microscale will be discretized into 10 j locations. Each j location needs its own variable for q_c, $C_{p,O2}$, and $C_{p,CO2}$. Typically, we will identify each j location variable by suffixing a variable name with an id number. For instance, in this example the variable named 'qc0' represents the surface carbon concentration at the 0^{th} location in the particles, where the 0^{th} location is at a radius of 0 (i.e., the center of the particle). Then, the outer most location of the particle will be denoted as 'qc9' since there are only 10 locations (nodes 0 through 9) we divide the microscale space into. Setting up the input file for these types of simulations is very tedious and long, so we will not write down all the details here. Additional instructions on how to structure input files for microscale simulations can be found in the 'Microscale' kernels section starting here. Additionally, the input file containing all these kernels and expressions is available at (https://github.com/aladshaw3/cats/tree/master/user_examples).

Lastly, for convenience, it will be advantageous to introduce auxiliary variables to represent the average pore-space concentrations at the microscale. Since we have divided up the microscale into 3 sets of 10 separate variables, it will be difficult to interpret exactly how much material is inside of the particles and/or the extent of the reactions inside the particles. To facilitate the need to interpret this information, provided in CATS are auxiliary kernels for 'MicroscaleIntegralAvg' and 'MicroscaleIntegralTotal'. These kernels perform an integration over the microscale domain to average or total the amount of material inside of the particles. The integration uses a trapezoid rule for accuracy, but these integrals will not be exact. As a consequence, it will not be uncommon to see some small mass errors (0.5 - 2 %) if you are looking to account for all mass in the system.

As a final note, we did not go over line-by-line each argument in the input file for this example due to how large this case is (the input file requires over 2000 lines of arguments). Instead, you should use the prior, simpler examples to build from first. After you become more familiar with the input file system and how to call specific kernels for specific purposes, then understanding this input file and simulation case should be a bit easier. Be sure to read through the 'Microscale' kernels (here) if you get lost on why certain input arguments are being invoked and check out the full input file for this simulation case at (https://github.com/aladshaw3/cats/tree/master/user_examples). Throughout that

input file are multiple developer comments which discuss what the kernels and input arguments are doing and why they are there.

Electrochemical Cell Example: Vanadium Flow Battery

This example follows the work from Shah et al.[1] and Clausen et al.[2] for simulating a charging and discharging cycle for a vanadium based flow battery. In this simulation, we have a positive and negative electrode separated by a membrane. The concentrations of the electrolyte solutions are constantly recycled from the outlet to the inlet at a rate defined by the volumetric flow rates and reservoir volumes for each electrode. The simulation domain is meshed into 5 regions (Figure 3) that are as follows (from left to right): (i) negative current collector, (ii) negative electrode, (iii) membrane, (iv) positive electrode, and (v) positive current collector.

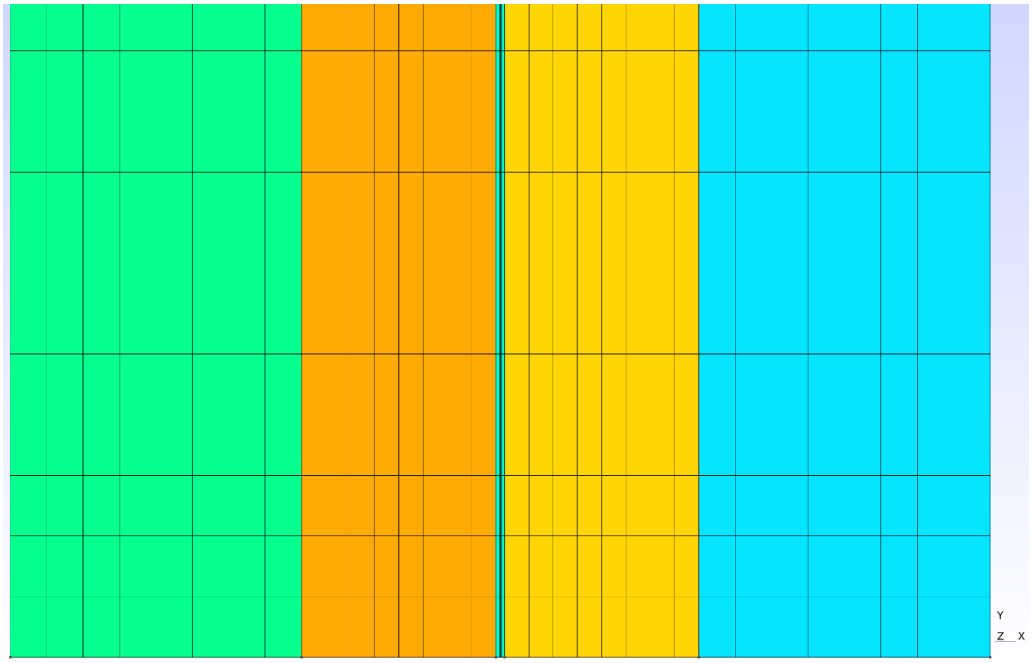

Figure 3: Meshed domain for vanadium flow battery

The negative electrode contains ions for V(II) and V(III) (as well as H^+, H2O, and HSO_4^-). The positive electrode contains ions for V(IV) and V(V) as VO^{2+} and VO_2^+, respectively (as well as H^+, H2O, and HSO_4^-). The redox reactions are as follows:

$$\text{Negative electrode:} \quad V(II) \leftrightarrows V(III) + e^-$$

$$\text{Positive electrode:} \quad VO^{2+} + H_2O \leftrightarrows VO_2^+ + 2\,H^+ + e^-$$

NOTE: Although the reaction in the positive electrode includes H_2O and H^+ species, the rates of the reactions are zero order with respect to those species, as defined in references [1] and [2].

The example file 'vanadium_flow_battery.i' under 'user_examples' of the repository contains the input file for simulating the 1.08 M total vanadium case described in references [1] and [2]. All parameters used for that simulation case are defined in those 2 references and will not be expanded upon here further. Figure 4 below shows the validation of the model by comparing the simulated results against literature.

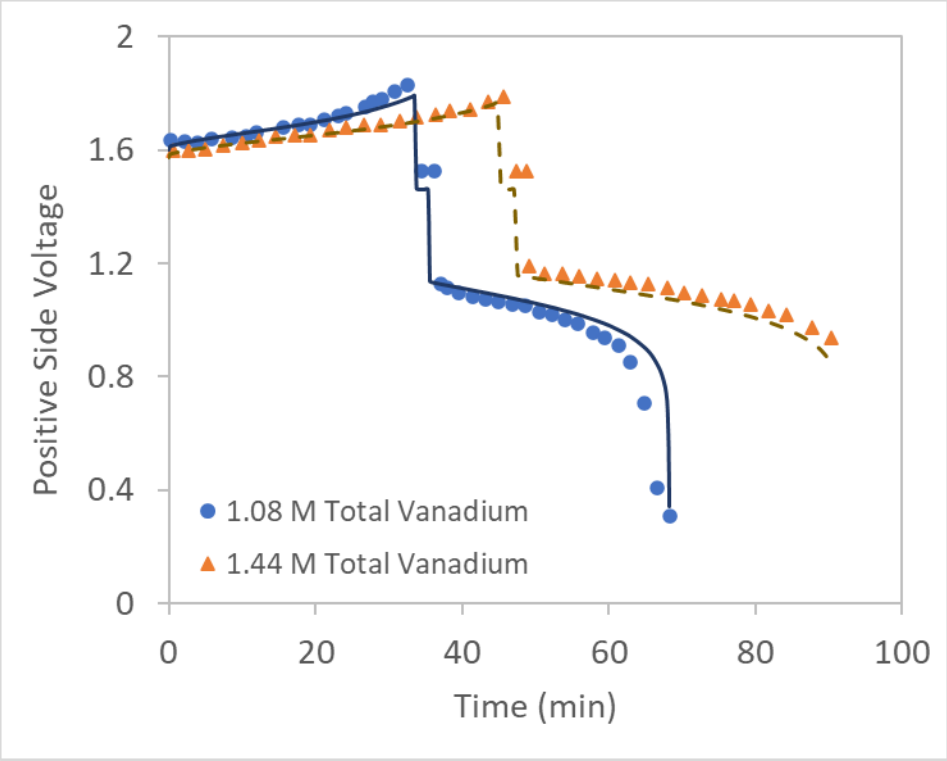

Figure 4: Simulated results vs literature using the electrochemical kernels in CATS

All of the kernels and physics for this system were discussed generically in Electrochemical Catalysis. Flow profiles in the system were resolved using Darcy's law (in porous media and in membrane). For a full view of all kernels used, please review the example file named 'vanadium_flow_battery.i'. You can use the names of the kernels invoked from within that file to search this document for more specifics on the implementation of each kernel used.

Link to example file: vanadium_flow_battery.i

References

[1] A.A. Shah, M.J. Watt-Smith, F.C. Walsh, Electrochimica Acta, 53 (2008) 8087-8100.

[2] J.R. Clausen, V.E. Brunini, H.K. Moffat, M.J. Martinez, Sandia Report, SAND2014-0190.

Running the Code

CATS requires the MOOSE framework to function. Before attempting to run simulations in CATS, you must download and install MOOSE. Follow the instructions on the MOOSE website to get started (https://mooseframework.org/). After you have installed MOOSE, you can use the 'git' commands to download and install the CATS module. The commands below need to be run in the same directory that the 'moose' folder is located in.

```
git clone https://github.com/aladshaw3/cats.git

cd cats/

make -j4
```

The 'make -j4' command will instruct your computer to build the CATS executable from the source files using 4 cores. After these instructions complete, and assuming no errors occurred, you should now see an executable called 'cats-opt' in the 'cats/' directory. This is the program you will invoke to run simulations. However, before get started creating and running simulations, you should test the CATS build on your machine to ensure everything is working as intended. The next section discusses how to run the unit tests.

Unit Tests

Unit tests are small scale simulations run on individual kernels or sets of kernels. Each test case has a 'gold' set of output files that will be used for comparison. In general, after you install the CATS executable, you should always run the unit tests and check for any errors. To run the unit tests, use the following command from within the 'cats/' directory.

```
./run_tests -j4
```

If there are no errors, then all tests will pass and show a green 'OK' or 'PASSED' message. Note that some tests may be skipped depending on specific test or computer conditions. This is normal. When the tests fail, there are 2 modes of failure: (i) yellow 'EXODIFF' failure and (ii) red 'CRASH' failure. The 'EXODIFF' message occurs when the test ran to completion, but the end results of the simulation were not within a specified level of tolerance when compared to the 'gold' solution. This can sometimes occur if your PETSc version is different than the version that was originally used to create the test. In general, if these errors occur, you either need to update the 'gold' files or update MOOSE/PETSc.

If you get a red 'CRASH' error, or something similar, then this is a situation in which the code could not run at all. When this happens, it could mean (i) you are missing MOOSE libraries, (ii) you are missing PETSc or libmesh libraries, and/or (iii) there is an underlying problem in some CATS kernels (this would be rare if you are on the 'master' branch. In general, should you encounter this problem, you should record your console output detailing all the error messages and report that to a CATS developer for debugging. You can report problems on the GitHub CATS page (https://github.com/aladshaw3/cats) or by emailing a developer list on the 'README' file.

Single Core Command Line

To run a simulation in CATS uses the command line. The basic structure for calling a simulation is as follows:

```
./cats-opt -i path/to/input.i
```

The above command calls the 'cats-opt' executable, denotes that you are giving an input file with the '-i' designation, then provides the input file to run along with the path to that file from the current directory.

As an additional argument, you can also suffix the above command with option '-log_view' that will print out some information after a completed run that can be used to see the statistics of the solvers (i.e., amount of time spent in each solver, percent of time in PETSc vs MOOSE, number of CPUs and flops per CPU, etc). This can be vital information when performing diagnostics checks. To invoke this option, your command line argument would be as follows:

```
./cats-opt -i path/to/input.i -log_view
```

Multi-Core Command Line

One of the major advantages of the MOOSE modeling framework is how easy it is to run simulations with multiple CPUs. MOOSE uses the Message Passing Interface (MPI) to distribute memory and operations among the cores of your computer. To use multiple cores to run a simulation, you simply prefix your command line argument with 'mpiexec --n 4' where the number 4 would represent the number of CPU cores to use. See the example below.

```
mpiexec --n 4 ./cats-opt -i path/to/input.i
```

Just like with single core simulations, you can invoke '-log_view' to get a diagnostics report for the multicore simulations. See the example below.

```
mpiexec --n 4 ./cats-opt -i path/to/input.i -log_view
```

To Do List

This section of the user guide is dedicated to items that are not yet developed and plan to be developed. It keeps a running list of things in the code base that needs changing or updating. Assume that any item discussed and/or listed here is not yet fully functional.

Specific Kernels

High Advection Stabilization

In recent testing, results have shown that even using DG methods that some oscillations in the solution wave front is still apparent under extreme circumstances, such as convergent fronts or fast-moving upwind fronts with slow moving downwind fronts. Thus, it may be necessary to investigate additional kernels that can be invoked to add stabilization for extreme cases. Such stabilization will need to be done by either (i) invoking a form of slope limiter or (ii) adding dispersion in those areas for numerical stabilization.

Finite Volumes CFD

The MOOSE framework has an experimental version of the Navier-Stokes equations using a finite volumes method built out of DG kernels. This will prove very useful for our purposes, thus, there is a desire to build our tests and examples around this new MOOSE framework capability. However, at current, this is low on the priorities list for this project.

Kernels for Ohmic Heating

CATS is adding capabilities for electrolyte and electrochemistry. The energy balances associated with these systems are fairly similar to other catalytic systems, however, there is a heating source term dubbed ‘Ohmic Heating’ which relates current through the system of a given resistance with a thermal heating of that system. This specific source term needs its own kernel.

Kernels

Kernels are the primary physics/chemistry portions of the CATS model framework. Each kernel provides a single residual to the overall problem you are seeking to solve. Simply invoke new kernels to add a single piece of physics or chemistry to your simulation. The best approach to developing your specific simulation is to first write out the strong form of the physics, then move all pieces of physics to the same side of the equation as your variable's time derivative (if there is one), then multiply through by a test function (φ). Search through the Residual Formulations below to see if that piece of physics exists in the framework and how to invoke it.

For example, let's say you want to simulate a simple advection diffusion problem. Mathematically, that would be represented by:

"Strong Form" $\frac{\partial u}{\partial t} + \nabla \cdot (v \cdot u) = \nabla \cdot (D \cdot \nabla u)$

Move all physics to the left-hand side and multiply by φ.

"Weak Form" $\varphi \frac{\partial u}{\partial t} + \varphi \nabla \cdot (v \cdot u) - \varphi \nabla \cdot (D \cdot \nabla u) = 0$

Search through the kernels listed below to find residual contributions for each piece of physics:

TimeDerivative ➔ $\varphi \frac{\partial u}{\partial t}$

GAdvection (DGAdvection) ➔ $\varphi \nabla \cdot (v \cdot u)$

GAnisotropicDiffusion (DGAnisotropicDiffusion) ➔ $-\varphi \nabla \cdot (D \cdot \nabla u)$

ActivityConstraint

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel creates an expression for the calculation of the activity of a chemical species in a phase (generally an electrolyte) as a function of a coupled activity coefficient (γ), the concentration of the species in that phase (C), and a reference or total concentration (C_{ref}). Generally, the reference concentration should be a constant value, but here it is coded as a variable in case it is necessary to calculate later. The primary purpose of this kernel is to create a means by which concentration values used in any reaction (ConstReaction, ArrheniusReaction, ModifiedButlerVolmerReaction, etc) can be replaced with activity variables while maintaining the same general form and still be fully coupled. It should be used in conjunction with the Reaction kernel to complete the constraint. You should also initialize the activity states by using InitialActivity for the activity variables. This will help improve convergence.

Residual Formulation

Residual (unitless) = $-\varphi \cdot \gamma \left(\frac{C}{C_{ref}}\right)$

φ = MOOSE variable test function (for FE formulation)

γ = variable for the activity coefficient of the species

(default = 1, assumes ideal)

C = variable for concentration of the species (same units as C_{ref})

C_{ref} = reference state or total concentration (same units as C)

(default = 1 M, the typical reference value for electrolytes)

Additional Computations ➔ None

Usage

Generally used in conjunction with a Reaction kernel to evaluate the activity of a species.

The example below shows how to calculate the activity of protons (a_H) from the proton concentration (C) and a proton activity coefficient variable (g_H). The standard C_{ref} of 1 M is used for this example, so the concentration variable (C) must have units of M as well.

```
[./a_H_equ]
        type = Reaction
        variable = a_H
[../]
[./a_H_fun]
        type = ActivityConstraint
        variable = a_H
        concentration = C_H
        activity_coeff = g_H
        ref_conc = 1   # in M
[../]
```

ArrheniusEquilibriumReaction

Inheritance ➔ ArrheniusReaction

Notes ➔ This kernel uses existing residuals from ArrheniusReaction, but calculates the rate terms for the reverse half of the reaction based on the enthalpies (ΔH) and entropies (ΔS) of the reaction. User must still provide the forward activation energy (E_f) and forward pre-exponential term (A_f).

Residual Formulation

(See ArrheniusReaction for residual formulation)

Additional Computations

$$k_f = A_f T^{\beta_f} exp\left[-\frac{E_f}{RT}\right] \qquad k_r = A_r T^{\beta_r} exp\left[-\frac{E_r}{RT}\right]$$

- Both β are assumed 0 for this kernel.

- All reverse components are calculated from forward components with ΔH & ΔS

$$E_r = E_f - \Delta H \qquad A_r = A_f \cdot exp\left\{-\frac{\Delta S}{R}\right\}$$

ΔH = reaction enthalpy (J/mol)

ΔS = reaction entropy (J/K/mol)

- The units for A_f and A_r must have the same units as k_f and k_r
- The units for E_f and E_r must be J/mol (same as ΔH)

Usage

Generally used in conjunction with a TimeDerivative or VariableCoefTimeDerivative kernel.

Code below creates a residual for species A for the following reaction:

A + B ← → C

```
[./rxn]
    type = ArrheniusEquilibriumReaction
    variable = A
    this_variable = A
    temperature = T
    scale = -1
    forward_activation_energy = 5E4
    forward_pre_exponential = 25
    enthalpy = -3.9E5
    entropy = 0
    reactants = 'A B'
    reactant_stoich = '1 1'
    products = 'C'
    product_stoich = '1'
[../]
```

ArrheniusEquilibriumReactionEnergyTransfer

Inheritance ➔ ArrheniusReactionEnergyTransfer

Notes ➔ This kernel using existing residuals from ArrheniusReactionEnergyTransfer, but calculates the rate terms for the reverse half of the reaction based on the enthalpies (ΔH) and entropies (ΔS) of the reaction. User must still provide the forward activation energy (E_f) and forward pre-exponential term (A_f).

Residual Formulation

(See ArrheniusReactionEnergyTransfer for residual formulation)

Additional Computations

The forward and reverse rate components are calculated the same way as ArrheniusEquilibriumReaction. See that kernel for details.

Usage

Generally used in conjunction with an energy balance set of kernels to add the physics of energy changes caused by a chemical reaction in the system. The usage also includes terms to convert the volumetric or per reactive area terms into the proper energy density residual.

Code below creates a residual for energy exchange via the following surface reaction:

A + B ← → C

```
[./rxn]
    type = ArrheniusEquilibriumReactionEnergyTransfer
    variable = E                # E = Energy density variable
    this_variable = E
    temperature = T
    volume_frac = s_frac        # Volume of solids per Total Volume
    specific_area = 8.6E7       # Reaction area of solid per volume of solid
    forward_activation_energy = 5E4
    forward_pre_exponential = 25
    enthalpy = -3.9E5
    entropy = 0
    reactants = 'A B'
    reactant_stoich = '1 1'
    products = 'C'
    product_stoich = '1'
[../]
```

ArrheniusReaction

Inheritance ➔ ConstReaction

Notes ➔ This kernel has all the same residuals from ConstReaction, but calculates the forward and reverse rates of that reaction based on the activation energies (E), pre-exponential factors (A), and beta terms (β) for both the forward and/or reverse halves of the reaction. All of those terms are optional. User only needs to provide the terms that the kernel actually needs.

Residual Formulation

(See ConstReaction for residual formulation)

Additional Computations

$$k_f = A_f T^{\beta_f} exp\left[-\frac{E_f}{RT}\right] \qquad k_r = A_r T^{\beta_r} exp\left[-\frac{E_r}{RT}\right]$$

- The units for A_f and A_r must have the same units as k_f and k_r
- The units for E_f and E_r must be J/mol

Usage

Generally used in conjunction with a TimeDerivative or VariableCoefTimeDerivative kernel.

Code below creates a residual for species A for the following reaction:

A + B → C

```
[./rxn]
    type = ArrheniusReaction
    variable = A
    this_variable = A
    temperature = T
    scale = -1
    forward_activation_energy = 5E4
    forward_pre_exponential = 25
    reactants = 'A B'
    reactant_stoich = '1 1'
    products = 'C'
    product_stoich = '1'
[../]
```

ArrheniusReactionEnergyTransfer

Inheritance → ArrheniusReaction

Notes → Although this kernel is used in energy balances and not mass balances, the kernels are mathematically very similar. Thus, this kernel uses the residual computed from ArrheniusReaction, but applies the enthalpy (ΔH) and 2 unit conversion factors to modify the scaling parameter of ArrheniusReaction. The net effect is to convert the existing reaction rate function from a mass term to an energy term.

Residual Formulation

The baseline residual comes from ArrheniusReaction. In this formulation, we will represent that portion of the residual as r (i.e., the rate of the reaction) with a scaling parameter of 1. See ArrheniusReaction for more details on scaling parameters. Then, this kernel uses that reaction residual (r) and applies a custom scaling based on ΔH and other unit conversions.

Residual (J/m^3/s) = $-\Delta H \cdot f_v \cdot A_s \cdot r$

f_v = volume fraction (i.e., volume of solids per total volume)

→ Can be set to 1 if no such conversion is needed

A_s = Specific reaction area per volume of solids (for a surface reaction)

→ Can be set to 1 if no such conversion is needed

ΔH = enthalpy of the reaction (J/mol)

r = reaction rate (mol/m^3/s) or (mol/m^2/s) [depends on reaction type]

➔ Use f_v and A_s to convert units such that the units for the residual return as energy rate per total volume (J/m^3/s)

Additional Computations

As noted above, the scaling parameter from the reaction is overridden to always be 1.

Usage

Generally used in conjunction with an energy balance set of kernels to add the physics of energy changes caused by a chemical reaction in the system. The usage also includes terms to convert the volumetric or per reactive area terms into the proper energy density residual.

Code below creates a residual for energy exchange via the following surface reaction:

A + B → C

```
[./rxn]
    type = ArrheniusReactionEnergyTransfer
    variable = E                # E = Energy density variable
    this_variable = E
    temperature = T
    volume_frac = s_frac        # Volume of solids per Total Volume
    specific_area = 8.6E7       # Reaction area of solid per volume of solid
    forward_activation_energy = 5E4
    forward_pre_exponential = 25
    enthalpy = -3.9E5
    reactants = 'A B'
    reactant_stoich = '1 1'
    products = 'C'
    product_stoich = '1'
[../]
```

ButlerVolmerCurrentDensity

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel creates an expression for the calculation of current from a Butler-Volmer type of reaction (such as ModifiedButlerVolmerReaction). In this kernel, you will couple the reaction rate from Butler-Volmer kinetics to the specific surface area of the electrodes to get a current density variable that can then be coupled into the electrode and electrolyte potential functions. Generally, that coupling into other kernels is accomplished using the ScaledWeightedCoupledSumFunction or similar kernel.

Residual Formulation

Residual (C / total volume / time) = $\varphi \cdot nA_s F(-r)$

φ = MOOSE variable test function (for FE formulation)

n = number of electrons transferred in the reaction

F = Faraday's constant (default = 96,485.3 C/mol)

r = reaction rate variable (moles / area / time)

A_s = specific surface area (electrode area / total volume)

Additional Computations ➔ None

Usage

Generally used in conjunction with a Reaction kernel to evaluate the current density (J). That current density can then be coupled with other kernels that evaluate the electric potentials in the domain.

The example below shows how to calculate the current density from a reaction (r) where 1 electron is exchanged. The electrodes have a specific surface area of As.

```
[./J_equ]
        type = Reaction
        variable = J
[../]
[./J_fun]
        type = ButlerVolmerCurrentDensity
        variable = J
        number_of_electrons = 1
        specific_area = As
        rate_var = r
[../]
```

ConstMassTransfer

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel can be used to create a generic exchange of mass (or energy) between 2 different non-linear variables (which may represent two phases of matter). For instance, a common usage would be to add the physics of mass transfer from a liquid phase to a gaseous phase inside bubbles in a CO_2 absorption column.

Residual Formulation

Residual (mass / volume / time) = $\varphi \cdot k \cdot (u - v)$

φ = MOOSE variable test function (for FE formulation)

k = transfer rate ($time^{-1}$)

u = this object's non-linear variable (mass / volume)

v = a coupled non-linear variable (mass / volume)

Additional Computations ➔ None

Usage

Generally used in conjunction with other physics or chemistry, this kernel facilitates the coupling of 2 different variables that exchange mass with each other at a given rate at all locations in the domain of interest.

```
[./mass_trans]
    type = ConstMassTransfer
    variable = u
    coupled = v
    transfer_rate = 1
[../]
```

ConstReaction

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel creates a generic residual for a single reaction. That reaction is assumed to be mechanistic (i.e., the stoichiometric coefficients are also the powers of the variables in the rate expression). The formulation of the kernel is such that it is valid for both reversible and irreversible reactions. A scaling parameter is provided to account for the molar amounts of a species either formed or consumed during the reaction. Multiple instances of this object can be invoked to create residuals for a series of reactions that all contribute to the losses/gains in a specific chemical species.

Residual Formulation

Residual (mass / volume (or area) / time)

$$= -\left(ak_f \cdot \prod_{react} C_i^{v_i} - ak_r \cdot \prod_{prod} C_j^{v_j}\right) \cdot \varphi$$

φ = MOOSE variable test function (for FE formulation)

a = scaling parameter

- ➔ Used as a conversion for the rates of a particular species
- ➔ Additionally, the scaling parameter is used to determine whether a species is lost or gained in a reaction. Negative scaling indicates the species is lost. Positive scaling indicates the species is gained. (i.e., if using this kernel to add a reaction rate for a species whose stoichiometry in the reaction is 2 and is lost in the reaction, then the scaling a = -2).

k_f = forward rate constant (units depend on number of reactants)

k_r = reverse rate constant (units depend on number of products)

C_i = the i^{th} reactant species variable

C_j = the j^{th} product species variable

v_i = the i^{th} reactant species' stoichiometry

v_j = the j^{th} product species' stoichiometry

➔ Both the product and reaction species and stoichiometries will be given as a list of variables and values in the input file. The stoichiometry values given MUST be in the same order that the variable names are given in.

Additional Computations ➔ None

Usage

Generally used in conjunction with a TimeDerivative or VariableCoefTimeDerivative kernel.

Code below creates a residual for species A for the following reaction:

2A + B → C

```
[./rxn]
    type = ConstReaction
    variable = A
    this_variable = A
    scale = -2
    forward_rate = 1
    reactants = 'A B'
    reactant_stoich = '2 1'
    products = 'C'
    product_stoich = '1'
[../]
```

CoupledCoeffTimeDerivative

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel is used to couple the time derivative of another non-linear variable to this variable's kernel set. It provides another means to couple the effects of mass or energy transfer, if the physics of transfer is directly related to the rate of change of another variable. For example, the mass transfer term in adsorption often is coupled to the bulk phase concentration as a sink term that is a direct function of the rate of adsorption (-a*dq/dt).

Residual Formulation

Residual (mass / volume / time) = $a \cdot \frac{dv}{dt} \cdot \varphi$

φ = MOOSE variable test function (for FE formulation)

a = time coefficient (useful for unit conversions or scaling)

➔ If a is a positive number, then this is a sink term to the main variable
➔ If a is a negative number, then this is a source term

v = coupled non-linear variable

(*Depreciated option*: "gaining" – Boolean used to determine whether or not to treat this as a source or sink term. Just change the sign of a to change from sink term to a source term).

Additional Computations ➔ None

Usage

Must be used in conjunction with a set of kernels for the main variable and the coupled variable.

Code below would create a sink term for the variable u based on the rate of change of v:

```
[./coupled_dv_dt]
        type = CoupledCoeffTimeDerivative
        variable = u
        coupled = v
        time_coeff = 1
[../]
```

CoupledPorePhaseTransfer

Inheritance ➔ CoupledCoeffTimeDerivative

Notes ➔ This kernel creates a specific form of CoupledCoeffTimeDerivative wherein the concentration of a bulk fluid species is coupled to the time derivative of a solid or surface species in a particle. The time coefficient is the porosity of the bulk phase, which is a common defining parameter for packed bed reactors. That porosity is used as a unit conversion from mass per volume of solids for the coupled variable to mass per total volume in the system.

Residual Formulation

(See CoupledCoeffTimeDerivative for residual formulation)

Additional Computations

The time coefficient (a) for the CoupledCoeffTimeDerivative is overridden to be as follows:

$a = (1 - \varepsilon)$ where ε is the bulk porosity in the domain (volume voids / total volume)

This kernel still uses the "gaining" Boolean argument to determine whether or not to treat this transfer as a source or sink term. That option is depreciated, so you do not need to use it.

Usage

Must be used in conjunction with a set of kernels for the bulk variable and the coupled variable.

Code below would create a sink term for the variable Cb based on the rate of change of q:

```
[./coupled_dq_dt]
        type = CoupledPorePhaseTransfer
        variable = Cb
        coupled =q
        porosity = eps
[../]
```

CoupledSumFunction

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel creates a residual based on a summation of other non-linear variables. It is intended to create a residual for a non-linear variable (not in the list of summed variables) to enforce the constraint that the value of the primary variable be a sum of other variables. For example, you can use this to establish the total amount of a material adsorbed as the sum of different adsorbed species that contribute to the total.

Residual Formulation

Residual (same units as the variables given) $= -\varphi \cdot \sum_{\forall i} u_i$

φ = MOOSE variable test function (for FE formulation)

u_i = i^{th} coupled non-linear variable

Additional Computations ➔ None

Usage

Generally used in conjunction with the built-in Reaction kernel (MOOSE framework kernel) in order to complete the summation expression that one non-linear variable be made as a sum of other non-linear variables.

In the example code below, this kernel is being used to sum up the adsorption from 3 species (q1, q2, and q3) to form the total adsorption (qT). Note that here we also must invoke the Reaction kernel on qT to force the finished summation expression.

```
[./qT_res]
        type = Reaction
        variable = qT
[../]
[./qT_sum]
        type = CoupledSumFunction
        variable = qT
        coupled_list = 'q1 q2 q3'
[../]
```

DivergenceFreeCondition

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel creates a residual based on a coupled divergence function. Generally, this kernel would be used to resolve pressure and/or potential gradients for the resolution of a flow field. This particular formulation is valid ONLY in cartesian coordinates, since the divergence condition is done in a piece-wise fashion. An optional variable for coupling a non-linear scalar variable is also provided within this formulation. As such, this can be used for both Incompressible and Compressible flows.

Residual Formulation

Residual (depends on given variables) $= \varphi \cdot (\nabla \cdot \rho \mathbf{v})$

φ = MOOSE variable test function (for FE formulation)

ρ = scalar coupled variable (such as density in Navier-Stokes)

v = coupled vector of variables in Cartesian coordinates (ux, uy, uz)

Additional Computations ➔ None

Usage

Generally used to enforce the continuity equation for fluid mass (or a continuity equation for current/eletroneutrality). By default, the 'ρ' parameter will be 1, in which case this simplifies to just the divergence of the given vector of variables.

In the demonstration below, we use this as the sole kernel acting on a pressure variable to solve the Navier-Stokes equations. In this formulation, we also give the 'coupled_scalar' as a 'rho' variable so that this is a valid expression in both incompressible and compressible flow.

```
[./continuity]
        type = DivergenceFreeCondition
        variable = pressure
        coupled_scalar = rho
        ux = vel_x
        uy = vel_y
        uz = vel_z
[../]
```

ElectrodeCurrentFromPotentialGradient

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel creates a residual for the electrical current in an electrode based on the potential gradient of that electrode. Current is calculated piecewise, thus, users must provide the direction that the kernel acts on for that component of electrical current. Users must also

provide a value/variable for the conductivity of the electrode as well as the 'solids fraction', if the electrode is porous.

Residual Formulation

Residual (C/area/time) $= \varphi \cdot f \cdot \sigma_s(\mathbf{n_i} \cdot \nabla \phi_s)$

φ = MOOSE variable test function (for FE formulation)

ϕ_s = variable for electrode potential (typical units: V or J/C)

$\mathbf{n_i}$ = unit vector in the direction of interest for the current vector

f = solids fraction (default = 1)

σ_s = conductivity of the electrode material (typical units: C/V/length/time)

Additional Computations ➔ None

Usage

Generally used to calculate electrical current in a porous electrode. The units of current will depend on the units given for the coefficients in the residual. Since current is a vector, you will need to apply this kernel for each current direction in the domain.

MUST be used in conjunction with the Reaction kernel to fully describe the model. The Reaction kernel supplies the 'equals to' operator for directly solving for current.

$$\mathbf{i_s} = -f\sigma_s \nabla \phi_s$$

The example below shows how to calculate the current in x and y directions. User provides the direction as an integer using the standard notation in MOOSE (x = 0, y = 1, and z = 2). Each parameter/coefficient can be a constant value or another variable.

```
[./is_x_equ]
        type = Reaction
        variable = is_x
[../]
[./is_x_pot]
        type = ElectrodeCurrentFromPotentialGradient
        variable = is_x
        direction = 0
        electric_potential = phi_s
        solid_frac = 0.5
        conductivity = sigma_s
[../]

[./is_y_equ]
        type = Reaction
        variable = is_y
[../]
[./is_y_pot]
```

```
        type = ElectrodeCurrentFromPotentialGradient
        variable = is_y
        direction = 1
        electric_potential = phi_s
        solid_frac = 0.5
        conductivity = sigma_s
[../]
```

ElectrodePotentialConductivity

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel creates a residual for the electrical potential in the electrode. Users can solve for potential by combining this kernel with kernels for ion reactions and boundary conditions for applied potentials at the walls/edges of the domain. Electric potential in the electrode comes from the divergence of ElectrodeCurrentFromPotentialGradient. However, instead of applying a DivergenceFreeCondition on that current, it is numerically more stable to model this as a 'Diffusion' process with standard finite elements. Thus, it is recommended to calculate potential in this manner.

Residual Formulation

Residual (C/volume/time) $= f \cdot \sigma_s(\nabla\varphi \cdot \nabla\phi_s)$

φ = MOOSE variable test function (for FE formulation)

ϕ_s = variable for electrode potential (typical units: V or J/C)

f = solids fraction (default = 1)

σ_s = conductivity of the electrode material (typical units: C/V/length/time)

Additional Computations ➔ None

Usage

Generally used to calculate electrical potential in a porous electrode, along with suitable boundary conditions and/or adding redox reactions to this kernel set. The units of potential will depend on the units given for the coefficients in the residual. Generally, your potential will have units of Voltz (V) or J/C.

In the case of no reactions, then all that is needed is this kernel plus appropriate BCs. Electric potential equations follow from the divergence of current to satisfy continuity of the flow of electricity.

$$\nabla \cdot \mathbf{i_s} = -\nabla \cdot (f\sigma_s \nabla\phi_s) = 0$$

If reactions are present, then the only modification to the above is to include a source/sink term. **NOTE**: For porous electrodes, the source/sink term will be of opposite sign for the electrolyte potential functions.

The example below shows how to invoke the kernel for electrode potential calculations. All the parameters/coefficients can be constants or other variables.

```
[./phi_s_pot]
        type = ElectrodePotentialConductivity
        variable = phi_s
        solid_frac = 0.5
        conductivity = sigma_s
[../]
```

ElectrolyteCurrentFromPotentialGradient

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel creates a residual for the electrical current in an electrolyte based on the potential gradient of that electrolyte. Current is calculated piecewise, thus, users must provide the direction that the kernel acts on for that component of electrical current. Users must also provide a values/variables for porosity of the domain, temperature, ion concentrations, diffusion coefficients for ions, and ion valences. Optionally, users may also provide values to override the gas constant and Faraday's constants if a different unit basis is needed.

NOTE: When using this kernel, you should ONLY provide ONE set of ions (either the set of negatively charged ions or positively charged ions). If you do NOT follow this convention, then the potentials in the domain will either always solve to zero (i.e., constant potential) or the solution will be infeasible. This is because the oppositely charged ions have equal and opposite effects on the potential. THUS, you are only ever calculating a REFERENCE potential in solution, where that reference is based on either the positive ions or the negative ions.

NOTE 2: For very dilute systems, it is HIGHLY recommended that you provide a background ion reference concentration. Otherwise, the calculation of effective conductivity will result in values too low to get a reasonable reference potential.

Special Note: The effective conductivity (K_{eff}) of the electrolyte phase is calculated from the diffusivity of ions and concentration of ions in solution. However, in circumstances where you do not know all the ions that contribute to the conductivity of the electrolyte, you can also provide a 'min_conductivity' argument that will add to the calculated K_{eff} value a given constant that would essentially represent the 'background' or 'standard state' effective conductivity for the media. For example, for a water system you could use the following values as the 'background' conductivity:

```
min_conductivity = 3.0E-6       #C/V/cm/min  (for DI water)

min_conductivity = 0.03         #C/V/cm/min (for tap water)

min_conductivity = 3.0          #C/V/cm/min (for seawater)
```

The units you give will vary depending on your needs.

Residual Formulation

Residual (C/area/time) $= \varphi \cdot K_{eff}(\mathbf{n_i} \cdot \nabla \phi_e)$

φ = MOOSE variable test function (for FE formulation)

ϕ_e = variable for electrolyte potential (typical units: V or J/C)

$\mathbf{n_i}$ = unit vector in the direction of interest for the current vector

K_{eff} = effective conductivity of the electrolyte (typical units: C/V/length/time)

$$K_{eff} = \varepsilon \frac{F^2}{RT}\left[\sum_{\substack{\forall\ same \\ charge\ ions}} {z_j}^2 D_j c_j\right]$$

ε = porosity (default = 1)

F = Faraday's Constant (default = 96,485.3 C/mol)

R = Gas Constant (default = 8.314462 J/K/mol)

T = temperature of electrolyte (K)

z_j = valence of the j-th ion

D_j = Diffusivity of the j-th ion (area/time)

c_j = concentration of the j-th ion (mol/volume)

Additional Computations ➔ None

Usage

Generally used to calculate electrical current in an electrolyte. The units of current will depend on the units given for the coefficients in the residual. Since current is a vector, you will need to apply this kernel for each current direction in the domain.

MUST be used in conjunction with the ElectrolyteCurrentFromIonGradient kernel AND Reaction kernel the kernel to fully describe the model. The Reaction kernel supplies the 'equals to' operator for directly solving for current.

$$\mathbf{i_e} = -K_{eff}\nabla\phi_e - F\varepsilon\left[\sum_{\substack{\forall\ same \\ charge\ ions}} z_j D_j \nabla c_j\right]$$

The example below shows how to calculate electrolyte current in just the x-direction. All parameters and coefficients (except for defined constants and valances) can be given as single values or as variables. In this case, we also supply a 'background' concentration of ions. We are also including the ElectrolyteCurrentFromIonGradient for completeness. In special cases, this additional piece can be neglected.

For aqueous systems, a good reference ion for the background is H^+, which at a neutral pH will be 1e-4 mol/m^3.

```
[./ie_x_equ]
        type = Reaction
        variable = ie_x
[../]
[./ie_x_pot]
        type = ElectrolyteCurrentFromPotentialGradient
        variable = ie_x
        direction = 0
        electric_potential = phi_e
        porosity = 0.5
        temperature = 298
        ion_conc = 'background pos_ion'
        diffusion = 'Db Dp'
        valence = '1 2'
[../]
[./ie_x_ion]
        type = ElectrolyteCurrentFromIonGradient
        variable = ie_x
        direction = 0
        porosity = 0.5
        ion_conc = 'background pos_ion'
        diffusion = 'Db Dp'
        valence = '1 2'
[../]
```

ElectrolyteCurrentFromIonGradient

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel creates a residual for the electrical current in an electrolyte based on the gradients of ions in the electrolyte domain. Current is calculated piecewise, thus, users must provide the direction that the kernel acts on for that component of electrical current. Users must also provide a values/variables for porosity of the domain, ion concentrations, diffusion coefficients for ions, and ion valences. Optionally, users may also provide values to override the gas constant and Faraday's constants if a different unit basis is needed.

NOTE: When using this kernel, you should ONLY provide ONE set of ions (either the set of negatively charged ions or positively charged ions). If you do NOT follow this convention, then the potentials in the domain will either always solve to zero (i.e., constant potential) or the solution will be infeasible. This is because the oppositely charged ions have equal and opposite effects on the potential. THUS, you are only ever calculating a REFERENCE potential in solution, where that reference is based on either the positive ions or the negative ions.

NOTE 2: For very dilute systems, it is HIGHLY recommended that you provide a background ion reference concentration. Otherwise, the calculation of effective conductivity will result in values too low to get a reasonable reference potential.

Residual Formulation

Residual (C/area/time) $= \varphi \cdot F\varepsilon \left[\sum_{\substack{\forall\, same \\ charge\ ions}} z_j D_j \left(\mathbf{n_i} \cdot \nabla c_j \right) \right]$

φ = MOOSE variable test function (for FE formulation)

$\mathbf{n_i}$ = unit vector in the direction of interest for the current vector

ε = porosity (default = 1)

F = Faraday's Constant (default = 96,485.3 C/mol)

z_j = valence of the j-th ion

D_j = Diffusivity of the j-th ion (area/time)

c_j = concentration of the j-th ion (mol/volume)

Additional Computations ➔ None

Usage

Generally used to calculate electrical current in an electrolyte. The units of current will depend on the units given for the coefficients in the residual. Since current is a vector, you will need to apply this kernel for each current direction in the domain.

MUST be used in conjunction with the ElectrolyteCurrentFromIonPotential kernel AND Reaction kernel the kernel to fully describe the model. The Reaction kernel supplies the 'equals to' operator for directly solving for current.

$$\mathbf{i_e} = -K_{eff} \nabla \phi_e - F\varepsilon \left[\sum_{\substack{\forall\, same \\ charge\ ions}} z_j D_j \nabla c_j \right]$$

The example below shows how to calculate electrolyte current in just the x-direction. All parameters and coefficients (except for defined constants and valances) can be given as single values or as variables. In this case, we also supply a 'background' concentration of ions. We are also including the ElectrolyteCurrentFromIonPotential for completeness.

```
[./ie_x_equ]
        type = Reaction
        variable = ie_x
[../]
[./ie_x_pot]
        type = ElectrolyteCurrentFromPotentialGradient
        variable = ie_x
```

```
        direction = 0
        electric_potential = phi_e
        porosity = 0.5
        temperature = 298
        ion_conc = 'background pos_ion'
        diffusion = 'Db Dp'
        valence = '1 2'
[../]
[./ie_x_ion]
        type = ElectrolyteCurrentFromIonGradient
        variable = ie_x
        direction = 0
        porosity = 0.5
        ion_conc = 'background pos_ion'
        diffusion = 'Db Dp'
        valence = '1 2'
[../]
```

ElectrolytePotentialConductivity

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel creates a residual for the electrical potential in the electrolyte. Users can solve for potential by combining this kernel with kernels for potential contributions from ion gradients, ion reactions, and boundary conditions for applied potentials at the walls/edges of the domain. Electric potential in the electrolyte comes from the divergence of ElectrolyteCurrentFromPotentialGradient plus ElectrolyteCurrentFromIonGradient. However, instead of applying a DivergenceFreeCondition on that current, it is numerically more stable to model this as a 'Diffusion' process with standard finite elements. Thus, it is recommended to calculate potential in this manner.

NOTE: When using this kernel, you should ONLY provide ONE set of ions (either the set of negatively charged ions or positively charged ions). If you do NOT follow this convention, then the potentials in the domain will either always solve to zero (i.e., constant potential) or the solution will be infeasible. This is because the oppositely charged ions have equal and opposite effects on the potential. THUS, you are only ever calculating a REFERENCE potential in solution, where that reference is based on either the positive ions or the negative ions.

NOTE 2: For very dilute systems, it is HIGHLY recommended that you provide a background ion reference concentration. Otherwise, the calculation of effective conductivity will result in values too low to get a reasonable reference potential. (See **Special Note** below)

NOTE 3: You will ALWAYS be required to supplied this kernel for electrolyte potential. However, coupling with ElectrolyteIonConductivity is optional and in some special cases can be neglected.

Special Note: The effective conductivity (K_{eff}) of the electrolyte phase is calculated from the diffusivity of ions and concentration of ions in solution. However, in circumstances where you do not know all the ions that contribute to the conductivity of the electrolyte, you can also provide a 'min_conductivity' argument that will add to the calculated K_{eff} value a given constant that would essentially represent the 'background' or 'standard state' effective conductivity for the media. For example, for a water system you could use the following values as the 'background' conductivity:

min_conductivity = 3.0E-6 #C/V/cm/min (for DI water)

min_conductivity = 0.03 #C/V/cm/min (for tap water)

min_conductivity = 3.0 #C/V/cm/min (for seawater)

The units you give will vary depending on your needs.

Residual Formulation

Residual (C/volume/time) $= K_{eff}(\nabla\varphi \cdot \nabla\phi_e)$

φ = MOOSE variable test function (for FE formulation)

ϕ_e = variable for electrolyte potential (typical units: V or J/C)

K_{eff} = effective conductivity of the electrolyte (typical units: C/V/length/time)

$$K_{eff} = \varepsilon \frac{F^2}{RT}\left[\sum_{\substack{\forall\ same \\ charge\ ions}} {z_j}^2 D_j c_j\right]$$

ε = porosity (default = 1)

F = Faraday's Constant (default = 96,485.3 C/mol)

R = Gas Constant (default = 8.314462 J/K/mol)

T = temperature of electrolyte (K)

z_j = valence of the j-th ion

D_j = Diffusivity of the j-th ion (area/time)

c_j = concentration of the j-th ion (mol/volume)

Additional Computations ➔ None

Usage

Generally used to calculate electrical potential of the electrolyte in a porous electrode, along with suitable boundary conditions and/or adding redox reactions to this kernel set. The units of potential will depend on the units given for the coefficients in the residual. Generally, your potential will have units of Voltz (V) or J/C.

In the case of no reactions, then all that is needed is this kernel plus appropriate BCs (and optionally the ElectrolyteIonConductivity kernel). Electric potential equations follow from the divergence of current to satisfy continuity of the flow of electricity.

$$\nabla \cdot \mathbf{i_e} = -\nabla \cdot \left(K_{eff} \nabla \phi_e\right) - \nabla \cdot \left(F\varepsilon \left[\sum_{\substack{\forall\ same \\ charge\ ions}} z_j D_j \nabla c_j \right] \right) = 0$$

If reactions are present, then the only modification to the above is to include a source/sink term. **NOTE**: For porous electrodes, the source/sink term will be of opposite sign for the electrode potential functions.

The example below shows how to invoke the kernel for electrolyte potential calculations, while neglecting contributions from ElectrolyteIonConductivity. All the parameters/coefficients can be constants or other variables.

For aqueous systems, we also provide a 'background' conductivity using the 'min_conductivity' argument.

```
[./phi_e_pot]
        type = ElectrolytePotentialConductivity
        variable = phi_e
        porosity = 0.5
        temperature = 298
        ion_conc = 'pos_ion'
        diffusion = 'Dp'
        valence = '2'
        min_conductivity = 0.03  # typical for tap water
[../]
```

ElectrolyteIonConductivity

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel creates a residual for the electrical potential in the electrolyte. Users can solve for potential by combining this kernel with kernels for potential contributions from ion gradients, ion reactions, and boundary conditions for applied potentials at the walls/edges of the domain. Electric potential in the electrolyte comes from the divergence of ElectrolyteCurrentFromPotentialGradient plus ElectrolyteCurrentFromIonGradient. However, instead of applying a DivergenceFreeCondition on that current, it is numerically more stable to model this as a 'Diffusion' process with standard finite elements. Thus, it is recommended to calculate potential in this manner.

NOTE: When using this kernel, you should ONLY provide ONE set of ions (either the set of negatively charged ions or positively charged ions). If you do NOT follow this convention, then the potentials in the domain will either always solve to zero (i.e., constant potential) or the solution will be infeasible. This is because the oppositely charged ions have equal and opposite

effects on the potential. THUS, you are only ever calculating a REFERENCE potential in solution, where that reference is based on either the positive ions or the negative ions.

NOTE 2: For very dilute systems, it is HIGHLY recommended that you provide a background ion reference concentration. Otherwise, the calculation of effective conductivity will result in values too low to get a reasonable reference potential.

NOTE 3: This kernel is optional for solving for electrolyte potential.

Residual Formulation

Residual (C/volume/time) $= F\varepsilon \left[\sum_{\substack{\forall\ same \\ charge\ ions}} z_j D_j \left(\nabla\varphi \cdot \nabla c_j \right) \right]$

φ = MOOSE variable test function (for FE formulation)

ε = porosity (default = 1)

F = Faraday's Constant (default = 96,485.3 C/mol)

z_j = valence of the j-th ion

D_j = Diffusivity of the j-th ion (area/time)

c_j = concentration of the j-th ion (mol/volume)

Additional Computations ➔ None

Usage

Generally used to calculate electrical potential of the electrolyte in a porous electrode, along with suitable boundary conditions and/or adding redox reactions to this kernel set. The units of potential will depend on the units given for the coefficients in the residual. Generally, your potential will have units of Voltz (V) or J/C.

In the case of no reactions, then all that is needed is this kernel plus appropriate BCs (and the REQUIRED ElectrolytePotentialConductivity kernel). Electric potential equations follow from the divergence of current to satisfy continuity of the flow of electricity.

$$\nabla \cdot \mathbf{i_e} = -\nabla \cdot \left(K_{eff} \nabla \phi_e \right) - \nabla \cdot \left(F\varepsilon \left[\sum_{\substack{\forall\ same \\ charge\ ions}} z_j D_j \nabla c_j \right] \right) = 0$$

If reactions are present, then the only modification to the above is to include a source/sink term. **NOTE**: For porous electrodes, the source/sink term will be of opposite sign for the electrode potential functions.

The example below shows how to invoke the kernel for electrolyte potential calculations. All the parameters/coefficients can be constants or other variables.

```
[./phi_e_pot]
        type = ElectrolytePotentialConductivity
```

```
        variable = phi_e
        porosity = 0.5
        temperature = 298
        ion_conc = 'pos_ion'
        diffusion = 'Dp'
        valence = '2'
        min_conductivity = 0.03 # typical for tap water
[../]
[./phi_e_ions]
        type = ElectrolyteIonConductivity
        variable = phi_e
        porosity = 0.5
        ion_conc = 'pos_ion'
        diffusion = 'Dp'
        valence = '2'
[../]
```

EquilibriumReaction

Inheritance ➔ ConstReaction

Notes ➔ This kernel uses the residual calculations from ConstReaction, but calculates the equilibrium constant for the reaction from a given ΔH and ΔS for the reaction. The scaling parameter from ConstReaction is overridden to 1, since it is useless for an equilibrium reaction, and the reverse rate is overridden to 1, then the forward rate forced to become the equilibrium constant. Use this kernel when you want to specify some local equilibrium between reactants and products, or when rate constants are unknown, but ΔH and ΔS are known.

Residual Formulation

(See ConstReaction for residual formulation)

Additional Computations

The value of the scaling parameter (a) from ConstReaction is overridden to be 1.

The value of the reverse rate (k_r) from ConstReaction is overridden to be 1.

The value of the forward rate (k_f) from ConstReaction is transformed into the equilibrium constant for this reaction through the van't Hoff relationship:

$$k_f = K = exp\left\{-\frac{\Delta H}{RT} + \frac{\Delta S}{R}\right\}$$

ΔH = reaction enthalpy (J/mol)

ΔS = reaction entropy (J/K/mol)

Usage

Generally, this kernel is intended to be used by itself to establish local equilibrium for a chemical species in the simulation domain. However, it can also be used in conjunction with other kernels on that variable for motion or transport of that species.

Example code below applies local equilibrium to chemical species A given the reaction:

A + B ← → C

```
[./rxn]
    type = EquilibriumReaction
    variable = A
    this_variable = A
    temperature = T
    enthalpy = -3.9E5
    entropy = 0
    reactants = 'A B'
    reactant_stoich = '1 1'
    products = 'C'
    product_stoich = '1'
[../]
```

ExtendedLangmuirFunction

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel creates a residual for the function side of the Extended Langmuir function for competitive adsorption with ideal surface behavior. While it may be temping to use this kernel as a quick and easy way to establish adsorption behavior, this kernel is less computationally efficient then representing adsorption through a series of reactions using the ArrheniusReaction, ConstReaction, ArrheniusEquilibriumReaction, or other such combinations of reaction kernels. It is also worth noting that this kernel only provides the right-hand side of the Extended Langmuir function and thus needs to be combined with the standard MOOSE Reaction kernel to finalize the representation. You can also combine multiple instances of this kernel to create a Multi-site or Heterogeneous Extended Langmuir type of function.

Residual Formulation

Residual (units of adsorbed species) = $-\varphi \cdot q_{max} \cdot \frac{K_i C_i}{1+\sum_{\forall j} K_j C_j}$

φ = MOOSE variable test function (for FE formulation)

q_{max} = theoretical maximum adsorption for this site (site density)

C_i = concentration of the primary bulk species adsorbing

K_i = equilibrium Langmuir constant for C_i (inverse units of C_i)

C_j = concentrations of all species j involved in reaction (including C_i)

K_j = equilibrium Langmuir constant for C_j (inverse units of C_j)

Additional Computations ➔ None

Usage

Generally used in conjunction with the built-in Reaction kernel (MOOSE framework kernel) in order to complete the adsorption expression.

Note that the list of adsorbing species MUST align in order with the list of Langmuir constants for those species.

Example below calculates the adsorption of species A to form q in the presence of species B, which is competing for the same adsorption sites as A. The Langmuir constant for A is 0.25 and for B is 0.75. The maximum site density (q_{max}) is 0.5.

```
[./q_res]
        type = Reaction
        variable = q
[../]
[./q_lang]
        type = ExtendedLangmuirFunction
        variable = q
        site_density = 0.5
        main_coupled = A
        coupled_list = 'A B'
        langmuir_coeff = '0.25 0.75'
[../]
```

ExtendedLangmuirModel

Inheritance ➔ ExtendedLangmuirFunction

Notes ➔ This kernel uses the same residuals from ExtendedLangmuirFunction, but calculates the Langmuir coefficients from the DH and DS of each species' site reaction based on the van't Hoff relationship. All other notes from ExtendedLangmuirFunction also apply to this kernel.

Residual Formulation

(See ExtendedLangmuirFunction for residual formulation)

Additional Computations

The Langmuir coefficients for each species involved in the reactions is calculated from the van't Hoff expression. See EquilibriumReaction for additional computation details. Must provide reaction enthalpies and entropies.

ΔH = reaction enthalpy (J/mol)

ΔS = reaction entropy (J/K/mol)

Usage

Usage is essentially the same as ExtendedLangmuirFunction, but user provides enthalpies and entropies instead of Langmuir coefficients (example below).

Example below calculates the adsorption of species A to form q in the presence of species B, which is competing for the same adsorption sites as A.

```
[./q_res]
        type = Reaction
        variable = q
[../]
[./q_lang]
        type = ExtendedLangmuirModel
        variable = q
        site_density = 0.5
        main_coupled = A
        coupled_list = 'A B'
        coupled_temp = T
        enthalpies = '-5000 -4000'
        entropies = '100 50'
[../]
```

FilmMassTransfer

Inheritance ➔ ConstMassTransfer

Notes ➔ This kernel uses the residual calculations from ConstMassTransfer, but allows for a variable rate transfer rate coefficient (k_m) and applies a conversion factor (A_s) to account for the transfer of mass across a specific area of a film layer of another phase (i.e., such as a bubble or solid particle).

Residual Formulation

(See ConstMassTransfer for residual formulation)

Additional Computations

The original rate parameter (k) from ConstMassTransfer is overridden in this kernel prior to calling the base kernel residual function.

$$k = k_m A_s v_f$$

k_m = variable rate of film mass transfer (m/s)

A_s = film layer area per volume of other phase (m^{-1})

v_f = volume fraction (used to convert units from one volume basis to another)

Usage

This kernel has the same usage as it's base class ConstMassTransfer. The only difference is the inclusion of the A_s and v_f coupled variables, and coupling with the k_m variable.

The below example is for the transfer of mass from the bulk phase (C_b) to the pore-space (C_p) of a porous material with rate variable of km and area-to-volume ratio of 1000 m^{-1}. Note that to

balance the mass between the 2 phases, you must include this kernel for both the bulk phase (C_b) and the pore-space (C_p) separately, and also use in conjunction with other kernels describing the transport or changes in those variables respectively. Additionally, we use a variable named (solids_frac) to apply the volume fraction unit conversion to each kernel. Note that this factor is optional. The default value in the kernel is set to 1.

```
[./mass_trans_from_bulk_to_pores]
        type = FilmMassTransfer
        variable = Cb
        coupled = Cp
        av_ratio = 1000
        rate_variable = km
        volume_frac = solids_frac
[../]


[./mass_trans_from_pores_to_bulk]
        type = FilmMassTransfer
        variable = Cp
        coupled = Cb
        av_ratio = 1000
        rate_variable = km
        volume_frac = solids_frac
[../]
```

GAdvection (DGAdvection)

Inheritance: GAdvection ➔ Kernel & DGAdvection ➔ DGKernel

Special Notes

➔ All kernels prefixed with 'G' and 'DG' must be used together (Except for Boundary Conditions). Combined they fully describe 1 piece of transport physics. You CANNOT use either 'G' or 'DG' kernels without also invoking the other. If you do, your simulation will likely result in errors in conservation of mass or energy.

➔ Any variable in the MOOSE system that uses 'DG' kernels must also either use MONOMIAL or L2_LAGRANGE shape functions (i.e., family). These are invoked when creating the variables under the variables block (see below).

```
[Variables]
        [./u]
                order = FIRST
                family = MONOMIAL
        [../]
[]
```

➔ It is recommended that you use a variable 'order' as either FIRST or SECOND for accuracy. However, if solutions are unstable, then reduce the 'order' to CONSTANT.

Notes ➔ This set of kernels formulates the Discontinuous Galerkin form for conservative advection in a simulation domain. This is the base class for advective physics in the DG system of kernels. All other advection kernels will inherit from this kernel. The velocities are assumed to be predetermined constants for the base class.

Residual Formulation

Due to the complexity of forming transport physics in the DG system of kernels, the exact formulations are not provided here. Instead, only the "strong form" of the physics is provided for all DG and associated kernels.

Strong Form Residual (mass / volume / time) = $\varphi \cdot \nabla \cdot (\mathbf{v}u)$

φ = MOOSE variable test function (for FE formulation)

v = velocity vector (m/s)

u = a conserved quantity per volume (mass / volume) or (energy / volume)

Additional Computations ➔ None

Usage

Generally used in conjunction with a TimeDerivative or VariableCoefTimeDerivative kernel.

Must always have the 'G' and 'DG' kernels matching by name and have matching parameters (i.e., same velocities). 'G' kernels are placed under the [Kernels] block and 'DG' kernels are placed under the [DGKernels] block in the input files.

Below example is for conservative advection of u with a constant velocity in the x-direction.

```
[Kernels]
    [./g_adv_u]
        type = GAdvection
        variable = u
        vx = 1
        vy = 0
        vz = 0
    [../]
[]

[DGKernels]
    [./dg_adv_u]
        type = DGAdvection
        variable = u
        vx = 1
        vy = 0
        vz = 0
    [../]
[]
```

GAnisotropicDiffusion (DGAnisotropicDiffusion)

Inheritance: GAnisotropicDiffusion ➔ Kernel & DGAnisotopicDiffusion ➔ DGKernel

Special Notes

➔ All kernels prefixed with 'G' and 'DG' must be used together (Except for Boundary Conditions). Combined they fully describe 1 piece of transport physics. You CANNOT use either 'G' or 'DG' kernels without also invoking the other. If you do, your simulation will likely result in errors in conservation of mass or energy.

➔ Any variable in the MOOSE system that uses 'DG' kernels must also either use MONOMIAL or L2_LAGRANGE shape functions (i.e., family). These are invoked when creating the variables under the variables block (see below).

```
[Variables]
    [./u]
        order = FIRST
        family = MONOMIAL
    [../]
[]
```

➔ It is recommended that you use a variable 'order' as either FIRST or SECOND for accuracy. However, if solutions are unstable, then reduce the 'order' to CONSTANT.

➔ **NOTE**: If you use a CONSTANT order for the variables, then this set of kernels will provide almost no value to the simulation. This is because diffusion is based on the local gradients of the variables and CONSTANT order variables have no gradients on a given element in the domain. However, it is sometimes necessary to invoke this order anyway for maximum stability.

➔ Note: DG diffusion type kernels all require additional parametric options to penalize jumps in discontinuity between neighboring variable values. These include a 'sigma' parameter and a 'dg_scheme' parameter.

dg_scheme → Can be either 'nipg', 'sipg', or 'iipg'. Default is 'nipg'.

More information on dg_scheme can be found in the corresponding source code files.

sigma → penalty term which should be >= 0. Default is 10.

As this penalty term gets higher, it convergence suffers.

Generally will want to put these options under 'GlobalParameters'

```
[GlobalParameters]
    dg_scheme = nipg
    sigma = 10
[]
```

Notes ➔ This set of kernels formulates the Discontinuous Galerkin form for conservative diffusion or dispersion in a simulation domain. This is the base class for diffusion physics in the DG system of kernels. All other diffusion kernels will inherit from this kernel. The diffusivity is taken as a tensor matrix such that the kernel can simulate transport in a medium that has anisotropic properties. However, you can just given the same diffusivities for the diagonal elements of the tensor to simulate isotropic diffusion transport with this kernel as well.

Residual Formulation

Due to the complexity of forming transport physics in the DG system of kernels, the exact formulations are not provided here. Instead, only the "strong form" of the physics is provided for all DG and associated kernels.

Strong Form Residual (mass / volume / time) = $-\varphi \cdot \nabla \cdot (\mathbf{D} \cdot \nabla u)$

φ = MOOSE variable test function (for FE formulation)

D = diffusion tensor (m^2/s)

➔ The tensor is given element-by-element in input values named as Dij where i is the row and j is the column of the tensor

➔ Example: Dxy = xy-component of diffusion (which can be different from Dyx if desired)

➔ For isotropic diffusion, simply provide only Dxx, Dyy, and Dzz with the same values (off-diagonals are assumed 0).

u = a conserved quantity per volume (mass / volume) or (energy / volume)

Additional Computations ➔ None

Usage

Generally used in conjunction with a TimeDerivative or VariableCoefTimeDerivative kernel.

Must always have the 'G' and 'DG' kernels matching by name and have matching parameters (i.e., same velocities). 'G' kernels are placed under the [Kernels] block and 'DG' kernels are placed under the [DGKernels] block in the input files.

Below example is for conservative diffusion of u with a constant, isotropic diffusion.

```
[Kernels]
    [./g_diff_u]
        type = GAnisotropicDiffusion
        variable = u
        Dxx = 1
        Dyy = 1
        Dzz = 1
    [../]
[]
```

```
[DGKernels]
    [./dg_diff_u]
        type = DGAnisotropicDiffusion
        variable = u
        Dxx = 1
        Dyy = 1
        Dzz = 1
    [../]
[]
```

GConcentrationAdvection (DGConcentrationAdvection)

Inheritance:

GConcentrationAdvection ➔ GAdvection

DGConcentrationAdvection ➔ DGAdvection

Special Notes ➔ (See GAdvection (DGAdvection) for special notes)

Notes ➔ This kernel set uses the residuals from GAdvection (DGAdvection), but allows for a variable velocity, which is useful for coupling mass transfer to a velocity field that comes from other conservation of momentum simulations (i.e., Navier-Stokes modules).

Residual Formulation ➔ (See GAdvection (DGAdvection) for residual formulation)

Additional Computations ➔ The velocity vector variable from GAdvection (DGAdvection) is overridden with the corresponding variable values for each velocity vector component.

Usage

Generally used in conjunction with a TimeDerivative or VariableCoefTimeDerivative kernel.

Must always have the 'G' and 'DG' kernels matching by name and have matching parameters (i.e., same velocities). 'G' kernels are placed under the [Kernels] block and 'DG' kernels are placed under the [DGKernels] block in the input files.

Below example is for conservative advection of u with a variable velocity vector constructed from vector components.

```
[Kernels]
    [./g_adv_u]
        type = GConcentrationAdvection
        variable = u
        ux = vel_x
        uy = vel_y
        uz = vel_z
    [../]
[]
```

```
[DGKernels]
    [./dg_adv_u]
        type = DGConcentrationAdvection
        variable = u
        ux = vel_x
        uy = vel_y
        uz = vel_z
    [../]
[]
```

GNernstPlanckDiffusion (DGNernstPlanckDiffusion)

Inheritance:

GVarPoreDiffusion ➔ GVariableDiffusion

DGVarPoreDiffusion ➔ DGVariableDiffusion

Special Notes ➔ This kernel acts on a concentration of ions in solution, but couples that diffusion with an electric potential gradient to base the diffusive flux off of.

NOTE: This kernel ONLY includes diffusive flux based on an electric potential gradient. It should be combined with another diffusion kernel (such as GVariableDiffusion (DGVariableDiffusion)) or similar to represent the full flux term.

Notes ➔ This set of kernels calculates the diffusive flux of ions caused by an electric potential gradient. If there is no electric potential gradient, then this term does not introduce any flux. Users must also provide the valence of the ions (which will help to dictate both the magnitude and direction of ionic flux). This kernel includes 2 constants (Faraday's constant and the Gas Law constant) that the user may provide other values for other than their defaults. This may be useful for converting this kernel to use a different unit basis in the overall ion mass balance.

Residual Formulation

Due to the complexity of forming transport physics in the DG system of kernels, the exact formulations are not provided here. Instead, only the "strong form" of the physics is provided for all DG and associated kernels.

Strong Form Residual (mass / volume / time) = $-\varphi \cdot \nabla \cdot \left(\frac{zF}{RT}\mathbf{D} \cdot \varepsilon \cdot c\nabla\phi\right)$

φ = MOOSE variable test function (for FE formulation)

D = diffusion tensor (m^2/s)

➔ See GVariableDiffusion (DGVariableDiffusion) for more notes

c = a conserved ion quantity per volume (mass / volume)

ε = porosity of the domain

z = valance of the ions (negative values for anions, positive values for cations)

T = coupled temperature variable (in K)

ϕ = electric potential variable in domain (units depend on F and R)

[Default units would be V or J/C]

F = Faraday's constant (default = 96,485.3 C/mol)

R = Gas Law constant (default = 8.314462 J/K/mol)

Additional Computations ➔ None

Usage

Generally used in conjunction with a TimeDerivative or VariableCoefTimeDerivative kernel.

Must always have the 'G' and 'DG' kernels matching by name and have matching parameters. 'G' kernels are placed under the [Kernels] block and 'DG' kernels are placed under the [DGKernels] block in the input files.

Below example is for diffusion of positive ions in a porous media with the Nernst-Planck diffusion added to the standard diffusion. The porosity value is 0.5 and each kernel is coupled with variable for diffusivity (D), temperature (temp), and electric potential (phi).

```
[Kernels]
    [./g_diff_u]
        type = GVarPoreDiffusion
        variable = u
        porosity = 0.5
        Dx = D
        Dy = D
        Dz = D
    [../]
    [./g_np_diff_u]
        type = GNernstPlanckDiffusion
        variable = u
        porosity = 0.5
        temperature = temp
        electric_potential = phi
        valence = 1
        Dx = D
        Dy = D
        Dz = D
    [../]
[]

[DGKernels]
    [./dg_diff_u]
        type = DGVarPoreDiffusion
        variable = u
        porosity = 0.5
```

```
        Dx = D
        Dy = D
        Dz = D
    [../]
    [./dg_np_diff_u]
        type = DGNernstPlanckDiffusion
        variable = u
        porosity = 0.5
        temperature = temp
        electric_potential = phi
        valence = 1
        Dx = D
        Dy = D
        Dz = D
    [../]
[]
```

GNSMomentumAdvection (DGNSMomentumAdvection)

Inheritance: GNSMomentumAdvection ➔ GConcentrationAdvection

DGNSMomentumAdvection ➔ DGConcentrationAdvection

Special Notes

➔ All kernels prefixed with 'G' and 'DG' must be used together (Except for Boundary Conditions). Combined they fully describe 1 piece of transport physics. You CANNOT use either 'G' or 'DG' kernels without also invoking the other. If you do, your simulation will likely result in errors in conservation of mass or energy.

➔ Any variable in the MOOSE system that uses 'DG' kernels must also either use MONOMIAL or L2_LAGRANGE shape functions (i.e., family). These are invoked when creating the variables under the variables block (see below).

➔ Specifically for Navier-Stokes equations with DG methods. Users should use 'SECOND' order functions for each velocity term.

➔ Navier-Stokes equations also require the usage of a pressure variable. That variable MUST BE 'FIRST' order 'LAGRANGE' functions for improved stability and evaluations of boundary pressure.

```
[Variables]
    [./vel_x]
        order = SECOND
        family = MONOMIAL
    [../]
    [./vel_y]
        order = SECOND
        family = MONOMIAL
```

```
        [../]
        [./pressure]
            order = FIRST
            family = LAGRANGE
        [../]
[]
```

Notes ➔ This set of kernels formulates the Discontinuous Galerkin form for conservative momentum advection in a simulation domain. These kernels act on velocity components in a piece-wise manner. Other sections of the Navier-Stokes equations are satisfied from other existing kernels such as GVariableDiffusion (DGVariableDiffusion) and VectorCoupledGradient.

Residual Formulation

Due to the complexity of forming Navier-Stokes transport physics in the DG system of kernels, the exact formulations are not provided here. Instead, only the "strong form" of the physics is provided for all DG and associated kernels.

Strong Form Residual (momentum / volume / time) = $\varphi \cdot \nabla \cdot (\rho \mathbf{v} \cdot u_i)$

φ = MOOSE variable test function (for FE formulation)

v = velocity vector (length / time)

[NOTE: this vector is built-up from each u_i velocity variable component]

ρ = density variable (mass / volume)

u_i = velocity variable in the i-th direction (where i = x, y, or z)

Additional Computations ➔ None

Usage

This kernel set is designed specifically for the use in the Navier-Stokes equations to formulate a DG implementation and solution to that system of equations. By itself, this kernel set does not do much. It should be combined with a VariableCoefTimeDerivative, VectorCoupledGradient, and GVariableDiffusion (DGVariableDiffusion) to complete the Navier-Stokes residuals. In addition, you must use these in conjunction with VectorCoupledGradient and/or DivergenceFreeCondition objects that act on the pressure variable to resolve the continuity equation portion of the Navier-Stokes equation set.

Must always have the 'G' and 'DG' kernels matching by name and have matching parameters (i.e., same velocities). 'G' kernels are placed under the [Kernels] block and 'DG' kernels are placed under the [DGKernels] block in the input files.

Below is an example for how to setup the momentum balance from Navier-Stokes for only the conservation of momentum in the x-direction. (**NOTE**: This example does not show the continuity equation. See the example in VectorCoupledGradient and/or DivergenceFreeCondition for that formulation).

```
[Kernels]
        # rho * d(vel_x)/dt
        [./x_dot]
                type = VariableCoefTimeDerivative
                variable = vel_x
                coupled_coef = rho
        [../]
        # -grad(P)_x
        [./x_press]
                type = VectorCoupledGradient
                variable = vel_x
                coupled = pressure
                vx = 1
        [../]
        # Div*(mu*grad(vel_x))
        [./x_gvis]
                type = GVariableDiffusion
                variable = vel_x
                Dx = mu
                Dy = mu
                Dz = mu
        [../]
        # Div*(rho*vel*vel_x)
        [./x_gadv]
                type = GNSMomentumAdvection
                variable = vel_x
                this_variable = vel_x
                density = rho
                ux = vel_x
                uy = vel_y
                uz = vel_z
        [../]
[]

[DGKernels]
        # Div*(mu*grad(vel_x))
        [./x_dgvis]
                type = DGVariableDiffusion
                variable = vel_x
                Dx = mu
                Dy = mu
                Dz = mu
        [../]
        # Div*(rho*vel*vel_x)
        [./x_dgadv]
                type = DGNSMomentumAdvection
                variable = vel_x
                this_variable = vel_x
```

```
        density = rho
        ux = vel_x
        uy = vel_y
        uz = vel_z
    [../]
[]
```

GNSViscousVelocityDivergence (DGNSViscousVelocityDivergence)

Inheritance: GNSMomentumAdvection ➔ GConcentrationAdvection

DGNSMomentumAdvection ➔ DGConcentrationAdvection

Special Notes

➔ All kernels prefixed with 'G' and 'DG' must be used together (Except for Boundary Conditions). Combined they fully describe 1 piece of transport physics. You CANNOT use either 'G' or 'DG' kernels without also invoking the other. If you do, your simulation will likely result in errors in conservation of mass or energy.

➔ Any variable in the MOOSE system that uses 'DG' kernels must also either use MONOMIAL or L2_LAGRANGE shape functions (i.e., family). These are invoked when creating the variables under the variables block (see below).

➔ Specifically for Navier-Stokes equations with DG methods. Users should use 'SECOND' order functions for each velocity term.

➔ Navier-Stokes equations also require the usage of a pressure variable. That variable MUST BE 'FIRST' order 'LAGRANGE' functions for improved stability and evaluations of boundary pressure.

```
[Variables]
    [./vel_x]
        order = SECOND
        family = MONOMIAL
    [../]
    [./vel_y]
        order = SECOND
        family = MONOMIAL
    [../]
    [./pressure]
        order = FIRST
        family = LAGRANGE
    [../]
[]
```

Notes ➔ This set of kernels formulates the Discontinuous Galerkin form for conservative momentum advection in a simulation domain. These kernels act on velocity components in a piece-wise manner. Other sections of the Navier-Stokes equations are satisfied from other existing kernels such as GVariableDiffusion (DGVariableDiffusion) and VectorCoupledGradient.

Residual Formulation

Due to the complexity of forming Navier-Stokes transport physics in the DG system of kernels, the exact formulations are not provided here. Instead, only the "strong form" of the physics is provided for all DG and associated kernels.

Strong Form Residual (momentum / volume / time) = $\varphi \cdot \frac{1}{3}\mu\nabla(\nabla \cdot \mathbf{v})$

φ = MOOSE variable test function (for FE formulation)

v = velocity vector (length / time)

[NOTE: this vector is built-up from each u_i velocity variable component]

μ = viscosity variable (mass / length / time)

NOTE: When doing Navier-Stokes in a piece-wise fashion, each velocity component is its own non-linear variable and each will have an instance of this kernel. When doing so, the above residual formulation breaks down into the following:

Residual = $\frac{1}{3}\mu \cdot \left[\mathbf{D}_{ix} \cdot \nabla u_x + \mathbf{D}_{iy} \cdot \nabla u_y + \mathbf{D}_{iz} \cdot \nabla u_z\right]$

$\mathbf{D}_{ix}$ = Tensor matrix whose entries are all 0 expect D(i,x), which has value of 1.

$\mathbf{D}_{iy}$ = Tensor matrix whose entries are all 0 expect D(i,y), which has value of 1.

$\mathbf{D}_{iz}$ = Tensor matrix whose entries are all 0 expect D(i,z), which has value of 1.

i = Cartesian direction of the velocity variable that this kernel acts on

Additional Computations ➔ None

Usage

This kernel set is designed specifically for the use in the Navier-Stokes equations to formulate a DG implementation and solution to that system of equations. By itself, this kernel set does not do much. It should be combined with a VariableCoefTimeDerivative, VectorCoupledGradient, and GVariableDiffusion (DGVariableDiffusion) to complete the Navier-Stokes residuals. In addition, you must use these in conjunction with VectorCoupledGradient and/or DivergenceFreeCondition objects that act on the pressure variable to resolve the continuity equation portion of the Navier-Stokes equation set.

Must always have the 'G' and 'DG' kernels matching by name and have matching parameters (i.e., same velocities). 'G' kernels are placed under the [Kernels] block and 'DG' kernels are placed under the [DGKernels] block in the input files.

Below is an example for how to setup the momentum balance from Navier-Stokes for only the conservation of momentum in the x-direction. (**NOTE**: This example does not show the continuity equation. See the example in VectorCoupledGradient and/or DivergenceFreeCondition for that formulation).

NOTE: This kernel is technically ONLY needed for compressible flow.

```
[Kernels]
        # rho * d(vel_x)/dt
        [./x_dot]
                type = VariableCoefTimeDerivative
                variable = vel_x
                coupled_coef = rho
        [../]
        # -grad(P)_x
        [./x_press]
                type = VectorCoupledGradient
                variable = vel_x
                coupled = pressure
                vx = 1
        [../]
        # Div*(mu*grad(vel_x))
        [./x_gvis]
                type = GVariableDiffusion
                variable = vel_x
                Dx = mu
                Dy = mu
                Dz = mu
        [../]
        # Div*(rho*vel*vel_x)
        [./x_gadv]
                type = GNSMomentumAdvection
                variable = vel_x
                this_variable = vel_x
                density = rho
                ux = vel_x
                uy = vel_y
                uz = vel_z
        [../]
        # (1/3)*mu*(grad(Div*vel)_x)
        [./x_gvisdiv]
                type = GNSViscousVelocityDivergence
                variable = vel_x
                this_variable = vel_x
                viscosity = mu
                ux = vel_x
                uy = vel_y
                uz = vel_z
        [../]

[]
[DGKernels]
        # Div*(mu*grad(vel_x))
        [./x_dgvis]
                type = DGVariableDiffusion
```

```
        variable = vel_x
        Dx = mu
        Dy = mu
        Dz = mu
    [../]
    # Div*(rho*vel*vel_x)
    [./x_dgadv]
        type = DGNSMomentumAdvection
        variable = vel_x
        this_variable = vel_x
        density = rho
        ux = vel_x
        uy = vel_y
        uz = vel_z
    [../]
    # (1/3)*mu*(grad(Div*vel)_x)
    [./x_dgvisdiv]
        type = DGNSViscousVelocityDivergence
        variable = vel_x
        this_variable = vel_x
        viscosity = mu
        ux = vel_x
        uy = vel_y
        uz = vel_z
    [../]

[]
```

GPhaseThermalConductivity (DGPhaseThermalConductivity)

Inheritance

GPhaseThermalConductivity ➔ GThermalConductivity

DGPhaseThermalConductivity ➔ DGThermalConductivity

Special Notes ➔ (See GThermalConductivity (DGThermalConductivity) for special notes)

Notes ➔ This kernel uses the same residuals from GThermalConductivity (DGThermalConductivity), but scales the conductivity transport of energy by a volume fraction that represents the fraction of the domain the represents the specific phase holding the energy. For instance, in the case of porous flow, you may have 1 energy density in the solids and another energy density in the fluid phases. Thus, the volume fraction would be representative of the solids-to-total volume or the fluid-to-total volume ratio.

Residual Formulation

Due to the complexity of forming transport physics in the DG system of kernels, the exact formulations are not provided here. Instead, only the "strong form" of the physics is provided for all DG and associated kernels.

Strong Form Residual (J / m^3 / s) = $-\varphi \cdot \nabla \cdot (\mathbf{K} \cdot f_v \cdot \nabla T)$

φ = MOOSE variable test function (for FE formulation)

K = thermal conductivity tensor (W/m/K)

➔ See GThermalConductivity (DGThermalConductivity) for more notes

T = temperature of the phase (K)

f_v = volume fraction of the phase (e.g., volume voids / total volume)

Additional Computations ➔ None

Usage

NOTE: This kernel acts on an energy variable, but is coupled to the temperature variable

Generally used in conjunction with a TimeDerivative or VariableCoefTimeDerivative kernel.

Must always have the 'G' and 'DG' kernels matching by name and have matching parameters (i.e., same velocities). 'G' kernels are placed under the [Kernels] block and 'DG' kernels are placed under the [DGKernels] block in the input files.

Below example is for conservative thermal conductivity of fluid phase energy (Ef) in a system with a porosity of 0.5 and an isotropic conductivity behavior in x, y, and z directions with a variable conductivity (Kg).

```
[Kernels]
    [./g_cond_T]
        type = GPhaseThermalConductivity
        variable = Ef
        temperature = T
        volume_frac = 0.5
        Dx = Kg
        Dy = Kg
        Dz = Kg
    [../]
[]

[DGKernels]
    [./dg_cond_T]
        type = DGPhaseThermalConductivity
        variable = Ef
        temperature = T
        volume_frac = 0.5
        Dx = Kg
```

```
        Dy = Kg
        Dz = Kg
    [../]
[]
```

GPoreConcAdvection (DGPoreConcAdvection)

Inheritance:

GPoreConcAdvection ➔ GConcentrationAdvection

DGPoreConcAdvection ➔ DGConcentrationAdvection

Special Notes ➔ (See GConcentrationAdvection (DGConcentrationAdvection) for special notes)

Notes ➔ This kernel set uses the residuals from GConcentrationAdvection (DGConcentrationAdvection), but scales the advective flow by a volume fraction or a porosity variable to represent the fraction of the total domain that is open for advective transport. This is most commonly used for the advective flow of mass or energy in packed beds or porous media.

Residual Formulation

Due to the complexity of forming transport physics in the DG system of kernels, the exact formulations are not provided here. Instead, only the "strong form" of the physics is provided for all DG and associated kernels.

Strong Form Residual (mass / volume / time) = $\varphi \cdot \nabla \cdot (\mathbf{v} \cdot \varepsilon u)$

φ = MOOSE variable test function (for FE formulation)

v = velocity vector (m/s)

u = a conserved quantity per volume (mass / volume) or (energy / volume)

ε = volume fraction or porosity variable for the void spaces

Additional Computations ➔ None

Usage

Generally used in conjunction with a TimeDerivative or VariableCoefTimeDerivative kernel.

Must always have the 'G' and 'DG' kernels matching by name and have matching parameters (i.e., same velocities). 'G' kernels are placed under the [Kernels] block and 'DG' kernels are placed under the [DGKernels] block in the input files.

Below example is for conservative advection of fluid energy (Ef) with a variable velocity vector constructed from vector components and a variable porosity (eps).

```
[Kernels]
    [./g_adv_E]
        type = GPoreConcAdvection
```

```
            variable = Ef
            porosity = eps
            ux = vel_x
            uy = vel_y
            uz = vel_z
        [../]
[]

[DGKernels]
        [./dg_adv_E]
            type = DGPoreConcAdvection
            variable = Ef
            porosity = eps
            ux = vel_x
            uy = vel_y
            uz = vel_z
        [../]
[]
```

GThermalConductivity (DGThermalConductivity)

Inheritance

GThermalConductivity ➔ GVariableDiffusion

DGThermalConductivity ➔ DGVariableDiffusion

Special Notes ➔ (See GAnisotropicDiffusion (DGAnisotropicDiffusion) for special notes)

➔ While the physics of thermal conductivity and mass diffusion are extraordinarily similar, there is a major difference in their usages. This is because an energy balance's variable is energy density and not temperature. However, the conductivity of energy is driven entirely by temperature, which itself is a variable. Thus, this kernel is not a function of the energy variable.

Notes ➔ This kernel uses the same basic strong form residuals from GVariableDiffusion (DGVariableDiffusion), which is also the same as GAnisotropicDiffusion (DGAnisotropicDiffusion), but acts on a variable that is not the primary variable. Thus, this kernel did have to be specially constructed for our usage. Here, the diffusion tensor (**D**) is replaced with a conductivity tensor (**K**) and the variable gradient is no longer with respect to the conserved quantity, but the temperature (T) of the media.

Residual Formulation

Due to the complexity of forming transport physics in the DG system of kernels, the exact formulations are not provided here. Instead, only the "strong form" of the physics is provided for all DG and associated kernels.

Strong Form Residual (J / m^3 / s) = $-\varphi \cdot \nabla \cdot (\mathbf{K} \cdot \nabla T)$

φ = MOOSE variable test function (for FE formulation)

K = thermal conductivity tensor (W/m/K)

➔ See GThermalConductivity (DGThermalConductivity) for more notes

T = temperature of the phase (K)

Additional Computations ➔ None

Usage

Generally used in conjunction with a TimeDerivative or VariableCoefTimeDerivative kernel.

Must always have the 'G' and 'DG' kernels matching by name and have matching parameters (i.e., same velocities). 'G' kernels are placed under the [Kernels] block and 'DG' kernels are placed under the [DGKernels] block in the input files.

Below example is for conservative thermal conductivity of fluid phase energy (Ef) in a system with an isotropic conductivity behavior in x, y, and z directions with a variable conductivity (Kg).

```
[Kernels]
    [./g_cond_T]
        type = GThermalConductivity
        variable = Ef
        temperature = T
        Dx = Kg
        Dy = Kg
        Dz = Kg
    [../]
[]

[DGKernels]
    [./dg_cond_T]
        type = DGThermalConductivity
        variable = Ef
        temperature = T
        Dx = Kg
        Dy = Kg
        Dz = Kg
    [../]
[]
```

GVarPoreDiffusion (DGVarPoreDiffusion)

Inheritance:

GVarPoreDiffusion ➔ GVariableDiffusion

DGVarPoreDiffusion ➔ DGVariableDiffusion

Special Notes ➔ (See GAnisotropicDiffusion (DGAnisotropicDiffusion) for special notes)

Notes ➔ This set of kernels uses the same residual calculation from GVariableDiffusion (DGVariableDiffusion), but scales the diffusive transport flux by the system porosity. Thus, representing only the diffusion that occurs through the pore spaces of a media. That porosity is allowed to be either a constant or a variable itself.

Residual Formulation

Due to the complexity of forming transport physics in the DG system of kernels, the exact formulations are not provided here. Instead, only the "strong form" of the physics is provided for all DG and associated kernels.

Strong Form Residual (mass / volume / time) = $-\varphi \cdot \nabla \cdot (\mathbf{D} \cdot \varepsilon \nabla u)$

φ = MOOSE variable test function (for FE formulation)

D = diffusion tensor (m^2/s)

➔ See GVariableDiffusion (DGVariableDiffusion) for more notes

u = a conserved quantity per volume (mass / volume)

ε = porosity of the domain

Additional Computations ➔ None

Usage

Generally used in conjunction with a TimeDerivative or VariableCoefTimeDerivative kernel.

Must always have the 'G' and 'DG' kernels matching by name and have matching parameters (i.e., same velocities). 'G' kernels are placed under the [Kernels] block and 'DG' kernels are placed under the [DGKernels] block in the input files.

Below example is for conservative diffusion of u with a variable, anisotropic diffusion in x, y, and z, as well as a constant porosity of 0.5.

```
[Kernels]
    [./g_diff_u]
        type = GVarPoreDiffusion
        variable = u
        porosity = 0.5
        Dx = Diff_x
        Dy = Diff_y
        Dz = Diff_z
    [../]
[]

[DGKernels]
```

```
        [./dg_diff_u]
            type = DGVarPoreDiffusion
            variable = u
            porosity = 0.5
            Dx = Diff_x
            Dy = Diff_y
            Dz = Diff_z
        [../]
[]
```

GVariableDiffusion (DGVariableDiffusion)

Inheritance:

GVariableDiffusion ➔ GAnisotropicDiffusion

DGVariableDiffusion ➔ DGAnisotropicDiffusion

Special Notes ➔ (See GAnisotropicDiffusion (DGAnisotropicDiffusion) for special notes)

Notes ➔ This set of kernels uses the same residual calculation from GAnisotropicDiffusion (DGAnisotropicDiffusion), but overrides the diffusion tensor (**D**) with given variables for diffusion in x, y, and z directions. From those three variables, the diffusion tensor is constructed

NOTE: The diffusion tensor MUST use coupled variables (i.e., Dz, Dx, and Dy CANNOT be constants reals in the input file). This is because of how MOOSE handles DG variables and kernels for the neighboring cells. If you provide a constant real value in the input file, then the code will SegFault.

Residual Formulation ➔ (See GAnisotropicDiffusion (DGAnisotropicDiffusion) for residual)

Additional Computations ➔ The diffusion tensor (**D**) is overridden with diffusion variable components D_x, D_y, and D_z.

Usage

Generally used in conjunction with a TimeDerivative or VariableCoefTimeDerivative kernel.

Must always have the 'G' and 'DG' kernels matching by name and have matching parameters (i.e., same velocities). 'G' kernels are placed under the [Kernels] block and 'DG' kernels are placed under the [DGKernels] block in the input files.

Below example is for conservative diffusion of u with a variable, isotropic diffusion.

```
[Kernels]
    [./g_diff_u]
        type = GVariableDiffusion
        variable = u
        Dx = Diff
        Dy = Diff
        Dz = Diff
```

```
        [../]
[]

[DGKernels]
        [./dg_diff_u]
                type = DGVariableDiffusion
                variable = u
                Dx = Diff
                Dy = Diff
                Dz = Diff
        [../]
[]
```

InhibitedArrheniusReaction

Inheritance ➔ ArrheniusReaction

Notes ➔ This kernel has all the same residuals from ArrheniusReaction, but changes the effective forward and reverse rates by dividing each of those rates by non-linear variables that will represent the inhibition term for each half of the reaction. The inhibition term can be given as a constant, but it is more useful to calculate that term in another kernel, or set of kernels, and couple it to this kernel. Some currently available inhibition kernels include LangmuirInhibition, PairedLangmuirInhibition, and InhibitionProducts.

Residual Formulation

(See ArrheniusReaction for residual formulation)

Additional Computations

$$k_f = \frac{A_f T^{\beta_f} exp\left[-\frac{E_f}{RT}\right]}{R_f} \qquad k_r = \frac{A_r T^{\beta_r} exp\left[-\frac{E_r}{RT}\right]}{R_r}$$

- The units for A_f and A_r must have the same units as k_f and k_r
- The units for E_f and E_r must be J/mol
  - The above parameters come from ArrheniusReaction
- R_f and R_r are non-linear variables that represent an inhibition term

Usage

Generally used in conjunction with a TimeDerivative or VariableCoefTimeDerivative kernel.

Code below creates a residual for species A for the following reaction:

A + B → C

That reaction rate is inhibited by a variable R, which needs its own kernel elsewhere.

```
[./rxn]
        type = InhibitedArrheniusReaction
        variable = A
```

this_variable = A
temperature = T
scale = -1
forward_activation_energy = 5E4
forward_pre_exponential = 25
forward_inhibition = R
reactants = 'A B'
reactant_stoich = '1 1'
products = 'C'
product_stoich = '1'
[../]

InhibitedArrheniusReactionEnergyTransfer

Inheritance ➔ InhibitedArrheniusReaction

Notes ➔ Although this kernel is used in energy balances and not mass balances, the kernels are mathematically very similar. Thus, this kernel uses the residual computed from InhibitedArrheniusReaction, but applies the enthalpy (ΔH) and 2 unit conversion factors to modify the scaling parameter of InhibitedArrheniusReaction. The net effect is to convert the existing reaction rate function from a mass term to an energy term. In addition, the forward and reverse rates are divided by the inhibition terms, which can be passed as non-linear variables or given constant values (see InhibitedArrheniusReaction for more details).

Residual Formulation

The baseline residual comes from InhibitedArrheniusReaction. In this formulation, we will represent that portion of the residual as r (i.e., the rate of the reaction) with a scaling parameter of 1. See InhibitedArrheniusReaction for more details on scaling parameters. Then, this kernel uses that reaction residual (r) and applies a custom scaling based on ΔH and other unit conversions.

Residual (J/m^3/s) = $-\Delta H \cdot f_v \cdot A_s \cdot r$

f_v = volume fraction (i.e., volume of solids per total volume)

➔ Can be set to 1 if no such conversion is needed

A_s = Specific reaction area per volume of solids (for a surface reaction)

➔ Can be set to 1 if no such conversion is needed

ΔH = enthalpy of the reaction (J/mol)

r = reaction rate (mol/m^3/s) or (mol/m^2/s) [depends on reaction type]

➔ Use f_v and A_s to convert units such that the units for the residual return as energy rate per total volume (J/m^3/s)

Additional Computations

As noted above, the scaling parameter from the reaction is overridden to always be 1.

Usage

Generally used in conjunction with an energy balance set of kernels to add the physics of energy changes caused by a chemical reaction in the system. The usage also includes terms to convert the volumetric or per reactive area terms into the proper energy density residual.

Code below creates a residual for energy exchange via the following surface reaction:

A + B → C

That reaction rate is inhibited by a variable R, which needs its own kernel elsewhere.

```
[./rxn]
    type = ArrheniusReactionEnergyTransfer
    variable = E                # E = Energy density variable
    this_variable = E
    temperature = T
    volume_frac = s_frac        # Volume of solids per Total Volume
    specific_area = 8.6E7       # Reaction area of solid per volume of solid
    forward_activation_energy = 5E4
    forward_pre_exponential = 25
    forward_inhibition = R
    enthalpy = -3.9E5
    reactants = 'A B'
    reactant_stoich = '1 1'
    products = 'C'
    product_stoich = '1'
[../]
```

InhibitionProducts

Inheritance → Kernel (i.e., the MOOSE Kernel base system)

Notes → This kernel is used to create a residual to represent an inhibition term for InhibitedArrheniusReaction. The inhibition term is its own non-linear variable and formulated as a product function of other inhibition terms. As such, this kernel must be used in conjunction with a set of other inhibition variables, whose values are fulfilled by other inhibition models such as LangmuirInhibition and PairedLangmuirInhibition.

Residual Formulation

The residual is formed from a list of coupled inhibition terms and a list of powers that is applied to each term.

Residual (-) = $-\varphi \cdot \left(\prod_{\forall i} R_i^{p_i}\right)$

φ = MOOSE variable test function (for FE formulation)

R_i = i-th inhibition variable

p_i = power that the i-th inhibition term is raised to

Additional Computations ➔ None

Usage

MUST be used in conjunction with a "Reaction" kernel from the MOOSE base system of kernels in order to fully describe the inhibition term (R) for a reaction.

Code below creates an inhibition term that is a function of inhibition terms RA and RB. Those other inhibition terms must be defined elsewhere in the input file.

```
[./R_equ]
        type = Reaction
        variable = R
[../]
[./R_prod]
        type = InhibitionProducts
        variable = R
        coupled_list = 'RA RB'
        power_list = '1 1'
[../]
```

LangmuirInhibition

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel is used to create a residual to represent an inhibition term for InhibitedArrheniusReaction. The inhibition term is its own non-linear variable and is used in conjunction with other inhibition terms or a "Reaction" kernel to finish the full definition of the term in the MOOSE residual functions (see **Usage** below).

Residual Formulation

The residual is formed from a list of coupled concentration terms and a coupled temperature. The inhibition term does not have any units.

Residual (-) = $-\varphi \cdot (1 + \sum_{\forall i} K_i C_i)$

φ = MOOSE variable test function (for FE formulation)

C_i = concentration variable for species i

K_i = Langmuir coefficient for the i-th species in the list

Additional Computations

The Langmuir coefficients are calculated from the same type of expression as the Arrhenius reaction term.

$$K_i = A_i T^{\beta_i} exp\left[-\frac{E_i}{RT}\right]$$

- The units for A_i must have the same units as K_i
- K_i must have units of inverse concentration C_i
- The units for E_i must be J/mol
- β_i are powers on temperature (which default to 0)

Usage

MUST be used in conjunction with a "Reaction" kernel from the MOOSE base system of kernels in order to fully describe the inhibition term (R) for a reaction.

Code below creates an inhibition term that is a function of concentrations for A and B. The coefficients (K) in the inhibition term are each set to a value of 1. This is accomplished by providing 0s for beta and activation energy terms, which removes the temperature dependence for those parameters.

```
[./R_equ]
        type = Reaction
        variable = R
[../]
[./R_lang]
        type = LangmuirInhibition
        variable = R
        temperature = T
        coupled_list = 'A B'
        pre_exponentials = '1 1'
        betas = '0 0'
        activation_energies = '0 0'
[../]
```

MaterialBalance

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel creates a residual for a generic mass/material balance on a set of non-linear variables. The point of this kernel is to reduce complexities in solving chemical reaction problems by closing the system of equations with a single kernel instead of having to create several sets of rate or equilibrium kernels for all chemical reactions. For instance, in the case of a simple adsorption reaction, the rate of adsorption is a function of the concentration of available sites. Those available sites will decrease overtime as adsorption continues. You can account for this decrease in site availability by invoking a set of rate kernels for both the adsorbed species and the surface sites. However, you can also just invoke rates for the adsorbed species, then close the system by invoking this material balance kernel to specific that the amount of available sites is a function of the initial site availability minus the amount of sites used by adsorption.

Residual Formulation

Residual (same units for all coupled variables) = $\varphi \cdot (C_T - \sum_{\forall i} w_i C_i)$

φ = MOOSE variable test function (for FE formulation)

C_T = total concentration variable

C_i = concentration variable for species i

w_i = weight factor for amount of Ci that contributes to CT

Additional Computations ➔ None

Usage

Generally, this kernel is used in combination with other reaction or transport kernels on the variables of interest. You use this kernel to completely solve for 1 other variable in your domain that you do not have any other kernels for.

For example, consider an adsorption process wherein we have solved for the concentration of the adsorbate (C) and adsorbed concentration (q) with other sets of kernels. The rates of adsorption depends on a site density variable (S). We know that the maximum or total site density is Smax, so we can use MaterialBalance to solve for S. Here, we state that 1 mole of q adsorbed uses 2 reaction sites of Smax and S represents 1 mole of total sites.

```
[./site_dens]
        type = MaterialBalance
        variable = S
        this_variable = S
        total_material = Smax
        coupled_list = 'S q'
        weights = '1 2'
[../]
```

MicroscaleCoefTimeDerivative

Inheritance ➔ TimeDerivative

Special Notes ➔ All kernels preceded with a 'Microscale' prefix are experimental kernels developed specifically for setting up and solving a hybrid Finite Difference/Finite Element problem involving micro-scale subdomains in a macro-scale domain. The general idea is to discretize the micro-scale domain with finite differences, then use that discretization to create a finite element weak form in MOOSE. This then allows us to represent the residuals in MOOSE as a set of ODEs instead of needing to explicitly create small subdomains or MultiApp grids in MOOSE to solve micro-scale physics alongside the macro-scale domain.

Mathematical Expression of the Microscale (not all physics shown)

$$r^d R \frac{\partial u}{\partial t} = \frac{\partial}{\partial r}\left(r^d D \frac{\partial u}{\partial r}\right) - r^d K \frac{\partial v}{\partial t} + r^d \lambda \sum w_i r_i$$

r = dimension of the micro-scale (i.e., radius of a pellet)

d = conversion factor between coordinate systems for micro-scale

d = 0 ➔ z-cartesian

d = 1 ➔ r-cylindrical

d = 2 ➔ r-spherical

R = time derivative coefficient

D = diffusion coefficient

K = coupled time derivative coefficient

λ = scale factor for coupled sum

w_i = weight factors in coupled sum

r_i = variable in coupled sum (generally represents a reaction rate)

u = primary variable (i.e., pore-space concentration)

v = coupled variable or physics (i.e., surface rxn)

➔ All 'Microscale' kernels require the following parameters…

'micro_length' (i.e., length of the micro-scale domain, such as pellet radius)

'num_nodes' (i.e., the number of nodes to divide the micro-scale into)

'coord_id' (i.e., the conversion factor that is 0 for cartesian, 1 for cylindrical pellets, or 2 for spherical pellets)

The above parameters can be defined individually in each kernel or defined as GlobalParams, which is generally more appropriate.

```
[GlobalParams]
    micro_length = 1
    num_nodes = 3
    coord_id = 2
[]
```

Notes ➔ This kernel is for the hybrid FD/FE representation of the time derivatives at all nodal locations in the micro-scale. Every node in the micro-scale requires an instance of this kernel (i.e., if you discretize the micro-scale into 10 nodes, you need 10 of these kernels).

Residual Formulation

Residual (mass / volume / time) = $\varphi \cdot r_l^d R_l \frac{\partial(u_l)}{\partial t}$

φ = MOOSE variable test function (for FE formulation)

r_l = nodal position in the micro-scale that this kernel acts on

R_l = nodal time coefficient for this kernel

l = node id for this position

u_l = conserved quantity variable at this nodal position

d = coordinate id (0 = z-cartesian, 1 = r-cylindrical, 2 = r-spherical)

Additional Computations ➔ None

Usage

All 'Microscale' kernels must have an instance of themselves for each node you choose to divide the micro-scale into. Generally, you would use this in conjunction with a series of other 'Microscale' kernels with all the same parameters, but at different nodes.

Each different node requires a different nodal variable to be declared. For instance, if your total micro-scale quantity is being represented by u, and if you divide the micro-scale into 3 nodes, then your variables might be named u0, u1, and u2 (i.e., value of u at node 0, value of u at node 1, and value of u and node 2).

Code example below is for the time derivatives of a micro-scale problem inside spherical pellets whose micro-scale domain was broken up into 3 nodes (0, 1, and 2). All time coefficients are assumed 1 at each nodal position.

```
[Kernels]
    [./u0_dot]
        type = MicroscaleCoefTimeDerivative
        variable = u0
        nodal_time_coef = 1
        node_id = 0
    [../]
    [./u1_dot]
        type = MicroscaleCoefTimeDerivative
        variable = u1
        nodal_time_coef = 1
        node_id = 1
    [../]
    [./u2_dot]
        type = MicroscaleCoefTimeDerivative
        variable = u2
        nodal_time_coef = 1
        node_id = 2
    [../]
[]
```

MicroscaleCoupledCoefTimeDerivative

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Special Notes ➔ (See MicroscaleCoefTimeDerivative for special notes)

Notes ➔ This kernel is for introducing a coupled time derivative for mass transfer within a specified particle nodal location for the hybrid FD/FE method. Just like the primary micro-scale variable, all coupled micro-scale variables must also have a variable for each nodal position.

Residual Formulation

Residual (mass / volume / time) = $\varphi \cdot r_l^d K_l \frac{\partial(v_l)}{\partial t}$

φ = MOOSE variable test function (for FE formulation)

r_l = nodal position in the micro-scale that this kernel acts on

K_l = nodal time coefficient for the coupled time derivative

l = node id for this position

v_l = coupled variable at this nodal position

d = coordinate id (0 = z-cartesian, 1 = r-cylindrical, 2 = r-spherical)

Additional Computations ➔ None

Usage

All 'Microscale' kernels must have an instance of themselves for each node you choose to divide the micro-scale into. Generally, you would use this in conjunction with a series of other 'Microscale' kernels with all the same parameters, but at different nodes.

Each different node requires a different nodal variable to be declared. For instance, if your total micro-scale quantity is being represented by u, and if you divide the micro-scale into 3 nodes, then your variables might be named u0, u1, and u2 (i.e., value of u at node 0, etc).

The coupled time derivatives are kernels that act on your primary variable (in this example: u0, u1, and u2), but are coupled to the time derivatives of other variables in the micro-scale subdomain (in this example: v0, v1, and v2). MAKE SURE the correct variables are coupled together (e.g., in this example: u0 couples to v0, not any other v).

Code example below is for the coupled time derivatives of a micro-scale problem inside spherical pellets whose micro-scale domain was broken up into 3 nodes (0, 1, and 2). All time coefficients are assumed 1 at each nodal position.

```
[Kernels]
    [./v0_trans]
        type = MicroscaleCoupledCoefTimeDerivative
        variable = u0
        coupled_at_node = v0
        nodal_time_coef = 1
        node_id = 0
    [../]
    [./v1_trans]
        type = MicroscaleCoupledCoefTimeDerivative
```

```
            variable = u1
            coupled_at_node = v1
            nodal_time_coef = 1
            node_id = 1
        [../]
        [./v2_trans]
            type = MicroscaleCoupledCoefTimeDerivative
            variable = u2
            coupled_at_node = v2
            nodal_time_coef = 1
            node_id = 2
        [../]
[]
```

MicroscaleCoupledVariableCoefTimeDerivative

Inheritance ➔ MicroscaleCoupledCoefTimeDerivative

Special Notes ➔ (See MicroscaleCoefTimeDerivative for special notes)

Notes ➔ This kernel is for introducing a coupled time derivative for mass transfer within a specified particle nodal location for the hybrid FD/FE method. Just like the primary micro-scale variable, all coupled micro-scale variables must also have a variable for each nodal position. The parameter K_l from MicroscaleCoupledCoefTimeDerivative is now a non-linear variable, rather than a given constant.

Residual Formulation ➔ (See MicroscaleCoupledCoefTimeDerivative for residual formulation)

Additional Computations ➔ K_l is replaced with a non-linear variable coefficient

Usage

All 'Microscale' kernels must have an instance of themselves for each node you choose to divide the micro-scale into. Generally, you would use this in conjunction with a series of other 'Microscale' kernels with all the same parameters, but at different nodes.

Each different node requires a different nodal variable to be declared. For instance, if your total micro-scale quantity is being represented by u, and if you divide the micro-scale into 3 nodes, then your variables might be named u0, u1, and u2 (i.e., value of u at node 0, etc).

The coupled time derivatives are kernels that act on your primary variable (in this example: u0, u1, and u2), but are coupled to the time derivatives of other variables in the micro-scale subdomain (in this example: v0, v1, and v2). MAKE SURE the correct variables are coupled together (e.g., in this example: u0 couples to v0, not any other v).

Code example below is for the coupled time derivatives of a micro-scale problem inside spherical pellets whose micro-scale domain was broken up into 3 nodes (0, 1, and 2). The time coefficients are nodal variables named K0, K1, and K2 at each node.

```
[Kernels]
    [./v0_trans]
```

```
        type = MicroscaleCoupledVariableCoefTimeDerivative
        variable = u0
        coupled_at_node = v0
        nodal_time_var = K0
        node_id = 0
    [../]
    [./v1_trans]
        type = MicroscaleCoupledVariableCoefTimeDerivative
        variable = u1
        coupled_at_node = v1
        nodal_time_var = K1
        node_id = 1
    [../]
    [./v2_trans]
        type = MicroscaleCoupledVariableCoefTimeDerivative
        variable = u2
        coupled_at_node = v2
        nodal_time_var = K2
        node_id = 2
    [../]
[]
```

MicroscaleDiffusion

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Special Notes ➔ (See MicroscaleCoefTimeDerivative for special notes)

Notes ➔ This kernel sets up the diffusion portion of the residuals in the micro-scale using a Monotonic Upstream-centered Scheme for Conservation Laws (MUSCL) method. User must invoke this kernel for each interior node in the micro-scale subdomain. For instance, if you have 3 nodes for the micro-scale, then there is 1 interior node and 2 boundary nodes. Boundary nodes must use the MicroscaleDiffusionInnerBC or MicroscaleDiffusionOuterBC kernels.

Residual Formulation

Residual (mass / volume / time)

$$= \varphi \cdot \left[\frac{r^d_{l+1/2} D_{l+1/2}}{(\Delta r)^2}\right](u_l - u_{l+1}) + \varphi \cdot \left[\frac{r^d_{l-1/2} D_{l-1/2}}{(\Delta r)^2}\right](u_l - u_{l-1})$$

φ = MOOSE variable test function (for FE formulation)

r_l = nodal position in the micro-scale that this kernel acts on

D_l = nodal diffusion coefficient for this kernel

l = node id for this position

u_l = conserved quantity variable at the l nodal position

u_{l-1} = conserved quantity variable at the l-1 nodal position

u_{l+1} = conserved quantity variable at the l+1 nodal position

d = coordinate id (0 = z-cartesian, 1 = r-cylindrical, 2 = r-spherical)

Additional Computations ➔ None

Usage

Each interior node of the micro-scale needs a MicroscaleDiffusion kernel to represent the diffusive transport between the neighboring nodes in the micro-scale domain. The primary variable for a given MicroscaleDiffusion kernel should be the center node, then you couple to the other variables that represent the upper and lower neighbors respectively. Generally, you should order the nodal variables such that higher numbers are upper neighbors and lower numbers are lower neighbors.

Code example below is for the micro-scale diffusion occurring at node 1 in a 3 node system. The variables are u0, u1, and u2. We only invoke one instance of MicroscaleDiffusion since node 1 is the only interior node for this example. The diffusion coefficient is a single given constant.

This kernel must also be using in conjunction with the MicroscaleCoefTimeDerivative and/or MicroscaleCoupledCoefTimeDerivative for the u1 variable.

```
[Kernels]
    [./u1_diff]
        type = MicroscaleDiffusion
        variable = u1
        node_id = 1
        upper_neighbor = u2
        lower_neighbor = u0
        diffusion_const = 1
    [../]
[]
```

MicroscaleDiffusionInnerBC

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Special Notes ➔ (See MicroscaleCoefTimeDerivative for special notes)

Notes ➔ This kernel creates a combined residual for MicroscaleDiffusion in addition to applying a zero-flux boundary condition for the inner boundary in the micro-scale particles. The zero-flux boundary condition is a standard inner boundary condition for all micro-scale transport physics.

Mathematical Representation: $D\frac{\partial u}{\partial r} = 0$

Residual Formulation

Residual (mass / volume / time)

$$= \varphi \cdot \left[\frac{r_0^d D_0}{(\Delta r)^2} + \frac{r_{1/2}^d D_{1/2}}{(\Delta r)^2}\right](u_0 - u_1)$$

φ = MOOSE variable test function (for FE formulation)

r_I = nodal position in the micro-scale that this kernel acts on

D_I = nodal diffusion coefficient for this kernel

0 = node id for this position

u_0 = conserved quantity variable at the 0 nodal position (boundary node)

u_1 = conserved quantity variable at the 1 nodal position (1st interior node)

d = coordinate id (0 = z-cartesian, 1 = r-cylindrical, 2 = r-spherical)

Additional Computations ➔ None

Usage

The inner boundary node must use this kernel instead of the MicroscaleDiffusion kernel used for the interior nodes.

Code example below is for the micro-scale diffusion occurring at node 0 (i.e., the inner boundary node) in a 3 node system. The variables are u0, u1, and u2. This kernel only couples u0 and u1 since there is no lower neighbor at the interior.

This kernel must also be using in conjunction with the MicroscaleCoefTimeDerivative and/or MicroscaleCoupledCoefTimeDerivative for the u1 variable.

```
[Kernels]
    [./u0_diff]
        type = MicroscaleDiffusionInnerBC
        variable = u0
        node_id = 0
        upper_neighbor = u1
        diffusion_const = 1
    [../]
[]
```

MicroscaleDiffusionOuterBC

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Special Notes ➔ (See MicroscaleCoefTimeDerivative for special notes)

Notes ➔ This kernel creates a combined residual for MicroscaleDiffusion in addition to applying a mass-transfer flux boundary condition for the outer boundary in the micro-scale particles. The mass-transfer flux boundary condition is a standard outer boundary condition for all micro-scale transport physics. Dirichlet boundaries are not implemented because they are not relevant.

Mathematical Representation: $D\frac{\partial u}{\partial r} = k_m(u_b - u_L)$

Residual Formulation

Residual (mass / volume / time)

$$= \varphi \cdot \left[\frac{r_{L+1/2}^{d} 2k_m}{(\Delta r)}\right](u_L - u_b) + \varphi \cdot \left[\frac{r_{L-1/2}^{d} D_{L-1/2}}{(\Delta r)^2} + \frac{r_{L+1/2}^{d} D_L}{(\Delta r)^2}\right](u_L - u_{L-1})$$

φ = MOOSE variable test function (for FE formulation)

r_l = nodal position in the micro-scale that this kernel acts on

D_l = nodal diffusion coefficient for this kernel

L = node id for this position

u_b = conserved quantify in the macro-scale domain (i.e., bulk concentration)

u_L = conserved quantity variable at the L nodal position (boundary node)

u_{L-1} = conserved quantity variable at the L-1 nodal position (L-1 interior node)

d = coordinate id (0 = z-cartesian, 1 = r-cylindrical, 2 = r-spherical)

Additional Computations ➔ None

Usage

The outer boundary node must use this kernel instead of the MicroscaleDiffusion kernel used for the interior nodes.

Code example below is for the micro-scale diffusion occurring at node 2 (i.e., the outer boundary node) in a 3 node system. The variables are u0, u1, and u2. This kernel only couples u1, u2, and ub, where ub represents the coupled variable in the macro-scale to simulate transfer of mass from the macro-scale to the micro-scale.

This kernel must also be using in conjunction with the MicroscaleCoefTimeDerivative and/or MicroscaleCoupledCoefTimeDerivative for the u2 variable.

```
[Kernels]
    [./u2_diff]
        type = MicroscaleDiffusionOuterBC
        variable = u2
        node_id = 2
        macro_variable = ub
        lower_neighbor = u1
        diffusion_const = 1
    [../]
[]
```

MicroscaleScaledWeightedCoupledSumFunction

Inheritance ➔ ScaledWeightedCoupledSumFunction

Special Notes ➔ (See MicroscaleCoefTimeDerivative for special notes)

Notes ➔ This kernel is for introducing a scaled, weighted sum of non-linear variables into microscale kernel set. This generally will be used to introduce reaction variables into the microscale. It is much more convenient to use this to couple multiple reactions rather than to use several difference MicroscaleCoupledCoefTimeDerivative kernels to cross-coupled numerous reactions.

Residual Formulation

Residual (mass / volume / time) = $-\varphi \cdot r_l^d \cdot \lambda \cdot \sum_{\forall i} w_i v_i$

φ = MOOSE variable test function (for FE formulation)

r_l = nodal position in the micro-scale that this kernel acts on

l = node id for this position

w_i = weight factor for the i-th non-linear variable

λ = coupled scaling factor for the variable sum

v_i = i-th coupled variable at this nodal position

d = coordinate id (0 = z-cartesian, 1 = r-cylindrical, 2 = r-spherical)

Additional Computations ➔ None

Usage

All 'Microscale' kernels must have an instance of themselves for each node you choose to divide the micro-scale into. Generally, you would use this in conjunction with a series of other 'Microscale' kernels with all the same parameters, but at different nodes.

Each different node requires a different nodal variable to be declared. For instance, if your total micro-scale quantity is being represented by u, and if you divide the micro-scale into 3 nodes, then your variables might be named u0, u1, and u2 (i.e., value of u at node 0, etc).

The scaled, weighted sum function is used to couple together a list of non-linear variables, that all exist on the microscale, to add them into the current variables' residual function. Every node in your microscale will likely need an instance of this sum kernel and the node_id values should line up for all non-linear variables.

In this example, we are coupling a reaction (r) at each node id (0, 1, 2) to the total residual of a non-linear variable (u). This is needed at each node and each reaction variable (r) is also

discretized by microscale location (r_0, r_1, r_2). You can use the 'scale' as a constant or a non-linear variable to represent a unit conversion.

```
[Kernels]
    [./u0_rxn]
        type = MicroscaleScaledWeightedCoupledSumFunction
        variable = u0
        node_id = 0
        coupled_list = 'r_0'
        weights = '-1'
        scale = 1
    [../]
    [./u1_rxn]
        type = MicroscaleScaledWeightedCoupledSumFunction
        variable = u1
        node_id = 1
        coupled_list = 'r_1'
        weights = '-1'
        scale = 1
    [../]
    [./u2_rxn]
        type = MicroscaleScaledWeightedCoupledSumFunction
        variable = u2
        node_id = 2
        coupled_list = 'r_2'
        weights = '-1'
        scale = 1
    [../]

[]
```

MicroscaleVariableCoefTimeDerivative

Inheritance ➔ MicroscaleCoefTimeDerivative

Special Notes ➔ (See MicroscaleCoefTimeDerivative for special notes)

Notes ➔ This kernel is for introducing a time derivative for a microscale mass balance within a specified particle nodal location for the hybrid FD/FE method. The parameter R_I from MicroscaleCoefTimeDerivative is now a non-linear variable, rather than a given constant.

Residual Formulation ➔ (See MicroscaleCoefTimeDerivative for residual formulation)

Additional Computations ➔ R_I is replaced with a non-linear variable coefficient

Usage

All 'Microscale' kernels must have an instance of themselves for each node you choose to divide the micro-scale into. Generally, you would use this in conjunction with a series of other 'Microscale' kernels with all the same parameters, but at different nodes.

Each different node requires a different nodal variable to be declared. For instance, if your total micro-scale quantity is being represented by u, and if you divide the micro-scale into 3 nodes, then your variables might be named u0, u1, and u2 (i.e., value of u at node 0, etc).

Code example below is for the variable time derivatives of a micro-scale problem inside spherical pellets whose micro-scale domain was broken up into 3 nodes (0, 1, and 2). The time coefficients are nodal variables named R0, R1, and R2 at each node.

```
[Kernels]
    [./u0_dot]
        type = MicroscaleVariableCoefTimeDerivative
        variable = u0
        nodal_time_var = R0
        node_id = 0
    [../]
    [./u1_dot]
        type = MicroscaleVariableCoefTimeDerivative
        variable = u1
        nodal_time_var = R1
        node_id = 1
    [../]
    [./u2_dot]
        type = MicroscaleVariableCoefTimeDerivative
        variable = u2
        nodal_time_var = R2
        node_id = 2
    [../]
[]
```

MicroscaleVariableDiffusion

Inheritance ➔ MicroscaleDiffusion

Special Notes ➔ (See MicroscaleCoefTimeDerivative for special notes)

Notes ➔ This kernel sets up the diffusion portion of the residuals in the micro-scale using a Monotonic Upstream-centered Scheme for Conservation Laws (MUSCL) method. User must invoke this kernel for each interior node in the micro-scale subdomain. For instance, if you have 3 nodes for the micro-scale, then there is 1 interior node and 2 boundary nodes. Boundary nodes must use the MicroscaleVariableDiffusionInnerBC or MicroscaleVariableDiffusionOuterBC kernels. This version of the microscale diffusion kernel replaces the constant diffusion value from MicroscaleDiffusion with a set of variables representing the diffusion coefficients associated with this and all neighboring nodes in the microscale domain.

Residual Formulation ➔ (See MicroscaleDiffusion for residual formulation)

Additional Computations

$$D_{l+1/2} = 0.5(D_{l+1} + D_l) \qquad D_{l-1/2} = 0.5(D_{l-1} + D_l)$$

D_{l+1} = upper diffusion coefficient

D_l = current diffusion coefficient

D_{l-1} = lower diffusion coefficient

Usage

Each interior node of the micro-scale needs a MicroscaleVariableDiffusion kernel to represent the diffusive transport between the neighboring nodes in the micro-scale domain. The primary variable for a given MicroscaleVariableDiffusion kernel should be the center node, then you couple to the other variables that represent the upper and lower neighbors respectively. Generally, you should order the nodal variables such that higher numbers are upper neighbors and lower numbers are lower neighbors.

Code example below is for the micro-scale diffusion occurring at node 1 in a 3 node system. The variables are u0, u1, and u2. We only invoke one instance of MicroscaleDiffusion since node 1 is the only interior node for this example. The diffusion coefficients are given as D0, D1, and D3 for the variation in diffusion at their respective nodes.

This kernel must also be using in conjunction with the MicroscaleVariableCoefTimeDerivative and/or MicroscaleCoupledVariableCoefTimeDerivative for the u1 variable.

```
[Kernels]
        [./u1_diff]
                type = MicroscaleVariableDiffusion
                variable = u1
                node_id = 1
                upper_neighbor = u2
                lower_neighbor = u0
                current_diff = D1
                upper_diff = D2
                lower_diff = D0
        [../]
[]
```

MicroscaleVariableDiffusionInnerBC

Inheritance ➔ MicroscaleDiffusionInnerBC

Special Notes ➔ (See MicroscaleCoefTimeDerivative for special notes)

Notes ➔ This kernel creates a combined residual for MicroscaleDiffusion in addition to applying a zero-flux boundary condition for the inner boundary in the micro-scale particles. The zero-flux boundary condition is a standard inner boundary condition for all micro-scale transport physics. The diffusion coefficients are replaced with diffusion variables for the nodes.

(See MicroscaleDiffusionInnerBC for additional notes)

Residual Formulation ➔ (See MicroscaleDiffusionInnerBC for residual formulation)

Additional Computations

$$D_{1/2} = 0.5(D_1 + D_0)$$

D_1 = upper diffusion coefficient D_0 = current diffusion coefficient

Usage

The inner boundary node must use this kernel instead of the MicroscaleVariableDiffusion kernel used for the interior nodes.

Code example below is for the micro-scale diffusion occurring at node 0 (i.e., the inner boundary node) in a 3 node system. The variables are u0, u1, and u2. This kernel only couples u0 and u1 since there is no lower neighbor at the interior.

This kernel must also be using in conjunction with the MicroscaleVariableCoefTimeDerivative and/or MicroscaleCoupledVariableCoefTimeDerivative for the u1 variable.

```
[Kernels]
    [./u0_diff]
        type = MicroscaleDiffusionInnerBC
        variable = u0
        node_id = 0
        upper_neighbor = u1
        current_diff =  D0
        upper_diff = D1
    [../]
[]
```

MicroscaleVariableDiffusionOuterBC

Inheritance ➔ MicroscaleDiffusionOuterBC

Special Notes ➔ (See MicroscaleCoefTimeDerivative for special notes)

Notes ➔ This kernel creates a combined residual for MicroscaleVariableDiffusion in addition to applying a mass-transfer flux boundary condition for the outer boundary in the micro-scale particles. The mass-transfer flux boundary condition is a standard outer boundary condition for all micro-scale transport physics. Dirichlet boundaries are not implemented because they are not relevant. Both the diffusion coefficients and mass transfer coefficients are variables instead of just constants as they were in the base class.

(See MicroscaleDiffusionOuterBC for additional notes)

Residual Formulation ➔ (See MicroscaleDiffusionOuterBC for residual formulation)

Additional Computations ➔ k_m is replaced with a non-linear variable

$$D_{L-1/2} = 0.5(D_{L-1} + D_L)$$

D_L = current diffusion coefficient D_{L-1} = lower diffusion coefficient

Usage

The outer boundary node must use this kernel instead of the MicroscaleVariableDiffusion kernel used for the interior nodes.

Code example below is for the micro-scale diffusion occurring at node 2 (i.e., the outer boundary node) in a 3 node system. The variables are u0, u1, and u2. This kernel only couples u1, u2, and ub, where ub represents the coupled variable in the macro-scale to simulate transfer of mass from the macro-scale to the micro-scale.

Note: When coupling with the macro-scale, you would couple the outer boundary variable (u2) with the bulk parameter (ub) through the FilmMassTransfer kernel using the same variable for the mass transfer coefficient (rate_variable = km).

This kernel must also be using in conjunction with the MicroscaleVariableCoefTimeDerivative and/or MicroscaleCoupledVariableCoefTimeDerivative for the u2 variable.

```
[Kernels]
    [./u2_diff]
        type = MicroscaleVariableDiffusionOuterBC
        variable = u2
        node_id = 2
        macro_variable = ub
        lower_neighbor = u1
        current_diff = D2
        lower_diff = D1
        rate_variable = km
    [../]
[]
```

ModifiedButlerVolmerReaction

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel creates a reaction rate expression based on a modification to the classical 'Butler-Volmer' kinetics expression for redox half-reactions in electrochemistry. In the classical formulation, the concept of 'overpotential' is used to relate the oxidation rate (k_a) and reduction rate (k_c) of the reaction to an equilibrium potential. However, this convention brings in a 'natural log' function of the ratio of concentrations, which is numerically unstable. In this formulation, we circumvent the need for that 'natural log' function and instead directly calculate rates of the reaction through a more standard reaction function (assuming the reaction is elementary) and using the potential difference between electrode and electrolyte as the basis for the barrier energy for the reaction.

The redox half-reaction for this formulation can be written generically as follows:

$$\sum_{jR} s_{jR} c_{jR} \leftrightarrow \sum_{iO} s_{iO} c_{iO} + ne^-$$

n = number of electrons transferred

s_{jR} = stoichiometric coefficient for the 'reduced state' variables

c_{jR} = concentration of 'reduced state' variables (moles per volume)

s_{iO} = stoichiometric coefficient for the 'oxidized state' variables

c_{iO} = concentration of 'oxidized state' variables (moles per volume)

Residual Formulation

Residual (moles / area / time)

$$= -\psi \cdot b \cdot k_a C_R exp\left(\frac{[1-\alpha]nF[\Delta\phi]}{RT}\right) + \psi \cdot b \cdot k_c C_O exp\left(\frac{-\alpha nF[\Delta\phi]}{RT}\right)$$

φ = MOOSE variable test function (for FE formulation)

b = scaling factor (default = 1)

k_a = oxidation rate (can be a given constant or calculated from equilibrium potential and an overall rate constant) [*units*: length/time, but varies]

k_c = reduction rate (can be a given constant or calculated from equilibrium potential and an overall rate constant) [*units*: length/time, but varies]

F = Faraday's constant (default = 96,485.3 C/mol)

R = Gas law constant (default = 8.314462 J/K/mol)

T = temperature variable (K)

$\Delta\phi$ = variable for potential difference (V or J/C)

→ Generally calculated as $\Delta\phi = \phi_s - \phi_e$

ϕ_s = electrode potential, ϕ_e = electrolyte potential

α = electron transfer coefficient (default = 0.5 for symmetric electron transfer)

C_R, C_O = see formulation below

Additional Computations

$$C_O = \prod_{iO} c_{iO}^{s_{iO}} \qquad C_R = \prod_{jR} c_{jR}^{s_{jR}}$$

$$k_c = k \cdot exp\left(\frac{\alpha E_0' nF}{RT}\right)$$

$$k_a = k \cdot exp\left(-\frac{(1-\alpha)E_0' nF}{RT}\right)$$

k = overall rate transfer constant [units: length / time, but varies]

E_0' = equilibrium potential for the reaction (V or J/C)

NOTE: Users can choose to just give a constant value for ka and kc, but by default the above calculation is done because that follows suit with standard parameters in literature.

Usage

Generally used in conjunction with a Reaction kernel to evaluate the rate (r). That rate can then be coupled into ion mass balances or coupled with the ButlerVolmerCurrentDensity kernel so that the electron exchange can be coupled into other kernels for potentials in the domain.

The example below shows how to calculate the reaction rate of a simple redox reaction of the form of: R ←→ O + e-. In this reaction, a single electron is transferred. We also couple with reaction with the potential difference variable (pot_diff) in the domain.

```
[./r_equ]
        type = Reaction
        variable = r
[../]
[./r_rxn]
        type = ModifiedButlerVolmerReaction
        variable = r

        reaction_rate_const = 1
        equilibrium_potential = 0
        number_of_electrons = 1

        reduced_state_vars = 'R'
        reduced_state_stoich = '1'
        oxidized_state_vars = 'O'
        oxidized_state_stoich = '1'

        temperature = T
        electric_potential_difference = pot_diff
[../]
```

PairedLangmuirInhibition

Inheritance ➔ LangmuirInhibition

Notes ➔ This kernel is used to create a residual to represent an inhibition term for InhibitedArrheniusReaction. The inhibition term is its own non-linear variable and is used in conjunction with other inhibition terms or a "Reaction" kernel to finish the full definition of the term in the MOOSE residual functions (see **Usage** below). The difference between this kernel and LangmuirInhibition is the inclusion of 2nd order coefficients and binary interactions between concentrations of species.

Residual Formulation

The residual is formed from a list of coupled concentration terms and a coupled temperature. The inhibition term does not have any units. In addition, lists of "paired" species (C_i and C_j) are given with associated "paired" Langmuir coefficients (Kij).

$$\text{Residual (-)} = -\varphi \cdot \left(1 + \sum_{\forall i} K_i C_i + \sum_{\forall ij} K_{ij} C_i C_j\right)$$

φ = MOOSE variable test function (for FE formulation)

C_i = concentration variable for species i

K_i = Langmuir coefficient for the i-th species in the list

C_i,C_j = concentration variables for a pairing of i and j species

K_{ij} = associated Langmuir coefficients for the i,j pairs

Additional Computations

The Langmuir coefficients are calculated from the same expression in LangmuirInhibition. In addition, the paired Langmuir coefficients (K_{ij}) are calculated with a similar expression.

$$K_{ij} = A_{ij} T^{\beta_{ij}} exp\left[-\frac{E_{ij}}{RT}\right]$$

- The units for A_{ij} must have the same units as K_{ij}
- K_{ij} must have inverse units of the products of concentration C_i and C_j
- The units for E_{ij} must be J/mol
- β_{ij} are powers on temperature (which default to 0)

Usage

MUST be used in conjunction with a "Reaction" kernel from the MOOSE base system of kernels in order to fully describe the inhibition term (R) for a reaction.

Code below creates an inhibition term that is a function of concentrations for A and B, as well as a pairing of A and B. The coefficients (K) in the inhibition term are each set to a value of 1. This is accomplished by providing 0s for beta and activation energy terms, which removes the temperature dependence for those parameters.

```
[./R_equ]
        type = Reaction
        variable = R
[../]
[./R_lang]
        type = PairedLangmuirInhibition
        variable = R
        temperature = T
        coupled_list = 'A B'
        pre_exponentials = '1 1'
        betas = '0 0'
```

activation_energies = '0 0'
coupled_i_list = 'A'
coupled_j_list = 'B'
binary_pre_exp = '1'
binary_betas = '0'
binary_energies = '0'
[../]

PhaseEnergyTransfer

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes

➔ This kernel facilitates the transfer of energy from one system phase to another. For instance, in a packed bed there is a solid phase and a fluid phase. If those two phases cannot be assumed locally isothermal, then each will have a different temperature and must transfer energy between each other through a heat transfer coefficient and effective area of contact. That area of contact will be a function of the volume fraction in the domain between the two phases and the specific area per volume of the phases.

➔ Each phase will need an instance of this kernel to complete the energy transfer expression for each phase. In addition, this kernel is intended to act upon the energy density variable of each phase and not the temperature of each phase. It is coupled to temperature, but used to solve for internal energy.

Residual Formulation

Residual (J / m^3 / s) = $\varphi \cdot h \cdot f_v \cdot A_s \cdot (T_a - T_b)$

φ = MOOSE variable test function (for FE formulation)

h = heat transfer rate ($W/m^2/K$)

f_v = volume fraction between the phases (i.e., volume solids / total volume)

A_s = specific area of contact per volume of a phase (m^{-1})

(i.e., surface area per volume of solids)

T_a = temperature of this phase energy variable (K)

T_b = temperature of the other phase energy variable (K)

Additional Computations ➔ None

Usage

Generally used in conjunction with other kernels describing the conservation of energy in different phases. Each phase that is transferring energy needs to invoke this kernel in their respective energy balances. Note that the variable this kernel acts on is the energy density of the given phase, NOT the temperature of the phase. Temperature is a coupled variable.

Example below is for transfer of energy to the fluid phase energy density (Ef) from the solid phase. The parameters for heat transfer, specific area, and volume fraction are all constants. The temperature of this phase is the temperature of the fluid (Tf) and the other temperature is the temperature variable for the solid (Ts).

```
[./energy_trans]
        type = PhaseEnergyTransfer
        variable = Ef
        this_phase_temp = Tf
        other_phase_temp = Ts
        transfer_coef = 10
        volume_frac = 0.5
        specific_area = 1000
[../]
```

PhaseTemperature

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel is used in conjunction with an energy density variable and kernel set in order to calculate the temperature of the phase that the energy density variable represents. This is done because the energy balance is most logically performed on the energy density and not the temperature. Thus, it is necessary to add a kernel that can calculate the temperature for the phase, which is used in other computations. Temperature of a phase is based variables for the density of the phase, the specific heat of the phase, and the volume fraction of the phase.

Residual Formulation

Residual (J / m^3) = $\varphi \cdot (\rho c T - E)$

φ = MOOSE variable test function (for FE formulation)

ρ = density of the phase (kg/m^3)

c = specific heat of the phase (J/kg/K)

T = temperature of the phase (K)

E = energy density of the phase (J/m^3)

Additional Computations ➔ None

Usage

Generally used in conjunction with kernel that calculation energy density of a phase from conservation laws. Can also be combined with variables or auxiliary variables for the other parameters, such as density. The temperature is the primary variable for this kernel.

Example below calculates the temperature (T) of a phase based on the energy density (E) of that phase, as well as variables for all other properties of the phase.

```
[./phase_temp]
        type = PhaseTemperature
        variable = T
        energy = E
        specific_heat = cp
        density = rho
[../]
```

Reaction

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel is actually part of the MOOSE framework, but is discussed here since it is used to create and evaluate some reaction rate variables for complex reaction systems. It provides a residual for a basic, linear reaction with an optional coefficient (which we will not be using for most of our problems).

Residual Formulation

Residual (mass / volume / time) = $\varphi \cdot \lambda u$

φ = MOOSE variable test function (for FE formulation)

u = a non-linear variable (which will represent a reaction rate in our models)

λ = (optional) rate parameter/coefficient [we won't use this]

Additional Computations ➔ None

Usage

Used whenever we want to create a non-linear variable for a reaction rate. Will be used in conjunction with a rate function such as ArrheniusReaction or ConstReaction. For example, here we create non-linear variable 'r1' and specify that that reaction rate is evaluated as a simple forward reaction, e.g., r1 = kf*C, where kf = 1.

That reaction rate can then be directly coupled into other mass balances or rate expressions.

```
[./r1_rxn]
        type = Reaction
        variable = r1
 [../]
[./r1_rate]
        type = ConstReaction
        variable = r1
        this_variable = r1
        scale = 1
        forward_rate = 1
        reactants = 'C'
```

reactant_stoich = '1'

[../]

ScaledWeightedCoupledSumFunction

Inheritance ➔ WeightedCoupledSumFunction

Notes ➔ This kernel inherits from WeightedCoupledSumFunction and can be used in the exact same way if needed. However, it adds to the summation a 'scale' factor variable that can be used as a way to convert units and/or couple between mass balances.

For example, in surface reaction problems, the binding of bulk phase molecules to a surface removes those molecules from that bulk phase. However, the amount removed is not directly a function of the surface reactions, but a 'scaled' representation of those reactions, wherein the scaling factor is based on a surface-to-volume or solids-to-volume ratio.

Example: $\varepsilon_b \frac{\partial C_{b,i}}{\partial t} + \nabla \cdot \left(\varepsilon_b v C_{b,i}\right) + (1 - \varepsilon_b) \sum_{\forall j} w_j r_j = 0$

In the above formulation, the factor of (1 - ε_b) would be our 'scaling' factor that converts the rate of reactions from surface concentrations to bulk concentrations to finish the mass balance. The r_j are the reaction variables that represent each surface reaction and w_j are the weights (usually molar amounts) that that reaction contributes to the total effect on the bulk species.

Residual Formulation

Residual (same units as the variables given) $= -\varphi \cdot \lambda \cdot \sum_{\forall i} w_i v_i$

φ = MOOSE variable test function (for FE formulation)

v_i = i^{th} coupled non-linear variable

w_i = i^{th} weight coefficient for the sum (can be positive or negative)

λ = scaling factor (can be a non-linear variable or a constant)

Additional Computations ➔ None

Usage

Our example here is a continuation of the example from WeightedCoupledSumFunction, wherein we have 3 reactions that contribute to a surface concentration 'q', which is resolved in the previous example. Here, we build on that example and demonstrate how to used ScaledWeightedCoupledSumFunction to account for the disappearance of bulk species 'A' and 'B' from those reactions. This kernel must be used in conjunction with other kernels describing the bulk behaviors of 'A' and 'B'.

```
# This kernel defines disappearance of 'A' from 'r1'
# Weights are negative to represent that 'A' is lost during these reactions
[./A_losses]
        type = ScaledWeightedCoupledSumFunction
        variable = A
        coupled_list = 'r1'
        weights = '-1'
        scale = 0.5
 [../]

# This kernel defines disappearance of 'B' from 'r2'
# Weights are negative to represent that 'B' is lost during these reactions
[./B_losses]
        type = ScaledWeightedCoupledSumFunction
        variable = B
        coupled_list = 'r2'
        weights = '-1'
        scale = 0.5
 [../]
```

TimeDerivative

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel is actually part of the MOOSE framework, but is discussed here since it is commonly used in all MOOSE modules/simulations. It provides a residual for a basic time derivative of the non-linear variable.

Residual Formulation

Residual (mass / volume / time) = $\varphi \cdot \frac{\partial u}{\partial t}$

φ = MOOSE variable test function (for FE formulation)

u = a non-linear variable that changes in time

Additional Computations ➔ None

Usage

Used whenever a non-linear variable varies in time. Generally needs to be accompanied by other kernels that describe how the variable changes in time.

```
[./u_dot]
        type = TimeDerivative
        variable = u
 [../]
```

VarSiteDensityExtLangModel

Inheritance ➔ ExtendedLangmuirModel

Notes ➔ This kernel uses the residuals from ExtendedLangmuirModel, but allows the maximum adsorption capacity for the Langmuir model to be a non-linear variable. This way that maximum capacity can change as some other function of space and time. This is useful if we need to simulate aging effects that reduce the maximum site densities overtime. HOWEVER, this kernel is unnecessary as it is more efficient to just use a series of reaction kernels (such as ArrheniusReaction or ArrheniusEquilibriumReaction or even MaterialBalance) to account for variations in site density from aging.

Residual Formulation ➔ (See ExtendedLangmuirModel for residual formulation)

Additional Computations ➔ The maximum capacity parameter form ExtendedLangmuirModel is overridden to be the value of another non-linear variable provided.

Usage

Usage is essentially the same as ExtendedLangmuirModel, but user also provides a non-linear variable for site density instead of a constant for maximum capacity.

Example below calculates the adsorption of species A to form q in the presence of species B, which is competing for the same adsorption sites as A.

```
[./q_res]
        type = Reaction
        variable = q
[../]
[./q_lang]
        type = VarSiteDensityExtLangModel
        variable = q
        coupled_site_density = qmax
        main_coupled = A
        coupled_list = 'A B'
        coupled_temp = T
        enthalpies = '-5000 -4000'
        entropies = '100 50'
[../]
```

VariableCoefTimeDerivative

Inheritance ➔ CoefTimeDerivative (MOOSE framework kernel)

Notes ➔ This kernel uses the existing residual calculations from CoefTimeDerivative (in the MOOSE framework), but overrides the time coefficient of that kernel to be another non-linear variable. This is useful for simulations that involve a time coefficient that might vary in space (such as a variable porosity in a porous media).

Residual Formulation

Residual (mass / volume / time) = $\varphi \cdot R \frac{\partial u}{\partial t}$

φ = MOOSE variable test function (for FE formulation)

u = a non-linear variable that changes in time

R = a non-linear variable for the time coefficient

Additional Computations ➔ Overrides the base kernel parameter for the time coefficent

Usage

Used whenever a non-linear variable varies in time and has a non-linear variable for a time coefficient that varies in space (but not in time). Generally needs to be accompanied by other kernels that describe how the variable changes in time.

```
[./u_dot]
        type = VariableCoefTimeDerivative
        variable = u
        coupled_coef = eps
 [../]
```

VariableCoupledCoeffTimeDerivative

Inheritance ➔ CoupledCoeffTimeDerivative

Notes ➔ This kernel creates a specific form of CoupledCoeffTimeDerivative wherein the concentration of a bulk fluid species is coupled to the time derivative of a solid or surface species in a particle. The time coefficient is the bulk density of solids in the column, which is a common defining parameter for packed bed reactors. That bulk density is used as a unit conversion from mass per volume of solids for the coupled variable to mass per volume of gases in the system.

Residual Formulation

(See CoupledCoeffTimeDerivative for residual formulation)

Additional Computations

The time coefficient (a) for the CoupledCoeffTimeDerivative is overridden to be as follows:

$a = \rho_b$ where ρ_b is the bulk density in the domain (mass solids / total volume)

This kernel still uses the "gaining" Boolean argument to determine whether or not to treat this transfer as a source or sink term. That option is depreciated, so you do not need to use it.

Usage

Must be used in conjunction with a set of kernels for the bulk variable and the coupled variable.

Code below would create a sink term for the variable Cb based on the rate of change of q:

```
[./coupled_dq_dt]
        type = VariableCoupledCoeffTimeDerivative
        variable = Cb
        coupled =q
        var_time_coeff = rho_b
[../]
```

VectorCoupledGradient

Inheritance ➔ Kernel (MOOSE framework base kernel)

Notes ➔ This kernel creates a way to couple one variable to the gradient of another variable through a vector dot product operation. The most basic way to use this kernel would be as a means to enforce continuity on a fluid (i.e., $\nabla \cdot \mathbf{u} = 0$, where $\mathbf{u}$ is the velocity vector) or to calculate velocity based on a pressure gradient as in Darcy's Law.

Residual Formulation

Residual (mass / volume / time) = $\varphi \cdot (\mathbf{v} \cdot \nabla p)$

φ = MOOSE variable test function (for FE formulation)

p = a non-linear variable whose gradient we couple to

v = a constant vector that is dotted with the coupled gradient

= < v_x, v_y, v_z>

NOTE: User should supply each vector component in the input file.

Additional Computations ➔ None

Usage

One such usage of this kernel is to enforce the condition of the continuity of flow in an incompressible fluid. We would do this in a piecewise manner and couple with each individual velocity variable component and give unit vectors for the specific direction that an individual piece is integrating on. In this example, the kernels act on the pressure variable and couple with velocity variables in each direction

Example 1: Continuity in 2D

```
[./p_x]
        type = VectorCoupledGradient
        variable = pressure
        coupled = vel_x
        vx = 1
        vy = 0
        vz = 0
 [../]
[./p_y]
```

```
        type = VectorCoupledGradient
        variable = pressure
        coupled = vel_y
        vx = 0
        vy = 1
        vz = 0
[../]
```

Another way to use this kernel is Darcy's Law. According to Darcy's Law, the velocity in a porous media can be calculated as a function of the pressure gradient multiplied by a permeability coefficient (K).

Darcy's Law: $\mathbf{u} = -K\nabla p$

This statement is then completed by also providing pressure at the inlet and outlet of a domain and enforcing linearity of the pressure gradient (i.e., $\nabla^2 p = 0$).

NOTE: In the kernel, the velocity vector components (vx, vy, vz) would be a scalar multiple of the unit vector in the direction of interest. For example, if my permeability is 5, then replace 'vx' with 5 for the velocity in the x-direction and make all other velocity components 0.

Example 2: Darcy Flow in 2D

```
# Force linear pressure gradient (no other active kernels for pressure)
[./p_lin]
        type = Diffusion
        variable = pressure
[../]

# Weak form statement for velocity in x-direction
[./vel_x_equ]
        type = Reaction
        variable = vel_x
[../]
[./vel_x_Darcy]
        type = VectorCoupledGradient
        variable = vel_x
        coupled = pressure
        vx = 5
        vy = 0
        vz = 0
[../]


# Weak form statement for velocity in y-direction
[./vel_y_equ]
        type = Reaction
        variable = vel_y
[../]
```

```
[./vel_y_Darcy]
        type = VectorCoupledGradient
        variable = vel_y
        coupled = pressure
        vx = 0
        vy = 5
        vz = 0
[../]
```

VariableLaplacian

Inheritance ➔ Kernel (MOOSE framework base kernel)

Notes ➔ This kernel creates a way to couple one variable as a coefficient to a Laplace's or Poisson's type of model.

Laplace's Model: $K \cdot \nabla^2 u = 0$

Poisson's Model: $K \cdot \nabla^2 u = f$

Residual Formulation

Residual (units depend on the system) = $\nabla\varphi \cdot K \cdot \nabla u$

φ = MOOSE variable test function (for FE formulation)

u = the non-linear variable this kernel acts on

K = a non-linear variable coefficient for the equation

Additional Computations ➔ None

Usage

A common usage would be for solving a diffusion-reaction model (using standard Continuous Galerkin finite elements) where the diffusion coefficient is a variable itself.

```
[./var_diff]
        type = VariableLaplacian
        variable = u
        coupled_coef = K
[../]
[./simple_linear_reaction]
        type = Reaction
        variable = u
[../]
```

VariableVectorCoupledGradient

Inheritance ➔ VectorCoupledGradient

Notes ➔ This kernel creates a way to couple one variable to the gradient of another variable through a vector dot product operation. The most basic way to use this kernel would be as a means to enforce continuity on a fluid (i.e., $\nabla \cdot \mathbf{u} = 0$, where $\mathbf{u}$ is the velocity vector) or to calculate velocity based on a pressure gradient as in Darcy's Law.

Residual Formulation

Residual (mass / volume / time) = $\varphi \cdot (\mathbf{v} \cdot \nabla p)$

φ = MOOSE variable test function (for FE formulation)

p = a non-linear variable whose gradient we couple to

v = a variable vector that is dotted with the coupled gradient

= < u_x, u_y, u_z> where each u_i can be its own non-linear variable.

NOTE: User should supply each vector component variable in the input file.

Additional Computations ➔ None

Usage

The primary way to use this kernel is Darcy's Law. According to Darcy's Law, the velocity in a porous media can be calculated as a function of the pressure gradient multiplied by a permeability coefficient (K).

Darcy's Law: $\mathbf{u} = -K\nabla p$

This statement is then completed by also providing pressure at the inlet and outlet of a domain and enforcing linearity of the pressure gradient (i.e., $\nabla^2 p = 0$).

NOTE: In the kernel, the velocity vector components (ux, uy, uz) are variables that would represent permeability (K). For each velocity direction, provide 'K' in ONLY the direction that the kernel acts on (e.g., for velocity in x-direction, supply 'ux = K' and each other component, uy and uz, would be zero).

Example: Darcy Flow in 2D

```
# Force linear pressure gradient (no other active kernels for pressure)
[./p_lin]
        type = Diffusion
        variable = pressure
 [../]

# Weak form statement for velocity in x-direction
[./vel_x_equ]
        type = Reaction
        variable = vel_x
```

```
[../]
[./vel_x_Darcy]
        type = VariableVectorCoupledGradient
        variable = vel_x
        coupled = pressure
        ux = K
        uy = 0
        uz = 0
[../]


# Weak form statement for velocity in y-direction
[./vel_y_equ]
        type = Reaction
        variable = vel_y
[../]
[./vel_y_Darcy]
        type = VariableVectorCoupledGradient
        variable = vel_y
        coupled = pressure
        ux = 0
        uy = K
        uz = 0
[../]
```

WeightedCoupledSumFunction

Inheritance ➔ Kernel (i.e., the MOOSE Kernel base system)

Notes ➔ This kernel is similar to CoupledSumFunction, but does not inherit from it. It creates a residual for a weighted sum of other coupled variables. The variable that this kernel acts on should NOT be in the list of other coupled variables. Most common usage of this kernel would be in conjunction with the MOOSE standard Reaction kernel, TimeDerivative kernel, or VariableCoefTimeDerivative kernel. The culmination of this kernel with one of the previous will create an equation such as shown below.

Example Problem: $dq/dt = r_1 + r_2 - r_3$

To setup and solve this problem, we create variables for 'r1', 'r2', 'r3', and 'q'. Each reaction variable is setup using the Reaction kernel with an associated ArrheniusReaction, ConstReaction, and/or ArrheniusEquilibriumReaction. Then, the 'q' variable is setup as a combination of a TimeDerivative kernel and this kernel, the WeightedCoupledSumFunction. See **Usage** below for input file formatting.

Residual Formulation

Residual (same units as the variables given) $= -\varphi \cdot \sum_{\forall i} w_i v_i$

φ = MOOSE variable test function (for FE formulation)

v_i = i^{th} coupled non-linear variable

w_i = i^{th} weight coefficient for the sum (can be positive or negative)

Additional Computations ➔ None

Usage

Our example here highlights how to setup the example problem described above in **Notes**. The rate functions for each reaction will be ConstReaction time kernels just to keep the example relatively simple.

```
# Define r1:   r1 = 1*C
[./r1_rxn]
        type = Reaction
        variable = r1
 [../]
[./r1_rate]
        type = ConstReaction
        variable = r1
        this_variable = r1
        scale = 1
        forward_rate = 1
        reactants = 'C'
        reactant_stoich = '1'
 [../]

# Define r2:   r2 = 0.5*B
[./r2_rxn]
        type = Reaction
        variable = r2
 [../]
[./r2_rate]
        type = ConstReaction
        variable = r2
        this_variable = r2
        scale = 1
        forward_rate = 0.5
        reactants = 'B'
        reactant_stoich = '1'
 [../]

# Define r3:   r3 = 2.5*q
[./r3_rxn]
        type = Reaction
        variable = r3
 [../]
[./r3_rate]
        type = ConstReaction
```

```
        variable = r3
        this_variable = r3
        scale = 1
        forward_rate = 2.5
        reactants = 'q'
        reactant_stoich = '1'
 [../]

# Define q:       dq/dt = r1 + r2 - r3 == 1*C + 0.5*B - 2.5*q
[./q_dot]
        type = TimeDerivative
        variable = q
 [../]
[./q_rate_sum]
        type = WeightedCoupledSumFunction
        variable = q
        coupled_list = 'r1 r2 r3'
        weights = '1 1 -1'
 [../]
```

Boundary Conditions

Boundary conditions are required for any physics that involves either motion or spatially variant non-linear residual kernels (such as diffusion or advection kernels). In DG methods, it is impossible to provide Dirichlet type boundary conditions because solutions are not defined on nodes, but elements instead. However, we can emulate Dirichlet type boundary conditions if necessary.

In CATS, there are 2 main types of boundaries: (i) 'Flux' boundary and (ii) 'FluxLimited' boundary. The 'Flux' boundaries should be employed for any "open" type of boundary conditions where energy or mass is flowing into the domain carried by a velocity (or heat transfer coefficient). The 'FluxLimited' boundaries use the same parametric information as their 'Flux' counter-parts, but are used to emulate a Dirichlet type boundary. In general, it is recommended to use the 'Flux' type boundary conditions unless you absolutely need a Dirichlet boundary condition or if simulations are oscillatory.

All specialized boundary conditions inherit from either DGFluxBC or DGFluxLimitedBC. Those are the most generic of the boundaries. The other kernels create more specific conditions at the boundaries for a variety of different situations such as variable velocities or porosities, etc.

NOTE: Users can use any MOOSE IntegratedBCs they want (including things like PenaltyDirichletBC and all of its derivatives). Provide here are BCs specific to modeling problems in CATS, where generally speaking your boundaries are all defined by some form of 'flux' across/through a boundary.

NOTE 2: Using either DGFE or CGFE methods in MOOSE, the default BC will always be that the gradients at the boundaries are 0. As a consequence, this means that the boundaries for which you do NOT apply a specific condition, that will be treated as a "closed" boundary.

CoupledDirichletBC

Inheritance ➔ NodalBC (i.e., the MOOSE base kernel for flux-based boundaries)

Notes ➔ This kernel can be used ONLY with continuous Galerkin shape functions. Generally, in CATS this is reserved exclusively for pressure and potential variables (as all other system are usually solved with Discontinuous Galerkin shape functions). It creates a standard Dirichlet type BC where the value at the boundary can be a coupled variable.

Residual Formulation

Note: The residuals are integrated across the area of the boundary.

Residual (-) = $u - v$

v = coupled variable

Additional Computations ➔ None

Usage

Generally you invoke this kernel in the same way you would use the MOOSE standard Dirichlet BC, but the value at the boundary can be a variable

```
[./dirichlet]
        type = CoupledDirichletBC
        variable = u
        boundary = 'inlet'
        coupled = u_in
[../]
```

CoupledNeumannBC

Inheritance ➔ IntegratedBC (i.e., the MOOSE base kernel for flux-based boundaries)

Notes ➔ This kernel can be used to couple another non-linear MOOSE variable at the boundary of the domain. The primary purpose would be to use this kernel in conjunction with a pressure or potential variable wherein the value of the slope at the boundary is equal to the velocity (for pressure) or current (for potential). Or, this kernel can be used like the regular Neumann BC in MOOSE.

Residual Formulation

Residual (units depend on the variable coupled and variable acted on) = $-\varphi \cdot v$

φ = MOOSE variable test function (for FE formulation)

v = coupled variable

Additional Computations ➔ None

Usage

Generally you invoke this kernel in the same way you would use the MOOSE standard Neumann BC, but the value at the boundary can be a variable. The example below is a typical way to setup the inlet BC for a Darcy Flow problem.

```
[./neumann]
        type = CoupledNeumannBC
        variable = pressure
        boundary = 'inlet'
        coupled = vel_in
[../]
```

CoupledVariableFluxBC

Inheritance ➔ IntegratedBC (i.e., the MOOSE base kernel for flux-based boundaries)

Notes ➔ This kernel can be used for both inlet and outlet boundaries for the flow of material carried into and out of a domain through a 'Flux' variable or given set of values. The material flux at a boundary determined from the dot product of the 'Flux' vector with the normal vector at the boundary. The 'Fluxes' should be given in units that reflect 'quantity/area/time' where the 'quantity' can be any conserved value. Since this kernel couples with a 'flux' variable inside

the domain to determine exit fluxes, the user is then responsible for determining a calculation for that flux variable within the set of Kernels for the run file.

Residual Formulation

Note: The residuals are integrated across the area of the boundary.

Residual (quantity / area / time) = $\varphi \cdot (\mathbf{F} \cdot \mathbf{n})$

φ = MOOSE variable test function (for FE formulation)

F = flux vector at the boundaries (quantity/area/time)

n = normal vector at the boundaries

Additional Computations ➔ None

Usage

You can invoke this same kernel for both inlet and outlet boundaries (i.e., any open boundaries in the domain) and the sign of the flux variables dictates automatically whether this is an input or output. You would need another kernel or auxiliary kernel to define how the flux varies spatially in the domain and at each respective boundary.

```
[./flux]
    type = CoupledVariableFluxBC
    variable = u
    boundary = 'inlet outlet'
    fx = flux_x
    fy = flux_y
    fz = flux_z
[../]
```

CoupledVariableGradientFluxBC

Inheritance ➔ IntegratedBC (i.e., the MOOSE base kernel for flux-based boundaries)

Notes ➔ This kernel can be used for both inlet and outlet boundaries for the flow of material carried into and out of a domain through the gradient of another variable (not itself). The material flux at a boundary determined from the dot product of the coupled variable's gradient with the normal vector at the boundary. User may also optionally provide a coupled coefficient (or constant value) for the multiplier, *K*.

Residual Formulation

Note: The residuals are integrated across the area of the boundary.

Residual (quantity / area / time) = $-\varphi \cdot K(\nabla \mathrm{v} \cdot \mathbf{n})$

φ = MOOSE variable test function (for FE formulation)

v = coupled variable whose gradient dictates the flux of material

K = coefficient multiplier (default = 1)

n = normal vector at the boundaries

Additional Computations ➔ None

Usage

You can invoke this same kernel for either inlet or outlet boundaries. The coupled variable is required by the boundary condition and must be solved in the domain using other kernels. The coefficient can be another coupled variable or a given constant.

```
[./flux]
        type = CoupledVariableGradientFluxBC
        variable = u
        boundary = 'inlet outlet'
        coupled = v
        coef = 5
[../]
```

DGConcFluxLimitedStepwiseBC

Inheritance ➔ DGConcentrationFluxLimitedBC

Notes ➔ This kernel uses the residuals form DGConcentrationFluxLimitedBC, but changes the u_input term in a stepwise fashion based on a list of values to update it to and a corresponding list of times when it gets updated. Optionally, the user can provide another list of arguments for the span of time over which the stepwise increase occurs. This allows for the simulation to slowly ramp up the inlet values from its previous time value to its future time value.

(See DGConcentrationFluxLimitedBC for additional notes)

Residual Formulation ➔ (See DGConcentrationFluxLimitedBC for residual formulation)

Additional Computations ➔ Value of u_input is updated at specified times from the user.

Usage

Generally you invoke this same kernel for both inlet and outlet boundaries. The kernel knows implicitly whether the boundary is an inlet or outlet based on the velocity vector given.

This example is for a variable velocity vector (given component-wise) where the non-linear variable (u) has an inlet value of 1. The user must provide a list of the names of the boundaries that this kernel applies to. Since this is a 'FluxLimited' boundary, you must also provide diffusion values to fill out the penalty terms.

The value of u_input will start out as 1, then be stepped down to 0.5 at a time of 1, then down to 0.25 at a time of 2. During stepdown, the u_input will gradually change for a period of 0.1 time units centered around each time when the step changes occur.

```
[./flux]
        type = DGConcFluxLimitedStepwiseBC
        variable = u
        boundary = 'inlet outlet'
        u_input = 1
        input_vals =   '0.5   0.25'
        input_times = '1      2'
        time_spans =  '0.1    0.1'
        ux = vel_x
        uy = vel_y
        uz = vel_z
        Dxx = 1
        Dyy = 1
        Dzz = 1
[../]
```

DGConcFluxStepwiseBC

Inheritance ➔ DGConcentrationFluxBC

Notes ➔ This kernel uses the residuals form DGConcentrationFluxBC, but changes the u_input term in a stepwise fashion based on a list of values to update it to and a corresponding list of times when it gets updated. Optionally, the user can provide another list of arguments for the span of time over which the stepwise increase occurs. This allows for the simulation to slowly ramp up the inlet values from its previous time value to its future time value.

(See DGConcentrationFluxBC for additional notes)

Residual Formulation ➔ (See DGConcentrationFluxBC for residual formulation)

Additional Computations ➔ Value of u_input is updated at specified times from the user.

Usage

Generally you invoke this same kernel for both inlet and outlet boundaries. The kernel knows implicitly whether the boundary is an inlet or outlet based on the velocity vector given.

This example is for a variable velocity vector (given component-wise) where the non-linear variable (u) has an inlet value of 1. The user must provide a list of the names of the boundaries that this kernel applies to.

The value of u_input will start out as 1, then be stepped down to 0.5 at a time of 1, then down to 0.25 at a time of 2. During stepdown, the u_input will gradually change for a period of 0.1 time units centered around each time when the step changes occur.

```
[./flux]
        type = DGFluxStepwiseBC
        variable = u
        boundary = 'inlet outlet'
        u_input = 1
```

```
        input_vals =   '0.5   0.25'
        input_times = '1      2'
        time_spans =  '0.1    0.1'
        ux = vel_x
        uy = vel_y
        uz = vel_z
[../]
```

DGConcentrationFluxBC

Inheritance ➔ DGFluxBC

Notes ➔ This kernel uses the residuals form DGFluxBC, but overrides the velocity vector with variables for the velocity in each component direction. This is most useful when the velocity field is calculated by other kernels or MOOSE modules, such as Navier-Stokes.

(See DGFluxBC for additional notes)

Residual Formulation ➔ (See DGFluxBC for residual formulation)

Additional Computations ➔ The velocity vector is overridden with new variable components.

Usage

Generally you invoke this same kernel for both inlet and outlet boundaries. The kernel knows implicitly whether the boundary is an inlet or outlet based on the velocity vector given.

This example is for a variable velocity vector (given component-wise) where the non-linear variable (u) has an inlet value of 1. The user must provide a list of the names of the boundaries that this kernel applies to.

```
[./flux]
        type = DGConcentrationFluxBC
        variable = u
        boundary = 'inlet outlet'
        u_input = 1
        ux = vel_x
        uy = vel_y
        uz = vel_z
[../]
```

DGConcentrationFluxLimitedBC

Inheritance ➔ DGFluxLimitedBC

Notes ➔ This kernel uses the residuals form DGFluxLimitedBC, but overrides the velocity vector with variables for the velocity in each component direction. This is most useful when the velocity field is calculated by other kernels or MOOSE modules, such as Navier-Stokes.

(See DGFluxLimitedBC for additional notes)

Residual Formulation ➔ (See DGFluxLimitedBC for residual formulation)

Additional Computations ➔ The velocity vector is overridden with new variable components.

Usage

Generally you invoke this same kernel for both inlet and outlet boundaries. The kernel knows implicitly whether the boundary is an inlet or outlet based on the velocity vector given.

This example is for a variable velocity vector (given component-wise) where the non-linear variable (u) has an inlet value of 1. The user must provide a list of the names of the boundaries that this kernel applies to. Since this is a 'FluxLimited' boundary, you must also provide diffusion values to fill out the penalty terms.

```
[./flux]
        type = DGConcentrationFluxLimitedBC
        variable = u
        boundary = 'inlet outlet'
        u_input = 1
        ux = vel_x
        uy = vel_y
        uz = vel_z
        Dxx = 1
        Dyy = 1
        Dzz = 1
[../]
```

DGDiffuseFlowMassFluxBC

Inheritance ➔ DGPoreDiffFluxLimitedBC

Notes ➔ This kernel uses the residuals form DGPoreDiffFluxLimitedBC, but replaces the used provided 'u_input' constant with a coupled variable. This is the most generic of the mass flux BCs and is widely applicable to many material balances in CATS. The advantage of using this BC over DGPoreDiffFluxLimitedBC is that the user can define the input as any variable or auxiliary variable calculated as a function of time.

(See DGPoreDiffFluxLimitedBC for additional notes)

Residual Formulation ➔ See DGPoreDiffFluxLimitedBC

Additional Computations ➔ 'u_input' is replaced by a variable the user provides

Usage

Generally you invoke this same kernel for both inlet and outlet boundaries. The kernel knows implicitly whether the boundary is an inlet or outlet based on the velocity vector given. In the example below, the user provides the value at the inlet boundary as variable 'u_in'.

```
[./u_flux]
        type = DGDiffuseFlowMassFluxBC
```

variable = u
boundary = 'inlet outlet'
porosity = 0.5
ux = vel_x
uy = vel_y
uz = vel_z
Dx = 0.1
Dy = 0.1
Dz = 0.1
input_var = u_in
[../]

DGDiffusionFluxBC

Inheritance ➔ IntegratedBC (i.e., the MOOSE base kernel for flux-based boundaries)

Notes ➔ This kernel is used for both inlet and outlet boundaries for the flow of mass or energy carried into and out of a domain through a diffusion process ***ONLY***. Generally, problems in CATS always involve some advective flux, so you would usually use DGFluxBC or DGFluxLimitedBC or any of their derivatives, to resolve flux-based BCs. However, in the cases where you have a diffusion only process, this kernel can be applied to enforce flux out of (or into) a domain. When doing so, material will flux into an 'empty' space, thus, you will always have non-zero gradients at exit boundaries. This kernel requires the same parameter arguments from GAnisotropicDiffusion (DGAnisotropicDiffusion) in order to penalize variation at the boundary from the given input or outside value. Thus, the kernel attempts to enforce the non-linear variable to a given constant input value based on the DG formulation and penalty terms.

NOTE: By default, the input value is assume 0, i.e., there is no concentration of the non-linear variable outside the domain, thus, material fluxes into an 'empty' space.

(See GAnisotropicDiffusion (DGAnisotropicDiffusion) for special notes on penalty terms)

Residual Formulation

Note: The residuals are integrated across the area of the boundary.

Note: The formulation of the inlet boundary is very complex and long, so the full residual expression is not show here.

Note: If an inlet term (u_{inlet}) for the non-linear variable is given, then this BC essentially applies a 'penalized' Dirichlet type of BC.

Residual (mass / area / time) =

if $u_{inlet} = 0$ $\quad \varphi \cdot (-\mathbf{D} \cdot \nabla u \cdot \mathbf{n})$

if $u_{inlet} \neq 0$ $\quad \approx \varphi \cdot p \cdot (u - u_{inlet})$

φ = MOOSE variable test function (for FE formulation)

D = constant diffusion tensor at the boundaries

n = normal vector at the boundaries

p = penalty term based on diffusion coefficients and other

DGAnisotropicDiffusion parameter arguments

u = non-linear variable at the interior of the domain

u_{inlet} = input value for the variable at the boundary (default = 0)

Additional Computations ➔ None

Usage

Generally, this is not a BC you would use for a real type of problem, but would act as a base for derived kernels to build off of. Only use this kernel if you have a diffusion only process with DG methods and want to enforce constant diffusion flux at the boundary.

ONLY provide the 'u_input' argument if the boundary is to behave as an inlet. Otherwise, when not provided, the boundary behaves as an outlet into a empty domain.

```
[./flux_in]
        type = DGDiffusionFluxBC
        variable = u
        boundary = 'inlet'
        u_input = 1
        Dxx = 1
        Dyy = 1
        Dzz = 1
[../]
[./flux_out]
        type = DGDiffusionFluxBC
        variable = u
        boundary = 'outlet'
        Dxx = 1
        Dyy = 1
        Dzz = 1
[../]
```

DGFlowEnergyFluxBC

Inheritance ➔ DGConcentrationFluxBC

Notes ➔ This kernel uses the residuals form DGConcentrationFluxBC, but is specific for an energy balance where the conserved quantity is the energy density of a specific phase in the domain. The flux of energy into an open boundary is dictated by the velocity variable that carries the fluid phase into the domain. Then, the energy density that is carried into, or out of,

the domain is a function of the variables for phase density, phase porosity, phase specific heat, and phase temperature.

(See DGConcentrationFluxBC for additional notes)

Residual Formulation

Note: The residuals are integrated across the area of the boundary.

Residual (J / m^2 / s) =

for $(\mathbf{v} \cdot \mathbf{n}) > 0$ $\varphi \cdot (\varepsilon \mathbf{v} \cdot \mathbf{n}) \cdot E$

for $(\mathbf{v} \cdot \mathbf{n}) \leq 0$ $\varphi \cdot (\varepsilon \mathbf{v} \cdot \mathbf{n}) \cdot (\rho c T_{inlet})$

φ = MOOSE variable test function (for FE formulation)

v = velocity vector at the boundaries

n = normal vector at the boundaries

E = non-linear variable for energy density (J/m^3)

ε = domain porosity (volume voids per total volume)

ρ = phase density (kg/m^3)

c = phase specific heat ($J/kg/m^3$)

T_{inlet} = input value for the temperature at the boundary (K)

Additional Computations ➔ None

Usage

Generally you invoke this same kernel for both inlet and outlet boundaries. The kernel knows implicitly whether the boundary is an inlet or outlet based on the velocity vector given.

This example is for a variable velocity vector (given component-wise) where the energy density variable (E) is calculated at the inlet base on an inlet temperature (Tin) of 298 K along with the porosity, density, and specific heats given at the inlet. The user must provide a list of the names of the boundaries that this kernel applies to.

```
[./flux]
    type = DGFlowEnergyFluxBC
    variable = E
    boundary = 'inlet outlet'
    porosity = 0.5
    specific_heat = 1000
    density = 1
    inlet_temp = 298
    ux = vel_x
    uy = vel_y
```

```
        uz = vel_z
[../]
```

DGFlowMassFluxBC

Inheritance ➔ DGPoreConcFluxBC

Notes ➔ This kernel uses the residuals form DGPoreConcFluxBC, but replaces the used provided 'u_input' constant with a coupled variable. This is the most generic of the mass flux BCs and is widely applicable to many material balances in CATS. The advantage of using this BC over DGPoreConcFluxBC is that the user can define the input as any variable or auxiliary variable calculated as a function of time.

(See DGPoreConcFluxBC for additional notes)

Residual Formulation ➔ See DGPoreConcFluxBC

Additional Computations ➔ 'u_input' is replaced by a variable the user provides

Usage

Generally you invoke this same kernel for both inlet and outlet boundaries. The kernel knows implicitly whether the boundary is an inlet or outlet based on the velocity vector given. In the example below, the user provides the value at the inlet boundary as variable 'u_in'.

```
[./u_flux]
        type = DGFlowMassFluxBC
        variable = u
        boundary = 'inlet outlet'
        porosity = 0.5
        ux = vel_x
        uy = vel_y
        uz = vel_z
        input_var = u_in
[../]
```

DGFluxBC

Inheritance ➔ IntegratedBC (i.e., the MOOSE base kernel for flux-based boundaries)

Notes ➔ This kernel is used for both inlet and outlet boundaries for the flow of mass or energy carried into and out of a domain through an advective process. It should be invoked at all "open" boundaries. It uses the velocity vector and the normal vector to the boundary to determine whether or not mass/energy is flowing into or out of the system. When it finds that the boundary is an outlet, it uses the interior values of the non-linear variable to establish exit flux. When it finds that the boundary is an inlet, it uses the user specified inlet value to establish flux of mass/energy into the domain.

Residual Formulation

Note: The residuals are integrated across the area of the boundary.

Residual (mass / area / time) =

for $(\mathbf{v} \cdot \mathbf{n}) > 0$ $\varphi \cdot (\mathbf{v} \cdot \mathbf{n}) \cdot u$

for $(\mathbf{v} \cdot \mathbf{n}) \leq 0$ $\varphi \cdot (\mathbf{v} \cdot \mathbf{n}) \cdot u_{inlet}$

φ = MOOSE variable test function (for FE formulation)

v = velocity vector at the boundaries

n = normal vector at the boundaries

u = non-linear variable at the interior of the domain

u_{inlet} = input value for the variable at an inlet

Additional Computations ➔ None

Usage

Generally you invoke this same kernel for both inlet and outlet boundaries (i.e., any open boundaries in the domain). The kernel knows implicitly whether the boundary is an inlet or outlet based on the velocity vector given.

This example is for a constant velocity vector (given component-wise) where the non-linear variable (u) has an inlet value of 1. The user must provide a list of the names of the boundaries that this kernel applies to.

```
[./flux]
        type = DGFluxBC
        variable = u
        boundary = 'inlet outlet'
        u_input = 1
        vx = 1
        vy = 0
        vz = 0
[../]
```

DGFluxLimitedBC

Inheritance ➔ IntegratedBC (i.e., the MOOSE base kernel for flux-based boundaries)

Notes ➔ This kernel is used for both inlet and outlet boundaries for the flow of mass or energy carried into and out of a domain through an advective process. The outlet boundary will inherently use advective flux at the exit, but the inlet boundary will attempt to emulate a Dirichlet type boundary condition. This emulation requires the same parameter arguments from GAnisotropicDiffusion (DGAnisotropicDiffusion) in order to penalize variation at the inlet from the given inlet value. Thus, the kernel attempts to enforce the non-linear variable to a given constant inlet value based on the DG formulation and penalty terms.

(See GAnisotropicDiffusion (DGAnisotropicDiffusion) for special notes on penalty terms)

Residual Formulation

Note: The residuals are integrated across the area of the boundary.

Note: The formulation of the inlet boundary is very complex and long, so the full residual expression is not show here.

Residual (mass / area / time) =

for $(\mathbf{v} \cdot \mathbf{n}) > 0$ $\varphi \cdot (\mathbf{v} \cdot \mathbf{n}) \cdot u$

for $(\mathbf{v} \cdot \mathbf{n}) \leq 0$ $\approx \varphi \cdot p \cdot (u - u_{inlet})$

φ = MOOSE variable test function (for FE formulation)

v = velocity vector at the boundaries

n = normal vector at the boundaries

p = penalty term based on diffusion coefficients and other DGAnisotropicDiffusion parameter arguments

u = non-linear variable at the interior of the domain

u_{inlet} = input value for the variable at an inlet

Additional Computations ➔ None

Usage

Generally you invoke this same kernel for both inlet and outlet boundaries. The kernel knows implicitly whether the boundary is an inlet or outlet based on the velocity vector given.

This example is for a constant velocity vector (given component-wise) where the non-linear variable (u) has an inlet value of 1. The user must provide a list of the names of the boundaries that this kernel applies to. Since this is a 'FluxLimited' boundary, you must also provide diffusion values to fill out the penalty terms.

```
[./flux]
    type = DGFluxLimitedBC
    variable = u
    boundary = 'inlet outlet'
    u_input = 1
    vx = 1
    vy = 0
    vz = 0
    Dxx = 1
    Dyy = 1
    Dzz = 1
[../]
```

DGFluxLimitedStepwiseBC

Inheritance ➔ DGFluxLimitedBC

Notes ➔ This kernel uses the residuals form DGFluxLimitedBC, but changes the u_input term in a stepwise fashion based on a list of values to update it to and a corresponding list of times when it gets updated. Optionally, the user can provide another list of arguments for the span of time over which the stepwise increase occurs. This allows for the simulation to slowly ramp up the inlet values from its previous time value to its future time value.

(See DGFluxLimitedBC for additional notes)

Residual Formulation ➔ (See DGFluxLimitedBC for residual formulation)

Additional Computations ➔ Value of u_input is updated at specified times from the user.

Usage

Generally you invoke this same kernel for both inlet and outlet boundaries. The kernel knows implicitly whether the boundary is an inlet or outlet based on the velocity vector given.

This example is for a constant velocity vector (given component-wise) where the non-linear variable (u) has an inlet value of 1. The user must provide a list of the names of the boundaries that this kernel applies to. Since this is a 'FluxLimited' boundary, you must also provide diffusion values to fill out the penalty terms.

The value of u_input will start out as 1, then be stepped down to 0.5 at a time of 1, then down to 0.25 at a time of 2. During stepdown, the u_input will gradually change for a period of 0.1 time units centered around each time when the step changes occur.

```
[./flux]
        type = DGFluxLimitedStepwiseBC
        variable = u
        boundary = 'inlet outlet'
        u_input = 1
        input_vals =   '0.5   0.25'
        input_times = '1      2'
        time_spans =  '0.1    0.1'
        vx = 1
        vy = 0
        vz = 0
        Dxx = 1
        Dyy = 1
        Dzz = 1
[../]
```

DGFluxStepwiseBC

Inheritance ➔ DGFluxBC

Notes ➔ This kernel uses the residuals form DGFluxBC, but changes the u_input term in a stepwise fashion based on a list of values to update it to and a corresponding list of times when it gets updated. Optionally, the user can provide another list of arguments for the span of time over which the stepwise increase occurs. This allows for the simulation to slowly ramp up the inlet values from its previous time value to its future time value.

(See DGFluxBC for additional notes)

Residual Formulation ➔ (See DGFluxBC for residual formulation)

Additional Computations ➔ Value of u_input is updated at specified times from the user.

Usage

Generally you invoke this same kernel for both inlet and outlet boundaries. The kernel knows implicitly whether the boundary is an inlet or outlet based on the velocity vector given.

This example is for a constant velocity vector (given component-wise) where the non-linear variable (u) has an inlet value of 1. The user must provide a list of the names of the boundaries that this kernel applies to.

The value of u_input will start out as 1, then be stepped down to 0.5 at a time of 1, then down to 0.25 at a time of 2. During stepdown, the u_input will gradually change for a period of 0.1 time units centered around each time when the step changes occur.

```
[./flux]
        type = DGFluxStepwiseBC
        variable = u
        boundary = 'inlet outlet'
        u_input = 1
        input_vals =   '0.5   0.25'
        input_times = '1      2'
        time_spans =  '0.1    0.1'
        vx = 1
        vy = 0
        vz = 0
 [../]
```

DGNSMomentumOutflowBC

Inheritance ➔ DGConcentrationFluxBC

Notes ➔ This kernel uses the residuals form DGConcentrationFluxBC, but scales the flux by the density of the domain and modifies the Jacobian elements to account for the fact that the variable we are coupling to is also a variable in the velocity vector. This is a boundary condition that should apply to ANY outlet/open boundary for conservation of momentum using the DG implementation of Incompressible Navier-Stokes equations.

(See DGConcentrationFluxBC for additional notes)

Residual Formulation

Note: The residuals are integrated across the area of the boundary.

Residual (momentum / area / time) =

for $(\mathbf{v} \cdot \mathbf{n}) > 0$ $\quad residual = \varphi \cdot (\mathbf{v} \cdot \mathbf{n}) \cdot (\rho u_i)$

for $(\mathbf{v} \cdot \mathbf{n}) \leq 0$ $\quad residual = 0$

φ = MOOSE variable test function (for FE formulation)

v = velocity vector at the boundaries

[This vector will hold variables for velocity in each direction]

n = normal vector at the boundaries

ρ = fluid density (mass / volume)

u_i = non-linear variable for velocity in the i-th direction

where i = x, y, or z

Additional Computations ➔ None

Usage

Generally, whenever you use the DG implementation of Navier-Stokes equations, you need to allow the momentum in the domain to leave out the open boundary. This kernel will apply that boundary condition to each velocity variable (one at a time). Thus, for each velocity variable, you MUST invoke this BC kernel at EACH open boundary.

```
[./momentum_flux_in_x]
        type = DGNSMomentumOutflowBC
        variable = vel_x
        this_variable = vel_x
        boundary = 'outlet'
        density = rho
        ux = vel_x
        uy = vel_y
        uz = vel_z
 [../]
```

DGPoreConcFluxBC_ppm

Inheritance ➔ DGPoreConcFluxBC

Notes ➔ This kernel uses the residuals form DGPoreConcFluxBC, but changes the inlet values from a user provided value in ppm to a moles per liter (mol/L) inlet concentration using ideal gas law and coupled variables for pressure and temperature at the boundary. This kernel is very

convenient for simulations in which your inlet temperatures and/or pressure are changing, but you want to specify a constant mole-fraction or volume-per-volume inlet as constant.

(See DGPoreConcFluxBC for additional notes)

Residual Formulation

Note: The residuals are integrated across the area of the boundary.

Residual (mass / area / time) = See DGPoreConcFluxBC

Additional Computations ➔ The value of u_{inlet} (see DGPoreConcFluxBC) is calculated from the user provided inlet in ppm as well as variables for temperature (T) and pressure (P) using ideal gas law.

$$u_{inlet} = \frac{P \cdot (y_{ppm}/10^6)}{RT}$$

P = inlet pressure (in kPa)

T = inlet temperature (in K)

R = gas law constant = 8.3145 L*kPa/mol/K

y_{ppm} = inlet concentration in ppm

Usage

Generally you invoke this same kernel for both inlet and outlet boundaries. The kernel knows implicitly whether the boundary is an inlet or outlet based on the velocity vector given.

This example is for a variable velocity vector (given component-wise) where the non-linear variable (u) has an inlet value of 1000 ppmv. In addition, the porosity of the domain is a variable named eps and the temperature is a variable named temperature. The user must provide a list of the names of the boundaries that this kernel applies to.

```
[./flux]
        type = DGPoreConcFluxBC
        variable = u
        boundary = 'inlet outlet'
        porosity = eps
        ux = vel_x
        uy = vel_y
        uz = vel_z
        pressure = 101.35
        temperature = temp
        inlet_ppm = 1000
[../]
```

DGPoreConcFluxBC

Inheritance ➔ DGConcentrationFluxBC

Notes ➔ This kernel uses the residuals form DGConcentrationFluxBC, but scales the flux by the porosity of the domain. Thus, this kernel is most appropriate for boundary conditions for porous flow or packed bed domains.

(See DGConcentrationFluxBC for additional notes)

Residual Formulation

Note: The residuals are integrated across the area of the boundary.

Residual (mass / area / time) =

for $(\mathbf{v} \cdot \mathbf{n}) > 0$ $\varphi \cdot (\mathbf{v} \cdot \mathbf{n}) \cdot (\varepsilon u)$

for $(\mathbf{v} \cdot \mathbf{n}) \leq 0$ $\varphi \cdot (\mathbf{v} \cdot \mathbf{n}) \cdot (\varepsilon u_{inlet})$

φ = MOOSE variable test function (for FE formulation)

v = velocity vector at the boundaries

n = normal vector at the boundaries

ε = domain porosity (volume voids per total volume)

u = non-linear variable for concentration (mass/volume)

u_{inlet} = input value for concentration (mass/volume)

Additional Computations ➔ None

Usage

Generally you invoke this same kernel for both inlet and outlet boundaries. The kernel knows implicitly whether the boundary is an inlet or outlet based on the velocity vector given.

This example is for a variable velocity vector (given component-wise) where the non-linear variable (u) has an inlet value of 1. In addition, the porosity of the domain is a variable named eps. The user must provide a list of the names of the boundaries that this kernel applies to.

```
[./flux]
    type = DGPoreConcFluxBC
    variable = u
    boundary = 'inlet outlet'
    porosity = eps
    u_input = 1
    ux = vel_x
    uy = vel_y
    uz = vel_z
[../]
```

DGPoreConcFluxStepwiseBC

Inheritance ➔ DGPoreConcFluxBC

Notes ➔ This kernel uses the residuals form DGPoreConcFluxBC, but changes the u_input term in a stepwise fashion based on a list of values to update it to and a corresponding list of times when it gets updated. Optionally, the user can provide another list of arguments for the span of time over which the stepwise increase occurs. This allows for the simulation to slowly ramp up the inlet values from its previous time value to its future time value.

(See DGPoreConcFluxBC for additional notes)

Residual Formulation ➔ (See DGPoreConcFluxBC for residual formulation)

Additional Computations ➔ Value of u_input is updated at specified times from the user.

Usage

Generally you invoke this same kernel for both inlet and outlet boundaries. The kernel knows implicitly whether the boundary is an inlet or outlet based on the velocity vector given.

This example is for a variable velocity vector (given component-wise) where the non-linear variable (u) has an inlet value of 1. In addition, the porosity of the domain is a variable named eps. The user must provide a list of the names of the boundaries that this kernel applies to.

The value of u_input will start out as 1, then be stepped down to 0.5 at a time of 1, then down to 0.25 at a time of 2. During stepdown, the u_input will gradually change for a period of 0.1 time units centered around each time when the step changes occur.

```
[./flux]
        type = DGPoreConcFluxStepwiseBC
        variable = u
        porosity = eps
        boundary = 'inlet outlet'
        u_input = 1
        input_vals =   '0.5   0.25'
        input_times = '1      2'
        time_spans =  '0.1    0.1'
        ux = vel_x
        uy = vel_y
        uz = vel_z
 [../]
```

DGPoreDiffFluxLimitedBC

Inheritance ➔ DGVarVelDiffFluxLimitedBC

Notes ➔ This kernel uses the residuals form DGVarVelDiffFluxLimitedBC, but scales the flux by the porosity of the domain. Thus, this kernel is most appropriate for boundary conditions for porous flow or packed bed domains. Note that this is the version of the flux that emulates the Dirichlet boundary condition.

(See DGVarVelDiffFluxLimitedBC for additional notes)

(See GAnisotropicDiffusion (DGAnisotropicDiffusion) for notes on penalty terms)

Residual Formulation

Note: The residuals are integrated across the area of the boundary.

Residual (mass / area / time) =

for $(\mathbf{v} \cdot \mathbf{n}) > 0$ $\varphi \cdot (\mathbf{v} \cdot \mathbf{n}) \cdot (\varepsilon u)$

for $(\mathbf{v} \cdot \mathbf{n}) \leq 0$ $\approx \varphi \cdot p \cdot (u - u_{inlet})$

φ = MOOSE variable test function (for FE formulation)

v = velocity vector at the boundaries

n = normal vector at the boundaries

ε = domain porosity (volume voids per total volume)

p = penalty term applied to emulate Dirichlet boundary condition

u = non-linear variable for concentration (mass/volume)

u_{inlet} = input value for concentration (mass/volume)

Additional Computations ➔ None

Usage

Generally you invoke this same kernel for both inlet and outlet boundaries. The kernel knows implicitly whether the boundary is an inlet or outlet based on the velocity vector given.

This example is for a variable velocity vector (given component-wise) where the non-linear variable (u) has an inlet value of 1. In addition, the porosity of the domain is a variable named eps. The user must provide a list of the names of the boundaries that this kernel applies to.

```
[./flux]
        type = DGPoreDiffFluxLimitedBC
        variable = u
        boundary = 'inlet outlet'
        porosity = eps
        u_input = 1
        ux = vel_x
        uy = vel_y
        uz = vel_z
        Dx = Diff
        Dy = Diff
        Dz = Diff
[../]
```

DGPoreDiffFluxLimitedStepwiseBC

Inheritance ➔ DGPoreDiffFluxLimitedBC

Notes ➔ This kernel uses the residuals form DGPoreDiffFluxLimitedBC, but changes the u_input term in a stepwise fashion based on a list of values to update it to and a corresponding list of times when it gets updated. Optionally, the user can provide another list of arguments for the span of time over which the stepwise increase occurs. This allows for the simulation to slowly ramp up the inlet values from its previous time value to its future time value.

(See DGPoreDiffFluxLimitedBC for additional notes)

Residual Formulation ➔ (See DGPoreDiffFluxLimitedBC for residual formulation)

Additional Computations ➔ Value of u_input is updated at specified times from the user.

Usage

Generally you invoke this same kernel for both inlet and outlet boundaries. The kernel knows implicitly whether the boundary is an inlet or outlet based on the velocity vector given.

This example is for a constant velocity vector (given component-wise) where the non-linear variable (u) has an inlet value of 1. The user must provide a list of the names of the boundaries that this kernel applies to. In addition, the porosity of the domain is a variable named eps. Since this is a 'FluxLimited' boundary, you must also provide diffusion variables to fill out the penalty terms.

The value of u_input will start out as 1, then be stepped down to 0.5 at a time of 1, then down to 0.25 at a time of 2. During stepdown, the u_input will gradually change for a period of 0.1 time units centered around each time when the step changes occur.

```
[./flux]
        type = DGPoreDiffFluxLimitedStepwiseBC
        variable = u
        porosity = eps
        boundary = 'inlet outlet'
        u_input = 1
        input_vals =   '0.5   0.25'
        input_times = '1      2'
        time_spans =  '0.1    0.1'
        ux = vel_x
        uy = vel_y
        uz = vel_z
        Dx = Diff
        Dy = Diff
        Dz = Diff
 [../]
```

DGVarVelDiffFluxLimitedBC

Inheritance ➔ DGConcentrationFluxLimitedBC

Notes ➔ This kernel uses the residuals form DGConcentrationFluxLimitedBC, but allows the diffusion parameter to be a function of other non-linear variables for each x, y, z direction.

(See DGConcentrationFluxLimitedBC for additional notes)

(See GVariableDiffusion (DGVariableDiffusion) for notes on variable diffusion)

Residual Formulation ➔ (See DGConcentrationFluxLimitedBC for residual formulation)

Additional Computations ➔ Value of diffusion tensor replaced with variables Dx, Dy, and Dz.

Usage

Generally you invoke this same kernel for both inlet and outlet boundaries. The kernel knows implicitly whether the boundary is an inlet or outlet based on the velocity vector given.

This example is for a variable velocity vector (given component-wise) where the non-linear variable (u) has an inlet value of 1. The user must provide a list of the names of the boundaries that this kernel applies to. Since this is a 'FluxLimited' boundary, you must also provide diffusion values to fill out the penalty terms. In this formulation, the diffusion values are other variables.

```
[./flux]
        type = DGVarVelDiffFluxLimitedBC
        variable = u
        boundary = 'inlet outlet'
        u_input = 1
        ux = vel_x
        uy = vel_y
        uz = vel_z
        Dx = Diff
        Dy = Diff
        Dz = Diff
[../]
```

DGVarVelDiffFluxLimitedStepwiseBC

Inheritance ➔ DGVarVelDiffFluxLimitedBC

Notes ➔ This kernel uses the residuals form DGVarVelDiffFluxLimitedBC, but changes the u_input term in a stepwise fashion based on a list of values to update it to and a corresponding list of times when it gets updated. Optionally, the user can provide another list of arguments for the span of time over which the stepwise increase occurs. This allows for the simulation to slowly ramp up the inlet values from its previous time value to its future time value.

(See DGVarVelDiffFluxLimitedBC for additional notes)

Residual Formulation ➔ (See DGVarVelDiffFluxLimitedBC for residual formulation)

Additional Computations ➔ Value of u_input is updated at specified times from the user.

Usage

Generally you invoke this same kernel for both inlet and outlet boundaries. The kernel knows implicitly whether the boundary is an inlet or outlet based on the velocity vector given.

This example is for a variable velocity vector (given component-wise) where the non-linear variable (u) has an inlet value of 1. The user must provide a list of the names of the boundaries that this kernel applies to. Since this is a 'FluxLimited' boundary, you must also provide diffusion values to fill out the penalty terms. In this formulation, the diffusion values are other variables.

The value of u_input will start out as 1, then be stepped down to 0.5 at a time of 1, then down to 0.25 at a time of 2. During stepdown, the u_input will gradually change for a period of 0.1 time units centered around each time when the step changes occur.

```
[./flux]
        type = DGVarVelDiffFluxLimitedStepwiseBC
        variable = u
        boundary = 'inlet outlet'
        u_input = 1
        input_vals =   '0.5   0.25'
        input_times = '1      2'
        time_spans =  '0.1    0.1'
        ux = vel_x
        uy = vel_y
        uz = vel_z
        Dx = Diff
        Dy = Diff
        Dz = Diff
[../]
```

DGWallEnergyFluxBC

Inheritance ➔ IntegratedBC (i.e., the MOOSE base kernel for flux-based boundaries)

Notes ➔ This kernel provides residuals for a closed boundary wherein energy is transferred into the domain via a heat transfer coefficient and direct contact with an entity outside the domain. The residual is also scaled by an area fraction in the case of a multi-phase domain wherein only part of the phase is in direct contact with the wall. Both the fluid phase and solid phase would need this kernel if simulating energy density in 2 phases. Also note that this kernel acts on the energy density variable and not the phase temperature.

Residual Formulation

Residual (J / m^2 / s) = $\varphi \cdot h \cdot f_a \cdot (T - T_w)$

φ = MOOSE variable test function (for FE formulation)

h = heat transfer coefficient at the wall (W/m^2/K)

f_a = area fraction for the phase in contact with the wall

T = variable for phase temperature at the interior of the domain (K)

T_w = variable for temperature of the wall (K)

Additional Computations ➔ None

Usage

Used to create the transfer of energy via contact of a phase with the wall. The variable this kernel acts on is the energy density, not the temperature.

In this example, the energy transfer at a wall boundary is dictated by a variable wall temperature (Tw) with a constant area fraction of 1 and a constant heat transfer of 50 $W/m^2/K$. The internal temperature variable is T and contributes to the residual for internal energy (E).

```
[./wallflux]
    type = DGWallEnergyFluxBC
    variable = E
    boundary = 'wall1 wall2'
    transfer_coef = 50
    area_frac = 1
    wall_temp = Tw
    temperature = T
[../]
```

DirichletBC

Inheritance ➔ NodalBC (i.e., the MOOSE base kernel for flux-based boundaries)

Notes ➔ This kernel provides residuals for a generic Dirichlet type boundary condition in the MOOSE system. **NOTE**: Since the majority of CATS uses DG methods for mass and energy balances, this kernel would NEVER be used for any of those balances and is ONLY valid for the incompressible Navier-Stokes module that CATS can be linked to. CATS uses this kernel only to specify a velocity value at a boundary for incompressible Navier-Stokes. You can use it to specify a value at an inlet/outlet boundary, however, it is most commonly used to specify a "no slip" boundary condition at the walls of a domain. For specification of inflow/outflow, you should use the INSNormalFlowBC or PenaltyDirichletBC.

Residual Formulation

Residual (any units) = $\varphi \cdot g$

φ = MOOSE variable test function (for FE formulation)

g = expected value at a boundary

Additional Computations ➔ None

Usage

Here, we impose that at the 'top' and 'bottom' boundary, the velocity in the y-direction should be 0 (i.e., "no slip").

```
[./strong_no_slip_y]
    type = DirichletBC
    variable = vel_y
    boundary = 'top bottom'
    value = 0
[../]
```

INSNormalFlowBC

Inheritance ➔ IntegratedBC (i.e., the MOOSE base kernel for flux-based boundaries)

Notes ➔ This kernel provides residuals for an open-boundary or closed for the MOOSE and CATS Navier-Stokes formulation to weakly specify that the flux of flow across the given boundary must equation a given scalar for the result of the dot product between the actual velocity vector and a normal vector for this boundary. In the case of an inflow boundary condition, this dot product will be a negative value. In the case of an outflow boundary condition, this dot product will be a positive value. Depending on the orientation of the boundary, you may need to provide this type of boundary condition for each component of velocity (i.e., any non-zero velocity component at this particular boundary).

Most common usage for this BC kernel is to impose a 'No Penetration' BC for the flow. By default, the expected value 'a' (see below) is set to zero. Thus, by default this kernel would enforce that the dot product of velocity with the normal vector of the boundary be zero. When that dot product is zero, no flow can cross the boundary. You can use this type of boundary condition as a substitute for the 'No Slip' type of boundary condition. In this case, the boundary allows full slip along the wall, but without also allowing a flux through that wall.

Residual Formulation

Residual (m / s) = $\varphi \cdot p \cdot \left((\mathbf{v} \cdot \boldsymbol{n}) - a\right)$

φ = MOOSE variable test function (for FE formulation)

p = a penalty term for deviations from this relationship

NOTE: Should provide a large enough value (i.e., 1e4 to 1e6)

v = velocity vector at the boundary (m/s)

n = normal vector at the boundary

a = expected value for the dot product (m/s)

Additional Computations ➔ None

Usage

Example 1: *Using this BC to enforce a constant flux through an open boundary*

In this example, all velocities should also be accompanied by “no slip” boundary conditions at the walls. Those conditions are necessary to complete the Navier-Stokes formulation (at least for the incompressible flow). The “no slip” boundary condition can be applied in a “strong” form using DirichletBC or in a “weak” form using PenaltyDirichletBC.

Here, we impose that at the ‘left’ boundary, the dot product between the velocity and the boundary normal should result in -1.15 m/s and the resulting velocity should act solely in the y-direction. Since this value is negative, this boundary would naturally represent an inflow condition. User MUST provide all other velocity component variables and the direction that this boundary residual acts on (0 = x, 1 = y, and 2 = z).

```
[./normal_flow]
    type = INSNormalFlowBC
    variable = vel_y
    direction = 1
    boundary = ‘left’
    u_dot_n = -1.15
    ux = vel_x
    uy = vel_y
    uz = vel_z
    penalty = 1e6
[../]
```

Example 2: *Using this BC to enforce a No Penetration BC for velocity*

In this example, all velocities should also be accompanied by either a define pressure at inlet and outlet, or a defined inlet velocity.

Here, we impose that at the ‘top’ and ‘bottom’ boundary, there should be no penetration of velocity in x or velocity in y across that domain. This effectively means that momentum cannot go through that boundary, but can freely slide parallel to it.

```
[./no_penetration_x]
    type = INSNormalFlowBC
    variable = vel_x
    direction = 0
    boundary = ‘top bottom’
    ux = vel_x
    uy = vel_y
    uz = vel_z
[../]
[./no_penetration_y]
    type = INSNormalFlowBC
    variable = vel_y
    direction = 1
    boundary = ‘top bottom’
    ux = vel_x
    uy = vel_y
```

uz = vel_z
[../]

PenaltyDirichletBC

Inheritance ➔ NodalBC (i.e., the MOOSE base kernel for flux-based boundaries)

Notes ➔ This kernel provides residuals for a generic penalty based, Dirichlet type boundary condition in the MOOSE system. **NOTE**: Since the majority of CATS uses DG methods for mass and energy balances, this kernel would NEVER be used for any of those balances and is ONLY valid for the incompressible Navier-Stokes module that CATS can be linked to. CATS uses this kernel only to specify a velocity value at a boundary for incompressible Navier-Stokes. It is most commonly used to weakly specify a "no slip" boundary condition at the walls of a domain, however, you can also use it to specify a specific velocity at inflow/outflow boundaries. The advantage of specifying this boundary weakly is improved convergence.

Residual Formulation

Residual (any units) = $\varphi \cdot p \cdot (u - g)$

φ = MOOSE variable test function (for FE formulation)

p = a penalty term for deviations from this relationship

NOTE: Should provide a large enough value (i.e., 1e6), but should be smaller than the penalty for INSNormalFlowBC

g = expected value at a boundary

u = non-linear nodal variable

Additional Computations ➔ None

Usage

Here, we impose that at the 'top' and 'bottom' boundary, the velocity in the y-direction should be very close to 0 (i.e., "no slip").

```
[./weak_no_slip_y]
    type = PenaltyDirichletBC
    variable = vel_y
    boundary = 'top bottom'
    value = 0
    penalty = 1000
[../]
```

Interface Kernels

These kernels are essentially like boundary conditions, but at boundaries that are internal to the overall simulation domain. When the simulation domain needs to be divided into subdomains, the interface kernels define how mass and energy is transferred to different variables in each subdomain respectively. This is particularly useful for 3D modeling of catalysts, as you can create a physical space to represent the open-air channels in a monolith catalyst as well as a separated domain for the solid, porous washcoat where the catalytic reactions take place.

Each subdomain can have its own set of non-linear variables, boundary conditions, parameters, properties, auxiliary variables, etc. Subdomains can be created using the MeshGenerator systems in MOOSE or can be created in other CAD software and imported into MOOSE. Below is an example of how to use the MeshGenerator system to create two different subdomains. In this example, we are creating a cylindrical shaped open channel (block 0) that is coated with a porous washcoat (block 1). We then specify that the side set of nodes between those 2 blocks as an interface boundary and name it so it can be referenced later in the interface kernels.

```
[Problem]
        #NOTE: For RZ coordinates, x ==> R and y ==> Z (and z ==> nothing)
         coord_type = RZ
[] #END Problem

[Mesh]
        # Master block (entire domain) with default block_id = 0
        [gen]
                type = GeneratedMeshGenerator
                dim = 2
                nx = 7
                ny = 10
                 xmin = 0.0
                xmax = 0.1015    # m radius
                ymin = 0.0
                ymax = 0.1346    # m length
         []
         #Create a bounding box from the entire domain to span the new subdomain (block = 1)
        [./subdomain1]
                 input = gen
                 type = SubdomainBoundingBoxGenerator
                 bottom_left = '0.0726 0 0'
                top_right = '0.1015 0.1346 0'
                 block_id = 1
         [../]
         #Designate a new boundary as the side sets that are shared between block 0 and block 1
        #   The new boundary is now labeled and can be used in boundary conditions or InterfaceKernels
        [./interface]
                type = SideSetsBetweenSubdomainsGenerator
                 input = subdomain1
                 master_block = '0'
```

```
            paired_block = '1'
            new_boundary = 'master0_interface'
        [../]
        #Break up the original boundaries (left right top bottom) to create separate boundaries
        #   for each subdomain new boundary names are (old_name)_to_(block_id)
        # For example, two new left side boundary names:   left_to_0 and left_to_1
        #      left_to_0 is the new left side bounary that is a part of block 0
        [./break_boundary]
             input = interface
            type = BreakBoundaryOnSubdomainGenerator
        [../]
[]

# Variables must be defined to exist on one domain or the other (or multiple domains)

[Variables]
        [./u]
            order = FIRST
            family = MONOMIAL
            block = 0                   #domain on which the variable is defined
        [../]
        [./v]
            order = FIRST
            family = MONOMIAL
            block = 1                   #domain on which the variable is defined
        [../]
[]
```

InterfaceConstReaction

Inheritance ➔ InterfaceKernel (i.e., the MOOSE base class for interface kernels)

Notes ➔ This kernel creates residuals for the master block (block 0) variable and the neighbor block (block 1) variable. It represents the physics of a simple chemical reaction at the boundary that results in a phase change. For instance, this kernel is analogous to a Henry's Law type of reaction for the transfer of mass from the gas phase to a liquid phase (or solid phase) at a specific interior boundary.

Residual Formulation

Residual (mass / area / time) =

master variable (u): $\varphi \cdot (k_u u - k_v v)$

neighbor variable (v): $-\varphi \cdot (k_u u - k_v v)$

φ = MOOSE variable test function (for FE formulation)

k_u = reaction rate coefficient for master variable

k_v = reaction rate coefficient for the neighbor variable

u = master variable on the master block (block 0)

v = neighbor variable on the neighbor block (block 1)

Note: each subdomain is not necessarily named block 0 and block 1

Additional Computations ➔ None

Usage

All interface kernels only need to be invoked once for each interface and they will automatically provide the residuals for both the master and neighbor variables.

In this example, the physics is for a reaction that occurs at the surface of a boundary named master0_interface. The master variable is u and the neighbor variable is v. The rate parameters for each variable are 1 and 2, respectively.

```
[InterfaceKernels]
    [./interface]
        type = InterfaceConstReaction
        variable = u              #variable must be the variable in the master block
        neighbor_var = v          #neighbor_var must the variable in the neighbor block
        boundary = master0_interface   #name of the interface boundary
        variable_rate = 1
        neighbor_rate = 2
    [../]
[] #END InterfaceKernels
```

InterfaceEnergyTransfer

Inheritance ➔ InterfaceKernel (i.e., the MOOSE base class for interface kernels)

Notes ➔ This kernel creates residuals for the master block (block 0) variable and the neighbor block (block 1) variable. It represents the physics of energy transfer between the two different subdomains of different energy density variables with different phase temperatures. They transfer energy through a temperature difference at the point of contact with a heat transfer coefficient and contact area fraction, similar to the DGWallEnergyFluxBC.

Residual Formulation

Residual (J / m^2 / s) =

master variable (Eu): $\varphi \cdot h \cdot f_a(T_u - T_v)$

neighbor variable (Ev): $-\varphi \cdot h \cdot f_a(T_u - T_v)$

φ = MOOSE variable test function (for FE formulation)

h = heat transfer coefficient (W/m^2/K)

f_a = contact area fraction (if not the full boundary area)

T_u = temperature for master energy density variable

T_v = temperature for the neighbor energy density variable

Eu = master energy variable on the master block (block 0)

Ev = neighbor energy variable on the neighbor block (block 1)

Note: each subdomain is not necessarily named block 0 and block 1

Additional Computations ➔ None

Usage

All interface kernels only need to be invoked once for each interface and they will automatically provide the residuals for both the master and neighbor variables.

In this example, the physics is for a transfer of energy that occurs at the surface of a boundary named master0_interface. The master variable is Eu and the neighbor variable is Ev. The temperatures variables for each are Tu and Tv, respectively. The area fraction is 1 and the heat transfer coefficient is 50.

```
[InterfaceKernels]
    [./interface]
        type = InterfaceEnergyTransfer
        variable = Eu              #variable must be the variable in the master block
        neighbor_var = Ev          #neighbor_var must the variable in the neighbor block
        boundary = master0_interface   #name of the interface boundary
        transfer_coef = 50
        area_frac = 1
        master_temp = Tu
        neighbor_temp = Tv
    [../]
[] #END InterfaceKernels
```

InterfaceMassTransfer

Inheritance ➔ InterfaceKernel (i.e., the MOOSE base class for interface kernels)

Notes ➔ This kernel creates residuals for the master block (block 0) variable and the neighbor block (block 1) variable. It represents the physics of mass transfer between the two different subdomains. They transfer energy through a concentration difference at the point of contact with a constant mass transfer rate.

Residual Formulation

Residual (mass / area / time) =

master variable (Eu): $\varphi \cdot f \cdot k(u - v)$

neighbor variable (Ev): $-\varphi \cdot k(u - v)$

φ = MOOSE variable test function (for FE formulation)

u = master concentration variable on the master block (block 0)

v = neighbor concentration variable on the neighbor block (block 1)

k = rate variable for mass-transfer (length / time)

f = contact area fraction (default = 1)

Note: each subdomain is not necessarily named block 0 and block 1

Additional Computations ➔ None

Usage

All interface kernels only need to be invoked once for each interface and they will automatically provide the residuals for both the master and neighbor variables.

In this example, the physics is for a transfer of mass that occurs at the surface of a boundary named master0_interface. The master variable is u and the neighbor variable is v. The mass transfer rate is 1. **NOTE**: 'transfer_rate' can be a variable.

```
[InterfaceKernels]
    [./interface]
        type = InterfaceMassTransfer
        variable = u              #variable must be the variable in the master block
        neighbor_var = v          #neighbor_var must the variable in the neighbor block
        boundary = master0_interface   #name of the interface boundary
        transfer_rate = 1
    [../]
[] #END InterfaceKernels
```

Initial Conditions

These are kernels that do not create residuals, but instead create initial conditions for non-linear variables in the domain. There are many, many built-in initial condition kernels in MOOSE including just setting variables to a constant or setting variables to values specified by a generic function of space (in x, y, z coordinates). Discussed here are some custom initial condition kernels specific to our problems of interest.

InitialActivity

Inheritance ➔ InitialCondition (i.e., MOOSE base kernel for initial conditions)

Notes ➔ When using activities in reactions (as opposed to concentrations), the system of equations is more difficult to solve, especially at the first time step. To get better, more efficient results, the state of the system needs to be properly initialized. Thus, the initial activities of species in a phase needs to be resolved. This initial condition kernel will establish the initial state of activities as a function of the initial concentrations (which should be known) and initial activity coefficients (which can be resolved with other kernels).

Computation

$$\text{Activity variable (unitless)} = \gamma \left(\frac{C}{C_{ref}}\right)$$

γ = activity coefficient variable for the species

(default = 1, assumes ideal state)

C = concentration variable for the species (same units as Cref)

C_{ref} = reference state concentration (same units as C)

(default = 1 M, typical for electrolytes)

Usage

Typically, the user would provide the initial condition kernel underneath the definition of the non-linear variable.

This kernel allows all the parameters (except for defined constants) that it depends on to be non-linear variables, thus the initial activity can vary in space depending on how the other parameters/variables are defined.

```
[Variables]
    [./a_H]
        order = FIRST
        family = LAGRANGE
        [./InitialCondition]
            type = InitialActivity
            concentration = C_H
            activity_coeff = g_H
```

```
                ref_conc = 1  # in M
            [../]
        [../]
[]
```

InitialButlerVolmerCurrentDensity

Inheritance ➔ InitialCondition (i.e., MOOSE base kernel for initial conditions)

Notes ➔ Solving the system of equations associated with electrochemical problems can be very tricky and may depend on our ability to form a good 'initial guess' to the starting state of the system. This includes any constraint functions we apply in the domain to resolve the kinetics of the redox reactions.

In this initial condition kernel, we initialize the starting state of the ButlerVolmerCurrentDensity variable and kernel. This requires the user to also provide initial states for potential differences between the electrode and electrolyte, as well as an initial state for the reaction rate. The computation of this initial state is exactly the same as the residual function in the ButlerVolmerCurrentDensity kernel, but without the test function.

Computation

Current Density variable (C / total volume / time) = $nA_sF(-r)$

n = number of electrons transferred in the reaction

F = Faraday's constant (default = 96,485.3 C/mol)

r = reaction rate variable (moles / area / time)

A_s = specific surface area (electrode area / total volume)

Usage

Typically, the user would provide the initial condition kernel underneath the definition of the non-linear variable.

This kernel allows all the parameters (except for defined constants) that it depends on to be non-linear variables, thus the initial current density can vary in space depending on how the other parameters are defined.

NOTE: Each initialized current density variable MUST use the same 'number_of_electrons' as defined in the 'rate_var' kernel. See InitialModifiedButlerVolmerReaction for details on how to initialize the 'rate_var'.

```
[Variables]
    [./J]
        order = FIRST
        family = LAGRANGE
        [./InitialCondition]
            type = InitialButlerVolmerCurrentDensity
```

```
                rate_var = r
                specific_area = As
                number_of_electrons = 1
            [../]
        [../]
[]
```

InitialDaviesActivityCoeff

Inheritance ➔ InitialCondition (i.e., MOOSE base kernel for initial conditions)

Notes ➔ When using activities in reactions (as opposed to concentrations), the system of equations is more difficult to solve, especially at the first time step. To get better, more efficient results, the state of the system needs to be properly initialized. Thus, the initial activities of species in a phase needs to be resolved. The InitialActivity kernels helps to resolve the initial activity values that couple to reactions, but that kernel is itself dependent on an activity coefficient (γ). Thus, the initial activity coefficients must also be properly initialized.

This kernel establishes the initial activity coefficient using the Davies model. It uses all the same arguments as the DaviesActivityCoeff auxiliary kernel, but performs the calculation at the initialization stage.

Computation

Activity coefficient (unitless) = See DaviesActivityCoeff for the full formulation.

Usage

Typically, the user would provide the initial condition kernel underneath the definition of the non-linear variable.

This kernel allows all the parameters (except for defined constants) that it depends on to be non-linear variables, thus the initial activity coefficient can vary in space depending on how the other parameters/variables are defined.

NOTE: This is generally applied in the AuxVariables block because we typically calculate activity coefficients in the auxiliary system.

```
[AuxVariables]
    [./g_H]
        order = FIRST
        family = LAGRANGE
        [./InitialCondition]
            type = InitialDaviesActivityCoeff
            temperature = 300 #in K
            ionic_strength = IonStr #in M
            ion_valence = 1
        [../]
    [../]
[]
```

InitialInhibitionProducts

Inheritance ➔ InitialCondition (i.e., MOOSE base kernel for initial conditions)

Notes ➔ This kernel is designed to automate the process by which initial conditions are set for the inhibition product variables (see InhibitionProducts kernel for more details). This is useful for improving convergence of problems that use inhibition terms for chemical reactions. In this particular case, this form of the inhibition term is made up of a product of other terms (presumably other Langmuir inhibition terms).

Computation

Inhibition variable (unitless) = $\left(\prod_{\forall i} R_i^{p_i}\right)$

R_i = i-th inhibition variable

p_i = power that the i-th inhibition term is raised to

Usage

Typically, the user would provide the initial condition kernel underneath the definition of the non-linear variable.

This kernel allows all the parameters that it depends on to be non-linear variables, thus the initial inhibition product variable can vary depending on how the other parameters are defined.

Note that the usage and parameter/variable names should always be the same as they are defined in the InhibitionProducts kernel.

```
[Variables]
    [./R]
        order = FIRST
        family = MONOMIAL
        [./InitialCondition]
            type = InitialInhibitionProducts
            coupled_list = 'R1 R2'
            power_list = '1 1'
        [../]
    [../]
[]
```

InitialIonicStrength

Inheritance ➔ InitialCondition (i.e., MOOSE base kernel for initial conditions)

Notes ➔ When using activities in reactions (as opposed to concentrations), the system of equations is more difficult to solve, especially at the first time step. To get better, more efficient results, the state of the system needs to be properly initialized. Thus, the initial activities of species in a phase needs to be resolved. The InitialActivity kernels helps to resolve the initial

activity values that couple to reactions, but that kernel is itself dependent on an activity coefficient (γ), which also depends on ionic strength.

This kernel establishes the initial ionic strength that can then be coupled with an initial activity coefficient (such as InitialDaviesCoeff). It uses all the same arguments as the IonicStrength auxiliary kernel, but performs the calculation at the initialization stage.

Computation

Ionic Strength (in M) = See IonicStrength for the full formulation.

Usage

Typically, the user would provide the initial condition kernel underneath the definition of the non-linear variable.

This kernel allows all the parameters (except for defined constants) that it depends on to be non-linear variables, thus the initial ionic strength can vary in space depending on how the other parameters/variables are defined.

NOTE: This is generally applied in the AuxVariables block because we typically calculate ionic strength in the auxiliary system.

```
[AuxVariables]
    [./IonStr]
        order = FIRST
        family = LAGRANGE
        [./InitialCondition]
            type = InitialIonicStrength
            conversion_factor = 1  # if concentrations already in M
            ion_conc = 'C_H C_Cs'
            ion_valence = '1 1'
        [../]
    [../]
[]
```

InitialLangmuirInhibition

Inheritance ➔ InitialCondition (i.e., MOOSE base kernel for initial conditions)

Notes ➔ This kernel is designed to automate the process by which initial conditions are set for the Langmuir inhibition variables (see LangmuirInhibition kernel for more details). This is useful for improving convergence of problems that use inhibition terms for chemical reactions. In this particular case, this form of the inhibition term is made up of a sum of Langmuir terms that is used to represent surface coverage for a catalytic reaction.

Computation

Inhibition variable (unitless) = $(1 + \sum_{\forall i} K_i C_i)$

C_i = concentration variable for species i

K_i = Langmuir coefficient for the i-th species in the list

Additional Computations

The Langmuir coefficients are calculated from the same type of expression as the Arrhenius reaction term.

$$K_i = A_i T^{\beta_i} exp\left[-\frac{E_i}{RT}\right]$$

- The units for A_i must have the same units as K_i
- K_i must have units of inverse concentration C_i
- The units for E_i must be J/mol
- β_i are powers on temperature (which default to 0)

Usage

Typically, the user would provide the initial condition kernel underneath the definition of the non-linear variable.

This kernel allows all the parameters that it depends on to be non-linear variables, thus the initial Langmuir Inhibition variable can vary depending on other parameters/variables.

Note that the usage and parameter/variable names should always be the same as they are defined in the LangmuirInhibition kernel.

The "temperature" argument can be a value or a variable. Same with the concentrations in the "coupled_list" argument.

```
[Variables]
    [./R1]
        order = FIRST
        family = MONOMIAL
        [./InitialCondition]
            type = InitialLangmuirInhibition
            coupled_list = 'C1 C2'
            temperature = 298
            pre_exponentials = '1 2'
            activation_energies = '-20000 10000'
            betas = '0 0'
        [../]
    [../]
[]
```

InitialModifiedButlerVolmerReaction

Inheritance ➔ InitialCondition (i.e., MOOSE base kernel for initial conditions)

Notes ➔ Solving the system of equations associated with electrochemical problems can be very tricky and may depend on our ability to form a good 'initial guess' to the starting state of the system. This includes any constraint functions we apply in the domain to resolve the kinetics of the redox reactions.

In this initial condition kernel, we initialize the starting state of the ModifiedButlerVolmerReaction variable and kernel. This requires the user to also provide initial states for potential differences between the electrode and electrolyte. The computation of this initial state is exactly the same as the residual function in the ModifiedButlerVolmerReaction kernel, but without the test function.

Computation

Reaction rate variable (moles / area / time)

$$= -b \cdot k_a C_R exp\left(\frac{[1-\alpha]nF[\Delta\phi]}{RT}\right) + b \cdot k_c C_O exp\left(\frac{-\alpha nF[\Delta\phi]}{RT}\right)$$

b = scaling factor (default = 1)

k_a = oxidation rate (can be a given constant or calculated from equilibrium potential and an overall rate constant) [*units*: length/time, but varies]

k_c = reduction rate (can be a given constant or calculated from equilibrium potential and an overall rate constant) [*units*: length/time, but varies]

F = Faraday's constant (default = 96,485.3 C/mol)

R = Gas law constant (default = 8.314462 J/K/mol)

T = temperature variable (K)

$\Delta\phi$ = variable for potential difference (V or J/C)

→ Generally calculated as $\Delta\phi = \phi_s - \phi_e$

ϕ_s = electrode potential, ϕ_e = electrolyte potential

α = electron transfer coefficient (default = 0.5 for symmetric electron transfer)

C_R, C_O = see ModifiedButlerVolmerReaction for additional details

Usage

Typically, the user would provide the initial condition kernel underneath the definition of the non-linear variable.

This kernel allows all the parameters (except for defined constants) that it depends on to be non-linear variables, thus the initial reaction rate can vary in space depending on how the other parameters are defined.

NOTE: Each initialized rate variable MUST use all the same parameters that are used in the full ModifiedButlerVolmerReaction kernel. Otherwise, this rate variable will not be properly initialized before starting a solve.

```
[Variables]
    [./r]
        order = FIRST
        family = LAGRANGE
        [./InitialCondition]
            type = InitialModifiedButlerVolmerReaction
            reaction_rate_const = 1
            equilibrium_potential = 0
            number_of_electrons = 1

            reduced_state_vars = 'R'
            reduced_state_stoich = '1'
            oxidized_state_vars = 'O'
            oxidized_state_stoich = '1'

            temperature = T
            electric_potential_difference = pot_diff
        [../]
    [../]
[]
```

InitialPotentialDifference

Inheritance ➔ InitialCondition (i.e., MOOSE base kernel for initial conditions)

Notes ➔ Solving the system of equations associated with electrochemical problems can be very tricky and may depend on our ability to form a good 'initial guess' to the starting state of the system. This includes any constraint functions we apply in the domain to resolve the kinetics of the redox reactions.

In this initial condition kernel, we initialize the starting state of the potential difference variable ($\Delta\phi$ in ModifiedButlerVolmerReaction). During the full solve, we can use the WeightedCoupledSumFunction to evaluate this term, but here we initialize it with a specific kernel after having already established initial conditions for the potentials in the electrode and electrolyte respectively.

Computation

Potential difference variable (V or J/C) = $\phi_s - \phi_e$

ϕ_s = initial potential in the solid electrode (V)

ϕ_e = initial potential in the fluid electrolyte (V)

Usage

Typically, the user would provide the initial condition kernel underneath the definition of the non-linear variable.

This kernel allows all the parameters that it depends on to be non-linear variables, thus the initial energy density can vary in space depending on how the other parameters are defined.

```
[Variables]
    [./pot_diff]
        order = FIRST
        family = LAGRANGE
        [./InitialCondition]
            type = InitialPotentialDifference
            electrode_potential = phi_s
            electrolyte_potential = phi_e
        [../]
    [../]
[]
```

InitialPhaseEnergy

Inheritance ➔ InitialCondition (i.e., MOOSE base kernel for initial conditions)

Notes ➔ Generally, we do not know what the total internal energy is for a system. Instead, we would more typically know what the initial temperatures, densities, and other properties are in the domain. Thus, this kernel can be invoked to calculate the initial internal energy of a phase in the domain as a function of those known domain properties. Whenever we are performing an energy balance based on the energy density of a phase, we are required to provide an initial condition to that energy density. This kernel facilitates those calculations.

Computation

Initial Energy Density (J/m^3) = $\rho c T$

ρ = initial density of the phase in the domain (kg/m^3)

c = initial specific heat of the phase in the domain (J/kg/K)

T = initial temperature of the phase in the domain (K)

Usage

Typically, the user would provide the initial condition kernel underneath the definition of the non-linear variable.

This kernel allows all the parameters that it depends on to be non-linear variables, thus the initial energy density can vary in space depending on how the other parameters are defined.

```
[Variables]
    [./Ef]
        order = FIRST
        family = MONOMIAL
```

```
                [./InitialCondition]
                        type = InitialPhaseEnergy
                        specific_heat = cpg
                        density = rho
                        temperature = Tf
                [../]
        [../]
[]
```

Auxiliary Kernels

The auxiliary system is used to calculate material properties needed for the simulations. We put these calculations in the auxiliary system, rather than the material properties system, because this will allow us to later move the calculation of these parameters to the kernel system if need be. For instance, if we want pressure or density to be a non-linear variable, rather than a parameter or property, it is easier to declare them as auxiliary variables now. This will allow all kernels that depend on those properties to be valid whether they are constants, auxiliary variables, or other non-linear variables. On the other hand, if we had declared those parameters as material properties, then we would have to build new kernels to use them if and when we decided to move those properties to the kernel system.

In addition to calculation of system properties, the auxiliary system can also be used to update system parameters as generic functions of space and time. This is particularly useful for simulating isothermal processes that do experience a set temperature change, like in the case of Temperature Programmed Desorption (TPD) experiments. For such a simulation, we need to include temperature as a variable into the calculations, but since the reactor system is kept isothermal, it is unnecessary to invoke a full energy balance. Instead, we can declare temperature as an auxiliary variable and write an auxiliary function to represent what the average experimental temperature is at any point in time.

AuxAvgLinearVelocity

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel can be used to calculate the average linear velocity in a packed column or through the channels of a monolith catalyst. For simulations that do not have a set velocity field, or do not use Navier-Stokes modules to produce flow fields, you can invoke this kernel for the primary flow direction and estimate the gas velocity in that direction based on the total volumetric flow rate, total cross-sectional area, and/or the porosity or bulk void space of the domain. This only estimates 1 velocity term, not a velocity vector. Thus, it would then presume that all velocity moves in the direction perpendicular to the cross-sectional area of the domain.

Computation

Average linear velocity (m/s) = $\frac{Q}{\varepsilon A}$

Q = total volumetric flow rate (m^3/s)

A = total cross-sectional area (m^2)

ε = domain average void space or porosity

Usage

All auxiliary kernels must provide an argument for option ‘execute_on’ that is used to tell the software when to compute this value. You MUST ALWAYS specify to execute on ‘initial’ and ‘timestep_end’ only!

In the below example, we are calculating the velocity in the y-direction (vel_y) as a function of a constant flow rate and a variable cross-sectional area (A) with a porosity of 1.

[./vel_calc]
type = AuxAvgLinearVelocity
variable = vel_y
execute_on = 'initial timestep_end'
flow_rate = 0.5
xsec_area = A
porosity = 1
[../]

AuxElectrodeCurrent

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel can be used to calculate the current in an electrode through the auxiliary system rather than using Reaction and ElectrodeCurrentFromPotentialGradient in the Kernel system. The advantage of calculating current in this manner is that it requires less computational work from the solver, as this computation can be performed post-solve. Since the current is dependent on solve variables, and no variables are explicitly dependent on current, this will make the system of equations smaller during the non-linear solves. Also note that this calculation is done in a piece-wise fashion, thus, the user must provide the 'direction' (e.g., 0 = x, 1 = y, and 2 = z) for the component of the current vector this acts on.

Computation

Electrode Current Density (C/area/time) = $-\left((1-\varepsilon)\sigma_s(\mathbf{n}\cdot\nabla\phi_s)\right)$

σ_s = electrode conductivity (C/V/area/time)

n = unit vector in a specific direction

ϕ_s = electric potential in the electrode (V)

(1-ε) = solids void fraction

Usage

All auxiliary kernels must provide an argument for option 'execute_on' that is used to tell the software when to compute this value. You MUST ALWAYS specify to execute on 'initial' and 'timestep_end' only!

In the below example, we are calculating the current in the y-direction (is_y), thus we provide the 'direction' argument as 1. All parameter arguments can be given as constants or variables (except for the electric_potential, which must be a variable).

[./is_y_calc]
type = AuxElectrodeCurrent
variable = is_y
direction = 1
electric_potential = phi_s
solid_frac = sol_vol

conductivity = 300
execute_on = 'initial timestep_end'
[../]

AuxElectrolyteCurrent

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel can be used to calculate the current in an electrolyte through the auxiliary system rather than using Reaction with ElectrolyteCurrentFromPotentialGradient and ElectrolyteCurrentFromIonGradient kernels in the Kernel system. The advantage of calculating current in this manner is that it requires less computational work from the solver, as this computation can be performed post-solve. Since the current is dependent on solve variables, and no variables are explicitly dependent on current, this will make the system of equations smaller during the non-linear solves. Also note that this calculation is done in a piece-wise fashion, thus, the user must provide the 'direction' (e.g., 0 = x, 1 = y, and 2 = z) for the component of the current vector this acts on.

Special Note: The effective conductivity (K_{eff}) of the electrolyte phase is calculated from the diffusivity of ions and concentration of ions in solution. However, in circumstances where you do not know all the ions that contribute to the conductivity of the electrolyte, you can also provide a 'min_conductivity' argument that will add to the calculated K_{eff} value a given constant that would essentially represent the 'background' or 'standard state' effective conductivity for the media. For example, for a water system you could use the following values as the 'background' conductivity:

min_conductivity = 3.0E-6 #C/V/cm/min (for DI water)

min_conductivity = 0.03 #C/V/cm/min (for tap water)

min_conductivity = 3.0 #C/V/cm/min (for seawater)

The units you give will vary depending on your needs.

Computation

Electrolyte Current Density (C/area/time) =

$$-\left(K_{eff}(\mathbf{n}\cdot\nabla\phi_e)\right)-\left(F\varepsilon\left[\sum_{j\in\text{ions}} z_j D_j(\mathbf{n}\cdot\nabla c_j)\right]\right)$$

where

$$K_{eff}=\varepsilon\frac{F^2}{RT}\left[\sum_{j\in\text{ions}} {z_j}^2 D_j c_j\right]$$

ε = porosity (default = 1)

F = Faraday's Constant (default = 96,485.3 C/mol)

R = Gas Constant (default = 8.314462 J/K/mol)

T = temperature of electrolyte (K)

z_j = valence of the j-th ion

D_j = Diffusivity of the j-th ion (area/time)

c_j = concentration of the j-th ion (mol/volume)

n = unit vector in a specific direction

Usage

All auxiliary kernels must provide an argument for option 'execute_on' that is used to tell the software when to compute this value. You MUST ALWAYS specify to execute on 'initial' and 'timestep_end' only!

In the below example, we are calculating the current in the y-direction (ie_y), thus we provide the 'direction' argument as 1. All parameter arguments can be given as constants or variables (except for the electric_potential and ion_conc, which must be a variable).

```
[./ie_y_calc]
        type = AuxElectrolyteCurrent
        variable = is_y
        direction = 1
        electric_potential = phi_e
        porosity = eps
        temperature = T_e
        ionc_conc = 'C_H C_Cs'
        diffusion = 'D_H D_Cs'
        ion_valence = '1 1'
        execute_on = 'initial  timestep_end'
[../]
```

AuxErgunPressure

Inheritance ➔ GasPropertiesBase

Notes ➔ This kernel uses the Ergun equation to estimate pressure drop axially in a packed bed or any straight segment of a reactor. User must provide a direction in which the pressure drop applies (0=x, 1=y, 2=z) and a pressure value or variable representing either a known pressure at the inlet or outlet of the domain. If the outlet pressure is provided, then the user must also specify some parameters defining the length of the column/domain (i.e., the 'start_point' in meters and the 'end_point' in meters).

(See GasPropertiesBase for additional notes)

Computation

Ergun Equation for Pressure Drop → $-\frac{\Delta P}{\Delta z} = 150\frac{\mu(1-\varepsilon)^2}{\varepsilon^3 d^2}\varepsilon v + 1.75\frac{(1-\varepsilon)}{\varepsilon^3 d}\rho\varepsilon^2 v^2$

If given inlet pressure…

P (Pa) = $P_{in} - \left[150\frac{\mu(1-\varepsilon)^2}{\varepsilon^3 d^2}\varepsilon v + 1.75\frac{(1-\varepsilon)}{\varepsilon^3 d}\rho\varepsilon^2 v^2\right]\cdot(z - z_s)$

If given outlet pressure…

P (Pa) = $P_{out} + \left[150\frac{\mu(1-\varepsilon)^2}{\varepsilon^3 d^2}\varepsilon v + 1.75\frac{(1-\varepsilon)}{\varepsilon^3 d}\rho\varepsilon^2 v^2\right]\cdot(z_e - z)$

z = axial position in the system (m)

z_s = starting point in the domain (m)

z_e = ending point in the domain (m)

d = particle diameter or hydraulic diameter (m)

ε = porosity in the domain (constant or variable)

v = average linear velocity (m/s)

→ Calculated from the velocity vector components

μ = gas viscosity (See GasViscosity for calculation)

ρ = gas density (See GasDensity for calculation)

P = calculated pressure at all axial positions (Pa)

Usage

Recall that there are arguments for this kernel pushed to GlobalParams (See GasPropertiesBase)

Example below calculates the pressure drop in a packed column along the y-direction given a reference pressure variable (Po) that represents the outlet pressure. The hydraulic diameter is the size of the particles in the system, which is a constant 0.01 m. The size of the column is indicated by the starting and ending y-points of 0 and 0.13 m, respectively.

```
[./press_calc]
    type = AuxErgunPressure
    variable = P
    direction = 1
    porosity = 0.5
    temperature = T
    pressure = Po
    is_inlet_press = false
    start_point = 0
    end_point = 0.13
    hydraulic_diameter = 0.01
```

uy = vel_y

[../]

AuxFirstOrderRecycleBC

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel can be used to calculate the value of an inlet condition for a concentration variable when the system is operated with a recycle line. In doing so, we effectively couple the inlet BC with the outlet BC through an approximate 'recycle rate'. To accomplish this, user's must couple this kernel with a Postprocessor that calculates the integral average concentration at the outlet of the boundary. Then, this inlet value is estimated from that outlet, a recycle rate, and the step forward in time.

NOTE: The coupling of the inlet and outlet is done approximately to first order. This is for the sake of computational efficiency.

Computation

Approximate Inlet from Recycle (mass / volume) = $\frac{u_{old}+\Delta t \cdot R \cdot u_{out}}{1+\Delta t \cdot R}$

u_{old} = value of the inlet concentration from the previous time step

u_{out} = integrated value of concentration at the outlet

Δt = size of the current time step

R = rate of recycle (per time)

Usage

Because this Auxiliary kernel is used to setup a boundary condition and does not couple with other non-linear variables directly, it should always be executed on the 'initial', 'timestep_begin', and 'nonlinear' to ensure that the value at the boundary is properly setup.

In this example, 'u_out' is an integrated Postprocessor value (which should only be executed on 'initial' and 'timestep_end'). The recycle rate can be a constant or another auxiliary variable.

```
[./inlet_calc]
    type = AuxFirstOrderRecycleBC
    variable = u_in
    execute_on = 'initial  timestep_begin nonlinear'
    recycle_rate = 1
    outlet_postprocessor = u_out
[../]
```

AuxPostprocessorValue

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel can be used to set an auxiliary variable value to that of a postprocessor. This can be useful for when another auxiliary variable may be the result of some integrated value over the domain or over the boundary on a domain. You can use this kernel to save that postprocessor value as a variable and couple it to anything needed.

Computation

Aux value (-) = p

p = value of a postprocessor

Usage

Generally, it should always be executed on the 'initial', 'timestep_begin', and 'nonlinear'.

```
[./p_calc]
    type = AuxPostprocessorValue
    variable = val
    postprocessor = p
    execute_on = 'initial  timestep_begin nonlinear'
[../]
```

DarcyWeisbachCoefficient

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel is be used to calculate the Darcy-Weisbach coefficient to be used to establish pressure gradients and flow fields in pipes and conduits without doing Navier-Stokes. The relationship is valid for laminar and turbulent flows as a long as an appropriate friction factor value is used. Generally does not account for fluid expansion if pipe changes shape and size, unless user provides a hydraulic diameter variable that maps the pipe/domain size appropriately in the mesh.

This kernel can be used to calculate the a lumped variable term that can be coupled into the VariableVectorCoupledGradient or VariableLaplacian kernels to describe incompressible flow:

$$\nabla \cdot \mathbf{v} = 0$$

For Darcy's Law (with a Darcy-Weisbach coefficient), the velocity is defined as follows:

$$\mathbf{v} = -\frac{2d_h}{f_D \rho |\mathbf{v}|} \cdot \nabla p$$

By combining this expression with the incompressible flow condition, the pressure gradient in the domain is resolved by Laplace's equation:

$$\nabla \cdot \left[-\frac{2d_h}{f_D \rho |\mathbf{v}|} \cdot \nabla p \right] = 0$$

This kernel is used to calculate the common coefficient in these 2 expressions as a function of the other properties in the domain.

Computation

Darcy-Weisbach coefficient (mass / volume / time) = $\frac{2d_h}{f_D \rho |\mathbf{v}|}$

d_h = hydraulic diameter (length)

ρ = density of the fluid (mass / length^3)

f_D = Darcy friction factor (unitless)

$|\mathbf{v}|$ = magnitude of velocity (length / time)

Usage

All auxiliary kernels must provide an argument for option 'execute_on' that is used to tell the software when to compute this value. You MUST ALWAYS specify to execute on 'initial' and 'timestep_end' only!

The velocity coupled to this kernel should be the magnitude of the velocity vector. That can be calculated from VectorMagnitude.

```
[./dw_calc]
    type = DarcyWeisbachCoefficient
    variable = dw_coeff
    velocity = vel_mag
    density = rho
    hydraulic_diameter = dh
    friction_factor = 0.05
    execute_on = 'initial  timestep_end
[../]
```

DaviesActivityCoeff

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel is be used to calculate activity coefficients for ions in an electrolyte based on the classical Davies model. It is intended to be used in conjunction with the ActivityConstraint kernel to calculate activity of a species from that species concentration and activity coefficient (calculated here). For the Davies model, each ion of the same valence will have the same activity coefficient, thus, you only need to calculate a one activity coefficient for each valence of ion in the system you are modeling, rather than needing to calculate an activity coefficient for each individual ion.

Davies Model:

$$log(\gamma_i) = -a \cdot z_i^2 \cdot \left[\frac{\sqrt{I}}{1+\sqrt{I}} - 0.3I\right]$$

$$a = A \cdot (\varepsilon T)^{-3/2}$$

Computation

Davies activity coefficient (-) = $10^{\left(-a \cdot z_i^2 \cdot \left[\frac{\sqrt{I}}{1+\sqrt{I}} - 0.3I\right]\right)}$

A = Davies fitting parameter (default = 1.82E6)

ε = dielectric constant for the media (default = 78.325)

T = temperature of the media (in K)

I = ionic strength of the solution (in M)

z_i = valence of the ion this activity coefficient applies towards

Usage

All auxiliary kernels must provide an argument for option 'execute_on' that is used to tell the software when to compute this value. You MUST ALWAYS specify to execute on 'initial' and 'timestep_end' only!

The given 'ionic_strength' and/or 'temperature' can be a constant or a variable. Users can also supply their own 'fitting_param' and 'dielectric_const' if desired, however, it is generally recommended to use the defaults.

```
[./gamma1_calc]
        type = DaviesActivityCoeff
        variable = gamma1
        ionic_strength = IonStr
        temperature = 300
        ion_valence = 1
        execute_on = 'initial  timestep_end
[../]
```

ElectrolyteConductivity

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel can be used to calculate the electrical conductivity of an electrolyte solution using the system temperature, ion concentrations, diffusion coefficients, and ion valences. Optionally, users may also override the default Faraday and Gas Law Constants if the user wishes to make a calculation in another unit basis.

NOTE: When using this kernel, you should ONLY provide ONE set of ions (either the set of negatively charged ions or positively charged ions). If you do NOT follow this convention, then the potentials in the domain will either always solve to zero (i.e., constant potential) or the solution will be infeasible. This is because the oppositely charged ions have equal and opposite effects on the potential. THUS, you are only ever calculating a REFERENCE potential in solution, where that reference is based on either the positive ions or the negative ions.

NOTE 2: For very dilute systems, it is HIGHLY recommended that you provide a background ion reference concentration. Otherwise, the calculation of effective conductivity will result in values too low to get a reasonable reference potential. (See **Special Note** below)

Special Note: The effective conductivity (K_{eff}) of the electrolyte phase is calculated from the diffusivity of ions and concentration of ions in solution. However, in circumstances where you do not know all the ions that contribute to the conductivity of the electrolyte, you can also provide a 'min_conductivity' argument that will add to the calculated K_{eff} value a given constant that would essentially represent the 'background' or 'standard state' effective conductivity for the media. For example, for a water system you could use the following values as the 'background' conductivity:

min_conductivity = 3.0E-6 #C/V/cm/min (for DI water)

min_conductivity = 0.03 #C/V/cm/min (for tap water)

min_conductivity = 3.0 #C/V/cm/min (for seawater)

The units you give will vary depending on your needs.

Computation

Electrolyte conductivity (C/V/length/time) = $\varepsilon \frac{F^2}{RT}\left[\sum_{\substack{\forall\, same \\ charge\ ions}} {z_j}^2 D_j c_j\right]$

ε = porosity (default = 1)

F = Faraday's Constant (default = 96,485.3 C/mol)

R = Gas Constant (default = 8.314462 J/K/mol)

T = temperature of electrolyte (K)

z_j = valence of the j-th ion

D_j = Diffusivity of the j-th ion (area/time)

c_j = concentration of the j-th ion (mol/volume)

Usage

All auxiliary kernels must provide an argument for option 'execute_on' that is used to tell the software when to compute this value. You MUST ALWAYS specify to execute on 'initial' and 'timestep_end' only!

In the below example, we are calculating the conductivity from all the positive ions in the electrolyte while supplying a background conductivity through 'min_conductivity'.

```
[./sigma_e_calc]
    type = ElectrolyteConductivity
    variable = sigma_e
    execute_on = 'initial  timestep_end'
    porosity = 0.5
```

```
        temperature = 298
        ion_conc = 'pos_ion'
        diffusion = 'Dp'
        valence = '2'
        min_conductivity = 0.03 # typical for seawater
[../]
```

ErgunCoefficient

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel is be used to calculate the Ergun coefficient to be used to establish pressure gradients and flow fields in pipes and conduits without doing Navier-Stokes. The relationship is valid for any packed-bed system.

This kernel can be used to calculate the a lumped variable term that can be coupled into the VariableVectorCoupledGradient or VariableLaplacian kernels to describe incompressible flow:

$$\nabla \cdot \mathbf{v} = 0$$

For Darcy's Law (with a Ergun coefficient), the velocity is defined as follows:

$$\mathbf{v} = -\left(\frac{150\mu(1-\varepsilon)^2}{d_p^2\varepsilon^3} + \frac{1.75\rho(1-\varepsilon)|\mathbf{v}|}{d_p\varepsilon^3}\right)^{-1} \cdot \nabla p$$

By combining this expression with the incompressible flow condition, the pressure gradient in the domain is resolved by Laplace's equation:

$$\nabla \cdot \left[-\left(\frac{150\mu(1-\varepsilon)^2}{d_p^2\varepsilon^3} + \frac{1.75\rho(1-\varepsilon)|\mathbf{v}|}{d_p\varepsilon^3}\right)^{-1} \cdot \nabla p\right] = 0$$

This kernel is used to calculate the common coefficient in these 2 expressions as a function of the other properties in the domain.

Computation

Ergun coefficient (mass / volume / time) = $\left(\frac{150\mu(1-\varepsilon)^2}{d_p^2\varepsilon^3} + \frac{1.75\rho(1-\varepsilon)|\mathbf{v}|}{d_p\varepsilon^3}\right)^{-1}$

d_p = particle diameter (length)

μ = viscosity of the fluid (mass / length / time)

ρ = density of the fluid (mass / length^3)

ε = porosity of the domain (-)

$|\mathbf{v}|$ = magnitude of velocity (length / time)

Usage

All auxiliary kernels must provide an argument for option 'execute_on' that is used to tell the software when to compute this value. You MUST ALWAYS specify to execute on 'initial' and 'timestep_end' only!

The velocity coupled to this kernel should be the magnitude of the velocity vector. That can be calculated from VectorMagnitude.

```
[./ergun_calc]
        type = ErgunCoefficient
        variable = ergun_coeff
        velocity = vel_mag
        density = rho
        viscosity = mu
        particle_diameter = dp
        porosity = eps
        execute_on = 'initial  timestep_end
[../]
```

GasDensity

Inheritance ➔ GasPropertiesBase

Notes ➔ The density of the gas phase is computed from the known concentrations of species in the gas phase. In this kernel, it is NOT a function of ideal gas law, since the domain is assumed to be confined. If the user desires to have the gas density a function of ideal gas law, then a new kernel will need to be added. As an optional argument, the user may provide a variable or value representing the concentration of a carrier gas and it's respective molecular weight.

(See GasPropertiesBase for additional notes)

Computation

Gas density (kg/m^3) = $\sum_{\forall i} C_i \frac{MW_i}{1000} + C_{carrier} \frac{MW_{carrier}}{1000}$

C_i = molar concentration of the ith gas species (mol/m^3)

MW_i = molar weight of the ith gas species (g/mol)

Usage

Recall that there are arguments for this kernel pushed to GlobalParams (See GasPropertiesBase)

Example below calculates the density in the domain given an N2 carrier gas with a molecular weight of 28 g/mol.

NOTE: If N2 is from the original list of gas species, then DO NOT provide here. This will cause errors in the calculation of density.

```
[./dens_calc]
```

```
        type = GasDensity
        variable = rho
        carrier_gas = N2
        carrier_gas_mw = 28
        temperature = T                 #Note: not used, but a required argument
        pressure = P                    #Note: not used, but a required argument
        hydraulic_diameter = d          #Note: not used, but a required argument
        uy = vel_y                      #Note: not used, but a required argument
[../]
```

GasEffectiveThermalConductivity

Inheritance ➔ GasPropertiesBase

Notes ➔ This kernel is used to calculate an effective thermal conductivity that is to be a volume averaged conductivity between the conductivity of 2 different phases in a domain. For instance, in the case of packed beds, the energy density and temperatures are usually solved in a single combined term which assumes local isothermal equilibria between the solid and fluid. Thus, it is necessary to create an effective conductivity parameter that averages the fluid and solid conductivities based on the domain porosity.

(See GasPropertiesBase for additional notes)

Computation

Effective Conductivity (W/m/K) = $(1-\varepsilon)\cdot K_s + \varepsilon \cdot K_g$

ε = porosity in the domain (constant or variable)

K_s = solid phase thermal conductivity (W/m/K)

K_g = gas phase thermal conductivity (See GasThermalConductivity for details)

Usage

Recall that there are arguments for this kernel pushed to GlobalParams (See GasPropertiesBase)

Example below calculates the effective thermal conductivity of a packed column. The porosity in the system is 0.5 and the average linear velocity is calculated from the given velocity vectors. The 'heat_cap_ratio' is an optional parameter that represents the ratio of the constant pressure specific heat to the constant volume specific heat (i.e., c_p/c_v). Typical values for this ratio is about 1.4. It can range between 0.56 and 1.67. Some arguments are required by the base class object GasPropertiesBase, but are not actually used in the calculations.

```
[./Ke_calc]
        type = GasEffectiveThermalConductivity
        variable = Ke
        temperature = T
        pressure = P
        hydraulic_diameter = d          #Note: not used, but a required argument
```

```
        uy = vel_y                      #Note: not used, but a required argument
        heat_cap_ratio = 1.4
        solid_conductivity = Ks
        porosity = 0.5
[../]
```

GasPropertiesBase

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This is the base object that defines how all 'Gas' auxiliary values are calculated. It sets up the variables needed and provides a way to interface with the Egret utilities subroutines. Because of this, all the 'Gas' auxiliary kernels require a common set of input parameters, although some do require additional arguments beyond the base set of arguments. Since all those input arguments are common to all other 'Gas' auxiliary kernels, you should put those arguments in the 'GlobalParams' block of you input file. See the example below…

```
[GlobalParams]
        # Always have the arguments below in GlobalParams
        gases = 'N2  O2  CO2'
        molar_weights = '28  32  44'
        sutherland_temp = '300.55   292.25   293.15'
        sutherland_const = '111   127   240'
        sutherland_vis = '0.0001781   0.0002018   0.000148'
        spec_heat = '1.04   0.919   0.846'
        execute_on = 'initial  timestep_end'

        # Arguments in the below list can also go in GlobalParams
        #       or defined locally in other auxiliary kernels if desired
        #       NOTE: it is usually recommended to provide these locally
        temperature = T
        ux = vel_x
        uy = vel_y
        uz = vel_z

        # Arguments below are optional arguments to refine how gas
        #       properties are calculated (also used in GasDensity calculation)
        carrier_gas = Air
        carrier_gas_mw = 28.8
        is_ideal_gas = false
[]
```

gases = list of gas concentration variables (mol/m^3)

carrier_gas = gas concentration variable/constant for carrier gas (mol/m^3)

➔ **NOTE**: DO NOT put a gas species concentration here that already exists in the list of 'gases'. If all gases are accounted for in 'gases', then you can leave this optional parameter out or set to 0.

molar_weights = list of molecular weights of corresponding gas species (g/mol)

carrier_gas_mw = molecular weight of carrier gas (g/mol)

sutherland_temp = list of Sutherland's temperatures for each gas species (K)

sutherland_const = list of Sutherland's constants for each gas species (K)

sutherland_vis = list of Sutherland's viscosities for each gas species (g/cm/s)

spec_heat = list of specific heat capacities at constant pressure for each gas (J/g/k)

execute_on = when to calculate the properties (ALWAYS 'initial timestep_end')

temperature = temperature variable for the gas (K)

ux, uy, uz = velocity component variables in each cardinal direction (m/s)

is_ideal_gas = Boolean (true/false) used to determine whether or not kinetic theory of gases will uses the Ideal Gas Law to estimate gas densities or just use the given gas concentrations. If true, then ideal gas law is assumed for gas properties.

Computation ➔ None, this is a place holder object to prepare calculations.

Usage ➔ None, this object should not be invoked in the input files.

GasSolidHeatTransferCoef

Inheritance ➔ GasPropertiesBase

Notes ➔ This kernel calculates the gas-to-solid heat transfer coefficient (h_s) in W/m^2/K. This is a parameter that the user can provide as a constant for PhaseEnergyTransfer or can use this kernel to approximate that parameter from the Prandtl and Reynolds numbers, as well as an approximated gas thermal conductivity. The calculation is not a fundamental calculation, but instead is an empirical estimation based a relationship between the Prandtl and Reynolds numbers, as well as the thermal conductivity of the solids.

(See GasPropertiesBase for additional notes)

Computation

Prandtl Number (Pr) = $\frac{\mu c_p}{K_g}$

Reynolds Number (Re) = $\frac{\varepsilon v d \rho}{\mu}$

Gas-solid Heat Transfer (W/m^2/K) = $\frac{K_s}{500 \cdot d}[2 + 1.1 \cdot Re^{0.6} \cdot Pr^{0.3}]$

ε = porosity in the domain (constant or variable)

v = average linear velocity (m/s)

→ Calculated from the velocity vector components

μ = gas viscosity (See GasViscosity for calculation)

ρ = gas density (See GasDensity for calculation)

d = particle diameter or hydraulic diameter (m)

c_p = specific heat of the gas at constant pressure (J/kg/m^3)

K_s = solids thermal conductivity (W/m/K)

K_g = gas thermal conductivity (See GasThermalConductivity for calculation)

Usage

Recall that there are arguments for this kernel pushed to GlobalParams (See GasPropertiesBase)

Example below calculates the heat transfer coefficient for solids with diameter of 0.01 m and a variable conductivity (Ks). The porosity in the system is 0.5 and the average linear velocity is calculated from the given velocity vectors. The 'heat_cap_ratio' is an optional parameter that represents the ratio of the constant pressure specific heat to the constant volume specific heat (i.e., c_p/c_v). Typical values for this ratio is about 1.4. It can range between 0.56 and 1.67.

```
[./hs_calc]
    type = GasSolidHeatTransferCoef
    variable = hs
    temperature = T
    pressure = P
    hydraulic_diameter = 0.01
    ux = vel_x
    uy = vel_y
    uz = vel_z
    heat_cap_ratio = 1.4
    solid_conductivity = Ks
    porosity = 0.5
[../]
```

GasSpecHeat

Inheritance ➔ GasPropertiesBase

Notes ➔ This kernel calculates the specific heat of the gas (at constant pressure: c_p) in J/kg/K. That value is computed as a weighted average of the specific heats of each gas species given.

Recall that GasPropertiesBase requires you give the list of gas species and their respective standard state specific heats.

(See GasPropertiesBase for additional notes)

Computation

Specific heat at constant pressure (J/kg/K) = $\sum_{\forall i} y_i c_{p,i}$

y_i = mole fraction of gas species i

→ mole fractions are calculated from pressure, temperature, and mass concentrations for each species

$c_{p,i}$ = specific heat of gas species i

Usage

Recall that there are arguments for this kernel pushed to GlobalParams (See GasPropertiesBase)

Example below calculates the specific heat of the gas in the domain given variables for temperature and pressure in that domain. Because this kernel inherits from GasPropertiesBase, there are some arguments required that are not actually used. That is why it is recommended to push some of those arguments to GlobalParams, so you reduce redundancy in the input files.

```
[./cp_calc]
        type = GasSpecHeat
        variable = cp
        temperature = T
        pressure = P
        hydraulic_diameter = d          #Note: not used, but a required argument
        uy = vel_y                      #Note: not used, but a required argument
[../]
```

GasSpeciesAxialDispersion

Inheritance ➔ GasPropertiesBase

Notes ➔ This kernel calculates an effective dispersion coefficient (m^2/s) for a gas species as a function of temperature, pressure, and concentrations of all gas species of interest. That dispersion variable can then be used in GVariableDiffusion (DGVariableDiffusion) or GVarPoreDiffusion (DGVarPoreDiffusion) to update the diffusive flux based on dispersion in a packed column. It is a function of the true molecular diffusivity of the gas species, as well as a correction for mechanical mixing that occurs in the tortuous path the molecules take in the domain. Those corrections are based on empirical relationships with the Reynolds and Schmidt numbers, as well as the overall diameter or hydraulic diameter of the domain.

(See GasPropertiesBase for additional notes)

Computation

Schmidt Number (Sc) = $\frac{\mu}{\rho D_{m,i}}$

Reynolds Number (Re) = $\frac{vD\rho}{\mu}$

Effective Dispersion of Species (m^2/s) = $v \cdot d \cdot \left(0.5 + \frac{20}{Re \cdot Sc}\right)$

μ = gas viscosity (See GasViscosity for calculation)

ρ = gas density (See GasDensity for calculation)

d = particle diameter or hydraulic diameter (m)

D = bed diameter domain width variable (m)

$D_{m,i}$ = molecular diffusivity of species i (See GasSpeciesDiffusion for calculation)

v = average linear velocity (m/s)

→ Calculated from the velocity vector components

i = gas species index

→ This value must be provided by user and should correspond to the position of the gas species of interest in the 'gases' list from GasPropertiesBase (indexing starts from 0)

Usage

Recall that there are arguments for this kernel pushed to GlobalParams (See GasPropertiesBase)

Example below calculates the effective diffusivity of O_2 gas in the domain given variables for temperature and pressure in that domain. The column diameter is a variable named Dia and the particle diameter is another parameter named d.

```
[./De_calc]
    type = GasSpeciesAxialDispersion
    variable = De
    gases = 'N2  O2  CO2'
    species_index = 1                #1 – Corresponds to O2 in the 'gases' list
    macroscale_diameter = Dia
    temperature = T
    pressure = P
    hydraulic_diameter = d
    uy = vel_y
[../]
```

GasSpeciesDiffusion

Inheritance → GasPropertiesBase

Notes → This kernel calculates the gas phase molecular diffusivity of a given gas species based on kinetic theory of gases, the Sutherland's model of gas viscosity, and Ideal Gas Law. User

needs to provide the temperature, pressure, and index of the gas species of interest from the 'gases' list in GasPropertiesBase.

(See GasPropertiesBase for additional notes and parameter requirements)

Computation

Molecular Diffusion of Species (m^2/s) = $(1 - y_i) \Big/ \left(\sum_{j \neq i} \frac{y_j}{D_{ij}}\right)$

$$D_{ij} = \frac{(4/\sqrt{2})\sqrt{\left(MW_i^{-1} + MW_j^{-1}\right)}}{\left[\left(\rho_i^2 \Big/ \{1.92\mu_i^2 MW_i\}\right)^{0.25} + \left(\rho_j^2 \Big/ \{1.92\mu_j^2 MW_j\}\right)^{0.25}\right]^2}$$

MW_i = molecular weights of ith species in gas (g/mol)

μ_i = partial viscosity of the ith species (See GasViscosity for calculation)

y_i = mole fraction of gas species i

→ mole fractions are calculated from pressure, temperature, and mass concentrations for each species

P = gas pressure variable (Pa)

T = gas temperature variable (K)

R = gas law constant (8.3144621 L*kPa/K/mol)

Usage

Recall that there are arguments for this kernel pushed to GlobalParams (See GasPropertiesBase)

Example below calculates the molecular diffusivity of O_2 gas in the domain given variables for temperature and pressure in that domain.

```
[./D_calc]
        type = GasSpeciesDiffusion
        variable = D
        gases = 'N2  O2  CO2'
        species_index = 1              #1 – Corresponds to O2 in the 'gases' list
        temperature = T
        pressure = P
        hydraulic_diameter = d         #Note: not used, but a required argument
        uy = vel_y                     #Note: not used, but a required argument
[../]
```

GasSpeciesEffectiveTransferCoef

Inheritance ➔ GasPropertiesBase

Notes ➔ This kernel calculates the effective mass transfer rate for a gas species. The effective rate of mass transfer is calculated in a very similar manner to GasSpeciesMassTransCoef, however, it uses corrections for the impact of pore-diffusion. You would typically use this kernel over GasSpeciesMassTransCoef when pore-diffusion is an important factor in the transfer of mass from gas to solid spaces, but do not want to have to simulate the diffusion inside the pore space explicitly with a hybrid FD/FE subdomain approach as described in MicroscaleDiffusion (and related) kernels.

(See GasPropertiesBase for additional notes and parameter requirements)

Computation

Schmidt Number (Sc) = $\frac{\mu}{\rho D_{p,i}}$

Reynolds Number (Re) = $\frac{vd\rho}{\mu}$

Effective Mass Transfer (m/s) = $\frac{D_{p,i}}{d}[2 + 1.1 \cdot Re^{0.6} \cdot Sc^{0.3}]$

ε_p = porosity in the particle (constant or variable) [Required for Pore Diffusion]

$D_{p,i}$ = pore-space diffusivity (See GasSpeciesPoreDiffusion for calculation)

v = average linear velocity (m/s)

→ Calculated from the velocity vector components

μ = gas viscosity (See GasViscosity for calculation)

ρ = gas density (See GasDensity for calculation)

d = particle diameter or hydraulic diameter (m)

Usage

Recall that there are arguments for this kernel pushed to GlobalParams (See GasPropertiesBase)

Example below calculates the effective mass transfer of O_2 gas in the domain given variables for temperature and pressure in that domain. The particle porosity has a value of 0.2 and the particle as a diameter of d.

```
[./ke_calc]
        type = GasSpeciesEffectiveTransferCoef
        variable = ke
        gases = 'N2  O2  CO2'
        species_index = 1                #1 – Corresponds to O2 in the 'gases' list
        micro_porosity = 0.2
        temperature = T
        pressure = P
        hydraulic_diameter = d
        uy = vel_y
[../]
```

GasSpeciesKnudsenDiffusionCorrection

Inheritance ➔ GasPropertiesBase

Notes ➔ This kernel calculates and effective pore-diffusivity inside of micro-porous region of a particle using a correction for the effects of Knudsen diffusion. User needs to provide the gas species index for the species of interest as well as providing the micro-scale porosity and nominal/average micro-pore radius.

(See GasPropertiesBase for additional notes and parameter requirements)

Computation

Effective Pore Diffusivity of Species (m^2/s) = $\left[D_{p,i}^{-1} + D_{k,i}^{-1}\right]^{-1}$

$$D_{k,i} = 9700 \cdot r_p \left(\frac{T}{MW_i}\right)^{0.5}$$

ε_p = porosity in the particle (constant or variable) [Required for Pore Diffusion]

$D_{p,i}$ = pore-space diffusivity (See GasSpeciesPoreDiffusion for calculation)

MW_i = molecular weights of ith species in gas (g/mol)

r_p = micro-pore radius (m)

T = gas temperature variable (K)

Usage

Recall that there are arguments for this kernel pushed to GlobalParams (See GasPropertiesBase)

Example below calculates the effective pore-diffusion of O_2 gas in the domain given variables for temperature and pressure in that domain. The particle porosity has a value of 0.2 and the particle as a diameter of d. The nominal pore radius is 1E-7 m.

```
[./Deff_calc]
    type = GasSpeciesKnudsenDiffusionCorrection
    variable = Deff
    gases = 'N2  O2  CO2'
    species_index = 1              #1 – Corresponds to O2 in the 'gases' list
    micro_porosity = 0.2
    micro_pore_radius = 1E-7
    temperature = T
    pressure = P
    hydraulic_diameter = d         #Note: not used, but a required argument
[../]
```

GasSpeciesMassTransCoef

Inheritance ➔ GasPropertiesBase

Notes ➔ This kernel calculates the film mass transfer coefficient for a gas species. The film mass transfer coefficient can be used in conjunction with FilmMassTransfer (or similar) kernels to facilitate the simulation of mass transfer from the bulk gas to the gases in the pore-space of particles. It would most appropriately be used with the MicroscaleDiffusion type of kernels that also simulate pore-diffusion in porous particles or materials.

(See GasPropertiesBase for additional notes and parameter requirements)

Computation

Schmidt Number (Sc) = $\frac{\mu}{\rho D_{m,i}}$

Reynolds Number (Re) = $\frac{vd\rho}{\mu}$

Mass Transfer Rate (m/s) = $\frac{D_{m,i}}{d}[2 + 1.1 \cdot Re^{0.6} \cdot Sc^{0.3}]$

$D_{m,i}$ = molecular diffusivity (See GasSpeciesDiffusion for calculation)

v = average linear velocity (m/s)

→ Calculated from the velocity vector components

μ = gas viscosity (See GasViscosity for calculation)

ρ = gas density (See GasDensity for calculation)

d = particle diameter or hydraulic diameter (m)

Usage

Recall that there are arguments for this kernel pushed to GlobalParams (See GasPropertiesBase)

Example below calculates the mass transfer of O_2 gas in the domain given variables for temperature and pressure in that domain. The particle porosity has a diameter of d.

```
[./k_calc]
    type = GasSpeciesMassTransCoef
    variable = k
    gases = 'N2  O2  CO2'
    species_index = 1                #1 – Corresponds to O2 in the 'gases' list
    temperature = T
    pressure = P
    hydraulic_diameter = d
    uy = vel_y
[../]
```

GasSpeciesPoreDiffusion

Inheritance ➔ GasPropertiesBase

Notes ➔ This kernel estimates the pore-diffusion coefficient from the molecular diffusivity of a species and the micro-porosity of the particles. Pore diffusion is a function of the tortuosity of the path the molecules take through the pore spaces, but tortuosity is a measure that is difficult to quantify. Here, the tortuosity is approximated as $1/\varepsilon_p$, where ε_p is the particle porosity.

(See GasPropertiesBase for additional notes and parameter requirements)

Computation

Pore Diffusivity (m^2/s) = $\frac{\varepsilon_p D_{m,i}}{\tau}$

$D_{m,i}$ = molecular diffusivity (See GasSpeciesDiffusion for calculation)

ε_p = porosity in the particle (constant or variable)

τ = tortuosity for the particles ~ $1/\varepsilon_p$

Usage

Recall that there are arguments for this kernel pushed to GlobalParams (See GasPropertiesBase)

Example below calculates the effective pore-diffusion of O_2 gas in the domain given variables for temperature and pressure in that domain. The particle porosity has a value of 0.2.

```
[./Dp_calc]
        type = GasSpeciesPoreDiffusion
        variable = Dp
        gases = 'N2  O2  CO2'
        species_index = 1               #1 – Corresponds to O2 in the 'gases' list
        micro_porosity = 0.2
        temperature = T
        pressure = P
        hydraulic_diameter = d          #Note: not used, but a required argument
[../]
```

GasThermalConductivity

Inheritance ➔ GasPropertiesBase

Notes ➔ This kernel approximates the thermal conductivity of the gas phase based on the specific heat at constant pressure, the ratio of the specific heats at constant pressure and temperature (i.e., c_p/c_v), and the viscosity of the gas. The specific heat ratio is an optional parameter that defaults to 1.4. Typical range is between 0.56 and 1.67.

(See GasPropertiesBase for additional notes and parameter requirements)

Computation

Gas Thermal Conductivity (W/m/K) = $0.25 \cdot \left(9\left[{}^{c_p}/_{c_v}\right] - 5\right) \cdot \mu \cdot c_v$

c_p = specific heat of the gas at constant pressure (J/kg/m^3)

→ See GasSpecHeat for calculation

c_v = specific heat of the gas at constant volume (J/kg/m^3)

μ = gas viscosity (See GasViscosity for calculation)

Usage

Recall that there are arguments for this kernel pushed to GlobalParams (See GasPropertiesBase)

Example below calculates the thermal conductivity of the gas phase. The 'heat_cap_ratio' is an optional parameter that represents the ratio of the constant pressure specific heat to the constant volume specific heat (i.e., c_p/c_v). Typical values for this ratio is about 1.4. It can range between 0.56 and 1.67.

```
[./Kg_calc]
        type = GasThermalConductivity
        variable = Kg
        temperature = T
        pressure = P
        hydraulic_diameter = d          #Note: not used, but a required argument
        uy = vel_y                      #Note: not used, but a required argument
        heat_cap_ratio = 1.4
[../]
```

GasVelocityCylindricalReactor

Inheritance → AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes → This kernel can be used to calculate the average linear velocity in a packed column or through the channels of a monolith catalyst. For simulations that do not have a set velocity field, or do not use Navier-Stokes modules to produce flow fields, you can invoke this kernel for the primary flow direction and estimate the gas velocity in that direction based on the space-velocity at a given reference state, dimensions of the reactor, and the porosity or bulk void space of the domain. This only estimates 1 velocity term, not a velocity vector. Thus, it would then presume that all velocity moves in the direction perpendicular to the cross-sectional area of the domain. This computation differs from AuxAvgLinearVelocity in that it will account for variations in average velocity with changes in inlet temperatures and pressures using an ideal gas assumption.

NOTE: Optionally, user's may pass a 'by_total_total_reactor_volume' argument to change the behavior of this calculation depending on whether or not the 'space-velocity' is a function of reactor volume or catalyst volume. By default, kernel assumes volume is total volume.

Computation

Average linear velocity (length/time) = $\frac{Q_{true}}{\varepsilon A}$

Q_{true} = true total volumetric flow rate (volume/time)

= Q_{ref} * (P_{ref}/P_{in}) * (T_{in}/T_{ref})

$P_{ref/in}$ = pressure terms (kPa)

$T_{ref/in}$ = temperature terms (K)

Q_{ref} = reference state volumetric flow rate (volume/time)

= (SV)*V

SV = space-velocity at the reference state (reactor volumes / time)

V = total reactor volume (volume) or solids/catalyst volume

= A*L (or) = A*L*(1-ε)

L = length of the reactor (length)

A = total cross-sectional area (area)

=πr^2

r = radius of cylindrical reactor (length)

ε = domain average void space or porosity

Usage

All auxiliary kernels must provide an argument for option 'execute_on' that is used to tell the software when to compute this value. You MUST ALWAYS specify to execute on 'initial' and 'timestep_end' only!

In the below example, we are calculating the velocity in the y-direction (vel_y) as a function of a the given space-velocity, variables for inlet pressure and temperature (press and temp), and the given reactor dimension information (porosity, length, and radius)

```
[./vel_calc]
    type = GasVelocityCylindricalReactor
    variable = vel_y
    execute_on = 'initial  timestep_end'
    space_velocity = 500  #volumes / min
    ref_temperature = 298
    inlet_temperature = temp
    ref_pressure = 100
    inlet_pressure = press
    porosity = 0.33
    radius = 1
```

length = 5

[../]

GasViscosity

Inheritance ➔ GasPropertiesBase

Notes ➔ This kernel calculates the gas viscosity based on kinetic theory of gases and the Sutherland's model for the viscosities of pure gas species.

(See GasPropertiesBase for additional notes and parameter requirements)

Computation

Gas Viscosity (kg/m/s) = $\sum_{\forall i}\left\{\frac{\mu_i}{\left(1+\frac{113.65\cdot\chi\mu_i T}{y_i MW_i}\left[\sum_{j\neq i} {y_i}/{D'_{ij}}\right]\right)}\right\}$

$$\chi = 0.873143 + (7.23875\cdot 10^{-5})T$$

$$P_o D'_{ij} = PD_{ij} \qquad \mu_i = \mu_i^o \frac{T_i^o + C_i}{T + C_i}\left(\frac{T}{T_i^o}\right)^{1.5}$$

D_{ij} = binary diffusion parameter (See GasSpeciesDiffusion for calculation)

P_o = standard state pressure (100,000 Pa)

T = gas temperature variable (K)

P = gas pressure variable (Pa)

MW_i = molecular weights of ith species in gas (g/mol)

y_i = mole fraction of gas species i

→ Calculated from the velocity vector components

μ_i^o = Sutherland's reference viscosity for species i

→ See GasPropertiesBase for notes on input values

T_i^o = Sutherland's reference temperature for species i

→ See GasPropertiesBase for notes on input values

C_i = Sutherland's constant for species i

→ See GasPropertiesBase for notes on input values

Usage

Recall that there are arguments for this kernel pushed to GlobalParams (See GasPropertiesBase)

Example below calculates the viscosity in the domain given variables for temperature and pressure in that domain. Because this kernel inherits from GasPropertiesBase, there are some arguments required that are not actually used. That is why it is recommended to push some of those arguments to GlobalParams, so you reduce redundancy in the input files.

```
[./mu_calc]
        type = GasViscosity
        variable = mu
        temperature = T
        pressure = P
        hydraulic_diameter = d          #Note: not used, but a required argument
        uy = vel_y                      #Note: not used, but a required argument
[../]
```

GasVolSpecHeat

Inheritance ➔ GasPropertiesBase

Notes ➔ This kernel estimates the specific heat at constant volume for the gas. The calculation uses a given ratio of specific heat at constant pressure and constant volume (i.e., c_p/c_v) to return the value of c_v based on the calculated c_p value from GasSpecHeat.

(See GasPropertiesBase for additional notes and parameter requirements)

Computation

Specific Heat at Constant Volume (J/kg/K) = $\frac{c_p}{f}$

c_p = specific heat at constant pressure (See GasSpecHeat for calculation)

f = ratio of c_p/c_v given as a constant between 0.56 and 1.67 (Default = 1.4)

Usage

Recall that there are arguments for this kernel pushed to GlobalParams (See GasPropertiesBase)

Example below calculates the specific heat at constant volume for the gas. The 'heat_cap_ratio' is an optional parameter that represents the ratio of the constant pressure specific heat to the constant volume specific heat (i.e., c_p/c_v). Typical values for this ratio is about 1.4. It can range between 0.56 and 1.67.

```
[./cv_calc]
        type = GasVolSpecHeat
        variable = cv
        temperature = T
        pressure = P
        hydraulic_diameter = d          #Note: not used, but a required argument
        uy = vel_y                      #Note: not used, but a required argument
        heat_cap_ratio = 1.4
 [../]
```

IonicStrength

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel is be used to calculate the ionic strength (in M) of the electrolyte solution. We require users to calculate ionic strength in units of M because this is the unit basis that all activity coefficient models (such as DaviesActivityCoeff) will use. It should be trivial to provide an appropriate conversion factor for you system as all the concentrations in the system will all have the same units (moles per some volume), thus, all you are providing is a unit conversion factor for the units of volume.

For example, if concentrations in the system are in mol/cm^3 (and M == mol/L), then the conversion factor would be 1000 cm^3/L.

Computation

Ionic Strength (M) = $0.5 \cdot f \cdot \sum_{ions} c_i z_i^2$

f = conversion factor to go from (moles / volume) to (M)

= default is 1 (i.e., no conversion)

[User's need to know the conversion factor to apply to their model]

c_i = concentration of the i^{th} ion (moles / volume)

z_i = valence of the i^{th} ion

Usage

All auxiliary kernels must provide an argument for option 'execute_on' that is used to tell the software when to compute this value. You MUST ALWAYS specify to execute on 'initial' and 'timestep_end' only!

In this sample, the concentrations all have units of mol/cm^3, thus we provide a conversion factor (conversion_factor) of 1000 cm^3/L.

```
[./IonStr_calc]
        type = IonicStrength
        variable = IonStr
        conversion_factor = 1000 # cm^3/L
        ion_conc = 'C_H C_Cs'
        ion_valence = '1 1'
        execute_on = 'initial  timestep_end
[../]
```

KozenyCarmanDarcyCoefficient

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel can be used to calculate the a lumped variable term that can be coupled into the VariableVectorCoupledGradient or VariableLaplacian kernels to describe Darcy flow. In a porous media, incompressible flow adheres to the following:

$$\nabla \cdot \mathbf{v} = 0$$

For Darcy's Law (with a Kozeny-Carman coefficient), the velocity is defined as follows:

$$\mathbf{v} = -\frac{d_p^2}{K\mu}\frac{\varepsilon^3}{(1-\varepsilon)^3} \cdot \nabla p$$

By combining this expression with the incompressible flow condition, the pressure gradient in the domain is resolved by Laplace's equation:

$$\nabla \cdot \left[-\frac{d_p^2}{K\mu}\frac{\varepsilon^3}{(1-\varepsilon)^3} \cdot \nabla p\right] = 0$$

This kernel is used to calculate the common coefficient in these 2 expressions as a function of the other properties in the domain.

Computation

Lumped Kozeny-Carman Coefficient (length^2/pressure/time) = $\frac{d_p^2}{K\mu}\frac{\varepsilon^3}{(1-\varepsilon)^3}$

d_p = average diameter of particles in the porous domain (length)

μ = viscosity of the fluid (pressure * time)

ε = porosity of the domain

K = Kozeny-Carman constant [default = 5.55]

Usage

In the below example, we are calculating the coefficient for Darcy flow giving variables for porosity, viscosity, and a value for typical particle size. Note that we need to make sure our units in the end make sense. If our pressure variable is in kPa and our domain dimensions are in cm, then the particle diameter must be in cm and the viscosity must use the 'pressure' unit basis to be in kPa*time. See SimpleFluidViscosity for information on viscosity units.

```
[./darcy_calc]
    type = KozenyCarmanDarcyCoefficient
    variable = DarcyCoeff
    execute_on = 'initial  timestep_end'
    porosity = eps
    viscosity = mu
    particle_diameter = 0.01
    kozeny_carman_const = 5.55
[../]
```

LinearChangeInTime

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel is used to change the value of an auxiliary variable linearly in time. The user gives a simulation time to both start and end the linear increases, as well as a target value to have the auxiliary variable end on. This is particularly useful for simulations of Temperature Programmed Desorption (TPD) curves wherein the temperature in the catalyst system is increased isothermally and linearly over a specified time period.

Computation [Pseudo Code]

if (time < start_time) value = initial_value

if (start_time <= time <= end_time)

slope = (final_value – initial_value) / (end_time – start_time)

value = initial_value + slope*(time – start_time)

if (time > end_time) value = final_value

Usage

In the example below, we are simulating a TPD wherein the temperature (T) raises inside the catalyst starting at a time of 13,530 seconds and continuing until 18,300 seconds and reaches a final temperature of 810 K. The initial value (423 K) for the temperature needs to be declared during the creation of the variable under the AuxVariables block in the input file.

```
[AuxVariables]
    [./T]
        order = FIRST
        family = LAGRANGE
        initial_condition = 423
    [../]
[]
[AuxKernels]
    [./TPD]
        type = LinearChangeInTime
        variable = T
        start_time = 13530
        end_time = 18300
        end_value = 810
        execute_on = 'initial   timestep_end'
    [../]
[]
```

MicroscaleIntegralAvg

Inheritance ➔ MicroscaleIntegralTotal

Notes ➔ This kernel inherits from MicroscaleIntegralTotal to first compute the total space integral of the microscale, then divide that result by the total microscale volume (of a single pellet), thus providing the average concentration of a species contained within the particle.

(See MicroscaleIntegralTotal for special notes and other notes)

Computation ➔ See MicroscaleIntegralTotal for calculation information

Usage

Recall that the MicroscaleDiffusion kernel set requires specific GlobalParams to be defined. It is recommended that you define these parameters as global to reduce input code redundancy.

The example below is for calculation of the average concentration (uAvg) inside of the pellets based on a MicroscaleDiffusion problem that was divided into 10 nodal variables (u0 through u9). The first node ID is 0 and the space_factor is 1, because the particle is spherical.

```
[GlobalParams]
        micro_length = 1
        num_nodes = 10
        coord_id = 2
[]

[AuxKernels]
        [./avg]
                type = MicroscaleIntegralAvg
                variable = uAvg
                space_factor = 1
                first_node = 0
                micro_vars = 'u0 u1 u2 u3 u4 u5 u6 u7 u8 u9'
                execute_on = 'initial   timestep_end'
        [../]
[]
```

MicroscaleIntegralTotal

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel is to be used specifically in conjunction with MicroscaleDiffusion kernels in the hybrid FD/FE method for multiscale mass conservation physics. The user must provide the list of microscale variables at all the nodes in the microscale for a particular conserved quantity (i.e., u0 through uL or v0 through vL). The list of those variables MUST be given in ascending order as well as the node ID for the first node in that list (otherwise result will be erroneous). User must also provide the same specified GlobalParams from the MicroscaleDiffusion notes and special notes. Optionally, the user may want to specify a 'space_factor' that is used to complete the total integral.

space_factor ➔ if coord_id = 0, space_factor is cross-sectional area

➔ if coord_id = 1, space_factor is the length of a cylindrical particle

➔ if coord_id = 2, then space_factor is not needed

(See MicroscaleDiffusion for special notes and other notes)

Computation

Calculation of the integral is carried out using the trapezoid rule for discrete spatial integration of the non-linear variable. The units of the integral will be that of the total mass of material in the pellets (not the mass per volume). To get the mass per volume in the pellets, use MicroscaleIntegralAvg.

Usage

Recall that the MicroscaleDiffusion kernel set requires specific GlobalParams to be defined. It is recommended that you define these parameters as global to reduce input code redundancy.

The example below is for calculation of the total mass (uTotal) inside of the pellets based on a MicroscaleDiffusion problem that was divided into 10 nodal variables (u0 through u9). The first node ID is 0 and the space_factor is 1, because the particle is spherical.

```
[GlobalParams]
    micro_length = 1
    num_nodes = 10
    coord_id = 2
[]

[AuxKernels]
    [./total]
        type = MicroscaleIntegralTotal
        variable = uTotal
        space_factor = 1
        first_node = 0
        micro_vars = 'u0 u1 u2 u3 u4 u5 u6 u7 u8 u9'
        execute_on = 'initial   timestep_end'
    [../]
[]
```

MicroscalePoreVolumePerTotalVolume

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel is to be used to calculate an effective microscale pore volume per total reactor volume ratio. This is a specific ratio that you generally only see used in the accumulation term, or time derivative term, that represents the microscale or pore-space concentration in a material balance. You would use this kernel if you microscale or pore-space mass balance was based on the total volume instead of on the microscale volume. This is most commonly the case when you model the microscale on an average basis instead of discretizing the microscale.

Computation

Micropore Volume per total volume (volume of micropores / total volume) = $\varepsilon_p(1-\varepsilon_b)$

ε_p = porosity of the microscale (i.e., particle/pellet/washcoat porosity)

ε_b = porosity of the macroscale (i.e., bulk bed porosity)

Usage

Couple with variables for microscale and macroscale porosity to formulate an auxiliary variable and corresponding calculation for the micro-pore volume to total volume ratio. In this example, 'total_pore' is the variable representing the total micro-porosity per total volume ratio. The porosity is represented by 'eps_b' and the particle/micro-scale porosity is 'eps_p'.

```
[AuxKernels]
    [./total_pore_calc]
        type = MicroscalePoreVolumePerTotalVolume
        variable = total_pore
        porosity = eps_b
        microscale_porosity = eps_p
        execute_on = 'initial   timestep_end'
    [../]
[]
```

MonolithAreaVolumeRatio

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel is used to calculate the ratio of monolith surface area to monolith volume (or to the total volume depending on user input and application). User must provide the cell density of the monolith (# cells per area) and the bulk voids fraction (volume of open channels per total reactor volume) for the monolith structure in question. By default, the calculation will return the ratio of monolith area to monolith volume. However, the user can specify that they would like to return the ratio of monolith area to total volume instead. Units on return will be in the same unit basis as the cell density (i.e., if the cell density is given in cells per square cm, then the return will be cm^{-1}.

Computation

Area-volume Ratio of Monolith (surface area / volume of monolith) = G_a

If (want area per total volume)

$$G_a = 4 \cdot (cd) \cdot d_h$$

If (want area per monolith volume) - Default

$$G_a = 4 \cdot (cd) \cdot d_h/(1-\varepsilon_b)$$

ε_b = bulk channel voids to total volume ratio

(cd) = cell density (# cells per face area of monolith)

d_h = hydraulic diameter of open channels (as square with rounded corners)

$$= 0.5(d_c + d_s)$$

$$d_s = \left(\frac{\varepsilon_b}{(cd)}\right)^{1/2} \qquad d_c = 2\left(\frac{\varepsilon_b}{\pi(cd)}\right)^{1/2}$$

Usage

Couple with a variable for the channel volume ratio (eps_b), user provides the cell density and (optionally) a Boolean for whether to calculate on the ratio of the monolith/solids volume. In this example, the user wants to calculate on a total volume ratio, so that option is set to 'false'.

```
[AuxKernels]
    [./Ga_calc]
        type = MonolithAreaVolumeRatio
        variable = Ga
        channel_vol_ratio = eps_b
        cell_density = 50
        per_solids_volume = false
        execute_on = 'initial   timestep_end'
    [../]
[]
```

MonolithMicroscaleTotalThickness

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel is used to calculate the total thickness to be used as the 'micro_length' parameter for setting up a coupled Microscale simulation for a washcoated monolith. Due to how the Microscale kernels are implemented (see MicroscaleCoefTimeDerivative), this parameter cannot be automatically set by this kernel (i.e., you cannot set the 'micro_length' as an auxiliary variable). Thus, it is recommended that you calculate this value in an input file first, then input the calculated value as a constant into 'micro_length'. Units on output are of the same basis of the units on input (i.e., if cell density is in cm^{-2}, then thickness comes out in cm).

Why not just use washcoat thickness for 'micro_length'?

The manner in which the Microscale kernels work is to discretize the microscale with finite differences and apply diffusion kernels onto those microscale variables. In a monolith system, you are diffusing into more than 1 wall/washcoat at any location in the macroscale (e.g., if you have square channels, you have 4 walls to diffuse into). Therefore, if you only provide the washcoat thickness for 'micro_length', then you will effectively be only allowing diffusion into 1 of those 4 walls. This will result is a significant underestimation of the level of mass transfer occurring in the system. The 'MonolithMicroscaleTotalThickness' is a singular calculation for thickness that will implicitly include all walls (regardless of the number of walls) by creating an effective thickness to account for the total volume of washcoating.

Alternatively, you could use multiple sets of microscale variables to represent the concentration of the microscale for each monolith wall separately, then apply separate mass-transfer and diffusion kernels for each of those sets of concentrations. Doing it in this way would allow you to use the true washcoat thickness for the microscale, but requires significantly more work to setup in the input file. That is why we generally recommend just using this simple calculation to get an effective thickness. **NOTE**: This kernel can also be used to estimate the actual washcoat thickness by providing an options 'wall_factor' value that will divide the total thickness by this factor (i.e., a representation of the number of walls). See this example for details.

Computation

Effective Washcoat Thickness (units of length) = $\left[\frac{1}{(cd)} - d_h^2\right]^{1/2} / n_w$

$d_h = 0.5(d_c + d_s)$

$d_s = \left(\frac{\varepsilon_b}{(cd)}\right)^{1/2}$ $\quad$ $d_c = 2\left(\frac{\varepsilon_b}{\pi(cd)}\right)^{1/2}$

ε_b = bulk channel voids to total volume ratio

(cd) = cell density (# cells per face area of monolith)

n_w = (optional) effective number of walls

[Default value = 1] Only give the number of walls if you plan to expand out all the microscale diffusion kernels for each individual wall.

Usage

Couple with a variable for the channel volume ratio (eps_b) and user provides the cell density.

```
[AuxKernels]
    [./wt_calc]
        type = MonolithMicroscaleTotalThickness
        variable = wt
        channel_vol_ratio = eps_b
        cell_density = 50
        execute_on = 'initial   timestep_end'
    [../]
[]
```

MonolithHydraulicDiameter

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel is used to calculate the hydraulic diameter of the channels in a monolith. This is not commonly used in individual kernels, but is a calculation that may be used elsewhere in the auxiliary system (see SimpleGasPropertiesBase). We do not provide an equivalent calculation for the hydraulic diameter of particles since that diameter is usually just the diameter of the particle itself, and thus, there is nothing to calculate. Units on return will be in

the same unit basis as the cell density (i.e., if the cell density is given in cells per square cm, then the return will be cm^{-1}.

Computation

Hydraulic Diameter of Channel (units of length) = d_h

d_h = hydraulic diameter of open channels (as square with rounded corners)

$$= 0.5(d_c + d_s)$$

$$d_s = \left(\frac{\varepsilon_b}{(cd)}\right)^{1/2} \qquad d_c = 2\left(\frac{\varepsilon_b}{\pi(cd)}\right)^{1/2}$$

ε_b = bulk channel voids to total volume ratio

(cd) = cell density (# cells per face area of monolith)

Usage

Couple with a variable for the channel volume ratio (eps_b) and user provides the cell density.

```
[AuxKernels]
    [./dh_calc]
        type = MonolithHydraulicDiameter
        variable = dh
        channel_vol_ratio = eps_b
        cell_density = 50
        execute_on = 'initial   timestep_end'
    [../]
[]
```

SchloeglDarcyCoefficient

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel can be used to calculate the a lumped variable term that can be coupled into the VariableVectorCoupledGradient or VariableLaplacian kernels to describe Schloegl-Darcy flow for specifically the flow through a porous membrane. Incompressible flow adheres to the following:

$$\nabla \cdot \mathbf{v} = 0$$

For Darcy's Law (with a Schloegl relationship), the velocity is defined as follows:

$$\mathbf{v} = -\frac{k_\phi}{\mu} F \cdot C_{ions} \cdot \nabla\phi - \frac{k_p}{\mu} \cdot \nabla p$$

In this formulation, the velocity of fluid through the membrane is both a function of pressure gradients and electric potential gradients. This kernel is focused on the coefficient associated with the pressure gradient. The kernel to calculate the coefficient associated with electric potential gradients is here.

Inside the membrane, it is assumed that there are no electric potential altering reactions occurring and no current exchange between the fluid through the membrane and the membrane itself (as per the Schloegl conditions). Therefore, the electric potential in the membrane follows a Laplace's equation:

$$\nabla \cdot i = 0 \rightarrow 0 = \nabla \cdot \left[-\frac{F^2}{RT} D \cdot C_{ions} \cdot \nabla \phi \right]$$

By combining these expressions with the incompressible flow condition, the pressure gradient in the domain is then also resolved by Laplace's equation. This is because the divergence of the electric potential by itself would be zero (according to the relationship above).

$$\nabla \cdot \left[-\frac{k_p}{\mu} \cdot \nabla p \right] = 0$$

This kernel is used to calculate the common coefficient in the pressure Laplacian and the Schloegl-Darcy flow coefficient.

Computation

Lumped Schloegl-Darcy Coefficient (length^2/pressure/time) = $\frac{k_p}{\mu}$

k_p = hydraulic permeability of the membrane domain (length^2)

μ = viscosity of the fluid (pressure * time)

Usage

In the below example, we are calculating the coefficient for Darcy flow in a membrane giving variables for viscosity and hydraulic permeability. Note that we need to make sure our units in the end make sense. If our pressure variable is in kPa and our domain dimensions are in cm, then the particle diameter must be in cm and the viscosity must use the 'pressure' unit basis to be in kPa*time. See SimpleFluidViscosity for information on viscosity units.

```
[./schloegl_darcy_calc]
        type = SchloeglDarcyCoefficient
        variable = SchloeglDarcyCoeff
        execute_on = 'initial  timestep_end'
        hydraulic_permeability = kp
        viscosity = mu
[../]
```

SchloeglElectrokineticCoefficient

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel can be used to calculate the a lumped variable term that can be coupled into the VariableVectorCoupledGradient or VariableLaplacian kernels to describe Schloegl-Darcy

flow for specifically the flow through a porous membrane. Incompressible flow adheres to the following:

$$\nabla \cdot \mathbf{v} = 0$$

For Darcy's Law (with a Schloegl relationship), the velocity is defined as follows:

$$\mathbf{v} = -\frac{k_\phi}{\mu} F \cdot C_{ions} \cdot \nabla\phi - \frac{k_p}{\mu} \cdot \nabla p$$

In this formulation, the velocity of fluid through the membrane is both a function of pressure gradients and electric potential gradients. This kernel is focused on the coefficient associated with the electric potential gradient. The kernel to calculate the coefficient associated with pressure gradients is here.

Inside the membrane, it is assumed that there are no electric potential altering reactions occurring and no current exchange between the fluid through the membrane and the membrane itself (as per the Schloegl conditions). Therefore, the electric potential in the membrane follows a Laplace's equation:

$$\nabla \cdot i = 0 \rightarrow 0 = \nabla \cdot \left[-\frac{F^2}{RT} D \cdot C_{ions} \cdot \nabla\phi \right]$$

By combining these expressions with the incompressible flow condition, the pressure gradient in the domain is then also resolved by Laplace's equation. This is because the divergence of the electric potential by itself would be zero (according to the relationship above).

$$\nabla \cdot \left[-\frac{k_p}{\mu} \cdot \nabla p \right] = 0$$

This kernel is used to calculate the coefficient for velocity from electric potential gradients.

Computation

Lumped Schloegl-Electrokinetic Coefficient (length^2/Voltz/time) = $\frac{k_\phi}{\mu} F \cdot C_{ions} \cdot f$

k_ϕ = electrokinetic permeability of the membrane domain (length^2)

μ = viscosity of the fluid (pressure * time)

F = Faraday's constant (default = 96,485.3 C/mol)

C_{ions} = total molar concentration of ions that can cross through the membrane

(units: moles per length^3)

f = [optional] Unit conversion factor (see note below) (default = 1)

NOTE: It is fairly difficult to get the units to work out for this lumped coefficient because it really depends on the units of electric potential, ion concentration, viscosity, etc. Thus, a conversion factor argument is included in the calculation to make sure that the units can work out in the

end. The user is responsible for making the units work out for themselves. Below is an example of how you can use this conversion factor to apply corrections for strange units.

Unit Example:

Consider we have the following unit conventions:

ϕ [V], C_{ions} [mol/cm^3], k_ϕ [cm^2], μ [kPa*min], lengths in the mesh [cm]

This unit basis for our variables, when put into the velocity function above, will yield:

$$\mathbf{v} = -K \cdot \nabla\phi \quad \text{with} \quad K = \frac{k_\phi}{\mu} F \cdot C_{ions} \cdot f$$

$\nabla\phi$ [V/cm], **v** [cm/min], and K needs units of [cm^2/V/min]

To get the correct units for K, determination of the conversion factor f is as follows:

([cm^2]/[kPa*min]) * [C/mol] * [mol/cm^3] * [?] = [cm^2/V/min]

([cm^2]/[kPa*min]) * [C] * [cm^{-3}] * [?] = [cm^2/V/min]

[kPa^{-1}*min^{-1}] * [C] * [cm^{-1}] * [?] = [cm^2/V/min]

[?] = [kPa*cm^3/(V*C)]

Now we need to use some definitions associated with SI units to parse this.

1 V = 1 J/C and 1 Pa = 1 J/m^3 or 1 kPa = 1000 J/m^3 = 0.001 J/cm^3

Combining this information gives the following:

[?] = [kPa*cm^3/((J/C)*C)] = [kPa*cm^3/J]

Thus, in this example the units of the conversion factor are [kPa*cm^3/J], which has a value of 0.001 because 1 kPa = 0.001 J/cm^3.

Usage

In the below example, we are calculating the coefficient for Schloegl flow in a membrane giving variables for viscosity, electrokinetic permeability, and an ion concentration. Note that we need to make sure our units in the end make sense (see discussion above). Following from that discussion on unit conversions, we also provide the 'conversion_factor' of 0.001 to make sure that the units work out for this example.

```
[./schloegl_electrokinetic_calc]
    type = SchloeglElectrokineticCoefficient
    variable = SchloeglEleCoeff
    execute_on = 'initial  timestep_end'
    electrokinetic_permeability = kf
    viscosity = mu
    ion_conc = C_H
    conversion_factor = 0.001
[../]
```

SimpleFluidPropertiesBase

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This is the base object that defines how all 'SimpleFluid' auxiliary values are calculated. It sets up the variables needed and provides a way to interface with some utilities and subroutines for performing unit conversions based on user inputs. Unlike 'SimpleGas' properties (see SimpleGasPropertiesBase), which requires coupling the variables for pressure and temperature, this kernel places default values in all of the declared variables and parameters. Thus, the user will only need to provide arguments to the specific 'SimpleFluid' auxiliary kernel. However, for simplicity, we placed all those arguments in the base class, as well as the functions to calculate specific properties, to reduce the redundancy of calculations in the code.

Units for specific inputs are provided as strings. Errors will be reported if user provides units that are not currently supported. Unit conversions are provided for energy, mass, volume, length, pressure, and time units. Only specific unit conversions are provided, all using SI base units.

How is this different from SimpleGasPropertiesBase?

The SimpleGasPropertiesBase object was an early attempt to include unit conversion calculations with relatively simplified methods for calculating specific properties of interest in the gas phase. This kernel works very similarly, but currently supports more unit conversions and puts the property calculation functions all in this base kernel. This reduces some code redundancy from the way it was done in SimpleGasPropertiesBase.

Currently Supported Units

length = "m", "cm", "mm"

time = "hr", "min", "s"

mass = "kg", "g", "mg"

energy = "kJ", "J"

pressure = "kPa", "Pa", "mPa"

volume = "kL", "L", "mL", "uL", "m^3", "cm^3", "mm^3"

(More will be added as needed)

Computation

Discussion on computation of properties will be put into each individual 'SimpleFluid' kernel (see the auxiliary kernels below).

Usage ➔ None, this object should not be invoked in the input files.

SimpleFluidDensity

Inheritance ➔ SimpleFluidPropertiesBase

Notes ➔ This kernel calculates the density of the fluid phase using a polynomial approximation for variations with temperature and pressure (see Computation below). This expression was based on a multi-dimensional analysis of the variations of standard water density from data collected from:

Engineering Toolbox. Water - Density, Specific Weight, and Thermal Expansion Coefficients. (2003). https://www.engineeringtoolbox.com/water-density-specific-weight-d_595.html. [Accessed on 13-12-2021].

Computation

Density (mass / volume) = $\rho_o \cdot (1.0135 + 4.9582 \times 10^{-7} \cdot P) \cdot (aT^2 + bT + c)$

P = fluid pressure (kPa)

T = fluid temperature (K)

a, b, c = temperature polynomial coefficients

(default values are for standard water between 0 and 350 °C)

(defaults: a = -2.9335E-6, b = 0.001529811, c = 0.797973)

ρ_o = reference density of the fluid at 0 °C (mass / volume)

(default = 1000 kg/L)

Usage

Users need to provide pressure and temperature variables. Optionally, users can also provide values for the reference density and the polynomial coefficients. Default values used will calculate the density of water.

```
[./rho_calc]
        type = SimpleFluidDensity
        variable = rho

        pressure = 101.35
        pressure_unit = “kPa”

        temperature = Tf          # always in K

        output_volume_unit = “cm^3”
        output_mass_unit = “kg”
[../]
```

SimpleFluidDispersion

Inheritance ➔ SimpleFluidPropertiesBase

Notes ➔ This kernel can make 4 different calculations of dispersion depending on user provided input Boolean arguments. The dispersion calculations are adjusted for temperature variations using an Arrhenius-like expression for the molecular diffusivity of a specific species of interest.

All the user needs to provide is a reference molecular diffusivity and the temperature at which that reference diffusivity corresponds to. Then, the user may also request that the dispersion coefficients be corrected for via a dispersivity coefficient and/or corrected for based on the porosity of the media. Calculations are made according to the following references.

[1] A.J. Easteal, W.E. Price, L.A. Woolf. Diaphragm cell for high-temperature diffusion measurements. J. Chem. Soc: Faraday Trans. 85 (1989) 1091-1097.

[2] A. Freeze and J. Cherry. Groundwater. Prentice-Hall. (1979) Ch. 9.

Computation

dispersion/diffusion coefficient (length^2 / time) = $\begin{cases} D_m \\ D_{m,p} \\ D_v \\ D_{v,p} \end{cases}$

D_m = molecular diffusion

$$D_m = D_o \cdot exp\left(-1991.805 \cdot \left[\frac{1}{T} - \frac{1}{T_o}\right]\right)$$

T = fluid temperature (K)

D_o = reference diffusivity

T_o = reference temperature (K)

$D_{m,p}$ = molecular diffusion through pores

$$D_{m,p} = \varepsilon^n D_m$$

ε = porosity of the media [default = 1, i.e., free flow]

n = effective diffusivity factor (0 < n < 2) [default = 0.5]

D_v = Dispersion (combines molecular diffusion and mechanic mixing)

$$D_v = D_m + d \cdot |\mathbf{v}|$$

d = dispersivity coefficient of the media [default = 0.01 cm]

v = magnitude of fluid velocity (provides mechanical mixing)

$D_{v,p}$ = Dispersion through porous media

$$D_{v,p} = \varepsilon^n D_v$$

To swap between methods, a set of Boolean arguments can be given to change the output behavior of this kernel.

'include_dispersivity_correction' → Setting 'true' will yield either D_v or $D_{v,p}$

'include_porosity_correction' → Setting 'true' will yield either $D_{m,p}$ or $D_{v,p}$

Combining these 2 arguments together can produce any of the calculations from above. For example, if 'include_dispersivity_correction' is 'false' and 'include_porosity_correction' is 'false', then the calculation returns D_m.

Usage

Users need to provide temperature, porosity, and velocity variables. Then, they should also provide the reference diffusivity and reference temperature for the chemical species of interest. Lastly, you need to provide the Boolean arguments for 'include_dispersivity_correction' and 'include_porosity_correction' to return a specific type of value.

```
[./Dvp_calc]
    type = SimpleFluidDispersion
    variable = Dvp

    temperature = Tf          # always in K
    macro_porosity = eps
    ux = vel_x
    uy = vel_y
    uz = vel_z
    vel_length_unit = "cm"
    vel_time_unit = "s"

    ref_diffusivity = 2.2E-5
    diff_length_unit = "cm"
    diff_time_unit = "s"
    ref_diff_temp = 298.15  # always in K

    effective_diffusivity_factor = 0.5
    dispersivity = 0.01
    disp_length_unit = "cm"

    include_dispersivity_correction = true
    include_porosity_correction = true

    output_length_unit = "cm"
    output_time_unit = "min"
[../]
```

SimpleFluidViscosity

Inheritance → SimpleFluidPropertiesBase

Notes ➔ This kernel calculates the viscosity of the fluid as a function of temperature using the Vogel-Fulcher-Tammann equation [1]. This equation, much like the density calculation here, is an empirical function that involves fitting parameters to data collected at various temperatures. By default, this equation is setup to calculate the viscosity of water [1]. However, users may provide alternative parameters for another fluid.

[1] D.S. Viswananth, G. Natarajan. Data Book on the Viscosity of Liquids. Hemisphere Publishing Corp. (1989).

Computation

Viscosity (Units: pressure * time OR mass/length/time)

$$= \mu_o \cdot exp\left(\frac{B}{T-C}\right)$$

T = fluid temperature (K)

μ_o = viscosity pre-exponential term (should have units of pressure*time)

= (default = 0.02939 Pa*s)

B = fitting parameter B (K) [default = 507.88 K]

C = fitting parameter C (K) [default = 149.3 K] (Must be < 273.15)

Usage

Users need to provide the temperature variable. This kernel can return units of viscosity in either a 'mass' basis (i.e., mass/length/time) or a 'pressure' basis (i.e., pressure*time). This is done by setting the 'unit_basis' argument to either 'mass' or 'pressure'.

```
[./mu_calc]
    type = SimpleFluidViscosity
    variable = mu
    temperature = Tf        # always in K

    unit_basis = 'pressure'

    output_pressure_unit = "kPa"
    output_time_unit = "min"
[../]
```

SimpleFluidElectrolyteViscosity

Inheritance ➔ SimpleFluidViscosity

Notes ➔ This kernel calculates the viscosity of the fluid as a function of temperature using the Vogel-Fulcher-Tammann equation [1] (as was done in SimpleFluidViscosity), but then applies an empirical correction factor based on the ionic strength of the fluid [2]. This correction factor

involves 3 fitting parameters that can be given by the user. By default, the calculation assumes the electrolyte viscosity behaves as a water + NaCl solution.

[1] D.S. Viswananth, G. Natarajan. Data Book on the Viscosity of Liquids. Hemisphere Publishing Corp. (1989).

[2] D.S. Viswananth, S. Dabir, et al. Viscosity of Liquids: Theory, Estimation, Experiment, and Data. Springer. (2007).

Computation

Viscosity (Units: pressure * time OR mass/length/time)

$$= \mu \cdot \left(1 + A\sqrt{I} + B \cdot I + C \cdot I^2\right)$$

I = fluid ionic strength (moles / volume)

μ = viscosity calculated from SimpleFluidViscosity

A = fitting parameter A $(L/mol)^{0.5}$ [default = 0.0062]

B = fitting parameter B (L/mol) [default = 0.0793]

C = fitting parameter C $(L/mol)^2$ [default = 0.0080]

Usage

Users need to provide the temperature variable and an ionic strength variable, along with the volume units of that strength. This kernel can return units of viscosity in either a 'mass' basis (i.e., mass/length/time) or a 'pressure' basis (i.e., pressure*time). This is done by setting the 'unit_basis' argument to either 'mass' or 'pressure'. NOTE: 'ionic_strength' must always have moles as the unit basis.

```
[./mu_calc]
        type = SimpleFluidElectrolyteViscosity
        variable = mu
        temperature = Tf        # always in K
        ionic_strength = I
        ionic_strength_volume_unit = "cm^3"

        unit_basis = 'pressure'

        output_pressure_unit = "kPa"
        output_time_unit = "min"
[../]
```

SimpleFluidFlatSurfaceMassTransCoef

Inheritance ➔ SimpleFluidPropertiesBase

Notes ➔ This kernel calculates the film mass transfer coefficient for a liquid species in system assuming that the mass transfer surface is a flat plate. This is most likely useful in conjunction with the InterfaceMassTransfer kernel as the 'transfer_rate' variable. That calculation is based on relationships with the Sherwood number and is a function of both the Reynolds and Schmidt numbers. Users must include the unit basis they want for the calculated value at the end.

(See SimpleFluidPropertiesBase for additional notes and calculations)

Computation

Mass Transfer Rate (length / time) => $k_m = \frac{Sh \cdot D_{eff}}{d_h}$

D_{eff} = Effective diffusivity in the micropores

$= (\varepsilon_w)^f \cdot D_m$

ε_w = porosity of the microscale (i.e., spherical particles)

f = effective diffusivity factor (default = 1.4)

D_m = molecular diffusivity of the species (see SimpleFluidPropertiesBase)

Sh = Sherwood number for a flat plate/surface

$= 0.664 Re^{1/2} \cdot Sc^{1/3}$

Usage

Recall that there are arguments for this pushed to GlobalParams

(See SimpleFluidPropertiesBase)

Example below calculates the mass transfer coefficient of O_2 liquid species into the surface. All common parameters were set in 'GlobalParams' (see SimpleFluidPropertiesBase) and parameters unique to this kernel are set here. User provides O_2 diffusivity reference in cm^2/s and outputs the mass transfer term in cm/min.

```
[./km_O2_calc]
        type = SimpleFluidFlatSurfaceMassTransCoef
        variable = km_O2

        temperature = Tf          # always in K
        macro_porosity = eps
        ux = vel_x
        uy = vel_y
        uz = vel_z
        vel_length_unit = "cm"
        vel_time_unit = "s"

        ref_diffusivity = 2.296E-5          #diffusivity of O2 in water (@298K)
```

```
        diff_length_unit = “cm”
        diff_time_unit = “s”
        ref_diff_temp = 298      # in K

        output_length_unit = “cm”
        output_time_unit = “min”
[../]
```

SimpleFluidMonolithMassTransCoef

Inheritance ➔ SimpleFluidPropertiesBase

Notes ➔ This kernel calculates the film mass transfer coefficient for a liquid species in a monolith system. That calculation is based on relationships with the Sherwood number and is a function of both the Reynolds and Schmidt numbers. Users must include the unit basis they want for the calculated value at the end.

(See SimpleFluidPropertiesBase for additional notes and calculations)

Computation

Mass Transfer Rate (length / time) => $k_m = \frac{Sh \cdot D_{eff}}{d_h}$

D_{eff} = Effective diffusivity in the micropores

$= (\varepsilon_w)^f \cdot D_m$

ε_w = porosity of the microscale (i.e., washcoat)

f = effective diffusivity factor (default = 1.4)

D_m = molecular diffusivity of the species (see SimpleFluidPropertiesBase)

Sh = Sherwood number in a monolith geometry

$= 0.3 + \left\{0.62Re^{1/2}Sc^{1/3} \cdot \left[1 + (0.4/Sc)^{2/3}\right]^{-1/4}\right\} \cdot \left\{1 + (Re/282000)^{5/8}\right\}^{4/5}$

Usage

Recall that there are arguments for this pushed to GlobalParams

(See SimpleFluidPropertiesBase)

Example below calculates the mass transfer coefficient of O_2 liquid species into a cylindrical particle. All common parameters were set in ‘GlobalParams’ (see SimpleFluidPropertiesBase) and parameters unique to this kernel are set here. User provides O_2 diffusivity reference in cm^2/s and outputs the mass transfer term in cm/min.

```
[./km_O2_calc]
        type = SimpleFluidMonolithMassTransCoef
        variable = km_O2
```

```
        temperature = Tf          # always in K
        macro_porosity = eps
        ux = vel_x
        uy = vel_y
        uz = vel_z
        vel_length_unit = "cm"
        vel_time_unit = "s"

        ref_diffusivity = 2.296E-5          #diffusivity of O2 in water (@298K)
        diff_length_unit = "cm"
        diff_time_unit = "s"
        ref_diff_temp = 298       # in K

        output_length_unit = "cm"
        output_time_unit = "min"
[../]
```

SimpleFluidSphericalMassTransCoef

Inheritance ➔ SimpleFluidPropertiesBase

Notes ➔ This kernel calculates the film mass transfer coefficient for a liquid species in a fixed-bed system, assuming the particles are spherical in shape. That calculation is based on relationships with the Sherwood number and is a function of both the Reynolds and Schmidt numbers. Users must include the unit basis they want for the calculated value at the end.

(See SimpleFluidPropertiesBase for additional notes and calculations)

Computation

Mass Transfer Rate (length / time) => $k_m = \frac{Sh \cdot D_{eff}}{d_h}$

D_{eff} = Effective diffusivity in the micropores

$= (\varepsilon_w)^f \cdot D_m$

ε_w = porosity of the microscale (i.e., spherical particles)

f = effective diffusivity factor (default = 1.4)

D_m = molecular diffusivity of the species (see SimpleFluidPropertiesBase)

Sh = Sherwood number in a monolith geometry

$= 2 + \left\{0.4Re^{1/2} + 0.06Re^{0.67}\right\} \cdot Sc^{0.4}$

Usage

Recall that there are arguments for this pushed to GlobalParams

(See SimpleFluidPropertiesBase)

Example below calculates the mass transfer coefficient of O_2 liquid species into the particles. All common parameters were set in 'GlobalParams' (see SimpleFluidPropertiesBase) and parameters unique to this kernel are set here. User provides O_2 diffusivity reference in cm^2/s and outputs the mass transfer term in cm/min.

```
[./km_O2_calc]
        type = SimpleFluidSphericalMassTransCoef
        variable = km_O2

        temperature = Tf          # always in K
        macro_porosity = eps
        ux = vel_x
        uy = vel_y
        uz = vel_z
        vel_length_unit = "cm"
        vel_time_unit = "s"

        ref_diffusivity = 2.296E-5          #diffusivity of O2 in water (@298K)
        diff_length_unit = "cm"
        diff_time_unit = "s"
        ref_diff_temp = 298      # in K

        output_length_unit = "cm"
        output_time_unit = "min"
[../]
```

SimpleGasPropertiesBase

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This is the base object that defines how all 'SimpleGas' auxiliary values are calculated. It sets up the variables needed and provides a way to interface with some utilities and subroutines for performing unit conversions based on user inputs. Because of this, all the 'SimpleGas' auxiliary kernels require a common set of input parameters, although some do require additional arguments beyond the base set of arguments.

Since most of those input arguments are common to all other 'SimpleGas' auxiliary kernels, you can put those arguments in the 'GlobalParams' block of you input file (see example below). However, some of the arguments in the derived 'SimpleGas' auxiliary kernels may be specific to a particular gas species or physical property.

Units for specific inputs are provided as strings. Errors will be reported if user provides units that are not currently supported.

```
[GlobalParams]
        # You may have the arguments below in GlobalParams
```

```
    temperature = temp_var          # in K
    pressure = press_var            # in kPa
    micro_porosity = eps_p
    macro_porosity = eps_b
    velocity = vel_y
    vel_length_unit = "cm"
    vel_time_unit = "min"
    execute_on = 'initial  timestep_end'

    # Arguments in the below list can also go in GlobalParams
    #       or defined locally in other auxiliary kernels if desired
    #       NOTE: it is usually recommended to provide these locally
    characteristic_length = dh      #hydraulic diameter
    char_length_unit = "cm"
    ref_diffusivity = 0.2       #diffusivity of H2O in air (@STP)
    diff_length_unit = "cm"
    diff_time_unit = "s"
    ref_diff_temp = 273       # in K

    # This factor is a power applied to the micro_porosity to
    #       estimate an effective pore diffusion rate. It ranges
    #       from 1 to 2, and defaults to 1.4.
    effective_diffusivity_factor = 1.4  #Default
[]
```

How is this different from GasPropertiesBase?

The original interface for calculating gas properties (GasPropertiesBase) is more difficult to use and more strict on units. Thus, the purpose of this auxiliary system was to simplify the user interface and allow for more flexibility on units. The downside is that these auxiliary kernels using the 'SimpleGas' approach will always make the assumption of an ideal gas and will always assume that your gas is a dilute system that is primarily made up of standard air (i.e., mostly N_2, O_2, and H_2O, with other gas species being on order of ppm to 1% by volume). These assumptions allow us to use simpler approximations for gas properties, like viscosity, density, and molecular diffusivity, and free up the strict unit convention to allow users to give inputs and receive outputs in units they need.

Currently Supported Units

length = "m", "cm", "mm"

time = "hr", "min", "s"

mass = "kg", "g", "mg"

(More will be added as needed)

Computation

For convenience, several standard calculations are presented below. These are the calculations used for gas density, gas viscosity, molecular diffusivity, and other calculations.

$$\rho = \frac{P}{287.058 \cdot T} \qquad \mu = 1.458 \times 10^{-5} \cdot \frac{T^{1.5}}{(110.4+T)} \qquad D_m = D_{ref} \cdot exp\left(-887.5\left[\frac{1}{T} - \frac{1}{T_{ref}}\right]\right)$$

P = gas pressure (in kPa)

T = gas temperature (in K)

ρ = gas density

μ = gas viscosity

D_{ref} = molecular diffusivity of a gas species of interest at a reference temperature

T_{ref} = reference temperature for molecular diffusivity

D_m = molecular diffusivity of a gas species of interest at system gas temperature

$$Re = \frac{\rho v d}{\mu} \qquad Sc = \frac{\mu}{\rho D_m} \qquad Pr = \frac{c_p \cdot \mu}{K_g}$$

Re = Reynolds number

v = average gas velocity

d = characteristic length (or hydraulic diameter)

Sc = Schmidt number

Pr = Prandtl number

c_p = isobaric heat capacity (see SimpleGasIsobaricHeatCapacity)

K_g = thermal conductivity (see SimpleGasThermalConductivity)

Usage ➔ None, this object should not be invoked in the input files.

SimpleGasFlatSurfaceMassTransCoef

Inheritance ➔ SimpleGasPropertiesBase

Notes ➔ This kernel calculates the film mass transfer coefficient for a gas species in system assuming that the mass transfer surface is a flat plate. This is most likely useful in conjunction with the InterfaceMassTransfer kernel as the 'transfer_rate' variable. That calculation is based on relationships with the Sherwood number and is a function of both the Reynolds and Schmidt numbers. Users must include the unit basis they want for the calculated value at the end.

(See SimpleGasPropertiesBase for additional notes and calculations)

Computation

Mass Transfer Rate (length / time) => $k_m = \frac{Sh \cdot D_{eff}}{d_h}$

D_{eff} = Effective diffusivity in the micropores

$= (\varepsilon_w)^f \cdot D_m$

ε_w = porosity of the microscale (i.e., spherical particles)

f = effective diffusivity factor (default = 1.4)

D_m = molecular diffusivity of the species (see SimpleGasPropertiesBase)

Sh = Sherwood number for a flat plate/surface

$= 0.664 Re^{1/2} \cdot Sc^{1/3}$

Usage

Recall that there are arguments for this pushed to GlobalParams (See SimpleGasPropertiesBase)

Example below calculates the mass transfer coefficient of O_2 gas into the particles. All common parameters were set in 'GlobalParams' (see SimpleGasPropertiesBase) and parameters unique to this kernel are set here. User provides O_2 diffusivity reference in cm^2/s and outputs the mass transfer term in cm/min.

```
[./km_O2_calc]
        type = SimpleGasFlatSurfaceMassTransCoef
        variable = km_O2

        #NOTE: we skipped the other parameters set here (assuming they
        #       got set in GlobalParams)

        ref_diffusivity = 0.561    #diffusivity of O2 in air (@473K)
        diff_length_unit = "cm"
        diff_time_unit = "s"
        ref_diff_temp = 473      # in K

        output_length_unit = "cm"
        output_time_unit = "min"
[../]
```

SimpleGasMonolithMassTransCoef

Inheritance ➔ SimpleGasPropertiesBase

Notes ➔ This kernel calculates the film mass transfer coefficient for a gas species in a monolith system. That calculation is based on relationships with the Sherwood number and is a function of both the Reynolds and Schmidt numbers. Users must include the unit basis they want for the calculated value at the end.

(See SimpleGasPropertiesBase for additional notes and calculations)

Computation

Mass Transfer Rate (length / time) => $k_m = \frac{Sh \cdot D_{eff}}{d_h}$

D_{eff} = Effective diffusivity in the micropores

$= (\varepsilon_w)^f \cdot D_m$

ε_w = porosity of the microscale (i.e., washcoat)

f = effective diffusivity factor (default = 1.4)

D_m = molecular diffusivity of the species (see SimpleGasPropertiesBase)

Sh = Sherwood number in a monolith geometry

$$= 0.3 + \left\{0.62Re^{1/2}Sc^{1/3} \cdot \left[1 + (0.4/Sc)^{2/3}\right]^{-1/4}\right\} \cdot \left\{1 + (Re/282000)^{5/8}\right\}^{4/5}$$

Usage

Recall that there are arguments for this pushed to GlobalParams (See SimpleGasPropertiesBase)

Example below calculates the mass transfer coefficient of O_2 gas into the washcoat. All common parameters were set in 'GlobalParams' (see SimpleGasPropertiesBase) and parameters unique to this kernel are set here. User provides O_2 diffusivity reference in cm^2/s and outputs the mass transfer term in cm/min.

```
[./km_O2_calc]
        type = SimpleGasMonolithMassTransCoef
        variable = km_O2

        #NOTE: we skipped the other parameters set here (assuming they
        #       got set in GlobalParams)

        ref_diffusivity = 0.561    #diffusivity of O2 in air (@473K)
        diff_length_unit = "cm"
        diff_time_unit = "s"
        ref_diff_temp = 473      # in K

        output_length_unit = "cm"
        output_time_unit = "min"
[../]
```

SimpleGasMonolithHeatTransCoef

Inheritance ➔ SimpleGasPropertiesBase

Notes ➔ This kernel calculates the heat transfer coefficient in a monolith system at the interface between the open channels and the washcoat/monolith walls. That calculation is based on relationships with the Nusselt number and is a function of both the Reynolds and Prandtl numbers. Users must include the unit basis they want for the calculated value at the end.

(See SimpleGasPropertiesBase for additional notes and calculations)

Computation

Heat Transfer Rate (energy / time / length^2 / temperature) => $h_s = \frac{Nu \cdot K_g}{d_h}$

K_g = thermal conductivity of gas (see SimpleGasThermalConductivity)

d_h = hydraulic diameter

Nu = Nusselt number in a monolith geometry

$= 0.35 \cdot Re^{1/2} Pr^{-1/6}$

Pr = Prandtl number (see SimpleGasPropertiesBase)

Usage

Recall that there are arguments for this pushed to GlobalParams (See SimpleGasPropertiesBase)

Example below calculates the heat transfer coefficient into the washcoat. All common parameters were set in 'GlobalParams' (see SimpleGasPropertiesBase) and parameters unique to this kernel are set here. User requests output in J/min/cm^2/K.

```
[./hs_calc]
        type = SimpleGasMonolithHeatTransCoef
        variable = hs

        #NOTE: we skipped the other parameters set here (assuming they
        #       got set in GlobalParams). Here only the important params
        #       are being set.
        pressure = 101.35
        temperature = Tf
        characteristic_length = dh
        char_length_unit = "cm"

        output_length_unit = "cm"
        output_time_unit = "min"
        output_energy_unit = "J"
[../]
```

SimpleGasSphericalMassTransCoef

Inheritance ➔ SimpleGasPropertiesBase

Notes ➔ This kernel calculates the film mass transfer coefficient for a gas species in a fixed-bed system, assuming the particles are spherical in shape. That calculation is based on relationships with the Sherwood number and is a function of both the Reynolds and Schmidt numbers. Users must include the unit basis they want for the calculated value at the end.

(See SimpleGasPropertiesBase for additional notes and calculations)

Computation

Mass Transfer Rate (length / time) => $k_m = \frac{Sh \cdot D_{eff}}{d_h}$

D_{eff} = Effective diffusivity in the micropores

$= (\varepsilon_w)^f \cdot D_m$

ε_w = porosity of the microscale (i.e., spherical particles)

f = effective diffusivity factor (default = 1.4)

D_m = molecular diffusivity of the species (see SimpleGasPropertiesBase)

Sh = Sherwood number in a monolith geometry

$= 2 + \{0.4Re^{1/2} + 0.06Re^{0.67}\} \cdot Sc^{0.4}$

Usage

Recall that there are arguments for this pushed to GlobalParams (See SimpleGasPropertiesBase)

Example below calculates the mass transfer coefficient of O_2 gas into the particles. All common parameters were set in 'GlobalParams' (see SimpleGasPropertiesBase) and parameters unique to this kernel are set here. User provides O_2 diffusivity reference in cm^2/s and outputs the mass transfer term in cm/min.

```
[./km_O2_calc]
        type = SimpleGasSphericalMassTransCoef
        variable = km_O2

        #NOTE: we skipped the other parameters set here (assuming they
        #       got set in GlobalParams)

        ref_diffusivity = 0.561    #diffusivity of O2 in air (@473K)
        diff_length_unit = "cm"
        diff_time_unit = "s"
        ref_diff_temp = 473      # in K

        output_length_unit = "cm"
        output_time_unit = "min"
[../]
```

SimpleGasSphericalHeatTransCoef

Inheritance ➔ SimpleGasPropertiesBase

Notes ➔ This kernel calculates the heat transfer coefficient in a fixed-bed system at the interface between the surface of spherical particles and the bulk fluid phase. That calculation is based on relationships with the Nusselt number and is a function of both the Reynolds and Prandtl numbers. Users must include the unit basis they want for the calculated value at the end.

(See SimpleGasPropertiesBase for additional notes and calculations)

Computation

Heat Transfer Rate (energy / time / length^2 / temperature) => $h_s = \frac{Nu \cdot K_g}{d_h}$

K_g = thermal conductivity of gas (see SimpleGasThermalConductivity)

d_h = hydraulic diameter

Nu = Nusselt number in a monolith geometry

$= 2 + 0.4 \cdot Re^{1/2} Pr^{1/3}$

Pr = Prandtl number (see SimpleGasPropertiesBase)

Usage

Recall that there are arguments for this pushed to GlobalParams (See SimpleGasPropertiesBase)

Example below calculates the heat transfer coefficient into the particles. All common parameters were set in 'GlobalParams' (see SimpleGasPropertiesBase) and parameters unique to this kernel are set here. User requests output in J/min/cm^2/K.

```
[./hs_calc]
        type = SimpleGasSphericalHeatTransCoef
        variable = hs

        #NOTE: we skipped the other parameters set here (assuming they
        #       got set in GlobalParams). Here only the important params
        #       are being set.
        pressure = 101.35
        temperature = Tf
        characteristic_length = 0.2
        char_length_unit = "cm"

        output_length_unit = "cm"
        output_time_unit = "min"
        output_energy_unit = "J"
[../]
```

SimpleGasCylinderWallHeatTransCoef

Inheritance ➔ SimpleGasPropertiesBase

Notes ➔ This kernel calculates the heat transfer coefficient at the walls of a cylindrical reactor. This will primarily be used to evaluate the heat transfer at a physical boundary of the domain. That calculation is based on relationships with the Nusselt number and is a function of both the Reynolds and Prandtl numbers. Users must include the unit basis they want for the calculated value at the end.

(See SimpleGasPropertiesBase for additional notes and calculations)

Computation

Heat Transfer Rate (energy / time / length^2 / temperature) => $h_s = \frac{Nu \cdot K_g}{d_h}$

K_g = thermal conductivity of gas (see SimpleGasThermalConductivity)

d_h = hydraulic diameter

Nu = Nusselt number in a monolith geometry

$= 0.27 \cdot Re^{4/5} Pr^{1/3}$

Pr = Prandtl number (see SimpleGasPropertiesBase)

Usage

Recall that there are arguments for this pushed to GlobalParams (See SimpleGasPropertiesBase)

Example below calculates the heat transfer coefficient at the walls of the domain (assuming a cylindrical reactor geometry). All common parameters were set in 'GlobalParams' (see SimpleGasPropertiesBase) and parameters unique to this kernel are set here. User requests output in J/min/cm^2/K.

NOTE: The characteristic length for this kernel is the diameter of the reactor (not the hydraulic diameter of a monolith or a particle size.)

```
[./hw_calc]
    type = SimpleGasCylinderWallHeatTransCoef
    variable = hw

    #NOTE: we skipped the other parameters set here (assuming they
    #      got set in GlobalParams). Here only the important params
    #      are being set.
    pressure = 101.35
    temperature = Tf
    characteristic_length = 2
    char_length_unit = "cm"
```

output_length_unit = “cm”
output_time_unit = “min”
output_energy_unit = “J”
[../]

SimpleGasEffectivePoreDiffusivity

Inheritance ➔ SimpleGasPropertiesBase

Notes ➔ This kernel calculates the effective pore diffusivity (or applied pore diffusivity) of a gas species in the microscale dimension. The difference from this diffusivity and SimpleGasPoreDiffusivity is that this kernel calculates a variable that would be used directly into the Microscale kernel material balance (MicroscaleVariableDiffusion, etc), where as SimpleGasPoreDiffusivity is just a calculation of the true pore diffusivity (D_p). Users should specify the volume basis of their microscale material balance to get the correct effective term calculated by setting the ‘per_solids_volume’ Boolean on input.

Users must also include the unit basis they want for the calculated value at the end.

(See SimpleGasPropertiesBase for additional notes and calculations)

Computation

Effective Pore Diffusivity (length^2 / time) = D_{eff}

If ‘per_solids_volume’ == true

$$D_{eff} = \varepsilon_w D_p$$

else

$$D_{eff} = \varepsilon_w (1 - \varepsilon_b) D_p$$

$$D_p = (\varepsilon_w)^f \cdot D_m$$

ε_w = porosity of the microscale (i.e., washcoat or particles)

f = effective diffusivity factor (default = 1.4)

ε_b = porosity of the macroscale (i.e., volume solids per total volume)

D_m = molecular diffusivity of the species (see SimpleGasPropertiesBase)

Usage

Recall that there are arguments for this pushed to GlobalParams (See SimpleGasPropertiesBase)

Example below calculates the effective diffusivity of O_2 gas through the pore space of the microscale to use in Microscale kernels. All common parameters were set in ‘GlobalParams’ (see SimpleGasPropertiesBase) and parameters unique to this kernel are set here. User provides O_2 diffusivity reference in cm^2/s and outputs the diffusion term in cm^2/min.

```
[./De_O2_calc]
        type = SimpleGasEffectivePoreDiffusivity
        variable = De_O2

        #NOTE: we skipped the other parameters set here (assuming they
        #       got set in GlobalParams)

        ref_diffusivity = 0.561    #diffusivity of O2 in air (@473K)
        diff_length_unit = "cm"
        diff_time_unit = "s"
        ref_diff_temp = 473      # in K

        per_solids_volume = false
        output_length_unit = "cm"
        output_time_unit = "min"
[../]
```

SimpleGasPoreDiffusivity

Inheritance ➔ SimpleGasPropertiesBase

Notes ➔ This kernel calculates the pore diffusivity of a gas species in the microscale dimension. Users must include the unit basis they want for the calculated value at the end.

(See SimpleGasPropertiesBase for additional notes and calculations)

Computation

Pore Diffusivity (length^2 / time) = $(\varepsilon_w)^f \cdot D_m$

ε_w = porosity of the microscale (i.e., washcoat or particles)

f = effective diffusivity factor (default = 1.4)

D_m = molecular diffusivity of the species (see SimpleGasPropertiesBase)

Usage

Recall that there are arguments for this pushed to GlobalParams (See SimpleGasPropertiesBase)

Example below calculates the diffusivity of O_2 gas through the pore space of the microscale. All common parameters were set in 'GlobalParams' (see SimpleGasPropertiesBase) and parameters unique to this kernel are set here. User provides O_2 diffusivity reference in cm^2/s and outputs the diffusion term in cm^2/min.

```
[./D_O2_calc]
        type = SimpleGasPoreDiffusivity
        variable = D_O2

        #NOTE: we skipped the other parameters set here (assuming they
```

```
#       got set in GlobalParams)

ref_diffusivity = 0.561    #diffusivity of O2 in air (@473K)
diff_length_unit = "cm"
diff_time_unit = "s"
ref_diff_temp = 473      # in K

output_length_unit = "cm"
output_time_unit = "min"
[../]
```

SimpleGasEffectiveKnudsenDiffusivity

Inheritance ➔ SimpleGasPropertiesBase

Notes ➔ This kernel calculates the effective Knudsen diffusivity (or applied Knudsen diffusivity) of a gas species in the microscale dimension. The Knudsen diffusivity generally only applies for microscale physics in situations where the pore-space molecules diffuse through are very, very narrow pores (i.e., not mesoporous materials). The total diffusivity is then a non-linear combination of pore diffusivity and a Knudsen diffusion correction. The difference from this diffusivity and SimpleGasKnudsenDiffusivity is that this kernel calculates a variable that would be used directly into the Microscale kernel material balance (MicroscaleVariableDiffusion, etc), where as SimpleGasKnudsenDiffusivity is just a calculation of the true Knudsen diffusivity. Users must include the unit basis they want for the calculated value at the end. Users should specify the volume basis of their microscale material balance to get the correct effective term calculated by setting the 'per_solids_volume' Boolean on input.

NOTE: The 'characteristic_length' for this auxiliary kernel must be average pore radius.

(See SimpleGasPropertiesBase for additional notes and calculations)

Computation

Effective Knudsen Diffusivity (length^2 / time) = $D_{k,eff}$

If 'per_solids_volume' == true

$$D_{k,eff} = \varepsilon_w \left[D_k^{-1} + D_p^{-1}\right]^{-1}$$

else

$$D_{k,eff} = \varepsilon_w (1 - \varepsilon_b) \left[D_k^{-1} + D_p^{-1}\right]^{-1}$$

$$D_k = 9700 r_p \sqrt{T/MW}$$

r_p = pore radius (in cm)

T = temperature (in K)

MW = molecular weight (in g/mol)

$$D_p = (\varepsilon_w)^f \cdot D_m$$

ε_w = porosity of the microscale (i.e., washcoat or particles)

f = effective diffusivity factor (default = 1.4)

D_m = molecular diffusivity of the species (see SimpleGasPropertiesBase)

Usage

Recall that there are arguments for this pushed to GlobalParams (See SimpleGasPropertiesBase)

Example below calculates the Knudsen diffusivity of O_2 gas through the pore space of the microscale. All common parameters were set in 'GlobalParams' (see SimpleGasPropertiesBase) and parameters unique to this kernel are set here. User provides O_2 diffusivity reference in cm^2/s and outputs the diffusion term in cm^2/min.

```
[./Dk_O2_calc]
    type = SimpleGasEffectiveKnudsenDiffusivity
    variable = Dk_O2

    #NOTE: we skipped the other parameters set here (assuming they
    #       got set in GlobalParams)

    ref_diffusivity = 0.561    #diffusivity of O2 in air (@473K)
    diff_length_unit = "cm"
    diff_time_unit = "s"
    ref_diff_temp = 473      # in K

    characteristic_length = 2E-6   # average pore radius
    char_length_unit = "mm"
    molar_weight = 32              # g/mol
    per_solids_volume = false

    output_length_unit = "cm"
    output_time_unit = "min"
[../]
```

SimpleGasKnudsenDiffusivity

Inheritance ➔ SimpleGasPropertiesBase

Notes ➔ This kernel calculates the Knudsen diffusivity of a gas species in the microscale dimension. The Knudsen diffusivity generally only applies for microscale physics in situations where the pore-space molecules diffuse through are very, very narrow pores (i.e., not mesoporous materials). The total diffusivity is then a non-linear combination of pore diffusivity

and a Knudsen diffusion correction. Users must include the unit basis they want for the calculated value at the end.

NOTE: The 'characteristic_length' for this auxiliary kernel must be average pore radius.

(See SimpleGasPropertiesBase for additional notes and calculations)

Computation

Knudsen Diffusivity (length^2 / time) = $\left[D_k^{-1} + D_p^{-1}\right]^{-1}$

$$D_k = 9700 r_p \sqrt{T/MW}$$

r_p = pore radius (in cm)

T = temperature (in K)

MW = molecular weight (in g/mol)

$$D_p = (\varepsilon_w)^f \cdot D_m$$

ε_w = porosity of the microscale (i.e., washcoat or particles)

f = effective diffusivity factor (default = 1.4)

D_m = molecular diffusivity of the species (see SimpleGasPropertiesBase)

Usage

Recall that there are arguments for this pushed to GlobalParams (See SimpleGasPropertiesBase)

Example below calculates the Knudsen diffusivity of O_2 gas through the pore space of the microscale to use in Microscale kernels. All common parameters were set in 'GlobalParams' (see SimpleGasPropertiesBase) and parameters unique to this kernel are set here. User provides O_2 diffusivity reference in cm^2/s and outputs the diffusion term in cm^2/min.

```
[./Dk_O2_calc]
        type = SimpleGasKnudsenDiffusivity
        variable = Dk_O2

        #NOTE: we skipped the other parameters set here (assuming they
        #       got set in GlobalParams)

        ref_diffusivity = 0.561    #diffusivity of O2 in air (@473K)
        diff_length_unit = "cm"
        diff_time_unit = "s"
        ref_diff_temp = 473      # in K

        characteristic_length = 2E-6   # average pore radius
        char_length_unit = "mm"
```

```
        molar_weight = 32              # g/mol

        output_length_unit = "cm"
        output_time_unit = "min"
[../]
```

SimpleGasDispersion

Inheritance ➔ SimpleGasPropertiesBase

Notes ➔ This kernel calculates the dispersion coefficient of a gas species in the macroscale dimension. When using this kernel, users must provide the diameter of the reactor for the 'characteristic_length'. The calculation of dispersion is based on an empirical relationship with average linear velocity, column/reactor diameter, and the dimensionless numbers for Reynolds and Schmidt. Users must include the unit basis they want for the calculated value at the end.

(See SimpleGasPropertiesBase for additional notes and calculations)

Computation

Dispersion Coefficient (length^2 / time) = $v \cdot d \cdot \left(0.5 + \frac{20}{Re \cdot Sc}\right)$

d = reactor diameter (i.e., characteristic_length)

v = average linear velocity

Re = Reynolds number

Sc = Schmidt number

(see SimpleGasPropertiesBase for other calculation information)

Usage

Recall that there are arguments for this pushed to GlobalParams (See SimpleGasPropertiesBase)

Example below calculates the dispersion coefficient of O_2 gas through the macroscale of the domain. All common parameters were set in 'GlobalParams' (see SimpleGasPropertiesBase) and parameters unique to this kernel are set here. User provides O_2 diffusion reference in cm^2/s and outputs the dispersion term in cm^2/min.

```
[./D_O2_calc]
        type = SimpleGasDispersion
        variable = D_O2

        #NOTE: we skipped the other parameters set here (assuming they
        #       got set in GlobalParams)

        ref_diffusivity = 0.561    #diffusivity of O2 in air (@473K)
```

```
        diff_length_unit = “cm”
        diff_time_unit = “s”
        ref_diff_temp = 473      # in K

        characteristic_length = 2   # diameter of reactor
        char_length_unit = “cm”

        output_length_unit = “cm”
        output_time_unit = “min”
[../]
```

SimpleGasDensity

Inheritance ➔ SimpleGasPropertiesBase

Notes ➔ This kernel calculates the density of the gas phase assuming it behaves as an ideal gas and is primarily standard air (i.e., roughly 79 % N_2 and 21 % O_2).

(See SimpleGasPropertiesBase for additional notes and calculations)

Computation

Density (mass / length^3) = $\frac{P}{R_g T}$

P = gas pressure (kPa)

T = gas temperature (K)

R_g = Standard Air gas constant = 287.058 kPa*cm^3/g/K

NOTE: This calculation gives g/cm^3 for density (but users can request different units on output)

(see SimpleGasPropertiesBase for other calculation information)

Usage

Recall that there are arguments for this pushed to GlobalParams (See SimpleGasPropertiesBase)

Example below calculates the density of air for variable ‘rho’ based on fluid temperature ‘Tf’. All common parameters were set in ‘GlobalParams’ (see SimpleGasPropertiesBase) and parameters unique to this kernel are set here. User sets the ‘output’ unit arguments to get density back in kg/cm^3.

```
[./rho_calc]
        type = SimpleGasDensity
        variable = rho

        #NOTE: we skipped the other parameters set here (assuming they
        #       got set in GlobalParams). Most relevant parameters are
        #       being set below.
        pressure = 101.35
```

temperature = Tf

output_length_unit = “cm”
output_mass_unit = “kg”
[../]

SimpleGasViscosity

Inheritance ➔ SimpleGasPropertiesBase

Notes ➔ This kernel calculates the viscosity of the gas phase assuming it behaves as an ideal gas and is primarily standard air (i.e., roughly 79 % N_2 and 21 % O_2).

(See SimpleGasPropertiesBase for additional notes and calculations)

Computation

Viscosity (mass / length / time) = $1.458 \times 10^{-5} \cdot \frac{T^{1.5}}{(110.4+T)}$

T = gas temperature (K)

NOTE: This calculation gives g/cm/s for viscosity (users can request different units on output)

(see SimpleGasPropertiesBase for other calculation information)

Usage

Recall that there are arguments for this pushed to GlobalParams (See SimpleGasPropertiesBase)

Example below calculates the density of air for variable ‘mu’ based on fluid temperature ‘Tf’. All common parameters were set in ‘GlobalParams’ (see SimpleGasPropertiesBase) and parameters unique to this kernel are set here. User sets the ‘output’ unit arguments to get density back in kg/cm/min.

[./mu_calc]
type = SimpleGasViscosity
variable = mu

#NOTE: we skipped the other parameters set here (assuming they
# got set in GlobalParams). Most relevant parameters are
# being set below.
pressure = 101.35
temperature = Tf

output_length_unit = “cm”
output_mass_unit = “kg”
output_time_unit = “min”
[../]

SimpleGasVolumeFractionToConcentration

Inheritance ➔ SimpleGasPropertiesBase

Notes ➔ This kernel calculates the concentration of a gas species from a given volume fraction. Volume fractions can be in %, ppm, or ppb. Concentration calculations are based on ideal gas law. These calculations can be used to establish inlet conditions more easily for situations where the volume fraction of species remains the same, but the temperatures and pressures at the inlet are varying.

(See SimpleGasPropertiesBase for additional notes and calculations)

Computation

Concentration (moles / volume) = $\frac{Py}{RT}$

T = gas temperature (K)

P = pressure (in kPa, Pa, or mPa)

y = volume fraction of the gas species

R = gas law constant

NOTE: This calculation can use pressure of various units.

(see SimpleGasPropertiesBase for other calculation information)

Usage

Recall that there are arguments for this pushed to GlobalParams (See SimpleGasPropertiesBase)

Volume fraction can be provided as a variable. All common parameters were set in 'GlobalParams' (see SimpleGasPropertiesBase) and parameters unique to this kernel are set here. User sets the 'output' unit arguments to get concentration back in mol/L.

```
[./NH3_in_calc]
        type = SimpleGasVolumeFractionToConcentration
        variable = NH3_inlet

        #NOTE: we skipped the other parameters set here (assuming they
        #       got set in GlobalParams). Most relevant parameters are
        #       being set below.
        pressure = 101.35
        pressure_unit = "kPa"
        temperature = 298 # K

        volfrac = NH3_ppm
        input_volfrac_unit = "ppm"

        output_volume_unit = "L"
[../]
```

SimpleGasIsobaricHeatCapacity

Inheritance ➔ SimpleGasPropertiesBase

Notes ➔ This kernel calculates the heat capacity (at constant pressure) of the gas phase assuming it behaves as an ideal gas and is primarily made up of standard air (i.e., roughly 79 % N_2 and 21 % O_2). This is an empirical formulation based on data for standard air heat capacity as a function of temperature and pressure. These approximations are good between 1 to 20 bar of pressure and 0 to 1600 °C.

(See SimpleGasPropertiesBase for additional notes and calculations)

Computation

Heat Capacity (energy/ mass / temperature) = $1 + c_{p,max} \frac{K_c T^{2.5}}{1+K_c T^{2.5}}$

$c_{p,max}$ = maximum heat capacity = 0.3 kJ/kg/K

K_c = heat capacity coefficient = 2.68588E-08 $K^{-2.5}$

T = gas temperature (K)

(see SimpleGasPropertiesBase for other calculation information)

Usage

Recall that there are arguments for this pushed to GlobalParams (See SimpleGasPropertiesBase)

Example below calculates the heat capacity of air for variable 'cp' based on fluid temperature 'Tf'. All common parameters were set in 'GlobalParams' (see SimpleGasPropertiesBase) and parameters unique to this kernel are set here. User sets the 'output' unit arguments to get heat capacity back in J/kg/K.

```
[./cp_calc]
        type = SimpleGasIsobaricHeatCapacity
        variable = cp

        #NOTE: we skipped the other parameters set here (assuming they
        #       got set in GlobalParams). Most relevant parameters are
        #       being set below.
        pressure = 101.35
        temperature = Tf

        output_energy_unit = "J"
        output_mass_unit = "kg"
[../]
```

SimpleGasThermalConductivity

Inheritance ➔ SimpleGasPropertiesBase

Notes ➔ This kernel calculates the thermal conductivity of the gas phase assuming it behaves as an ideal gas and is primarily made up of standard air (i.e., roughly 79 % N_2 and 21 % O_2). This is an empirical formulation based on data for standard air to calculate thermal conductivity as a function of the gas temperature.

(See SimpleGasPropertiesBase for additional notes and calculations)

Computation

Conductivity (energy / time / length / temperature) = $2.66 \times 10^{-4} \cdot T^{0.805}$

T = gas temperature (K)

(see SimpleGasPropertiesBase for other calculation information)

Usage

Recall that there are arguments for this pushed to GlobalParams (See SimpleGasPropertiesBase)

Example below calculates the thermal conductivity of air for variable 'Kg' based on fluid temperature 'Tf'. All common parameters were set in 'GlobalParams' (see SimpleGasPropertiesBase) and parameters unique to this kernel are set here. User sets the 'output' unit arguments to get conductivity back in J/min/cm/K.

```
[./Kg_calc]
        type = SimpleGasThermalConductivity
        variable = Kg

        #NOTE: we skipped the other parameters set here (assuming they
        #       got set in GlobalParams). Most relevant parameters are
        #       being set below.
        pressure = 101.35
        temperature = Tf

        output_energy_unit = "J"
        output_length_unit = "cm"
        output_time_unit = "min"
[../]
```

SolidsVolumeFraction

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel is to be used to calculate the solids volume ratio for the system given the voids volume ratio as an input variable/parameter. This is a specific ratio that you may use in other mass balances to convert units from a solids volume basis to a total volume basis. The units for the solids volume fraction is volume of solids per total system volume.

Computation

Solids volume fraction (volume of solids / total volume) = $(1 - \varepsilon_b)$

ε_b = porosity of the macroscale (i.e., bulk bed porosity)

Usage

Couple with the variable for porosity to formulate an auxiliary variable and corresponding calculation for the solids to total volume ratio. In this example, 'non_pore' is the variable representing the solids volume ratio. The porosity is represented by 'eps_b'.

```
[AuxKernels]
    [./non_pore_calc]
        type = SolidsVolumeFraction
        variable = non_pore
        porosity = eps_b
        execute_on = 'initial   timestep_end'
    [../]
[]
```

SphericalAreaVolumeRatio

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel is to be used to calculate the area to volume ratio for spherical particles. By default, this will return the area of the particles per the volume of the particles, however, user's may request to return the area of the particles per total volume. This can be useful depending on how your mass balances are formulated. Units on output will follow same unit basis on input (i.e., if user provides cm as units for particle diameter, then the area to volume ratio will come out as cm^{-1}).

Computation

Area-volume Ratio of spheres (surface area / volume of spheres) = G_a

If (want area per total volume)

$$G_a = (1 - \varepsilon_b) \cdot 6/d_p$$

If (want area per particle volume) - Default

$$G_a = 6/d_p$$

ε_b = porosity of the macroscale (i.e., bulk bed porosity)

d_p = particle diameter

Usage

Couple with the variable for porosity to formulate an auxiliary variable and corresponding calculation for the ratio of spherical area to volume. In this example, user is requesting to get output in terms of total volume, thus, user must provide porosity and set the 'per_solids_volume' option to 'false'.

```
[AuxKernels]
    [./Ga_calc]
        type = SphericalAreaVolumeRatio
        variable = Ga
        particle_diameter = 1
        porosity = eps_b
        per_solids_volume = false
        execute_on = 'initial   timestep_end'
    [../]
[]
```

TemporalStepFunction

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel can be used to set an auxiliary variable value to follow a step function in time based on a set of user inputs. The purpose of this function is to allow a user to vary inputs at a boundary in a stepwise function for boundaries that allow coupled variables (such as CoupledDirichletBC, CoupledNeumannBC, DGFlowMassFluxBC, etc). The manner in which the value is determined is identical to all the 'Stepwise' BCs discussed here.

Computation

$$\text{Aux value (-)} = \begin{cases} u_0 & t = t_0 \\ \vdots & \vdots \\ u_n & t = t_n \end{cases}$$

u_0 = initial value to set to

t_0 = initial time

t = current simulation time

u_n = value to set to at nth time

t_n = nth time value

Usage

Because this is meant to be coupled with a boundary condition, it should always be executed on the 'initial', 'timestep_begin', and 'nonlinear'.

User's provide an initial starting value, then a set of times and values that will change the result according to the current simulation time. This way, it creates a simulated step function.

Optionally, user's may also provide a set of 'time_spans' that allows for the step changes to be more gradual. The 'time_spans' represent the amount of time it takes to go from one value to the next in the step function.

```
[./step_calc]
    type = TemporalStepFunction
```

```
        variable = step
        start_value = 0
        aux_vals = '1 2 -3'
        'aux_times' = '1 5 10'
        'time_spans' = '0.1 0.25 0.4'
        execute_on = 'initial  timestep_begin nonlinear'
[../]
```

VectorMagnitude

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel is to be used to calculate the magnitude of a vector based on the given vector components.

Computation

Vector Magnitude (length / time) = $\left(u_x^2 + u_y^2 + u_z^2\right)^{1/2}$

u_i = vector in the i^{th} direction (length / time)

Usage

Users provide vector information component-wise for each direction. In this example, we are calculating the magnitude of the velocity vector in the domain.

```
[AuxKernels]
        [./vel_mag_calc]
                type = VectorMagnitude
                variable = vel_mag
                ux = vel_x
                uy = vel_y
                uz = vel_z
                execute_on = 'initial   timestep_end'
        [../]
[]
```

VoidsVolumeFraction

Inheritance ➔ AuxKernel (i.e., the MOOSE base class for the auxiliary system)

Notes ➔ This kernel is to be used to calculate the voids volume ratio for a packed-bed system given the bulk packing density of the bed and an approximate single particle density (calculated from average particle size and mass). This is a specific ratio that is also known as the bulk bed porosity and is used in many other kernels. Typically, you may just give this value in other kernels as a constant, but this auxiliary kernel will estimate it for you and check to see if the calculated value is reasonable for a packed-bed system. Warnings will be output if the calculated value is deemed unreasonable, and by default, it will override 'bad' values with theoretical maximums or minimums for spherical packing. User's are responsible for making sure that the

units in the system work out. If you give bulk packing density in g/cm^3, then you must give particle mass in g and particle size in cm.

Computation

Voids volume fraction (volume of bulk voids / total volume) = $\left(1 - \frac{\rho_b}{\rho_p}\right)$

ρ_b = bulk bed packing density (in mass / total volume)

ρ_p = average particle density (in mass of particle / volume of particle)

$= \frac{6m}{\pi d_p^3}$

m = average mass of a particle

d_p = average diameter of a particle

Theoretical Max and Min

Max voids fraction = 0.66

Min voids fraction = 0.26

(based on mathematical limitations of sphere packing)

Users may override these limits if desired.

Usage

Users provide bulk packing density, particle mass, and particle diameter. Optionally, users may set the 'override_limits' Boolean to allow for the calculation to produce voids fractions that go beyond the mathematical limits.

```
[AuxKernels]
    [./pore_calc]
        type = VoidsVolumeFraction
        variable = pore
        particle_diameter = 1
        particle_mass = 5
        packing_density = 1
        override_limits = false
        execute_on = 'initial   timestep_end'
    [../]
[]
```

Materials

The materials system in MOOSE is generally used as a way to setup parameters in a simulation for each subdomain in the system. In CATS, we accomplish this same functionality through the Auxiliary system, which has the added advantage of allowing us to seamlessly change parameters into variables without significant changes to the code base. However, since CATS uses the built-in MOOSE modules for Navier-Stokes modeling, and the Navier-Stokes modules use materials for parameters such as density and viscosity, we must provide some basic custom materials to interface with those older codes.

INSFluid

Inheritance ➔ Material (i.e., the MOOSE base class for the material system)

Notes ➔ This kernel is to be used specifically in conjunction with the incompressible Navier-Stokes module to provide an interface between our calculated fluid properties from the auxiliary system to the required material properties of density and viscosity. There are no calculations performed by this system. It simply creates the appropriate material property objects and sets their values to those dictated by our auxiliary variables.

Computation

Density (in kg/m^3) is calculated as in GasDensity.

Viscosity (in kg/m/s) is calculated as in GasViscosity.

(**Note**: Optionally, you can simply replace the required variables in the input value with specified constants for simplicity.)

Usage

This object is to be used in conjunction with the built-in Navier-Stokes incompressible flow module in MOOSE. In the example below, the density is being set to another variable in the system named “rho” and the viscosity is being set to a constant value.

Note: All material objects require the user to specify which “blocks” (i.e., subdomains) the material properties apply to. Whenever a material is declared, all blocks in the mesh must invoke a material object regardless of whether or not that subdomain needs that property.

```
[Materials]
        [./ins_material]
                type = INSFluid
                block = 'washcoat channel'
                density = rho
                viscosity = 1.81E-5
         [../]
[]
```

Utilities

Utilities are codes or subroutines that are not actually a part of the MOOSE framework set of kernels. In CATS, they are used for setting up other MOOSE kernels or performing parameter estimations used in MOOSE simulations, generally through the Auxiliary system.

Egret

Purpose ➔ Egret is the set of subroutines that actually perform most of the calculations associated with all the GasPropertiesBase calculations in the Auxiliary system. Those calculations include, but are not limited to:

- GasSpecHeat
- GasSpeciesDiffusion
- GasSpeciesMassTransCoef
- GasViscosity

Many of the other Auxiliary kernels for gas properties also use calculations from Egret as a part of their own calculations and computations.

Error

Purpose ➔ This is a helper file for Egret that is used to provide specific error messages upon failure. User does not need to interface with this subroutine.

Macaw

Purpose ➔ This is a helper file for Egret that is used to provide a templated dense matrix object used for vector calculations and/or data storage of diffusion tensors.