Proto Design Document

Phillip Colella
Daniel T. Graves
Christopher L. Gebhart
Brian Van Straalen

November 21, 2019

1 Introduction

`Proto` is a lightweight library designed for efficient solution of differential equations on domains composed of unions of structured, logically rectangular grids. The goal of `Proto` is to provide a fluent, intuitive interface that separates the complexities of *designing* and *scheduling* an algorithm. As both the tools and applications of high performance computing become continuously more intricate, libraries such as `Proto` will become more and more necessary to efficiently solve problems in a timely manner.

2 Variables, Notation, and Syntax

In general, variables which refer to tokens in `Proto` are written using a `monospaced font`. Vector-like objects are written in lower-case ***bold***, number types use black-board font ($\mathbb{R}$), and sets are written in math calligraphy font ($\mathcal{S}$). Occasionally words are typed in bold-face simply for **emphasis**.

Variable Name	Variable Definition
$\boldsymbol{D}$ or `DIM`	Number of space dimensions
$\mathbb{Z}^{\boldsymbol{D}}$	$\boldsymbol{D}$-dimensional Space of Integers
$\boldsymbol{B}$	Rectangular subset of $\mathbb{Z}^{\boldsymbol{D}}$
Box(V)	Domain of multidimensional rectangular array V
$[\boldsymbol{l}, \boldsymbol{h}]$	Box with low and high corners ($\boldsymbol{l}$, $\boldsymbol{h}$)
$\mathbb{T}$	Space of numbers of type `T`
$\boldsymbol{i}, \boldsymbol{j}, \boldsymbol{k}, \boldsymbol{r}, \boldsymbol{s}, \boldsymbol{t} \ldots$	Points in $\mathbb{Z}^{\boldsymbol{D}}$
$\boldsymbol{e}^d$	Unit vectors in direction d
$\boldsymbol{u}$	Point in $\mathbb{Z}^{\boldsymbol{D}}$ all of whose entries are 1

3 Spatial Discretizations on Rectangular Grids

We approximate solutions to partial differential equations in terms of **data arrays**, i.e. discrete values defined over rectangular regions (Boxes) in $\mathbb{Z}^{\boldsymbol{D}}$.

$$\phi : \boldsymbol{B} \rightarrow \mathcal{T} \text{ i.e., } \phi_{\boldsymbol{i}} \in \mathcal{T} \ , \ \boldsymbol{i} \in \boldsymbol{B} \subset \mathbb{Z}^{\boldsymbol{D}}, \tag{1}$$

$$\mathcal{T} = \mathcal{T}(C_1, \ldots, C_n) = \{t_{c_1,\ldots,c_n} : t \in \mathbb{T}, c_j \in [1, \ldots, C_j]\}, . \tag{2}$$

where $\mathbb{T}$ is one of the real numbers ($\mathbb{R}$), the complex numbers ($\mathbb{C}$), or the integers ($\mathbb{Z}$), and $\boldsymbol{B} = [\boldsymbol{l}, \boldsymbol{h}]$ is a rectangular subset of $\mathbb{Z}^{\boldsymbol{D}}$ with low and high corners $\boldsymbol{l}, \boldsymbol{h}$. Thus ϕ takes on values in a space of multidimensional arrays of $\mathbb{T}$s.While we could flatten this into a single array of dimension $\boldsymbol{D}+n$, we choose not to. This is because the different indices have different mathematical meanings. The Box indices represent a discretization of physical space, with those indices used to define difference approximations to derivatives. The indices in $\mathcal{T}$ represent different components in some state space. A second difference is that the range of indices for the Boxes are computed at run time, whereas we will assume that the range of component indices for any given ϕ are known at compile time. Multidimensional arrays have the usual algebraic operations associated with them:

- Assignment. If ϕ, ψ are data arrays with the same value space $\mathcal{T}$, we define the assignment operator

$$\psi := \phi \textbf{ on } \boldsymbol{B}' \Leftrightarrow \psi_{\boldsymbol{i}} := \phi_{\boldsymbol{i}} \ , \ \boldsymbol{i} \in \boldsymbol{B}'$$
$$\boldsymbol{B}' \subseteq \mathrm{Box}(\phi) \cap \mathrm{Box}(\psi)$$

  If the qualifier **on** ... is omitted, then the assignment is assumed to take place on $\mathrm{Box}(\phi) \cap \mathrm{Box}(\psi)$.

- Addition of two data arrays; multiplication of a data array by a scalar. If ϕ, ψ have the same value spaces, then $\phi+\psi$ is defined on $\mathrm{Box}(\phi) \cap \mathrm{Box}(\psi)$, with $(\psi+\phi)_{\boldsymbol{i}} = \psi_{\boldsymbol{i}}+\phi_{\boldsymbol{i}}$. If $t \in \mathbb{T}$, then $(t\phi)_{\boldsymbol{i}} = t\phi_{\boldsymbol{i}}$.

A broad range of discretized PDE operators can be represented as the composition of **stencil** and **pointwise** operations applied to data arrays.

The simplest example of a stencil operator is given by

$$\psi := S(\phi) \textbf{ on } \boldsymbol{B}' \ , \ S(\phi)_{\boldsymbol{i}} = \sum_{\boldsymbol{s} \in \mathcal{S}} a_{\boldsymbol{s}} \phi_{\boldsymbol{i}+\boldsymbol{s}},$$

where the coefficients $a_{\boldsymbol{s}} \in \mathbb{T}$, $\mathcal{S}$ is a finite subset of $\mathbb{Z}^{\boldsymbol{D}}$, and $\boldsymbol{B}' \subseteq \cap_{\boldsymbol{s} \in \mathcal{S}}(\boldsymbol{B}(\phi) - \boldsymbol{s})$. Such operators appear as finite difference approximations to constant-coefficient differential operators. For multilevel algorithms such as AMR and multigird, we will want more general

versions of stencil operators corresponding to strided access through the data array. A more general form of such a stencil is given by

$$\psi := S(\phi) \textbf{ on } \boldsymbol{B}'$$
$$S(\phi)_{\boldsymbol{r}^d \boldsymbol{i}+\boldsymbol{t}} = \sum_{\boldsymbol{s}\in\mathcal{S}} a_{\boldsymbol{s}} \phi_{\boldsymbol{r}^s \boldsymbol{i}+\boldsymbol{s}} \ , \ \boldsymbol{i} \in \boldsymbol{B}' \tag{3}$$

where $\boldsymbol{ri} = (r_0 i_0, \ldots, r_{\boldsymbol{D}-1} i_{\boldsymbol{D}-1}) \in \mathbb{Z}^{\boldsymbol{D}}$. Note that, if $\boldsymbol{r}^d \neq \boldsymbol{u}$, the application of the stencil assigns values to a strided subset of the data array on the left-hand side.

While the action of stencil on an a data array depend on the inputs, stencils themselves, as specified by the collection of offsets and weights $\{(\boldsymbol{s}, a_{\boldsymbol{s}}) : \boldsymbol{s} \in \mathcal{S}\}$ and the input and output stride and output offset $(\boldsymbol{r}^s, \boldsymbol{r}^d, \boldsymbol{t})$ have a mathematical meaning independent of the inputs. In particular, if $\boldsymbol{r}^s, \boldsymbol{r}^d = \boldsymbol{1}$, and $\boldsymbol{t} = \boldsymbol{0}$, stencils can be added, multiplied by scalars, and composed to obtain new stencils.

$$S = tS_1 + S_2 \Leftrightarrow S(\phi)_{\boldsymbol{i}} = \sum_{\boldsymbol{s}\in\mathcal{S}_1\cup\mathcal{S}_2} (ta^1_{\boldsymbol{s}} + a^2_{\boldsymbol{s}})\phi_{\boldsymbol{i}+\boldsymbol{s}} \tag{4}$$

$$S = S_1 \cdot S_2 \Leftrightarrow S(\phi)_{\boldsymbol{i}} = \sum_{\boldsymbol{s}\in\mathcal{S}}, (\sum_{\boldsymbol{s}_1,\boldsymbol{s}_2:\boldsymbol{s}_1+\boldsymbol{s}_2=\boldsymbol{s}} a^1_{\boldsymbol{s}_1} a^2_{\boldsymbol{s}_2})\phi_{\boldsymbol{i}+\boldsymbol{s}} \ , \ \mathcal{S} = \{\boldsymbol{s}_1 + \boldsymbol{s}_2 : \boldsymbol{s}_1 \in \mathcal{S}_1, \boldsymbol{s}_2 \in \mathcal{S}_2\}. \tag{5}$$

Algebraic operations and composition enables the construction of complex stencils from simpler ones as a symbolic preprocessing step.

Pointwise operators are defined using functions of one or more variables of the form (**??**). Given

$$F : \mathcal{T}_1 \times \cdots \times \mathcal{T}_P \to \mathcal{T}$$

we define $F@$, the pointwise operator derived from F, to be

$$\psi = F@(\phi^1, \ldots, \phi^P) \textbf{ on } \boldsymbol{B}' \Leftrightarrow \psi_{\boldsymbol{i}} = F(\phi^1_{\boldsymbol{i}}, \ldots, \phi^P_{\boldsymbol{i}}) \ , \ \boldsymbol{i} \in \boldsymbol{B}' \tag{6}$$
$$\boldsymbol{B}' \subseteq \boldsymbol{B}(\psi) \cap \boldsymbol{B}(\phi^1) \cap \cdots \cap \boldsymbol{B}(\phi^P)$$

4 `Proto` Classes

There are four C++ classes that implement a representation of spatial discretizations on rectangular grids described above.

- `Point` represents points in $\mathbb{Z}^{\boldsymbol{D}}$.
- `Box` represents rectangular subsets of $\mathbb{Z}^{\boldsymbol{D}}$.

- `BoxData<T,C,D,E>` represents data arrays of the form

$$\phi : \boldsymbol{B} \Rightarrow \mathcal{T} \ , \mathcal{T}(\mathbb{T}, C, D, E).$$

  `BoxData` is a templated class, with class templates specifying the values taken on by the array, and the dimensions of the range space $\mathcal{T}$ given by `C,D,E`.

- `Stencil` is a class that defines stencils as self-contained objects.

We provide a more detailed summary of these classes below. The full documentation is contained in the Doxygen-annotated header files for `Proto`.

4.1 Point

A `Point` is a vector in $\mathbb{Z}^{\boldsymbol{D}}$. `Proto` uses `Points` to define locations in a grid as well as offsets between grid cells. The number of component axes of a `Point` - and all the objects which depend on `Point` - is determined from the compile-time constant `DIM`.

- Construction. `Points` can be constructed from C arrays of length `DIM`, and from integer tuple literals `Point({1,2,1,2,1,2})`. In the latter case, the number of elements in the integer tuple can be greater than `DIM`, with the ones exceeding the length being ignored. There are also static member functions that return a `Point` whose components are all zeros (`Point::Zeros()`), all ones (`Point::Ones()`), and the unit vector in the d^{th} direction (`Point::Basis(d)`, $d = 0, \ldots,$ `DIM` $- 1$.

- Arithmetic Operators. `Points` have addition, componentwise multiplication, multiplication and division by an integer (in the latter case, with rounding toward $-\infty$).

- Logical Operators. `p1 == p2`, `p1 != p2`. The inequality operators e.g. `p1 > p2` return true if they are true for each pair of components, i.e. the usual partial ordering on $\mathbb{Z}^{\boldsymbol{D}}$.

- Indexing Operators. `p[d]`, $d = 0, \ldots, \boldsymbol{D} - 1$. returns the d^{th} component of `p`. `p[d]` can appear as the left-hand side in an assignment.

4.2 Box

A `Box` is an $\boldsymbol{D}$ dimensional rectangle in $\mathbb{Z}^{\boldsymbol{D}}$, defined by a pair of `Points` defining the low and high corners of the Box.

- Constructors. `Box(low,high)`, where `low, high` are `Points`. If `(high >= low) == false`, then the `Box` is an empty `Box`.

- Accessors and Queries `B1 == B2`, `B1 != B2`, `contains(Point a_pt)`, `onBoundary(Point a_pt)`, `empty()` all return `bool`s.

- Operators and Transformations. `B1 & B2` returns the intersection of the two `Box`es. `coarsen`, `refine`, `shift` and other transformation operations compute `Box`es needed to construct `BoxData` targets for various structured-grid algorithms.

- Iterators. `BoxIterator` iterates over the `Point`s in a `Box`. It is accessible using the standard iterator syntax: `bx.begin()` returns a `BoxIterator` for the `Box bx` initialized at the starting location; `it.done()` returns a `bool` telling whether the iterator has reached to end; and `++it` increments the iterator.

4.3 BoxData

A `BoxData<T,C,D,E>` represents a data array defined on a `Box` domain of the form (**??**)

$$U : \boldsymbol{B} \rightarrow \mathcal{T} \ , \ \mathcal{T} = \mathcal{T}(C, D, E),$$

where C, D, and E are positive integers which define the sizes of *component axes* 0, 1, and 2 respectively. These values define the nature of data in a `BoxData` as shown below:

- : $C = D = E = 1 \rightarrow$ Scalar
- : $C > 1, D = E = 1 \rightarrow$ Vector (1st Order Tensor) of length C
- : $C, D > 1, E = 1 \rightarrow C \times D$ Matrix (2nd Order Tensor)
- : $C, D, E > 1 \rightarrow C \times D \times E$ 3rd Order Tensor

`BoxData` has an accompanying class `Interval<C,D,E>` which represents the analog of `Box` in *component space*. `Interval` facilitates copy and slicing operations on `BoxData`s. Operations on `BoxData` include the following:

- Construction.

  `BoxData(const Box& a_bx)`, `define(const Box& a_bx)` define a `BoxData` over the `Box a_bx`. We can also control whether the data is allocated from the heap (`defineMalloc`) or from a specialized stack manager (`defineStack`) that provides a low-overhead way of allocating temporaries; the default is the latter.

  Copy construction and assignment are deleted. However, move construction ahd assignment is supported: `BoxData (BoxData< T, C, D, E > &&a_src)`, `BoxData& operator= (BoxData< T, C, D, E > &&a_src)`. `iota` creates a `BoxData<T,DIM>` from a `Box` and a step size of type `T` that represents a space of position vectors in $\mathbb{T}^{\boldsymbol{D}}$.

- Algebraic Operations. `operator{+,*,-,/}(const BoxData& a_bd)` returns a `BoxData` on the intersection of the `Box`es on which the two inputs are defined. `operator{+=,*=,-=,/=}(const BoxData& a_bd)` computes the same set of values,

but updates the left-hand side in place. We also define `operator(){+=,*=,-=,/=}(T a_t)` to apply the operator to each value in the `BoxData`, and update the left-hand side in place. `setVal(T a_t)` sets all of the values of the `BoxData` to `a_t`. `min`, `max`, and `absMax`: Computes the min, max, or absolute max of a `BoxData`.

- Slicing and Shifting. Operations on `BoxData`s are specific to the type; in particular, they cannot be performed on a `BoxData` with one set of component ranges to produce a `BoxData` with a different set of component ranges. For example,

```
BoxData<T,1,1,1> slice(const BoxData< T,C,D,E> & a_src,
                       unsigned int a_c,
                       unsigned int a_d = 0,
                       unsigned int a_e = 0)
```

  creates a scalar-valued `BoxData` that points to the slice corresponding to component `(a_c,a_d,a_e)`. The resulting `BoxData` can then be used in functions taking scalar-valued `BoxData`s. Similarly, the function

```
BoxData<T,C,D,E> alias(BoxData<T,C,D,E> &a_original,
                       const Point &a_shift)
```

  creates a `BoxData` with the same data pointer, but the `Box` offset by `a_shift`.

Var, Param, and Forall

One of the two types of aggregate operations on `BoxData`s is pointwise application of functions (**??**). This is represented in `Proto` using `forall`. For example, we want to compute $A' = F@A$, where

$$A : \boldsymbol{B_A} \rightarrow \mathcal{T} \ , \ F : \mathcal{T} \rightarrow \mathcal{T}'$$

This is done in `Proto` by defining the function `F` that represents the function F.

```
void
F(Var<T,Cprime>& AprimePoint, const Var<T,C>& APoint>
{
AprimePoint(0) = APoint(1);
AprimePoint(1) = APoint(2) * APoint(3);
...
}
```

This is then applied at all points $p \in \boldsymbol{B}_A$ using `forall`

```
BoxData<T,C,D,E> A = ...;
auto Aprime = forall<T,Cprime>(F,A);
```

Thus the class `Var<T,C,D,E>` is the value type of a `BoxData` , when the latter is viewed as a map from a `Box` to $\mathbb{T}^C \times \mathbb{T}^D \times \mathbb{T}^E$, and the pointwise function `F` is written in terms of `Var`s as input and output. More generally the signature of a `forall` function is given as follows.

```
forall<T,C,D,E>forall(const Func & a_F, Srcs &... a_srcs)
```

where the first argument is the function `F` that is to be applied pointwise, and the following arguments are set of `BoxData`s and scalar parameters. The matching signature for `F` is

```
void F(Var<T,C,D,E>& a_retval,
       Type1& a_src1,
       Type2& a_src2,...,
       TypeN& a_srcN)
{ ... }
```

where the first argument is the output value, and `Typej` is a type corresponding to the j^{th} entry in the `forall src ...` list: if the j^{th} entry is a scalar type, then `Typej = Tj`; if the j^{th} entry is of type `BoxData<Tj,Cj,Dj,Ej>`, then `Typej = Var<Tj,Cj,Dj,Ej>`. In this case, the return `BoxData<T,C,D,E>` from `forall` is defined on the intersection of all the `BoxData.box()` in the argument list. There are also variations of `forall` that enable restriction of the result to an input `Box`, and allows an argument to `F` corresponding to the `Point`s in a `Box`. Thus `forall` is a very powerful abstraction; for example, when combined with `alias` on `BoxData` it can be used to implement stencil operations, including nonlinear stencils, e.g. that occur in limiters for hyperbolic PDE. However, for the case of constant-coefficient stencils, we provide a specialized class `Stencil`, described below.

4.4 Stencil

A `Stencil<T>` is a object used to define and implement the operations described in **??**, with coefficients of type `T`. Typically, `Stencil`s are constructed using a companion class `Shift` which represents the grid shifts associated with a stencil operation. In addition, `Stencil` implements the operations (**??**), (**??**). Stencils are then applied to `BoxData` objects to update existing ones, or produce new ones.

- Construction and Definition. Default-constructed `Stencil`s are empty, containing no offsets or definitions. Stencil has non-default constructors, but they are mostly not used by applications. The most common way to construct a `Stencil` is as a (sum of) `Shift`s multiplied by weights.

  ```
  Point pt = ...; T wgt = ...; Stencil<T> = wgt*Shift(pt);
  ```

In addition, `Stencil`s can be constructed using the operators `+`, `*` that implement the corresponding operations in (**??**), (**??**).

```
Stencil<T> S1= ...; Stencil<T> S2 = ...; T wgt = ...;
auto SVec = S1 + wgt*S2; auto SComp = S1*S2;
```

There are a number of `Stencil`s precomputed and accessible through member functions of the class `Stencil`, including `Stencil<T>::Derivative(...)`, `Stencil<T>::Laplacian(...)`, and various interpolation `Stencil`s useful finite volume applications. We provide support for strided `Stencil`s, as in (**??**), either in the constructor, or by setting $\boldsymbol{r}^s$, $\boldsymbol{r}^d$, or $\boldsymbol{d}$ using the member functions `srcRatio()`, `destRatio()`, `destShift()`, respectively.

- Applying `Stencil`s. `Stencil`s can be applied to a `BoxData` to update the values in an existing `BoxData` .

```
BoxData<T> A1 = ...;BoxData<T> A2 = ...;Stencil<T> L = ...;T wgt = ...;
A1 |= L(A2);
A1 += L(A2,wgt);
```

In the first case, the values in `A1` are overwritten by those given by `L(A2)`. In the second case, the values in `A1` are incremented by those given by `L(A2)*wgt`. In both cases, the operations are performed on the points `pt` $\in$ `A2.box()&L(A2).box()`. `Stencil` application also can be used with assignment to create a new `BoxData` .

```
BoxData<T> A1 = ...;Stencil<T> L = ...;T wgt = ...;
auto A2 = L(A2,wgt);
```

4.5 Interface to `Chombo`

Proto is designed for a shared memory environment. To use Proto in a distributed memory environment, we use the Chombo4 infrastructure. Chombo4 uses Chombos LevelData to distribute data. Proto has data holders for both device-based and host-based data. Currently, communication and I/O are done on the host. Chombo4 provides `LevelBoxData`, a class which wraps `LevelData<BoxData<...> >` and provides functions to move data between the host and the device.

5 Examples

5.1 Point Relaxation

The code example **??** below implements one iteration of point Jacobi for L^h, the standard 5/7 point, second-order accurate discretization of the Laplacian on a grid of mesh spacing h

$$\phi : D \rightarrow \mathbb{R} \ , \ \rho : D_0 \rightarrow \mathbb{R}$$
$$\phi := \phi + \lambda(L^h(\phi) - \rho) \ , \ \lambda = \frac{h^2}{4\boldsymbol{D}}.$$

Listing 1: Point Jacobi Relaxation for Poisson

```
using namespace Proto;

void pointRelax(BoxData<double>& a_phi,
                BoxData<double>& a_rhs,
                double& a_h)
{
  double lambda = a_h*a_h/(4*DIM);
  Stencil<double> Identity = 1.0*Shift::Zeros();
  Stencil<double> laplacian = Stencil::Laplacian();
  BoxData<double> temp = laplacian(a_phi,1.0/(4*DIM));
  temp += (lambda * Identity)(a_rho);
  a_phi += temp;
}
```

The second example is that of a fourth-order accurate evaluation of the right-hand side for a finite-volume discretization of the compressible Euler equations, suitable for use in a method of lines integrator.

$$\langle U \rangle : \boldsymbol{B} \rightarrow \mathbb{R}^{\boldsymbol{D}+2} \ , \ \langle R \rangle : \boldsymbol{B}_0 \rightarrow \mathbb{R}^{\boldsymbol{D}+2} \ , \ \boldsymbol{B} \supseteq grow(\boldsymbol{B}_0, 5)$$
$$\bar{W} = \mathcal{C}(., \gamma)@\langle U \rangle)$$
$$U = \bar{U} - \frac{h^2}{24}\Delta^h(\langle U \rangle)$$
$$W = \mathcal{C}(., \gamma)@(U)$$
$$\langle W \rangle = W + \frac{h^2}{24}\Delta^h(\bar{W})$$
$$\langle R \rangle = 0$$

for $d = 0, \ldots \boldsymbol{D} - 1$:

$$W_L = D^d_L(\langle W \rangle) \ , \ W_R = D^d_R(\langle W \rangle)$$
$$\langle W \rangle_{face} = \mathcal{U}(.,.,\gamma)@(W_L, W_R)$$
$$\bar{F}_{face} = \mathcal{F}(.,d,\gamma)@(\langle W \rangle_{face})$$
$$W_{face} = \langle W \rangle_{face} + \frac{h^2}{24}\Delta^{h,d,\perp}(\langle W \rangle_{face})$$
$$F_{face} = \mathcal{F}(.,d,\gamma)@(W_{face})$$
$$\langle F \rangle_{face} = F_{face} + \frac{h^2}{24}\Delta^{h,d,\perp}(\bar{F}_{face})$$
$$\langle R \rangle \mathtt{+=} \frac{1}{h}(I - (\mathcal{S}^d)^{-1})(\langle F \rangle_{face}).$$

Here, the functions $\mathcal{C}, \mathcal{U}$, and $\mathcal{F}$ are, respectively, the mapping from conservative to primitive variables, an approximate solution to the Riemann problem given left and right state, and the flux in the d direction given a value for the primitive variables. The stencil operators Δ^h, $\Delta^{h,d,\perp}$, and $D^d_{\{L,R\}}$ are, respectively: the classical 5/7 point second-order discrete Laplacian; the second-order Laplacian minus the second differences in the d^{th} direction; and the interpolation of averages over d faces given cell averages using an upwind-biased fifth-order operator.

Listing 2: High-order upwind right-hand side for compressible Euler

```
void EulerOp::operator()(BoxData<Real,NUMCOMPS>& a_R,
                         const BoxData<Real,NUMCOMPS>& a_Uave)
{
  a_Rhs.setVal(0.0);

  Real gamma = s_gamma;

  auto W_bar = forall<Real,NUMCOMPS>(consToPrim,a_Uave,gamma);
  auto U = s_deconvolve(a_U);
  auto W = forall<Real,NUMCOMPS>(consToPrim,U,gamma);
  auto W_ave = s_laplacian(W_bar,1.0/24.0);
  W_ave += W;

  for (int d = 0; d < DIM; d++)
  {
    auto W_ave_low = s_interp_L[d](W_ave);
    auto W_ave_high = s_interp_H[d](W_ave);
    auto W_ave_f = forall<Real,NUMCOMPS>
                   (upwindState,W_ave_low,W_ave_high,d,gamma);

```

```
    auto F_bar_f = forall<Real,NUMCOMPS>(getFlux,W_ave_f,d,gamma);
    auto W_f = s_deconvolve_f[d](W_ave_f);
    auto W_f = W_ave_f;
    auto F_ave_f = forall<Real,NUMCOMPS>(getFlux,W_f,d,gamma);
    auto F_ave_f += s_laplacian_f[d](F_bar_f,1.0/24.0);
    a_Rhs += s_divergence[d](F_ave_f);
  }

  a_Rhs *= -1./s_dx;
}
```

Listing 3: Pointwise functions for Euler

```
typedef   ::Proto::Var<Real,NUMCOMPS> State;
PROTO_KERNEL_START
void
consToPrimF(State&          a_W,
            const State&    a_U,
            Real            a_gamma)
{
  Real rho = a_U(0);
  Real v2 = 0.0;
  Real gamma = a_gamma;
  a_W(0) = rho;
  for (int i = 1; i <= DIM; i++)
  {
    Real v;
    v = a_U(i) / rho;
    a_W(i) = v;
    v2 += v*v;
  }
  a_W(NUMCOMPS-1) =
          (a_U(NUMCOMPS-1) - .5 * rho * v2) * (gamma - 1.0);
}
PROTO_KERNEL_END(consToPrimF, consToPrim)
PROTO_KERNEL_START
void upwindStateF(State& a_out,
                  const State& a_low,
                  const State& a_high,
                  int    a_dir,
                  Real a_gamma)
{...}
PROTO_KERNEL_END(upwindStateF, upwindState)
```

```
31 PROTO_KERNEL_START
32 void getFluxF(State& a_F,
33                const State& a_W,
34                int     a_dir,
35                Real a_gamma)
36 {...}
37 PROTO_KERNEL_END(getFluxF, getFlux)
```