First VPLanet Developers Workshop

Lesson 7
Primary Variables

Overview

Lectures 7-9 address substantive changes to VPLanet

Five steps to adding a Primary Variable

- **Confirm the variables must be a primary variable**
- **Is it multi-module?**
- **Declarations**
- **Initializations**
- **Implementation**

Is it a Primary Variable?

Adding a PV is a lot of work, make sure it's necessary!

A PV is a parameter explicitly evolved by VPLanet

It becomes part of The Matrix; its evolution can affect timestepping

Adding one, adds a new governing equation to VPLanet

In most cases, the new parameter is **not** primary

- output
- auxiliary property
- option

Is the PV Multi-Module?

If the PV is affected by multiple modules, things are more complicated

The initial steps are the same, but multi-module PV's have more steps

For intermediate development, we focus on single-module PVs

Multi-module PVs require updates to module.c
- This lecture should provide the groundwork, then follow examples in the code

Declaring Primary Variables

The first step in adding a PV is to add it to the BODY struct

Let's call our example PV "PrimaryVariable

```
struct BODY {
  ...
  double dPrimaryVariable;  /**< Dummy variable for primary variable tutorial */
  ...
};
```

Nothing fancy here, but we obviously need it!

Declaring Primary Variables

Next we need to add several variable to the UPDATE struct in vplanet.h

This includes variables specific to the module(s) that can affect it

Let's call the single module that modifies our PV "Physics"

We need to add the following variables:

- iPrimaryVariable (an ID for the PV)
- iNumPrimaryVariable (number of modules that affect it)
- dPrimaryVariable (the current value of the PV)
- iPrimaryVariablePhysics (ID for the module affecting the PV)
- double *pdDPrimaryVariableDt (pointer to PV's derivative)

Declaring Primary Variables

Next, we must declare a typedef function for the PV in vplanet.h

typedef void (*fnFinalizeUpdatePrimaryVariableModule)(BODY*, UPDATE*,int*,int,int,int);

This is a bit complicated, but each PV needs a function that finishes off its addition to The Matrix (we'll return to this later)

Next we need to add a matrix of the function class to MODULE:

```
struct MODULE {
  ...
  fnFinalizeUpdatePrimaryVariableModule **fnFinalizeUpdatePrimaryVariable;
  ...
};
```

This line enables the code to call the function later

Declaring Primary Variables

The next step is to #define a number for this PV in vplanet.h:

#define VPRIMARYVARIABLE 2000

This is an ID that will go in update.iPrimaryVariable and/or update.iPrimaryVariablePhysics

VPLanet is very careful about accounting for which derivatives are in each element of The Matrix

Note that these PV ID macros all start with “V” for variable

Declaring Primary Variables

The final declaration step is to write the actual derivative function

```
double fdDPrimaryVariableDt(BODY *body, SYSTEM *system,
        int *iaBody) {
    *commands*

    *return derivative*
}
```

Note that the argument list is required as written

The array of ints is the list of body IDs that are needed to compute the derivative

Initialization

Now we're ready to start allocating memory for the UPDATE struct and assigning values

We begin in update.c and the InitializeUpdate function

This is a big function that contains the following pieces:

- Initialize number of modules affecting a PV to 0
- Initialize PVs that require multiple modules
- Allocate the 2nd dimension of The Matrix
- A loooong list of setting up PVs
- Note this function is nearly 3000 lines long!

For our new PV, the first step is easy!

```
update[iBody].iNumPrimaryVariable = 0;
```

Initialization

After setting all NumPV's to zero, we then count how many modules will affect each PV

```
154
155       /* First we must identify how many variables and models must be
156          assigned so we can malloc the update struct. */
157       for (iModule = 0; iModule < module->iNumModules[iBody]; iModule++) {
158         module->fnInitializeUpdate[iBody][iModule](body, update, iBody);
159       }
160
```

We'll return to these functions later, but for now just assume that iNumPrimaryVariable is set to 1 because InitializeUpdatePhysics set it to 1

After some more memory allocations, we're ready to set up the UPDATE struct for our new PV

Initialization

```
209
210       update[iBody].iVelX = -1;
211       if (update[iBody].iNumVelX) {
212         update[iBody].iVelX          = iVar;
213         update[iBody].iaVar[iVar]    = VVELX;
214         update[iBody].iNumEqns[iVar] = update[iBody].iNumVelX;
215         update[iBody].pdVar[iVar]    = &body[iBody].dVelX;
216         update[iBody].iNumBodies[iVar] =
217               malloc(update[iBody].iNumVelX * sizeof(int));
218         update[iBody].iaBody[iVar] =
219               malloc(update[iBody].iNumVelX * sizeof(int *));
220         update[iBody].iaType[iVar] = malloc(update[iBody].iNumVelX * sizeof(int));
221         update[iBody].iaModule[iVar] =
222               malloc(update[iBody].iNumVelX * sizeof(int));
223
```

The first block of line perform some basic functions

- 210: Assume PV doesn't get used
- But if it does, change its index to iVar (which increments per PV)
- Assign the PV ID to and accounting variable
- Assign number of PV eqns that modify it
- Allocate some memory

Note the pdVar member is a pointer; if you change it, you change the derivative value stored in memory, too!

Initialization

```
223
224         if (control->Evolve.iOneStep == RUNGEKUTTA) {
225           control->Evolve.tmpUpdate[iBody].pdVar[iVar] =
226                 &control->Evolve.tmpBody[iBody].dVelX;
227           control->Evolve.tmpUpdate[iBody].iNumBodies[iVar] =
228                 malloc(update[iBody].iNumVelX * sizeof(int));
229           control->Evolve.tmpUpdate[iBody].daDerivProc[iVar] =
230                 malloc(update[iBody].iNumVelX * sizeof(double));
231           control->Evolve.tmpUpdate[iBody].iaType[iVar] =
232                 malloc(update[iBody].iNumVelX * sizeof(int));
233           control->Evolve.tmpUpdate[iBody].iaModule[iVar] =
234                 malloc(update[iBody].iNumVelX * sizeof(int));
235           control->Evolve.tmpUpdate[iBody].iaBody[iVar] =
236                 malloc(update[iBody].iNumVelX * sizeof(int *));
237         }
238
```

The next block of lines allocates tmpUpdate struct memory

- Necessary for Runge-Kutta to capture midpoint derivatives
- Recall this is necessary to avoid allocating memory each timestep
- Memory allocations are very expensive in C

Initialization

```
238
239         iEqn = 0;
240         for (iModule = 0; iModule < module->iNumModules[iBody]; iModule++) {
241           module->fnFinalizeUpdateVelX[iBody][iModule](body, update, &iEqn, iVar,
242                                                        iBody, iFoo);
243         }
244
245         (*fnUpdate)[iBody][iVar]        = malloc(iEqn * sizeof(fnUpdateVariable));
246         update[iBody].daDerivProc[iVar] = malloc(iEqn * sizeof(double));
247         iVar++;
248       }
249
```

The last set of line perform more initializations

FinalizeUpdate assigns some accounting variables,
and increments iEqn (more later)

After calculating the number of equation, we can then allocate enough memory for The Matrix

Here we encounter a subtlety I glossed over:

There are actually 2 instances of “The Matrix”

fnUpdate is the matrix of function pointers

update.daDerivProc contains the values of those derivativces

Initialization

Now let's look at how to add the PV to the module "Physics"

```
void InitializePrimaryVariablePhysics(BODY *body, OPTIONS *options,
                        UPDATE *update, double dAge, int iBody) {

  // "Type" of update; 1 is an ordinary differential equation
  update[iBody].iaType[update[iBody].iPrimaryVariable]
                [update[iBody].iPrimaryVariablePhysics] = 1;
  // here we assume only one body is affecting the way the variable is updated
  update[iBody].iNumBodies[update[iBody].iPrimaryVariable]
                [update[iBody].iPrimaryVariablePhysics] = 1;
  // allocate memory
  update[iBody].iaBody[update[iBody].iPrimaryVariable]
                [update[iBody].iPrimaryVariablePhysics] = malloc(
                update[iBody].iNumBodies[update[iBody].iPrimaryVariable]
                [update[iBody].iPrimaryVariablePhysics] * sizeof(int));
  // Assign body(ies) that affect this derivative
  update[iBody].iaBody[update[iBody].iPrimaryVariable]
                [update[iBody].iPrimaryVariablePhysics][0] = iBody;
  // Assign pointer the derivative
  update[iBody].pdDPrimaryVariableDtPhysics =
                &update[iBody].daDeriv[update[iBody].iPrimaryVariable]
                [update[iBody].iPrimaryVariablePhysics];
}
```

Type tells VPLanet how to "update" the PV

iNumBodies is the number of bodies that affect the derivative

iaBody is the list of bodies

The pdDPrimary… member is a pointer to this module's derivative

Initialization

Now let's look at how to add the PV to the module "Physics"

```
void AssignPhysicsDerivatives(BODY *body, EVOLVE *evolve, UPDATE *update,
                              fnUpdateVariable ***fnUpdate, int iBody) {
  ...
  fnUpdate[iBody][update[iBody].iPrimaryVariable]
          [update[iBody].iPrimaryVariablePhysics] = &fdDPrimaryVariableDtPhysics;
  ...
}
```

Next, add a line in AssignPhysicsDerivatives that actually assigns the function to The Matrix

This function is called from Verify

Initialization

Now we can add some functions we mentioned earlier

```
void InitializeUpdatePhysics(BODY *body, UPDATE *update, int iBody) {
  ...
  if (iBody > 0) {
    if (update[iBody].iNumPrimaryVariable == 0) {
      update[iBody].iNumVars++;
    }
    update[iBody].iNumPrimaryVariable++;
  }
  ...
}
```

This function increments/sets some accounting variables

If PrimaryVariable was not previously set for this body, add 1 to the number of variable associated with the body

Regardless of that point, we know we must increment the number of equations that modify PrimaryVariable

Initialization

Next we add a NullDerivative function, a utility that might be helpful, so it's best to include while you're setting up the PV

```
void FinalizeUpdatePrimaryVariablePhysics(BODY *body, UPDATE *update,
                                          int *iEqn, int iVar,int iBody,
                                          int iFoo) {
  update[iBody].iaModule[iVar][*iEqn] = PHYSICS;
  update[iBody].iPrimaryVariablePhysics = *iEqn;
  (*iEqn)++;
  }
```

To null, we set the function pointer to return a function that returns a small number (~1e-308)

Setting it to zero would cause dynamical timestepping to break

Finishing Up

The final piece of mandatory code is assign the FinalizeUpdate function in AddModulePhysics

```
void AddModulePhysics(CONTROL *control, MODULE *module, int iBody,
                      int iModule) {
  ...
  module->fnFinalizeUpdatePrimaryVariable[iBody][iModule] =
      &FinalizeUpdatePrimaryVariablePhysics;
  ...
}
```

This sets the function pointer to complete the process of adding the PV

With this, VPLanet can now calculate the evolution of the PV
But you probably still need to add options and outputs to make it work
You may also need to add code to PropsAux, ForceBehavior, InitializeBody, InitializeTmpBody, and InitializeTmpUpdate
The final steps are to add examples and tests to complete the process!