First VPLanet Developers Workshop

Lesson 4
VPLanet Architecture

Overview

Today we begin discussing how you can modify VPLanet for your own research.

For this first lecture, we will cover three topics:

1. The VPLanet Flow Chart

2. VPLanet's Data Structures

3. Modules' models

4. An Introduction to Function Pointers

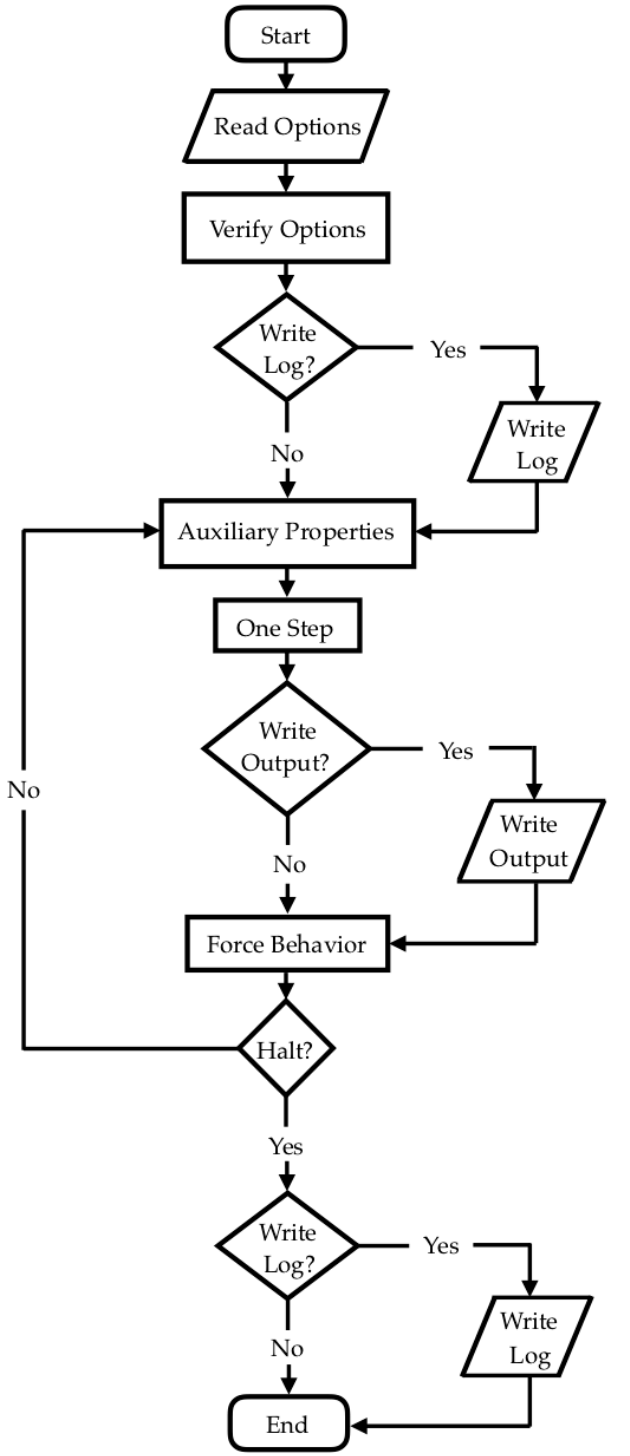
VPLanet Lexicon, Part II

Primary Variable: A parameter that controls the evolution of the system, i.e. a governing variable

Auxiliary Property: A parameter that is convenient to calculate for integrating governing equations

Verify: The process of checking that all input is self-consistent

The Matrix: The three-dimensional matrix of function pointers that advances the primary variables



ReadOptions: read infiles; perform basic checks

VerifyOptions: Ensure input is self-consistent

WriteLog: Log simulation parameters

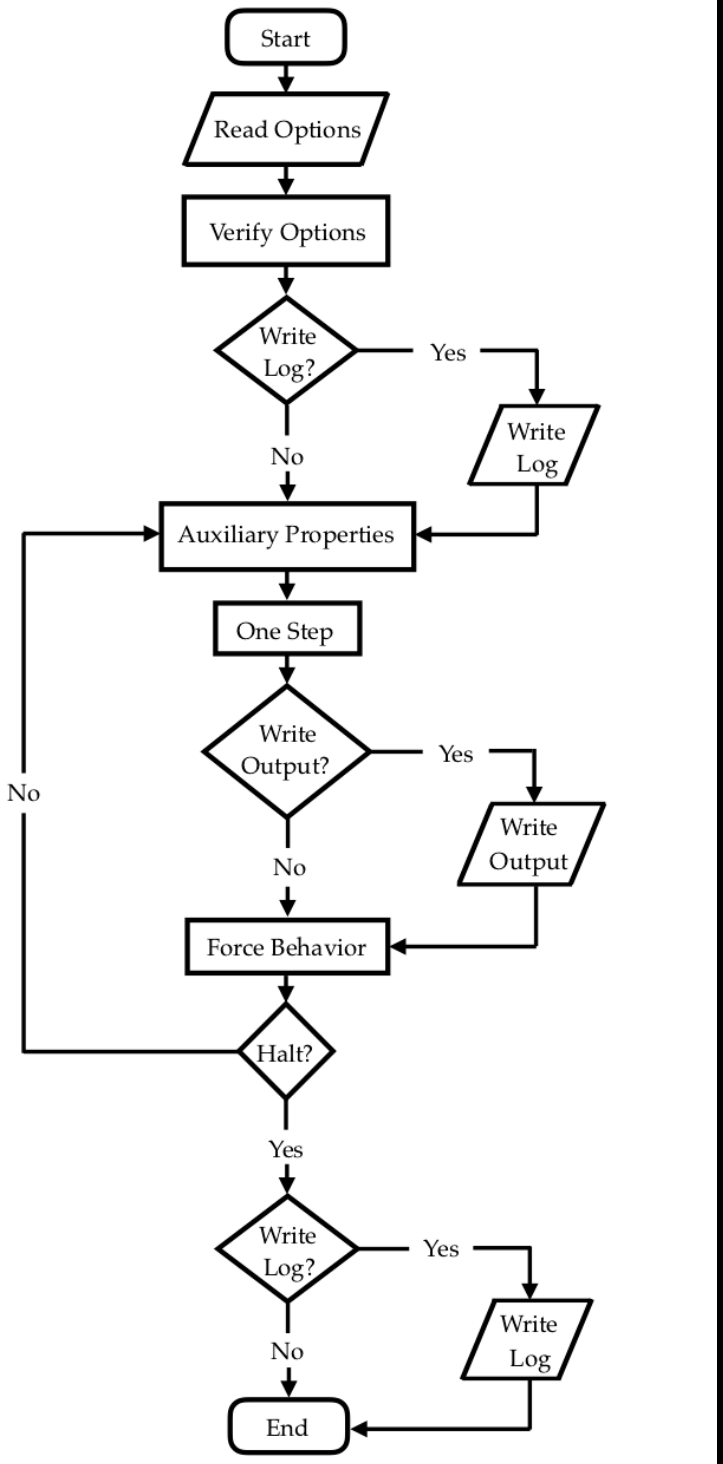
AuxProps: Calculate helper variables

OneStep: Integrate primary variables one step

WriteOutput: Write to forward files

ForceBehavior: Change the integration

Halt: Terminate the execution



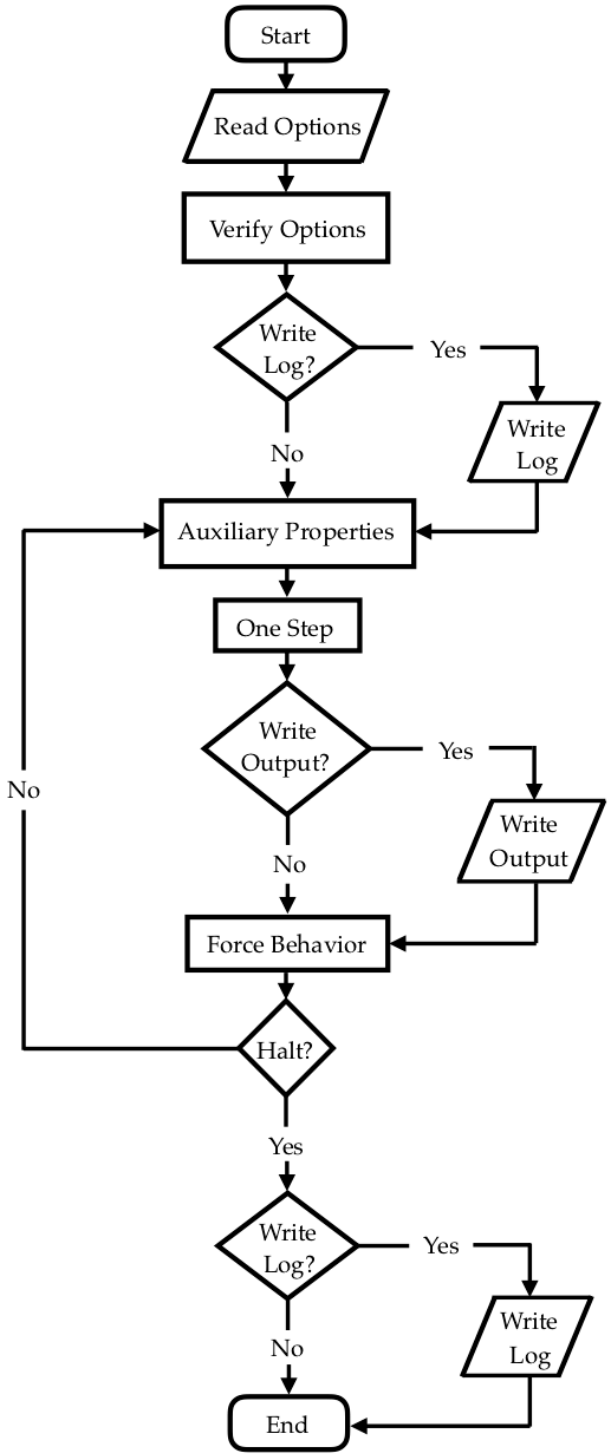
ReadOptions

ReadOption functions search .in files for options and their arguments

They make sure that arguments are the correct cast, e.g. a boolean vs. string array

They can make definitional checks on the argument, e.g. eccentricity $\prec 0$

If option omitted, they assign the default value



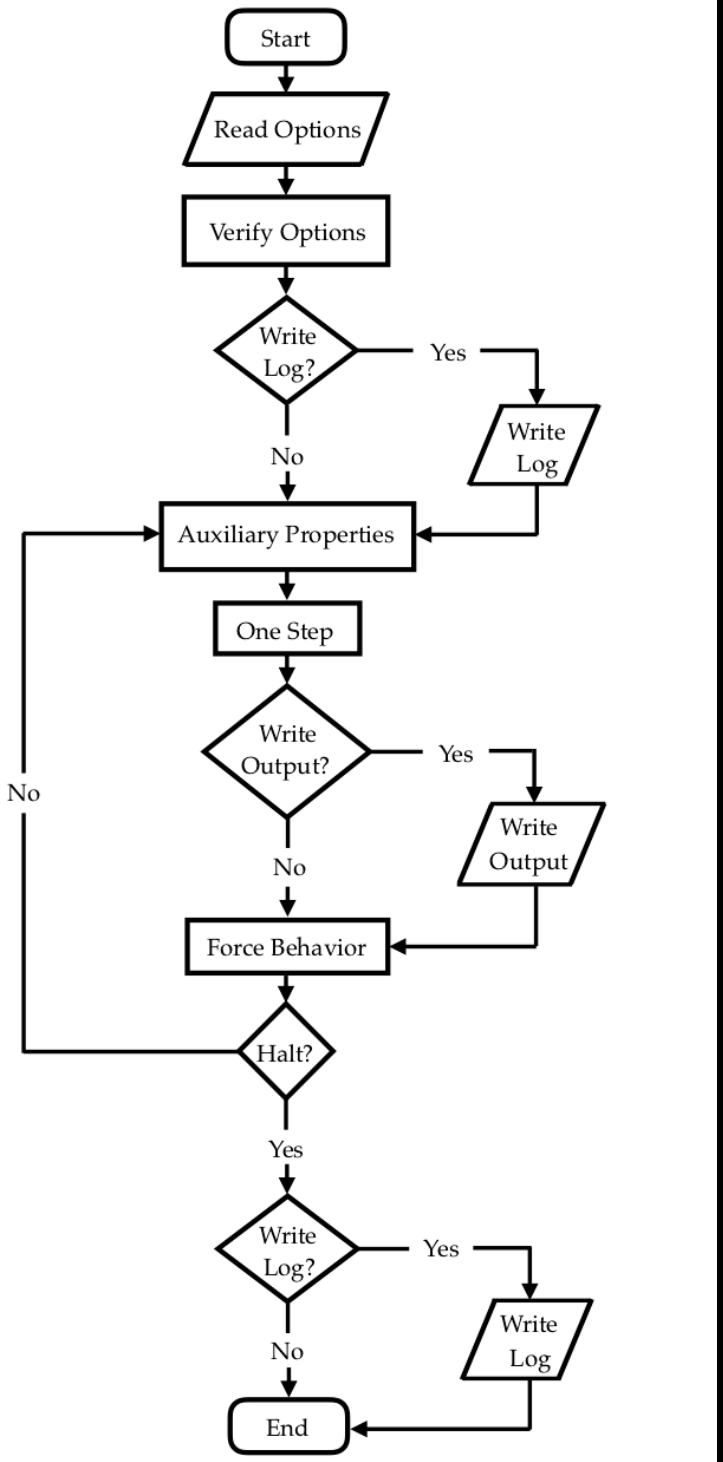
VerifyOptions

Verify checks that the *combination* of options is self-consistent

E.g. semi-major axis and orbital period are not both set

At the end of Verify, the simulation is ready to run

Verify is one of the hardest parts of VPLanet because you have to decide and code up what is legal and what is not



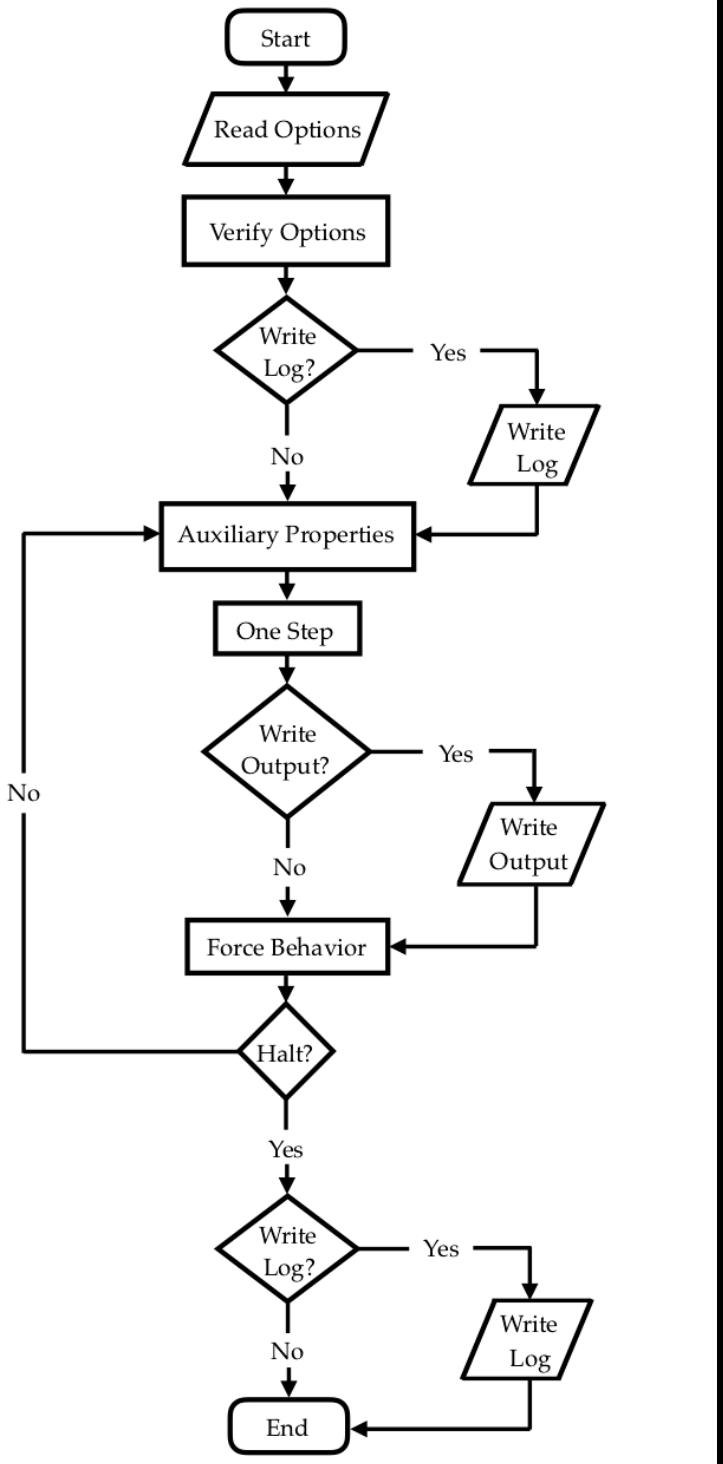
AuxProps

AuxProps is the first step in the integration loop

For code readability (and our own sanity) it is often best to define variables prior to integrating equations

Auxiliary properties can be general or module-specific

Add them sparingly, but there are no real limits to their use

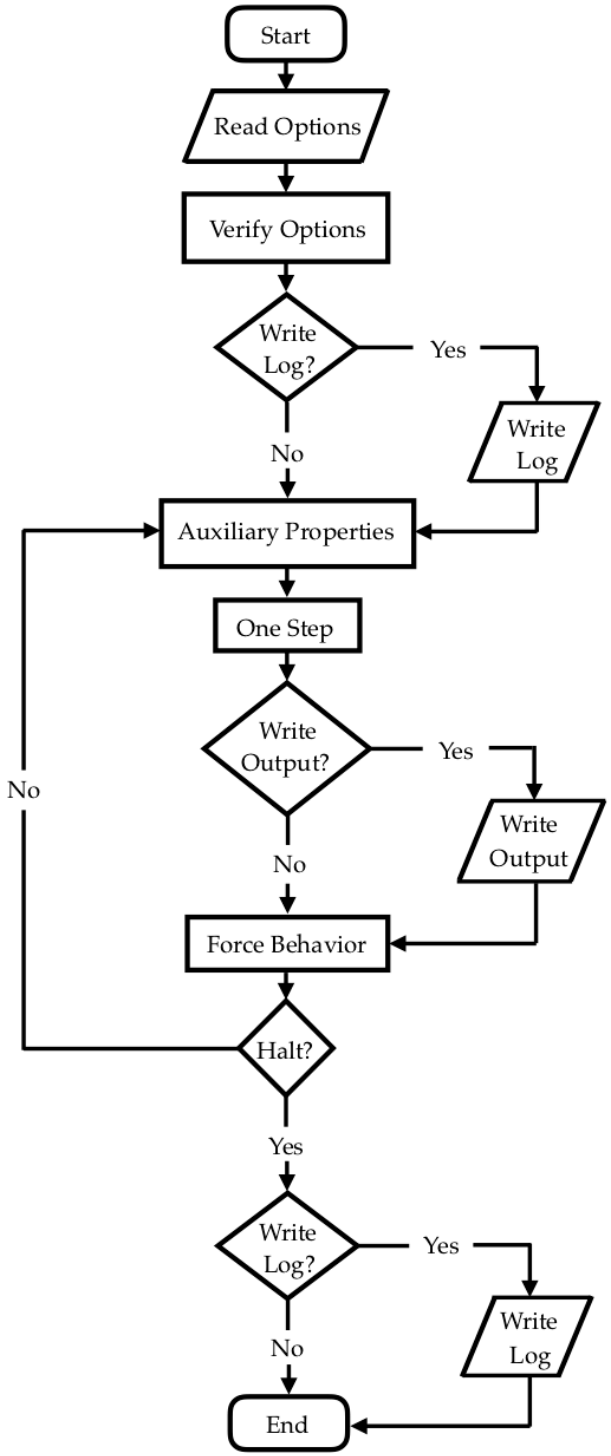


OneStep

This function evolves the system forward one whole step

Only two options are available: Euler and 4th order Runge-Kutta (don't use Euler!)

This functionality is in evolve.c. Please don't modify that file without checking with me first!

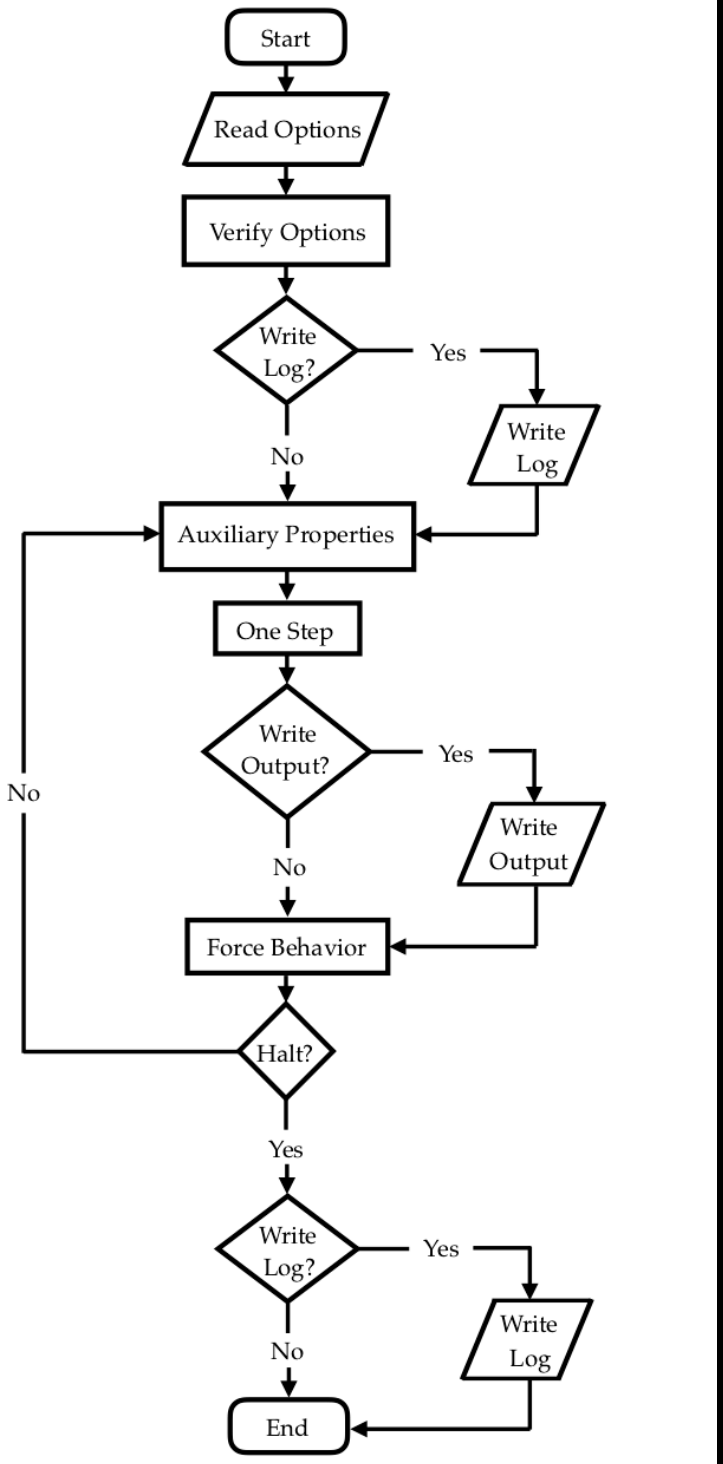


ForceBehavior

This step changes the integration according to user input

For example, if a planet tidally locks, then VPLanet should no longer calculate the rotational evolution

In principle, nearly any change is permitted, so this function can become complicated



Halt

Halts are any condition for which the user decided the code's exeuction should terminate

This includes reaching dStopTime

Some halts occur by default, e.g. mergers, but others require the user to set

Checking for halts is the final stage of the integration loop

VPLanet Data Stuctures

CONTROL: how the code runs

BODY: the orbital and physical properties of each object

UPDATE: how primary variables are integrated

SYSTEM: multi-body properties

MODULE: how modules interact

OPTIONS: options

OUTPUT: outputs

FILES: variable related to input and output files

fnUpdate: the function pointer matrix

CONTROL

Control contains all the variables for performing integrations, I/O, and units

It contains 4 substructs:

HALT: All the functions and values for halting code execution

IO: Parameters related to how data are written

EVOLVE: The variables for integrating the system

UNITS: All the units for each body/file

BODY

The Body struct is massive, and contains all the properties associated with an individual body

It is initialized as an array, with length equal to the # of bodies

It contains the parameters for all modules; there are no substructs

As you start developing, get ready to write body[iBody] a lot!

UPDATE

The Update struct contains a lot of accounting variables to keep track of the matrix

It is also an array with a length equal to the number of bodies

It includes pointers to derivatives

You'll probably only need to modify Update if you start adding primary variables or modules

fnUpdate

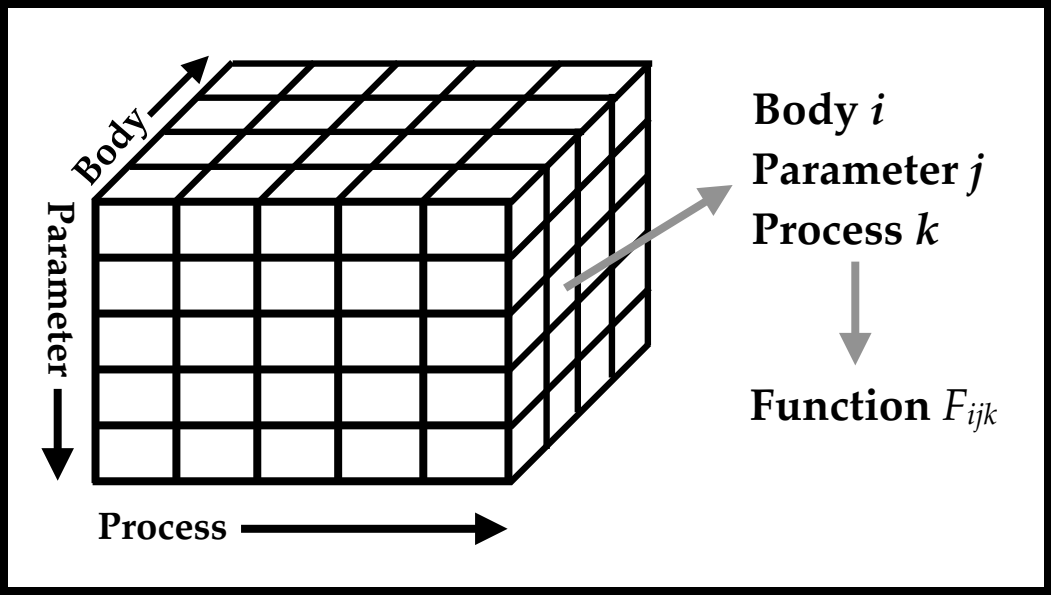
This variable is a three-dimensional array of function pointers

The first dimension corresponds to bodies

The second to the primary variables of the bodies

And the third to the processes that affect those variables

The size of the dimensions is allocated at runtime based on saModules



"Module Models"

Often it's convenient to employ different models that represent the same phenomenon

While one could design an entire new module, it can be easier to just change pieces of how an existing module works

These are models, and they have organically become part of VPLanet

But they are very powerful! You can compare how changing models affects the integration, i.e. model comparison

"VPLanet level": ReadOptions, Evolve, Halt, etc.
"Module level": AtmEsc, MagmOc, FLARE
"Model level": energy-limited, Roche lobe overflow (AtmEsc)

“Module Models”

Without rules, things can become messy

The STELLAR module, for example, includes models for XUV evolution and magnetic braking

It could include the a model for flaring, but that is now its own module

Orbital evolution is similar: DistOrb contains models for 2nd order and 4th order secular evolution, but SpiNBody is its own module

Use your best judgement as you develop, and you can always reach out!

Function Pointers in C

Function pointers are the backbone of VPLanet. They enable the dynamical assembly of modules at runtime

You may have seen *scalar* function pointers in code before, but vectors and matrices are pretty rare

So here's a brief introduction…

Function Pointers in C

Consider the following program:

```
#include <stdio.h>

typedef double (*fnptr)(double,double);

double foo(double a, double b) {

  return a*a + b;
}

int main() {
  fnptr fn;
  double x,y,z;

  x=4;
  y=0.1;

  fn = &foo;

  z = fn(x,y);

  printf("%lf\n",z);
  return 0;
}
```

We define a new cast for the function pointer

Function foo has the same arguments as our new fnptr cast — it can be called by it

Here we define variable fn as an fnptr cast

Here we set fn to the *address* of function foo

Now we call the variable fn, with arguments

This code will print 16.100000

Function Pointers in C

Now let's make it an array:

```
#include <stdio.h>
#include <stdlib.h>

typedef double (*fnptr)(double,double);

double foo(double a, double b) {

  return a*a + b;
}

double bar(double a, double b) {

  return a + b*b;
}
```

Here we create 2 functions that will be assigned to 2 elements in the array

```
int main() {
  fnptr *fn;
  double x,y,z;

  x=4;
  y=0.1;

  fn = malloc(2*sizeof(fnptr));

  fn[0] = &foo;
  fn[1] = &bar;

  z = fn[0](x,y);
  printf("%lf\n",z);

  z = fn[1](x,y);
  printf("%lf\n",z);

  return 0;
}
```

This code will print:

16.100000

4.010000

Function Pointers in C

As you can imagine, we can use that formalism to make an *N*-dimensional function pointer matrix

While this functionality is extremely powerful, it comes with danger!

C doesn't care if you write past the end of an array
- You can destroy values in memory!

If you misassign a function, it can be very difficult to debug!
- valgrind helps, but it can be very tedious to track down bugs

Now let's take a quick tour of vplanet.h