MdvDeriveModel - Engineering Notes

This application uses C++11 features. It depends on the Icing Library.

Assumptions

In general if something goes wrong, we log an error and exit. We don't try to continue, and it's the responsibility of the user to fix the problem and run again.

Icing Library

We made a design decision to make the calculators in icing library as dumb as possible. The application that calls them is generally responsible for finding the correct input data (e.g. which vertical levels), getting it into a useful structure (maybe a vector?), and checking for bad/missing inputs.

By default the icing library uses -9999 for the missing/bad data (although it generally tries to not need to use this at all). If you want a different bad/missing value, pass it in to the constructor when you create the Calculators object.

Adding a new output field

MdvDeriveModel uses TDRP. Edit the paramdef to add new configuration options. Any new inputs or outputs that must be supported are added to the data_field_t structure.

ModelData::loadData()

Add a case for the new output data where it calls your new _load*() method.

ModelData::_load*()

Each output field, needs it's own load method. Each load method starts with a #define macro and some other boilerplate:

```
bool ModelData::_loadSpecificHumidity()
{
  RUN_ONCE(Params::SPECH_FIELD)
  bool ret = true;
  _printLoading("specific humidity");
  if(_isFieldInInput(_inputNames[Params::SPECH_FIELD])) {
    _readField(Params::SPECH_FIELD);
  }
  else {
```

RUN_ONCE ensures each load method is not called more than once. This is necessary to prevent circular dependencies from creating infinite loops:

```
#define RUN_ONCE(field) if(_attemptedLoads.find(field) != _attemptedLoads.end()) return true; _attemptedLoads.
insert(field);
```

The rest of the boiler plate logs the load attempt, and tries the trivial case of the desired output already existing in the input.

NOTE: We would like to move all of this boilerplate into a function or macro or something eventually.

After the trivial situation is attempted, you add your specific calculators.

For example here is a field which can be derived from two different possible input fields, so we check for the existence of each before calling the relevant calculator:

```
 else if(_isFieldInInput(_inputNames[Params::ACPCP_2HR_FIELD])) {
    _readField(Params::ACPCP_2HR_FIELD);
    _calcACPCP(2);
  }
  else if(_isFieldInInput(_inputNames[Params::ACPCP_3HR_FIELD])) {
    _readField(Params::ACPCP_3HR_FIELD);
    _calcACPCP(3);
  }
```

Here is another example, where a derived field depends on other derived fields:

```
else if( _loadPressure() &&
    _loadTemperature() &&
    _loadRelativeHumidity() )
{
  _calcWetBulbTemp();
}
```

You can see in this example that _calcWetBulbTemp() depends on having pressure, temperature and RH already loaded, so they go into the IF test, before the calculator is called. If they have already successfully been loaded or calculated, then the load*() method calls are a quick return due to the RUN_ONCE macro.

If there were multiple ways to calculate wet bulb temperature, than _calcWetBulbTemp() should be renamed to something that more specifically describes what type of derivation is being done.

ModelData::_calc*()

Each

Adding a new input field

MdvDeriveModel uses TDRP. Edit the paramdef to add new configuration options. Any new inputs or outputs that must be supported are added to the data_field_t structure.

ToDo

- Do we use topo for anything? If not remove it from paramdef.
- Do we actually support REALTIME/ARCHIVE/FILELIST? If not remove from paramdef. If so add mention to user page.