Portable Python-wrapped FV3GFS atmospheric model

Vulcan Climate Modeling, Seattle, WA

Geophysical Fluid Dynamics Laboratory, Princeton, NJ

Jeremy McGibbon, Noah Brenowitz, Mark Cheeseman, **Spencer Clark**, Johann Dahm, Eddie Davis, Oliver Elbert, Rhea George, Brian M Henn, Anna Kwa, Andre Perkins, Oliver Watt-Meyer, Tobias Wicky, Christopher S. Bretherton, and Oliver Fuhrer

Agenda

- Review what the Python-wrapped FV3GFS model is and where to find documentation
- Illustrate how to use the wrapper through a sequence of examples and exercises.
    - Running the base version of the model.
    - Getting the values of fortran variables from within Python.
    - Setting the values of fortran variables from within Python.
    - Saving the model state and restarting the model.

Now we can run the model

We'll start by importing the `wrapper` and running the `initialize` method. This allocates the necessary memory for the model, and loads in the intial conditions.

[9]:
```
from fv3gfs import wrapper
wrapper.initialize()
```

Then to run the model for the amount of time specified in the namelist, we can write our own main loop in Python. The wrapper contains a convenient method for computing the number of timesteps required to run for the length of time in the namelist, `wrapper.get_step_count` (in this case four steps).

[10]:
```
if comm.rank == 0:
    print(wrapper.get_step_count())
```

```
[stdout:0] 4
```

The most basic main loop requires just three steps:

- `step_dynamics`
- `step_physics`
- `save_intermediate_restart_if_enabled`

These three steps, executed in this order within the main loop, ensure bit-for-bit reproducibility with the pure fortran model.

[11]:
```
for i in range(wrapper.get_step_count()):
    wrapper.step_dynamics()
    wrapper.step_physics()
    wrapper.save_intermediate_restart_if_enabled()
```

Finally, when the loop is finished, we can call `wrapper.cleanup`.

[12]:
```
wrapper.cleanup()
```

What is the Python-wrapped FV3GFS model?

- It is an alternative way of running and modifying the behavior of NOAA/EMC's FV3GFS model.
- It makes it easy to insert code into the time loop of a simulation in such a way that:
  - The model does not need to be recompiled
  - Provides access to an extensive ecosystem of pre-written, well-maintained tools for scientific computing
  - Through docker, makes it easy to reproduce on a variety of platforms (cloud, HPC, your laptop, etc.)

Useful links

GitHub repositories:

- fv3gfs-wrapper: https://github.com/VulcanClimateModeling/fv3gfs-wrapper
- fv3gfs-fortran: https://github.com/VulcanClimateModeling/fv3gfs-fortran
- fv3gfs-util: https://github.com/VulcanClimateModeling/fv3gfs-util

Documentation:

- fv3gfs-wrapper: https://vulcanclimatemodeling.github.io/fv3gfs-wrapper/index.html
- fv3gfs-util: https://fv3gfs-util.readthedocs.io/en/latest/index.html

Workshop materials: https://github.com/VulcanClimateModeling/fv3gfs-wrapper-workshop

Lesson 1: running the base version of the model

Reproducing the fortran model results from within Python

- From a valid run directory within a Docker container, running the seven lines on the right will bit-for-bit reproduce the results of the fortran model.

[11]:
```
from fv3gfs import wrapper

wrapper.initialize()
for i in range(wrapper.get_step_count()):
    wrapper.step_dynamics()
    wrapper.step_physics()
    wrapper.save_intermediate_restart_if_enabled()
wrapper.cleanup()
```

What exactly is the wrapper wrapping?

- The main loop is essentially breaking some code in coupler_main.F90 into separate pieces.

```
call fms_init()
call mpp_init()
initClock = mpp_clock_id( 'Initialization' )
call mpp_clock_begin (initClock) !nesting problem

call fms_init
call constants_init
call sat_vapor_pres_init

call coupler_init
call print_memuse_stats('after coupler init')

call mpp_set_current_pelist()
call mpp_clock_end (initClock) !end initialization
mainClock = mpp_clock_id( 'Main loop' )
termClock = mpp_clock_id( 'Termination' )
call mpp_clock_begin(mainClock) !begin main loop

do nc = 1, num_cpld_calls
   Time_atmos = Time_atmos + Time_step_atmos
   call update_atmos_model_dynamics (Atm)
   call update_atmos_radiation_physics (Atm)
   call update_atmos_model_state (Atm)

   if (intrm_rst) then
     if ((nc /= num_cpld_calls) .and. (Time_atmos == Time_restart)) then
       timestamp = date_to_string (Time_restart)
       call atmos_model_restart(Atm, timestamp)
       call coupler_res(timestamp)
       Time_restart = Time_restart + Time_step_restart
     endif
   endif
   call print_memuse_stats('after full step')
enddo
```

What exactly is the wrapper wrapping?

- The main loop is essentially breaking some code in coupler_main.F90 into separate pieces.
  - initialize() corresponds to the code before the do loop

```
call fms_init()
call mpp_init()
initClock = mpp_clock_id( 'Initialization' )
call mpp_clock_begin (initClock) !nesting problem

call fms_init
call constants_init
call sat_vapor_pres_init

call coupler_init
call print_memuse_stats('after coupler init')

call mpp_set_current_pelist()
call mpp_clock_end (initClock) !end initialization
mainClock = mpp_clock_id( 'Main loop' )
termClock = mpp_clock_id( 'Termination' )
call mpp_clock_begin(mainClock) !begin main loop
```

```
do nc = 1, num_cpld_calls
   Time_atmos = Time_atmos + Time_step_atmos
   call update_atmos_model_dynamics (Atm)
   call update_atmos_radiation_physics (Atm)
   call update_atmos_model_state (Atm)

   if (intrm_rst) then
     if ((nc /= num_cpld_calls) .and. (Time_atmos == Time_restart)) then
       timestamp = date_to_string (Time_restart)
       call atmos_model_restart(Atm, timestamp)
       call coupler_res(timestamp)
       Time_restart = Time_restart + Time_step_restart
     endif
   endif
   call print_memuse_stats('after full step')
enddo
```

What exactly is the wrapper wrapping?

- The main loop is essentially breaking some code in coupler_main.F90 into separate pieces.
  - initialize() corresponds to the code before the do loop
  - step_dynamics() corresponds to stepping the time forward and calling update_atmos_model_dynamics

```
call fms_init()
call mpp_init()
initClock = mpp_clock_id( 'Initialization' )
call mpp_clock_begin (initClock) !nesting problem

call fms_init
call constants_init
call sat_vapor_pres_init

call coupler_init
call print_memuse_stats('after coupler init')

call mpp_set_current_pelist()
call mpp_clock_end (initClock) !end initialization
mainClock = mpp_clock_id( 'Main loop' )
termClock = mpp_clock_id( 'Termination' )
call mpp_clock_begin(mainClock) !begin main loop

do nc = 1, num_cpld_calls
   Time_atmos = Time_atmos + Time_step_atmos
   call update_atmos_model_dynamics (Atm)
   call update_atmos_radiation_physics (Atm)
   call update_atmos_model_state (Atm)

   if (intrm_rst) then
     if ((nc /= num_cpld_calls) .and. (Time_atmos == Time_restart)) then
       timestamp = date_to_string (Time_restart)
       call atmos_model_restart(Atm, timestamp)
       call coupler_res(timestamp)
       Time_restart = Time_restart + Time_step_restart
     endif
   endif
   call print_memuse_stats('after full step')
enddo
```

What exactly is the wrapper wrapping?

- The main loop is essentially breaking some code in coupler_main.F90 into separate pieces.
  - initialize() corresponds to the code before the do loop
  - step_dynamics() corresponds to stepping the time forward and calling update_atmos_model_dynamics
  - step_physics() corresponds to calling update_atmos_radiation_physics and update_atmos_model_state

```
call fms_init()
call mpp_init()
initClock = mpp_clock_id( 'Initialization' )
call mpp_clock_begin (initClock) !nesting problem

call fms_init
call constants_init
call sat_vapor_pres_init

call coupler_init
call print_memuse_stats('after coupler init')

call mpp_set_current_pelist()
call mpp_clock_end (initClock) !end initialization
mainClock = mpp_clock_id( 'Main loop' )
termClock = mpp_clock_id( 'Termination' )
call mpp_clock_begin(mainClock) !begin main loop

do nc = 1, num_cpld_calls
   Time_atmos = Time_atmos + Time_step_atmos
   call update_atmos_model_dynamics (Atm)
   call update_atmos_radiation_physics (Atm)
   call update_atmos_model_state (Atm)

   if (intrm_rst) then
     if ((nc /= num_cpld_calls) .and. (Time_atmos == Time_restart)) then
       timestamp = date_to_string (Time_restart)
       call atmos_model_restart(Atm, timestamp)
       call coupler_res(timestamp)
       Time_restart = Time_restart + Time_step_restart
     endif
   endif
   call print_memuse_stats('after full step')
enddo
```

What exactly is the wrapper wrapping?

- The main loop is essentially breaking some code in coupler_main.F90 into separate pieces.
  - save_intermediate_restart_if_enabled() corresponds to the restart writing code at the end of the do loop

```
do nc = 1, num_cpld_calls
   Time_atmos = Time_atmos + Time_step_atmos
   call update_atmos_model_dynamics (Atm)
   call update_atmos_radiation_physics (Atm)
   call update_atmos_model_state (Atm)

   if (intrm_rst) then
     if ((nc /= num_cpld_calls) .and. (Time_atmos == Time_restart)) then
       timestamp = date_to_string (Time_restart)
       call atmos_model_restart(Atm, timestamp)
       call coupler_res(timestamp)
       Time_restart = Time_restart + Time_step_restart
     endif
   endif
   call print_memuse_stats('after full step')
enddo

#ifdef AVEC_TIMERS
call avec_timers_output
#endif
call mpp_set_current_pelist()
call mpp_clock_end(mainClock)
call mpp_clock_begin(termClock)

call coupler_end
call mpp_set_current_pelist()
call mpp_clock_end(termClock)

call fms_end
```

What exactly is the wrapper wrapping?

- The main loop is essentially breaking some code in coupler_main.F90 into separate pieces.
  - save_intermediate_restart_if_enabled() corresponds to the restart writing code at the end of the do loop
  - cleanup() corresponds to the code after the main loop

```
do nc = 1, num_cpld_calls
   Time_atmos = Time_atmos + Time_step_atmos
   call update_atmos_model_dynamics (Atm)
   call update_atmos_radiation_physics (Atm)
   call update_atmos_model_state (Atm)

   if (intrm_rst) then
     if ((nc /= num_cpld_calls) .and. (Time_atmos == Time_restart)) then
       timestamp = date_to_string (Time_restart)
       call atmos_model_restart(Atm, timestamp)
       call coupler_res(timestamp)
       Time_restart = Time_restart + Time_step_restart
     endif
   endif
   call print_memuse_stats('after full step')
enddo
```

```
#ifdef AVEC_TIMERS
call avec_timers_output
#endif
call mpp_set_current_pelist()
call mpp_clock_end(mainClock)
call mpp_clock_begin(termClock)

call coupler_end
call mpp_set_current_pelist()
call mpp_clock_end(termClock)

call fms_end
```

First notebook session

Lesson 2: getting the state of fortran variables from within Python

fv3gfs.wrapper.get_state

- Getting the state of the fortran model from within Python is useful for a variety of reasons:
  - The state of fortran variables often forms the basis of inputs for a machine learning model (or other kind of scheme that modifies the model).
  - Computing and/or saving diagnostics.
  - Making plots.

```
[89]: state = wrapper.get_state(["surface_pressure"])
      state
```

```
Out[0:84]:
{'surface_pressure': Quantity(
     data=
 [[1.20009249e-315 6.94174408e-310 1.20009249e-315 ... 1.14970578e-315
   9.88131292e-324 7.90505033e-323]
  [3.95252517e-323 6.94166886e-310 3.64126381e-321 ... 0.00000000e+000
   0.00000000e+000 0.00000000e+000]
  [0.00000000e+000 0.00000000e+000 0.00000000e+000 ... 1.14984182e-315
   1.14984135e-315 1.00000000e+000]
  ...
  [3.20000000e+002 3.13750000e+002 2.20000000e+002 ... 3.20000000e+002
   3.18828125e+002 2.20000000e+002]
  [3.19218750e+002 3.20000000e+002 3.19218750e+002 ... 4.32630757e-317
   6.94166970e-310 4.32852098e-317]
  [6.94166647e-310 6.94174388e-310 4.28711433e-317 ... 1.13342857e-315
   1.14805631e-315 7.90505033e-323]],
     dims=('y', 'x'),
     units=Pa,
     origin=(3, 3),
     extent=(48, 48)
 )}
```

fv3gfs.wrapper.get_state

- get_state is typically called with a sequence of variable names as an argument; these names must have "getters" defined for them in the wrapper.
  - Available fields in the dynamics can be found in dynamics_properties.json
  - Available fields in the physics can be found in physics_properties.json
  - Tracers are handled specially since they are dynamically set based on the model configuration through the field_table; tracer metadata can be found using fv3gfs.wrapper.get_tracer_metadata()

```
[89]: state = wrapper.get_state(["surface_pressure"])
      state
```

```
Out[0:84]:
{'surface_pressure': Quantity(
     data=
 [[1.20009249e-315 6.94174408e-310 1.20009249e-315 ... 1.14970578e-315
   9.88131292e-324 7.90505033e-323]
  [3.95252517e-323 6.94166886e-310 3.64126381e-321 ... 0.00000000e+000
   0.00000000e+000 0.00000000e+000]
  [0.00000000e+000 0.00000000e+000 0.00000000e+000 ... 1.14984182e-315
   1.14984135e-315 1.00000000e+000]
  ...
  [3.20000000e+002 3.13750000e+002 2.20000000e+002 ... 3.20000000e+002
   3.18828125e+002 2.20000000e+002]
  [3.19218750e+002 3.20000000e+002 3.19218750e+002 ... 4.32630757e-317
   6.94166970e-310 4.32852098e-317]
  [6.94166647e-310 6.94174388e-310 4.28711433e-317 ... 1.13342857e-315
   1.14805631e-315 7.90505033e-323]],
     dims=('y', 'x'),
     units=Pa,
     origin=(3, 3),
     extent=(48, 48)
 )}
```

fv3gfs.wrapper.get_state

- get_state is typically called with a sequence of variable names as an argument; these names must have "getters" defined for them in the wrapper.
    - It returns a dictionary mapping variable names to Quantity objects.

```
[89]: state = wrapper.get_state(["surface_pressure"])
      state
```

```
Out[0:84]:
{'surface_pressure': Quantity(
     data=
 [[1.20009249e-315 6.94174408e-310 1.20009249e-315 ... 1.14970578e-315
   9.88131292e-324 7.90505033e-323]
  [3.95252517e-323 6.94166886e-310 3.64126381e-321 ... 0.00000000e+000
   0.00000000e+000 0.00000000e+000]
  [0.00000000e+000 0.00000000e+000 0.00000000e+000 ... 1.14984182e-315
   1.14984135e-315 1.00000000e+000]
  ...
  [3.20000000e+002 3.13750000e+002 2.20000000e+002 ... 3.20000000e+002
   3.18828125e+002 2.20000000e+002]
  [3.19218750e+002 3.20000000e+002 3.19218750e+002 ... 4.32630757e-317
   6.94166970e-310 4.32852098e-317]
  [6.94166647e-310 6.94174388e-310 4.28711433e-317 ... 1.13342857e-315
   1.14805631e-315 7.90505033e-323]],
     dims=('y', 'x'),
     units=Pa,
     origin=(3, 3),
     extent=(48, 48)
 )}
```

fv3gfs.util.Quantity

- The Quantity object is a container for an array representing a portion of the model state for a variable.
    - In addition to the underlying data, it contains metadata about the variable's physical units, dimensions, shape, and halo points.

```
[89]: state = wrapper.get_state(["surface_pressure"])
      state
```

```
Out[0:84]:
{'surface_pressure': Quantity(
     data=
 [[1.20009249e-315 6.94174408e-310 1.20009249e-315 ... 1.14970578e-315
   9.88131292e-324 7.90505033e-323]
  [3.95252517e-323 6.94166886e-310 3.64126381e-321 ... 0.00000000e+000
   0.00000000e+000 0.00000000e+000]
  [0.00000000e+000 0.00000000e+000 0.00000000e+000 ... 1.14984182e-315
   1.14984135e-315 1.00000000e+000]
  ...
  [3.20000000e+002 3.13750000e+002 2.20000000e+002 ... 3.20000000e+002
   3.18828125e+002 2.20000000e+002]
  [3.19218750e+002 3.20000000e+002 3.19218750e+002 ... 4.32630757e-317
   6.94166970e-310 4.32852098e-317]
  [6.94166647e-310 6.94174388e-310 4.28711433e-317 ... 1.13342857e-315
   1.14805631e-315 7.90505033e-323]],
     dims=('y', 'x'),
     units=Pa,
     origin=(3, 3),
     extent=(48, 48)
 )}
```

fv3gfs.util.Quantity

- The Quantity object is a container for an array representing a portion of the model state for a variable.
  - In addition to the underlying data, it contains metadata about the variable's physical units, dimensions, shape, and halo points.
  - To access raw data, including halo points, you can use the Quantity.data attribute.

```
[11]: state["surface_pressure"].data

Out[0:6]:
array([[ 6.89883304e-310,  6.89883304e-310,  1.02004785e-315, ...,
        -6.26012713e-001, -6.37340568e-001, -6.48273040e-001],
       [-6.58791944e-001, -6.68878933e-001, -6.78515582e-001, ...,
        -6.60330499e-001, -6.68908498e-001, -6.76994882e-001],
       [-6.84570862e-001, -6.91618164e-001, -6.98119185e-001, ...,
        -6.75842822e-001, -6.81145395e-001, -6.85863419e-001],
       ...,
       [ 2.96439388e-323,  5.39530594e-316,  0.00000000e+000, ...,
         0.00000000e+000,  0.00000000e+000,  0.00000000e+000],
       [ 0.00000000e+000,  0.00000000e+000,  0.00000000e+000, ...,
         4.94065646e-324,  1.23516411e-322,  1.30237697e-314],
       [ 1.97626258e-323,  1.01726409e-315,  0.00000000e+000, ...,
         0.00000000e+000,  0.00000000e+000,  0.00000000e+000]])
```

fv3gfs.util.Quantity

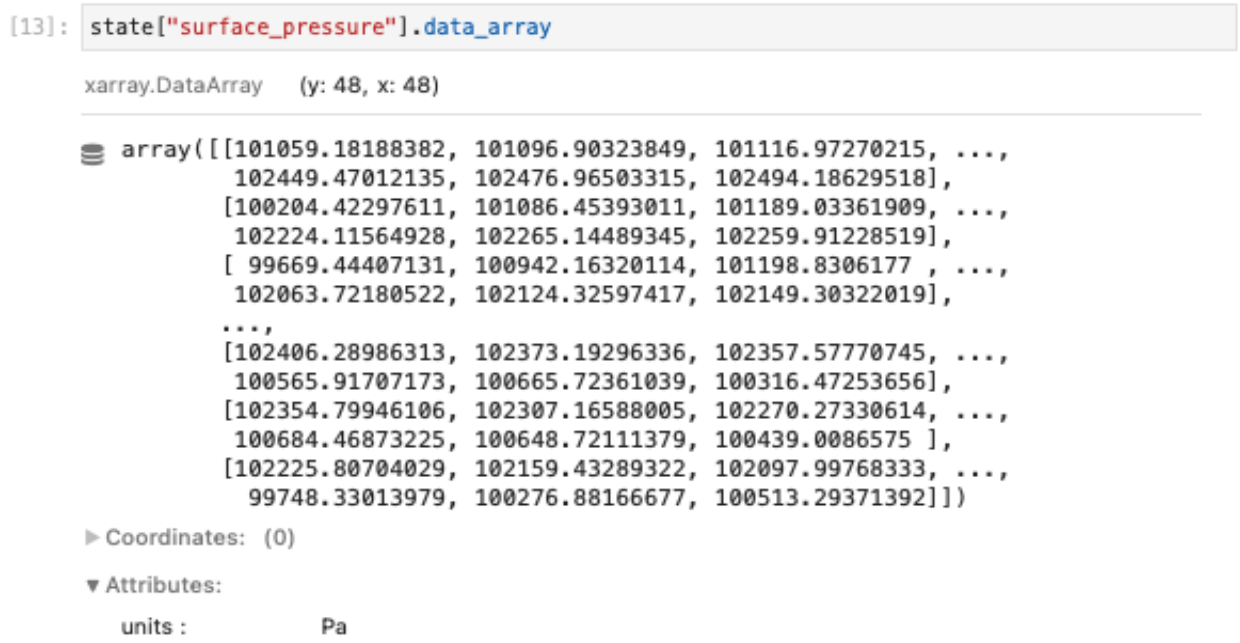
- The Quantity object is a container for an array representing a portion of the model state for a variable.
  - In addition to the underlying data, it contains metadata about the variable's physical units, dimensions, shape, and halo points.
  - To access raw data, including halo points, you can use the Quantity.data attribute.
  - To access data only on the compute domain of for the particular rank, use Quantity.view[:].

```
[12]: state["surface_pressure"].view[:]

Out[0:7]:
array([[101059.18188382, 101096.90323849, 101116.97270215, ...,
        102449.47012135, 102476.96503315, 102494.18629518],
       [100204.42297611, 101086.45393011, 101189.03361909, ...,
        102224.11564928, 102265.14489345, 102259.91228519],
       [ 99669.44407131, 100942.16320114, 101198.8306177 , ...,
        102063.72180522, 102124.32597417, 102149.30322019],
       ...,
       [102406.28986313, 102373.19296336, 102357.57770745, ...,
        100565.91707173, 100665.72361039, 100316.47253656],
       [102354.79946106, 102307.16588005, 102270.27330614, ...,
        100684.46873225, 100648.72111379, 100439.0086575 ],
       [102225.80704029, 102159.43289322, 102097.99768333, ...,
         99748.33013979, 100276.88166677, 100513.29371392]])
```

fv3gfs.util.Quantity

- The Quantity object is a container for an array representing a portion of the model state for a variable.
    - For those familiar with xarray, Quantity objects contain a convenience attribute for converting to a DataArray.



fv3gfs.util.Quantity

- The Quantity object is a container for an array representing a portion of the model state for a variable.
  - For those familiar with xarray, Quantity objects contain a convenience attribute for converting to a DataArray.
  - It is also straightforward to convert a DataArray back to a Quantity object through fv3gfs.util.Quantity.from_data_array.
  - Quantity objects currently do not support array arithmetic with each other, so it is often important to switch between NumPy, Xarray, and Quantity representations.

```
[15]: fv3gfs.util.Quantity.from_data_array(state["surface_pressure"].data_array)

Out[0:10]:
Quantity(
    data=
[[101059.18188382 101096.90323849 101116.97270215 ... 102449.47012135
  102476.96503315 102494.18629518]
 [100204.42297611 101086.45393011 101189.03361909 ... 102224.11564928
  102265.14489345 102259.91228519]
 [ 99669.44407131 100942.16320114 101198.8306177  ... 102063.72180522
  102124.32597417 102149.30322019]
 ...
 [102406.28986313 102373.19296336 102357.57770745 ... 100565.91707173
  100665.72361039 100316.47253656]
 [102354.79946106 102307.16588005 102270.27330614 ... 100684.46873225
  100648.72111379 100439.0086575 ]
 [102225.80704029 102159.43289322 102097.99768333 ...  99748.33013979
  100276.88166677 100513.29371392]],
    dims=('y', 'x'),
    units=Pa,
    origin=(0, 0),
    extent=(48, 48)
)
```

Second notebook session

Lesson 3: setting the state of fortran variables from within Python

fv3gfs.wrapper.set_state

- Being able to set the state of variables in the fortran model from Python is crucial for being able to modify the behavior of the model.
  - Any variable that you can "get" is a variable you can "set."
  - Setting the state can be done by calling fv3gfs.wrapper.set_state with a "state" dictionary as an argument (this is a dictionary mapping variable names to Quantity objects).

```
[19]: wrapper.set_state(state)
```

fv3gfs.wrapper.set_state_mass_conserving

- As you will see in the notebook session, setting the values of water tracer fields can be dangerous, because it also can change the mass of the atmosphere (a key prognostic variable in the non-hydrostatic version of the model).
    - set_state_mass_conserving adds safety by automatically adjusting the mass of the atmosphere accordingly when any water tracer field is updated.

```
[19]: wrapper.set_state_mass_conserving(state)
```

Third notebook session

Lesson 4: checkpointing the state and restarting the model

Writing and reading state

- Being able to checkpoint the full state of the model and restart the model later on is an important feature.
    - To checkpoint the full state of the model at a given time and save it out to disk, one can use fv3gfs.util.write_state.
    - To load in the state again, one can use fv3gfs.util.read_state.

[2]:
```
restart_state = wrapper.get_state(wrapper.get_restart_names())
fv3gfs.util.write_state(restart_state, filename)
```

[3]:
```
current_state = fv3gfs.util.read_state(filename)
```

Fourth notebook session

Questions / Progress