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

Questions / Progress

fv3gfs.util

https://fv3gfs-util.readthedocs.io/

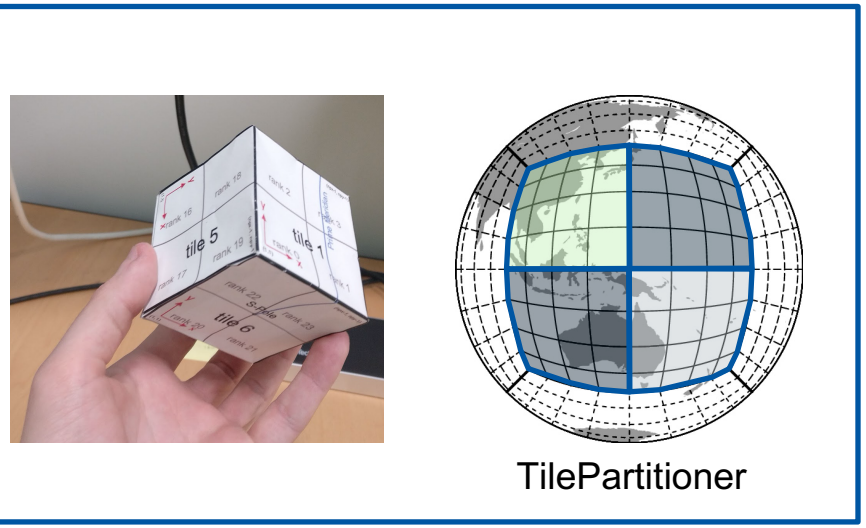
What is fv3gfs-util?

- Used by fv3gfs-wrapper and by new gt4py port of FV3
- Must support both “model” code like dynamics, and high-level analysis code
- “toolkit” of objects and utilities, as generic as possible


```
cube.halo_update(quantity, n_points=3)
```

“Partitioner”

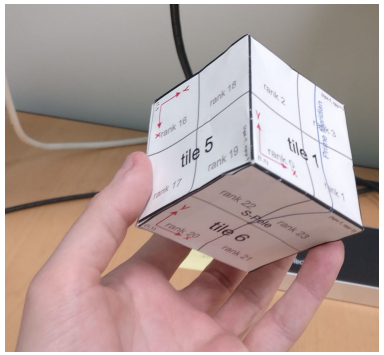
- TilePartitioner and CubedSpherePartitioner
- Initialized from a layout (e.g. 2 by 2)
- Responsible for defining how the domain is partitioned between processes
- partitioner = fv3gfs.util.CubedSpherePartitioner(
  fv3gfs.util.TilePartitioner((2, 2))
)

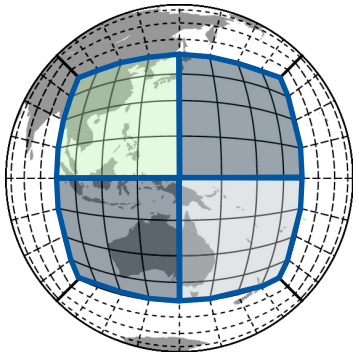


CubedSpherePartitioner

“Communicator”

- TileCommunicator and CubedSphereCommunicator
- Initialized from a partitioner and an mpi4py ”comm” object
- Responsible for MPI communication on a tile/cube
- Use help() and dir() to see methods, most important are scatter/gather, scatter_state/gather_state
  e.g. help(fv3gfs.util.CubedSphereCommunicator)



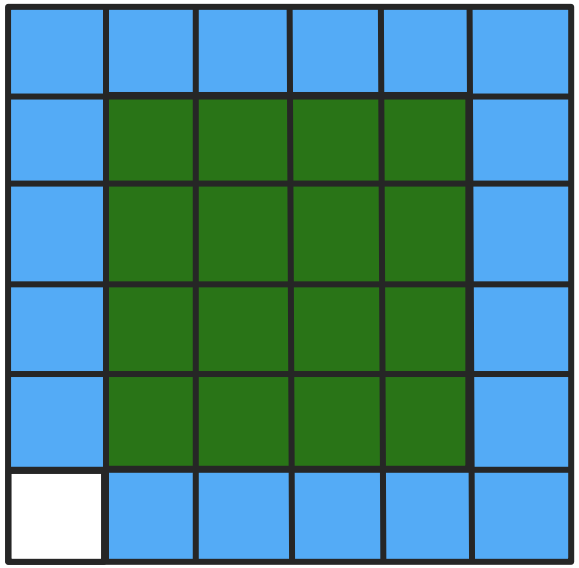


TilePartitioner

CubedSpherePartitioner

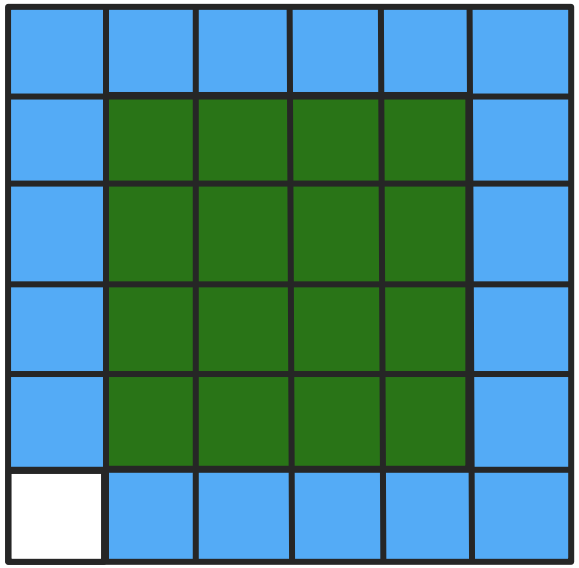
Quantity

- numpy/cupy array wrapped with metadata
  - origin
  - extent
  - dims
  - units
- quantity.data is the numpy array, contains halos
- quantity.view[:] gives a view of the compute domain
  - quantity.view is a custom object, *not* an array



Utilities

- fv3gfs.util.read_state/fv3gfs.util.write_state
  - Read or write processor's state to a netCDF file
- fv3gfs.wrapper.open_restart
  - Read state from a Fortran restart directory
  - Thin layer over fv3gfs.util.open_restart which passes tracer metadata from running model
- fv3gfs.util.ZarrMonitor
  - Will save a series of states to a Zarr store



Docker

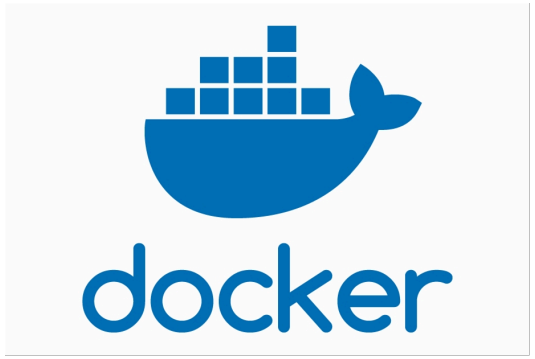
The Dockerfile

- Instructions to build an environment
- Base on an image with FROM
- Add sequential commands to set up the environment
- Not a complete definition
    - You build the environment at a specific point in your filesystem, used for COPY commands

```
FROM python:3.7.8-stretch

# install gcloud
RUN apt-get update && apt-get install -y  apt-transport-https ca-certificates gnupg curl gettext

RUN echo "deb [signed-by=/usr/share/keyrings/cloud.google.gpg] https://packages.cloud.google.com/
apt cloud-sdk main" | tee -a /etc/apt/sources.list.d/google-cloud-sdk.list &&\
    curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key --keyring /usr/share/
    keyrings/cloud.google.gpg add -
RUN apt-get update && apt-get install -y google-cloud-sdk

COPY constraints.txt /tmp/constraints.txt
COPY external/fv3fit/requirements.txt fv3fit/requirements.txt
RUN pip install -c /tmp/constraints.txt -r fv3fit/requirements.txt

COPY external/vcm/ /external/vcm/
COPY external/loaders/ /external/loaders/
COPY external/synth /external/synth
COPY external/fv3fit/ /fv3fit/

RUN apt-get update && apt-get install -y gfortran

COPY docker/fv3fit/entrypoint.sh /entrypoint.sh
RUN chmod +x /entrypoint.sh && \
    /entrypoint.sh

WORKDIR /fv3fit
ENTRYPOINT [ "/entrypoint.sh" ]
```

The Docker Image

- Fully defines an environment
- Built from a Dockerfile
  - `docker build –f Dockerfile –t image_name .`
- Can be:
  - used as a FROM target in Dockerfiles
  - pushed and pulled from the internet ("docker push" and "docker pull")
  - Used to start a Docker Container



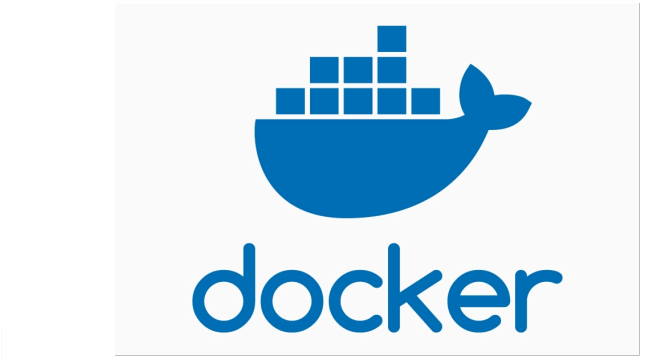
The Docker Container

- A running environment
- Started from a Docker image
  - `docker run image_name <command>`
  - `docker run –it image_name bash`
- Used to run code in a defined environment
- NOT a virtual machine – much less expensive to run


```
(base) mcgibbon ~ $ docker run ubuntu:18.04 echo "hello"
hello
```

Bind Mounts

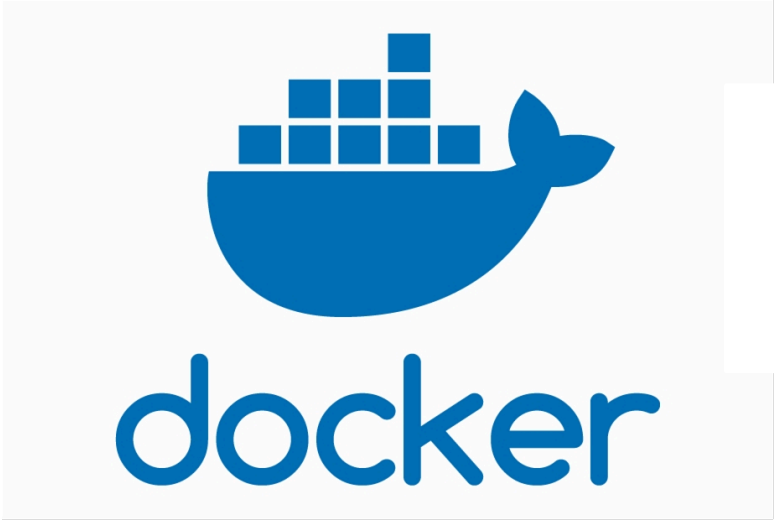
- https://docs.docker.com/storage/bind-mounts/
- docker run
  -v <absolute host path>:<absolute container path>
- Shares files, such as:
  - Input data/run directories
  - Output directories
  - Code you want to edit while the container runs




```
(base) mcgibbon ~ $ docker run ubuntu:18.04 echo "hello"
hello
```

Ask the internet!

- Lots of Docker users
- Great documentation for Docker
- Most questions are answered by a Google search
- If not, lots of people waiting to answer your question on Stack Overflow
- Or send me an e-mail jeremym@vulcan.com

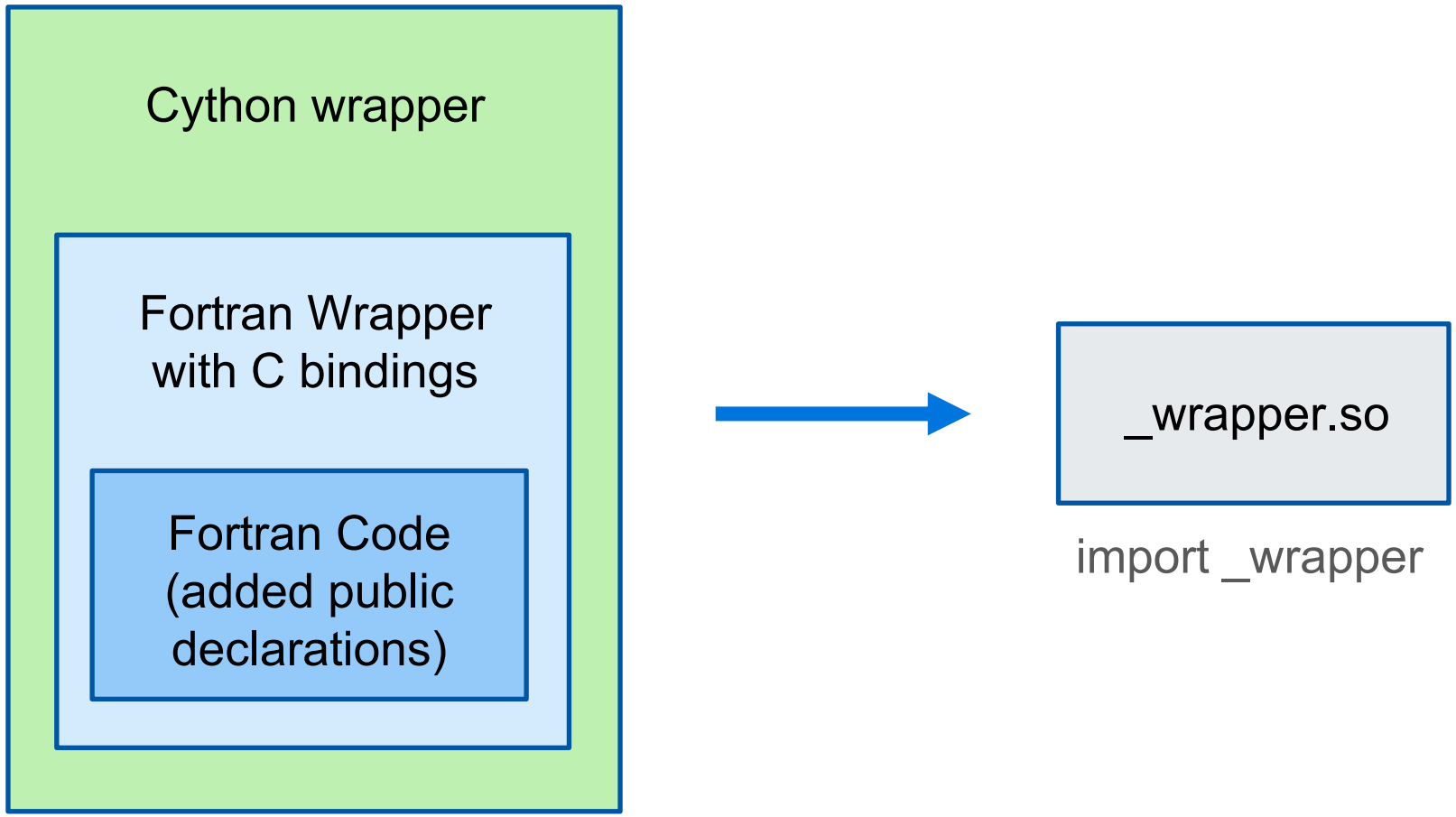


Wrapper Implementation

Using Cython

- https://cython.readthedocs.io/en/latest/
- Write a .pyx file (cython code)
- This gets converted to .c by Cython, and compiled to a .so file
    - Write a setup.py file saying how it needs to be compiled, including any external linked libraries
- Can import that .so file as though it's a Python file or package

Compiled library structure



Cython Wrapping

- cdef extern to define function signature of bind(c) Fortran routines
  - No arrays, instead use a pointer to the first element of the array
  - Does mean no safety checking on array sizes – look out for segmentation faults!
- get_u is a Fortran routine
- array_3d is a numpy array
- array_3d[0, 0, 0] is the first array element
- “&” provides a pointer to that element.

_wrapper.pyx

```
from mpi4py import MPI
ctypedef cnp.double_t REAL_t
ctypedef cnp.int_t INT_t
ctypedef cnp.npy_bool BOOL_t
real_type = np.float64
SURFACE_PRECIPITATION_RATE = 'surface_precipitation_rate'
MM_PER_M = 1000


cdef extern:
    void get_diagnostic_3d(int*, double *)
    void get_diagnostic_2d(int*, double *)
    void get_metadata_diagnostics(int* , int *, char*, char*, char*, char*)
    void get_diagnostics_count(int *)
    void initialize_subroutine(int *comm)
    void do_step_subroutine()
    void cleanup_subroutine()
    void do_dynamics()
    void compute_physics_subroutine()
    void apply_physics_subroutine()
    void save_intermediate_restart_if_enabled_subroutine()
    void save_intermediate_restart_subroutine()
    void initialize_time_subroutine(int *year, int *month, int *day, int *hour, int *minute, int *second)
    void get_time_subroutine(int *year, int *month, int *day, int *hour, int *minute, int *second, int *fms_calendar_type)
    void get_physics_timestep_subroutine(int *physics_timestep)
    void get_centered_grid_dimensions(int *nx, int *ny, int *nz)
    void get_n_ghost_cells_subroutine(int *n_ghost)
    void get_u(REAL_t *u_out)
    void set_u(REAL_t *u_in)
```

```
if 'x_wind' in input_names_set:
    quantity = _get_quantity(state, "x_wind", allocator, ['z', 'y_interface', 'x'], "m/s", dtype=real_type)
    with fv3gfs.util.recv_buffer(quantity.np.empty, quantity.view[:]) as array_3d:
        get_u(&array_3d[0, 0, 0])
```

Cython Wrapping

- use iso_c_binding to provide C interfaces for Fortran routines
- Declare output arrays as c types, like c_double
- Text is more complicated, see get_tracer_name in dynamics_data.F90 for an example

dynamics_data.F90

```
use atmosphere_mod, only: Atm, mytile
use tracer_manager_mod, only: get_tracer_names, get_number_tracers, get_tracer_index
use field_manager_mod,  only: MODEL_ATMOS
use iso_c_binding
```

```
pure function i_start() result(i)
    integer :: i
    i = Atm(mytile)%bd%is
end function i_start
```

```
subroutine set_u(u_in) bind(c)
    real(c_double), intent(in), dimension(i_start():i_end(), j_start():j_end()+1, 1:nz()) :: u_in
    Atm(mytile)%u(i_start():i_end(), j_start():j_end()+1, 1:nz()) = u_in(i_start():i_end(), j_start():j_end()+1, 1:nz())
end subroutine set_u

subroutine get_u(u_out) bind(c)
    real(c_double), intent(out), dimension(i_start():i_end(), j_start():j_end()+1, 1:nz()) :: u_out
    u_out(i_start():i_end(), j_start():j_end()+1, 1:nz()) = Atm(mytile)%u(i_start():i_end(), j_start():j_end()+1, 1:nz())
end subroutine get_u
```

Cython Wrapping

- use iso_c_binding to provide C interfaces for Fortran routines
- Declare output arrays as c types, like c_double
- Text is more complicated, see get_tracer_name in dynamics_data.F90 for an example

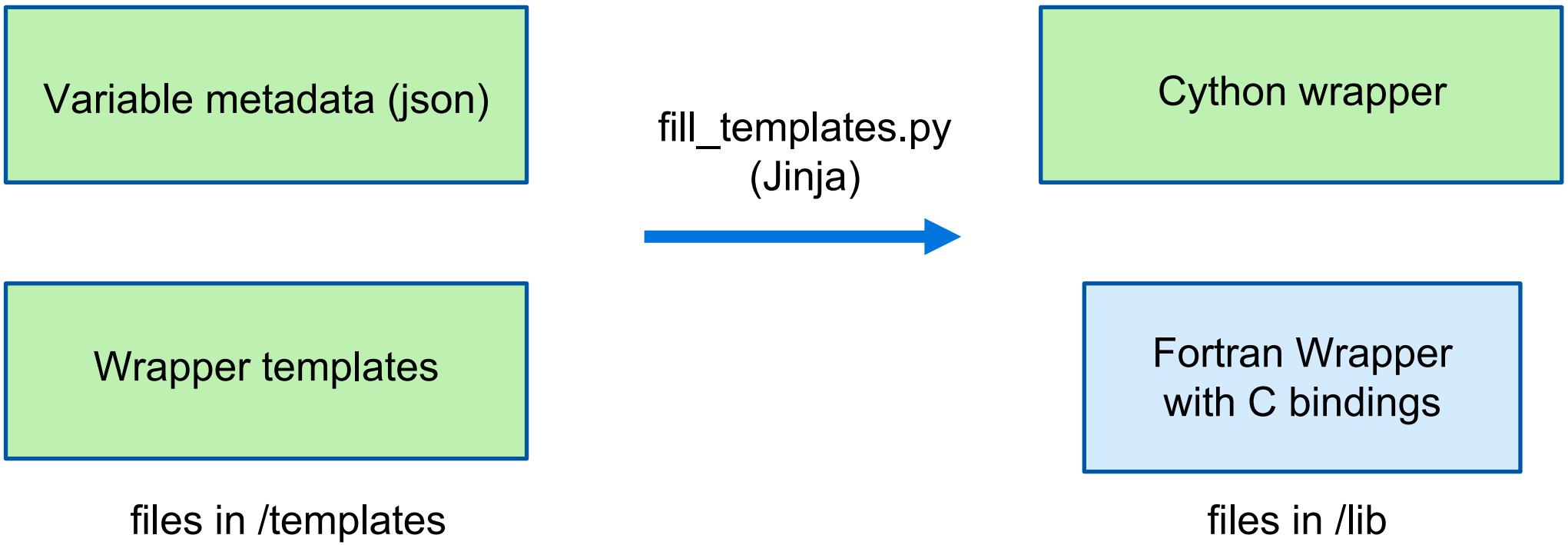
dynamics_data.F90

```
use atmosphere_mod, only: Atm, mytile
use tracer_manager_mod, only: get_tracer_names, get_number_tracers, get_tracer_index
use field_manager_mod,  only: MODEL_ATMOS
use iso_c_binding
```

```
pure function i_start() result(i)
    integer :: i
    i = Atm(mytile)%bd%is
end function i_start
```

```
subroutine set_u(u_in) bind(c)
    real(c_double), intent(in), dimension(i_start():i_end(), j_start():j_end()+1, 1:nz()) :: u_in
    Atm(mytile)%u(i_start():i_end(), j_start():j_end()+1, 1:nz()) = u_in(i_start():i_end(), j_start():j_end()+1, 1:nz())
end subroutine set_u

subroutine get_u(u_out) bind(c)
    real(c_double), intent(out), dimension(i_start():i_end(), j_start():j_end()+1, 1:nz()) :: u_out
    u_out(i_start():i_end(), j_start():j_end()+1, 1:nz()) = Atm(mytile)%u(i_start():i_end(), j_start():j_end()+1, 1:nz())
end subroutine get_u
```

Code Generation



Jinja Code Generation

- Jinja logic denoted by {% <code goes here> %}
- Inside a Jinja block, double curly brackets {{ }} denote an expression to parse
- Python code in fill_templates.py provides data for the template

dynamics_data.F90

```
{% for item in dynamics_properties %}
    subroutine set_{{ item.fortran_name }}({{ item.fortran_name }}_in) bind(c)
        real(c_double), intent(in), dimension({{ item.dim_ranges }}) :: {{ item.fortran_name }}_in
        Atm(mytile)%{{ item.fortran_name }}({{ item.dim_ranges }}) = {{ item.fortran_name }}_in({{ item.dim_ranges }})
    end subroutine set_{{ item.fortran_name }}

    subroutine get_{{ item.fortran_name }}({{ item.fortran_name }}_out) bind(c)
        real(c_double), intent(out), dimension({{ item.dim_ranges }}) :: {{ item.fortran_name }}_out
        {{ item.fortran_name }}_out({{ item.dim_ranges }}) = Atm(mytile)%{{ item.fortran_name }}({{ item.dim_ranges }})
    end subroutine get_{{ item.fortran_name }}
{% endfor %}
```

fill_templates.py

```
subroutine set_u(u_in) bind(c)
    real(c_double), intent(in), dimension(i_start():i_end(), j_start():j_end()+1, 1:nz()) :: u_in
    Atm(mytile)%u(i_start():i_end(), j_start():j_end()+1, 1:nz()) = u_in(i_start():i_end(), j_start():j_end()+1, 1:nz())
end subroutine set_u

subroutine get_u(u_out) bind(c)
    real(c_double), intent(out), dimension(i_start():i_end(), j_start():j_end()+1, 1:nz()) :: u_out
    u_out(i_start():i_end(), j_start():j_end()+1, 1:nz()) = Atm(mytile)%u(i_start():i_end(), j_start():j_end()+1, 1:nz())
end subroutine get_u
```

fill_templates.py

```
in_filename = os.path.join(setup_dir, f"templates/{base_filename}")
out_filename = os.path.join(setup_dir, f"lib/{base_filename}")
template = template_env.get_template(base_filename)
result = template.render(
    physics_2d_properties=physics_2d_properties,
    physics_3d_properties=physics_3d_properties,
    dynamics_properties=dynamics_properties,
    flagstruct_properties=flagstruct_properties,
)
with open(out_filename, "w") as f:
    f.write(result)
```

lists of data dictionaries

Jinja Code Generation

- Jinja logic denoted by {% <code goes here> %}
- Inside a Jinja block, double curly brackets {{ }} denote an expression to parse
- Python code in fill_templates.py provides data for the template
- This data is loaded from a json file, and added to by the script

dynamics_data.json

```
[
    {
        "name": "x_wind",
        "fortran_name": "u",
        "units": "m/s",
        "dims": ["z", "y_interface", "x"]
    },
    {
        "name": "y_wind",
```

dynamics_data.F90

```
{% for item in dynamics_properties %}
    subroutine set_{{ item.fortran_name }}({{ item.fortran_name }}_in) bind(c)
        real(c_double), intent(in), dimension({{ item.dim_ranges }}) :: {{ item.fortran_name }}_in
        Atm(mytile)%{{ item.fortran_name }}({{ item.dim_ranges }}) = {{ item.fortran_name }}_in({{ item.dim_ranges }})
    end subroutine set_{{ item.fortran_name }}

    subroutine get_{{ item.fortran_name }}({{ item.fortran_name }}_out) bind(c)
        real(c_double), intent(out), dimension({{ item.dim_ranges }}) :: {{ item.fortran_name }}_out
        {{ item.fortran_name }}_out({{ item.dim_ranges }}) = Atm(mytile)%{{ item.fortran_name }}({{ item.dim_ranges }})
    end subroutine get_{{ item.fortran_name }}
{% endfor %}
```

fill_templates.py

```
dynamics_data = json.load(
    open(os.path.join(PROPERTIES_DIR, "dynamics_properties.json"))
)
```

```
for properties in dynamics_data:
    properties["dim_ranges"] = get_dim_range_string(properties["dims"])
    properties["dim_colons"] = ", ".join(":" for dim in properties["dims"])
    dynamics_properties.append(properties)
```

```
in_filename = os.path.join(setup_dir, f"templates/{base_filename}")
out_filename = os.path.join(setup_dir, f"lib/{base_filename}")
template = template_env.get_template(base_filename)
result = template.render(
    physics_2d_properties=physics_2d_properties,
    physics_3d_properties=physics_3d_properties,
    dynamics_properties=dynamics_properties,
    flagstruct_properties=flagstruct_properties,
)
with open(out_filename, "w") as f:
    f.write(result)
```

lists of data dictionaries

Running the model

Running the model

- Basic idea: set up a run directory, mpirun a runfile inside
- Can get a run directory with:

```
wget https://zenodo.org/record/4429298/files/c48_6h.tar.gz
tar -xvf c48_6h.tar.gz
```

- Mount the run directory into your container using a bind mount (-v <abs local path>:<abs container path>)
- For non-interactive, you can use -w <directory> to run the command inside your run directory

Questions/Discussion?

Work time

Modify the basic model in some useful way, how is up to you. Two examples:

- The model as-is crashes after 18 hours due to an instability. This issue is often caused by the ML model introducing negative water into the model. Add a limiter which prevents this, and see if it fixes the instability.
- Add matplotlib code which runs on the first rank and plots something into a directory as the model runs. Use ffmpeg or another tool to turn the series of plots into a video.

Get in touch!

- Contact me at jeremym@vulcan.com or Spencer at spencerc@Vulcan.com
- Post issues to our Github repo https://github.com/VulcanClimateModeling/fv3gfs-wrapper
- Read more about our work https://www.vulcan.com/Special-Pages/Climate-Modeling.aspx

VULCAN

Geophysical Fluid Dynamics Laboratory
GFDL
NOAA