Workshop on

Domain-Specific Languages for Performance-Portable Weather and Climate Models

Content: Porting code to DSL
(serialization, validation, porting Fortran to Python)

Presenter: Rhea George and Chris Kung

Learning goals for this session

- Learn how Fortran code can be ported to a DSL and validated
    - Understand how serialization can be used to generate regression data
    - Be able to build a regression test in Python for existing Fortran

Introduction

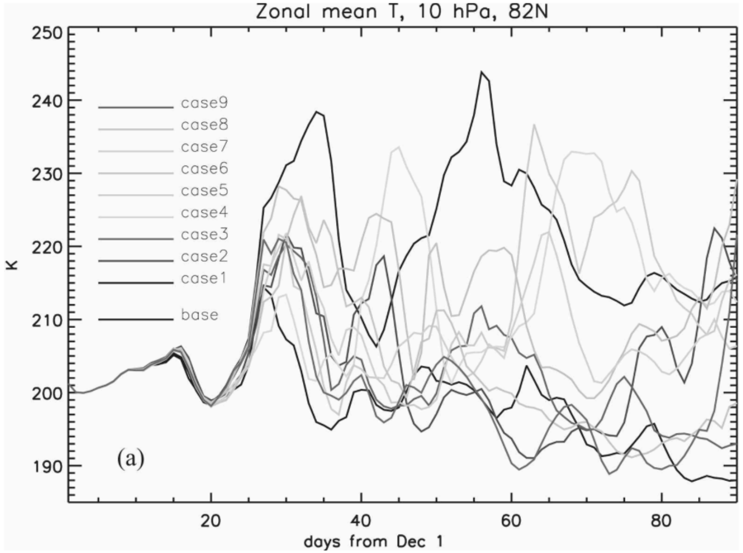
Atmospheric models contain many algorithmic patterns well suited for GT4Py and can be translated directly.

The difficult part is understanding the original code, resolving index alignments and bound settings, and tracking down the source of errors.

Validation of ported code

When you rewrite code, how are you sure it is right?

- How do you prove success?
- Can you continue to repeat the proof easily?



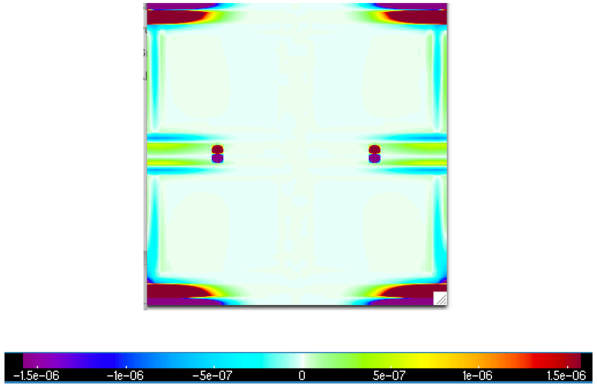
Source: Fig. 1 in Error Growth in a Whole Atmosphere Climate Model, Liu et. al 2009

- Original vs ported model output
  - If they are not identical, is it a bug?
  - If it's a bug where is it coming from?
- Small differences often result in large downstream differences in non-linear atmospheric models.
- Can collect and compare statistics of runs
  - Might obscure subtle and compensating bugs that aren't obvious in bulk stats
  - Time consuming to reproduce

Validation of ported code

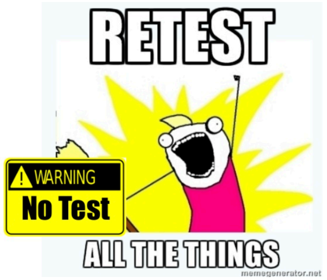
'Unit' testing

A bit reproducible port would be ideal, but can delay progress, not completely necessary to have confidence in the ported code.



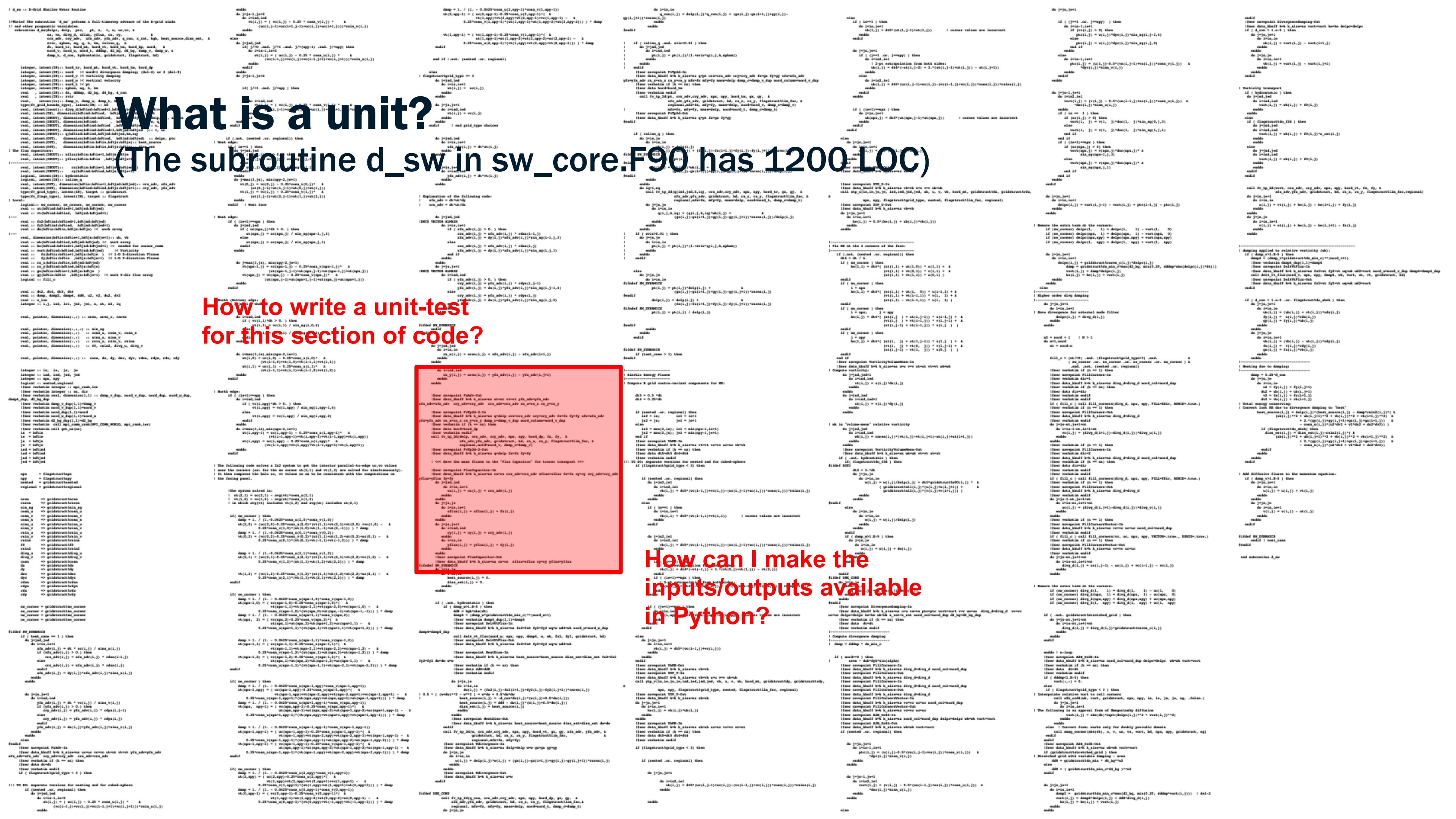
ΔTemperature (ported - original)

- Operating system, compiler, language standards, parallelization and math intrinsics are all reasons why achieving bit-reproducibility is possible but requires a significant effort (Arteaga et al. 2014, IPDPS).
- Using custom validation thresholds for different parts of the model is a pragmatic approach and has proven to be successful.
- 'Unit' test!
  - Find errors in small computational units before they infect a large chunk of code
  - Build up large regression testing set from the smaller units until you can test the whole model



What is a unit?

(The subroutine d_sw in sw_core.F90 has 1200 LOC)



```
[illegible]
```

How to write a unit-test for this section of code?

How can I make the inputs/outputs available in Python?

Serialbox

Serialization is the process of converting an object (e.g. data field) into a stream of bytes to store it.

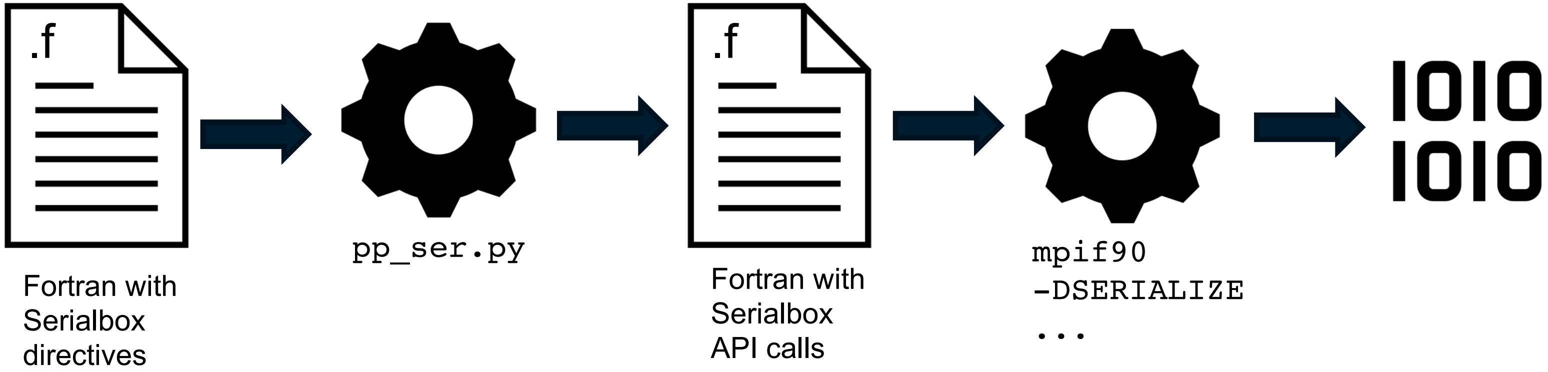
Serialbox is an open-source library which allows to serialize and de-serialize

https://github.com/GridTools/serialbox

- Serialization library for C/C++, Python(3), and Fortran codes that enables easy sharing of data between the various languages
- Supports primitive types character, Boolean, double, float, and integer
- Scalar and arrays (1d, 2d, 3d, 4d) of primitive types can be saved
- Functionality is exposed through a low-level API in C/C++/Fortran/Python and a high-level directive-like code format in Fortran

Serialbox: Fortran Workflow

- Compile the code containing Serialbox API calls using a standard Fortran compiler with the compiler flag `-DSERIALIZE`.
- Serialization/increased I/O results in slow runtime (don't try to run a large or long simulation)


.f
pp_ser.py
.f
1010
1010
Fortran with Serialbox directives
Fortran with Serialbox API calls
mpif90 -DSERIALIZE ...

Serialbox: Fortran Directives

`!$ser init directory=<path> prefix=<string>` Initialize Serialbox

`!$ser savepoint “<string>”` Create a savepoint to which data is associated

`!$ser data <name>=<variable>` Save data from Fortran <variable> to Serialbox as <name>

`!$ser verbatim <code>` Execute code only if `–DSERIALIZE` is set

`!$ser on` Turn serialization on
`!$ser off` Turn serialization off (subsequent `!$ser data` statements are ignored)

Custom data structures / derived data types cannot be serialized. However, primitive types within a derived data type can be individually serialized.

Basic commands to get started, more advanced directives and low-level API exist.

Serialbox: Fortran Pre-processing

- Using the high-level API allows developers to add serialization in a minimally invasive way
- `pp_ser.py` (a script available in Serialbox) translates Serialbox `!$ser` directives into Serialbox API calls
- The normal operation of the Fortran code is unaltered

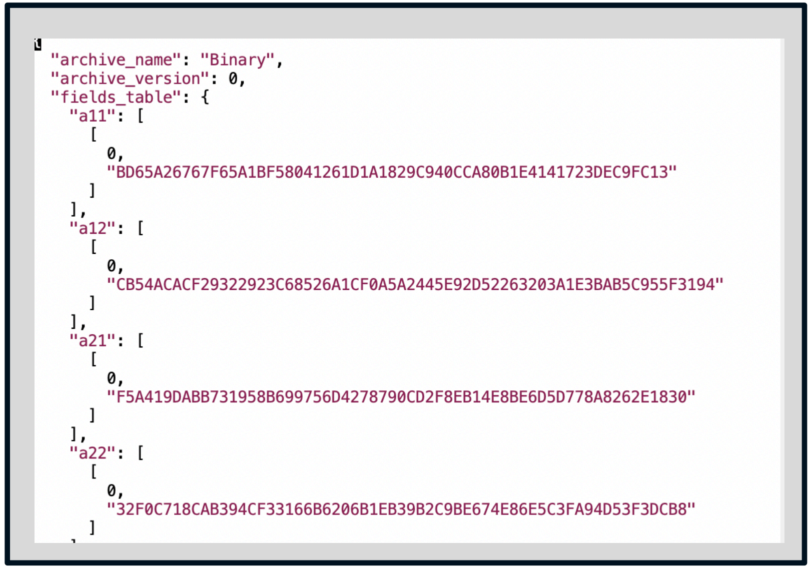
```
...
!$ser savepoint ”YPPM-In” k=k
!$ser data q=q c=cry jord=ord_in
call yppm(fy2, q, cry, ord_in, ...)
...
```

pp_ser.py

```
...
#ifdef SERIALIZE
call fs_create_savepoint('YPPM-In', ppser_savepoint)
call fs_add_savepoint_metainfo(ppser_savepoint, 'k', k)
SELECT CASE ( ppser_get_mode() )
  CASE(0)
    call fs_write_field(ppser_serializer, ppser_savepoint, 'q', q)
    call fs_write_field(ppser_serializer, ppser_savepoint, 'c', cry)
    call fs_write_field(ppser_serializer, ppser_savepoint, 'jord', ord_in)
END SELECT
#endif
call yppm(fy2, q, cry, ord_in, ...)
...
```

Serialbox: Data Format

Serialized data is be saved in binary format, one file per field and MPI rank.



```
{
  "archive_name": "Binary",
  "archive_version": 0,
  "fields_table": {
    "a11": [
      [
        0,
        "BD65A26767F65A1BF58041261D1A1829C940CCA80B1E4141723DEC9FC13"
      ]
    ],
    "a12": [
      [
        0,
        "CB54ACACF29322923C68526A1CF0A5A2445E92D52263203A1E3BAB5C955F3194"
      ]
    ],
    "a21": [
      [
        0,
        "F5A419DABB731958B699756D4278790CD2F8EB14E8BE6D5D778A8262E1830"
      ]
    ],
    "a22": [
      [
        0,
        "32F0C718CAB394CF33166B6206B1EB39B2C9BE674E86E5C3FA94D53F3DCB8"
      ]
```

`field_<rank>.dat`

Savepoint data in binary format

`ArchiveMetadata-Serializer_<rank>.json`

Record of lookup of field data hashes in dat files, not directly useful

`Metadata-Serializer_<rank>.json`

Field and Savepoint information

Hands-on Session

Session-3A/fortran-serialbox.ipynb

- Generate serialized regression data in Fortran using Serialbox

Session-3A/gt4py-port-test.ipynb

- How to port code to GT4py using regression data and the Python Serialbox API

See you on Slack! **Next huddle at 11:15 am EST.**

Serialbox: Python API

```
# import Serialbox package
import serialbox as ser

# create a serializer object (read-only)
serializer = ser.Serializer(
    ser.OpenModeKind.Read, “data_path”, “SerializerName_rank”
)

# create a savepoint object
savepoint = serializer.get_savepoint(“SavepointName”)

# read data for given name from savepoint
data = serializer.read(name, savepoint)
```

A basic unit test

```
!$ser init directory='./test_data/' prefix='Generator'
             mpi_rank=mpi_rank unique_id=.true.
!$ser mode write
!$ser on
 ...

 !$ser savepoint "PGradC-In"
 !$ser data delpc=delpc pkc=pkc gz=gz uc=uc vc=vc
 call p_grad_c(delpc, pkc, gz, uc, vc, ...)
 !$ser savepoint "PGradC-Out"
 !$ser data uc=uc vc=vc

 ...
```

- Note: Serialized data is imported into Python as Numpy arrays.

```
import serialbox as ser
serializer = ser. Serializer(ser.OpenModeKind.Read,
'test_data', 'Generator_rank0')

# read serialized inputs
sp = serializer.get_savepoint("PGradC-In")
delpc = serializer.read('delpc', sp[0])
pkc   = serializer.read('pkc', sp[0])
gz    = serializer.read('gz', sp[0])
uc    = serializer.read('uc', sp[0])
vc    = serializer.read('vc', sp[0])

# run computation
uc, vc = p_grad_c(delpc, pkc, gz, uc, vc, ...)

# read serialized outputs
sp = serializer.get_savepoint("PGradC-Out")
uc_ref = serializer.read('uc', sp)
vc_ref = serializer.read('vc', sp)

# check result
eps = 3.0e-14
assert np.all(np.abs(uc - uc_ref) < eps)
assert np.all(np.abs(vc - vc_ref) < eps)
```

Porting to GT4Py

Fortran

```
subroutine del2_cubed(q, cd, del6_v, del6_u, rarea, grid)
real :: fx(is:ie+1, js:je), fy(is:ie, js:je+1)
!$OMP parallel do default(none) shared(km, q, &
!$OMP is,ie,js,je, & cd) &
!$OMP private(fx, fy)

do k = 1, km
  do j = js, je
    do i = is, ie + 1
      fx(i,j) = del6_v(i,j) * ( q(i-1,j,k) - q(i,j,k) )
    enddo
  enddo

  do j = js, je + 1
    do i = is, ie
      fy(i,j) = del6_u(i,j) * ( q(i,j-1,k) - q(i,j,k) )
    enddo
  enddo

  do j = js, je
    do i = is, ie
      q(i,j,k) = q(i,j,k) + cd * rarea(i,j) * (
          fx(i,j) - fx(i+1,j) + fy(i,j) - fy(i,j+1) )
    enddo
  enddo
enddo
...
end subroutine del2_cubed
call del2_cubed(q, cd, del6_v, del6_u, rarea, grid)
```

GT4Py

```
@gtscript.function
def delx(q, weight):
    return weight * (q[-1, 0, 0] – q)

@gtscript.function
def dely(q, weight)
    return weight * (q[0, -1, 0] – q)

@gtscript.stencil(backend=‘numpy’)
def del2_cubed(q:field, rarea:field, del6_v:field,
               del6_u:field, cd:float):
    with computation(PARALLEL), interval(...):
        fx = delx(q, del6_v)
        fy = dely(q, del6_u)
        q = q + cd * rarea * (fx - fx[1, 0, 0] &
                             + fy - fy[0, 1, 0])

del2_cubed(q, del6_u, del6_v rarea, cd,
           origin=grid.compute_origin(),
           domain=grid.compute_domain())
```

Porting to GT4py

Porting from Fortran to Python

```
do k = 1, km
  do j = js, je
    do i = is, ie + 1
      fx(i,j) = del6_v(i,j) * ( q(i-1,j,k) - q(i,j,k) )
    enddo
  enddo

  do j = js, je + 1
    do i = is, ie
      fy(i,j) = del6_u(i,j) * ( q(i,j-1,k) - q(i,j,k) )
    enddo
  enddo

  do j = js, je
    do i = is, ie
      q(i,j,k) = q(i,j,k) + cd * rarea(i,j) * (
          fx(i,j) - fx(i+1,j) + fy(i,j) - fy(i,j+1) )
    enddo
  enddo
enddo
```

```
for k in range(km):
  for j in range(je-js + 1):
    for i in range(ie-is+2):
      fx[i,j] = del6_v[i,j] * ( q[i-1,j,k] – q[i,j,k] )

  for j in range(je-js+2):
    for i in range(ie-is+1):
      fy[i,j] = del6_u[i,j] * ( q[i,j-1,k] – q[i,j,k] )

  for j in range(je-js+1)
    for i in range(ie-is+1)
      q[i,j,k] = q[i,j,k] + cd * rarea[i,j] * (
          fx[i,j] – fx[i+1,j] + fy[i,j] – fy[i,j+1] )
```

- Note : Be mindful of array indexing when porting to Python!
  - Fortran : Array’s starting index = 1 (or arbitrary)
  - Python : Array’s starting index (in Numpy) = 0
- Resist the urge to use fast convenience Python libraries, which you will need to re-extract the loop structure from to get into GT4py

Porting to GT4py

Optional : Add k-dimension to 2D Numpy arrays

```
for k in range(km):
  for j in range(je-js + 1):
    for i in range(ie-is):
      fx[i,j] = del6_v[i,j] * ( q[i-1,j,k] – q[i,j,k] )


  for j in range(je-js+2):
    for i in range(ie-is):
      fy[i,j] = del6_u[i,j] * ( q[i,j-1,k] – q[i,j,k] )

  for j in range(je-js+1)
    for i in range(ie-is+1)
      q[i,j,k] = q[i,j,k] + cd * rarea[i,j] * (
          fx[i,j] – fx[i+1,j] + fy[i,j] – fy[i,j+1] )
```

```
for k in range(km):
  for j in range(je-js + 1):
    for i in range(ie-is):
      fx[i,j,k] = del6_v[i,j,k] * ( q[i-1,j,k] – q[i,j,k] )


  for j in range(je-js+2):
    for i in range(ie-is):
      fy[i,j,k] = del6_u[i,j,k] * ( q[i,j-1,k] – q[i,j,k] )

  for j in range(je-js+1)
    for i in range(ie-is+1)
      q[i,j,k] = q[i,j,k] + cd * rarea[i,j,k] * (
          fx[i,j,k] – fx[i+1,j,k] + fy[i,j,k] – fy[i,j+1,k] )
```

- Note : 2D fields are allowed and sometimes preferred, but reframing as a 3D problem can simplify porting efforts.

Porting to GT4py

From Python, port into GT4py

```
for k in range(km):
  for j in range(je-js + 1):
    for i in range(ie-is):
      fx[i,j,k] = del6_v[i,j,k] * ( q[i-1,j,k] – q[i,j,k] )


  for j in range(je-js+2):
    for i in range(ie-is):
      fy[i,j,k] = del6_u[i,j,k] * ( q[i,j-1,k] – q[i,j,k] )

  for j in range(je-js+1)
    for i in range(ie-is+1)
      q[i,j,k] = q[i,j,k] + cd * rarea[i,j,k] * (
          fx[i,j,k] – fx[i+1,j,k] + fy[i,j,k] – fy[i,j+1,k] )
```

```
with computation(PARALLEL), interval(...):
    fx = del6_v[0,0,0] * (q[-1,0,0] – q[0,0,0])
    fy = del6_u[0,0,0] * (q[0,-1,0] – q[0,0,0])
    q = q[0,0,0] + cd * rarea[0,0,0] * (fx[0,0,0] –  &
        fx[1,0,0] + fy[0,0,0] – fy[0,1,0])
```

- Note : A variable `x` at [i, j, k] can be written as either :
  - `x[0, 0, 0]` (to be super clear this is 3d)
  - `x` (to improve readability)

Porting to GT4py

Define GT4Py functions when appropriate

```
with computation(PARALLEL), interval(...):
    fx = del6_v[0,0,0] * (q[-1,0,0] – q[0,0,0])
    fy = del6_u[0,0,0] * (q[0,-1,0] – q[0,0,0])
    q = q[0,0,0] + cd * rarea[0,0,0] * (fx[0,0,0] –  &
        fx[1,0,0] + fy[0,0,0] – fy[0,1,0])
```

```
@gtscript.function
def delx(q, weight):
    return weight * (q[-1,0,0] – q)

@gtscript.function
def dely(q, weight)
    return weight * (q[0,-1,0] – q)
...
...
with computation(PARALLEL), interval(...):
    fx = delx(q, del6_v)
    fy = dely(q, del6_u)
    q = q[0,0,0] + cd * rarea[0,0,0] * (fx[0,0,0] –  &
        fx[1,0,0] + fy[0,0,0] – fy[0,1,0])
```

- Note: Abstraction is beautiful, but a function can’t be used in all situations and is typically meant for code that can be inlined.

Porting to GT4py

Putting it all together…

```
subroutine del2_cubed(q, cd, del6_v, del6_u, rarea, grid)

real :: fx(is:ie+1, js,je), fy(is:ie, js:je+1)
!$OMP parallel do default(none) shared(km, q,&
!$OMP is,ie,js,je, & cd) &
!$OMP private(fx, fy)
do k = 1, km
  do j = js, je
    do i = is, ie + 1
      fx(i,j) = del6_v(i,j) * ( q(i-1,j,k) - q(i,j,k) )
    enddo
  enddo

  do j = js, je + 1
    do i = is, ie
      fy(i,j) = del6_u(i,j) * ( q(i,j-1,k) - q(i,j,k) )
    enddo
  enddo

  do j = js, je
    do i = is, ie
      q(i,j,k) = q(i,j,k) + cd * rarea(i,j) * (
          fx(i,j) - fx(i+1,j) + fy(i,j) - fy(i,j+1) )
    enddo
  enddo
enddo

...

end subroutine del2_cubed

call del2_cubed(q, cd, del6_v, del6_u, rarea, grid)
```

```
@gtscript.function
def delx(q, weight):
    return weight * (q[-1, 0, 0] – q)

@gtscript.function
def dely(q, weight)
    return weight * (q[0, -1, 0] – q)

@gtscript.stencil(backend=‘numpy’)
def del2_cubed(q:field, rarea:field, del6_v:field, del6_u:field,
cd:float):
    with computation(PARALLEL), interval(...):
        fx = delx(q, del6_v)
        fy = dely(q, del6_u)
        q = q + cd * rarea * (fx – fx[1, 0, 0] + fy – fy[0, 1, 0])

del2_cubed(q, del6_u, del6_v rarea, cd,
           origin=grid.compute_origin(), domain=grid.compute_domain())
```

- Horizontal loops removed, schedule removed
- Optional specification of loop endpoints with ‘origin’ and ’domain’ in call to the stencil if needed
- Index offsets instead of absolute indices
- No explicit storage statements for temporary variables
- Overhead-free, reusable functions -- inlining
- Less code

Porting directly to GT4py

Loop bounds

Direct mapping of loop bounds using origin and domain is a good first step

Fortran

```
do j=js,je
    pe(is-1,j,1) = ptop
    do k=1,npz
        pe(is-1,j,k+1) = pe(is-1,j,k) + delp(is-1,j,k)
    enddo
enddo
```

- Origin: where in the data array the calculation starts
    - FV3 Fortran code uses tile global indices is, js to indicate the start of the compute domain
    - In, Python need to refer to local indices on rank, nhalo is the local horizontal start index is and js.

Python

```
@gtscript.stencil()
def edge_pe(pe: field, delp: field, ptop: float):
    with computation(FORWARD):
        with interval(0, 1):
            pe = ptop
        with interval(1, None):
            pe = pe[0, 0, -1] + delp[0, 0, -1]
def compute_edge_pe(pe, delp, ptop):
    edge_pe(pe, delp, ptop, origin=(nhalo - 1, nhalo, 0), domain=(1, ny, nz + 1))
```

Porting directly to GT4py using regions

Loop bounds

Also possible to not restrict the x-axis domain and use the ‘regions’ feature to specify the edge calculation.

Fortran

```
do j=js,je
    pe(is-1,j,1) = ptop
    do k=1,npz
        pe(is-1,j,k+1) = pe(is-1,j,k) + delp(is-1,j,k)
    enddo
enddo
```

Python with regions

```
@gtscript.stencil()
def larger_stencil(pe: field, delp: field, u: field, v: field, ptop: float):
    … other computations…
    with computation(FORWARD):
        with interval(0, 1):
              with parallel(region([I[0], :]):
                  pe = ptop
        with interval(1, None):
               with parallel(region([I[0], :]):
                  pe = pe[0, 0, -1] + delp[0, 0, -1]

def compute_large_section(pe, delp, ptop, u, v):
    large_stencil(pe, delp, u, v, ptop,
                  origin=(nhalo - 1, nhalo, 0), domain=(nx+2, ny, nz + 1)))
```

Error metrics

Validation criteria

How well do you expect the answers to match?

$$|V - R| < \varepsilon$$

$$|V - R| / R < \varepsilon$$

$$\frac{2 * |V - R|}{|V| + |R|} < \varepsilon$$

… etc

- Bit-for-bit matches makes it easier, though can limit refactors
- Considerations:
  - Relative error to avoid variable specific thresholds
  - Values near 0 – relative error can be misleading
  - Handle NaNs
  - When to investigate vs relax thresholds

Porting to GT4py

Final Thoughts

- Everyone will find a porting process that works for them
- Having a fundamental understanding of Python will help ease the porting process to GT4Py (and allows you to take advantage of Python features for debugging!)

Hands-on Session

Session-3A/fortran-serialbox.ipynb

- How to generate serialized regression data in Fortran using Serialbox

Session-3A/gt4py-port-test.ipynb

- How to port code to GT4py using regression data and the Python Serialbox API

Session-3A/optional-edge-pressure.ipynb

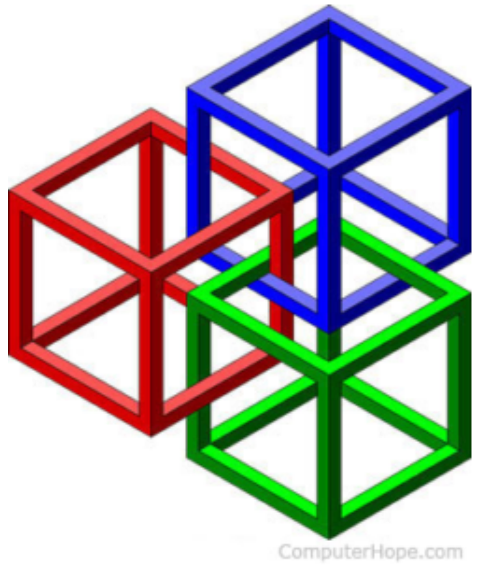
- Another porting example

See you on Slack! **Next huddle at 1:15 pm EST.**

Making your test writing easier with a framework

How would you alter your code for porting a lot of units?

Depending on how you wrote the first code, start to consider how you would want to extend it to port a large codebase.

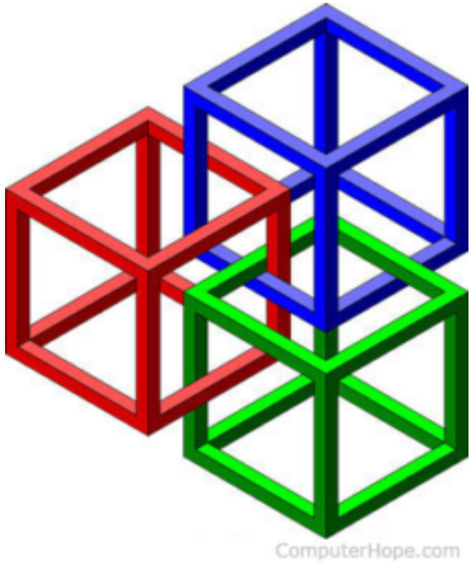


- What seems tedious/repetitive that you could make easier?

Making your test writing easier with a framework

How would you alter your code for porting a lot of units?

Depending on how you wrote the first code, start to consider how you would want to extend it to port a large codebase.



- What seems tedious/repetitive that you could make easier?
  - Translating from serialized data to the conventions of your new code
  - Indexing
  - Variable naming and identification

- Serialization considerations:
  - Data volume
  - Serialization complexity

FV3core Fortran to Python

Porting framework using classes

There are many possible ways to design a system for porting. For the dynamical core of FV3 we chose to go object-oriented.

- Allows for easy reuse of main functionality
- Allows for easy overriding of default behavior when needed
- Keeps the specification volume low

```
class TranslateFortranData2Py:
    max_error = 1e-14
    def __init__(self, grid):
        self.in_vars = {"data_vars": {}, "parameters": []}
        self.out_vars = {}
        self.grid = grid
        self.maxshape = (grid.npx + 1, grid.npy + 1, grid.npz + 1)

    def compute(self, inputs):
        self.make_storage_data(inputs)
        return self.slice_output(self.compute_func(**inputs))

    def make_storage_data(self, inputs):
        # …code that generates gt4py storages from serialized data
        # given specifications of each var in in_vars[“data_vars”]
```

```
import fv3core.stencils.del2cubed as del2cubed
from .translate import TranslateFortranData2Py

class TranslateDel2Cubed(TranslateFortranData2Py):
    def __init__(self, grid):
        super().__init__(grid)
        self.compute_func = del2cubed.compute
        self.in_vars["data_vars"] = {"qdel": {}}
        self.in_vars["parameters"] = ["nmax", "cd", "km"]
        self.out_vars = {"qdel": {}}
```

FV3core Fortran to Python

Porting with different variable sizes

Specification inside of translation class makes it easy to handle different patterns in the Fortran code.

- Directly specify array bounds (default assumed isd:ied, jsd:jed, 1:npz)
- Copy serialization data into gt4py storages of the same size and compare output
- Handles 'k' not in the 3rd dimension

Fortran

```
!$ser savepoint PE_Halo-In
!$ser data ptop=ptop pe=pe delp=delp remap_step=remap_step
if ( remap_step )  &
call pe_halo(ptop, pe, delp)
!$ser savepoint PE_Halo-Out
!$ser data pe=pe

…

subroutine pe_halo(ptop, pe, delp)
real, intent(in):: ptop
real, intent(in   ), dimension(isd:ied,jsd:jed,npz):: delp
real, intent(inout), dimension(is-1:ie+1,npz+1,js-1:je+1):: pe
```

Python translation of serialized data to gt4py storages

```
import fv3core.stencils.pe_halo as pe_halo
from .translate import TranslateFortranData2Py

class TranslatePE_Halo(TranslateFortranData2Py):
    def __init__(self, grid):
        super().__init__(grid)
        self.compute_func = pe_halo.compute
        self.in_vars["data_vars"] = {
            "delp": {},
            "pe": { "istart": grid.is_ - 1,
                    "iend": grid.ie + 1,
                    "jstart": grid.js - 1,
                    "jend": grid.je + 1,
                    "kend": grid.npz + 1,
                    "kaxis": 1,
            },
        }
        self.in_vars["parameters"] = ["ptop"]
        self.out_vars = {"pe": self.in_vars["data_vars"]["pe"]}
```

FV3core in Python

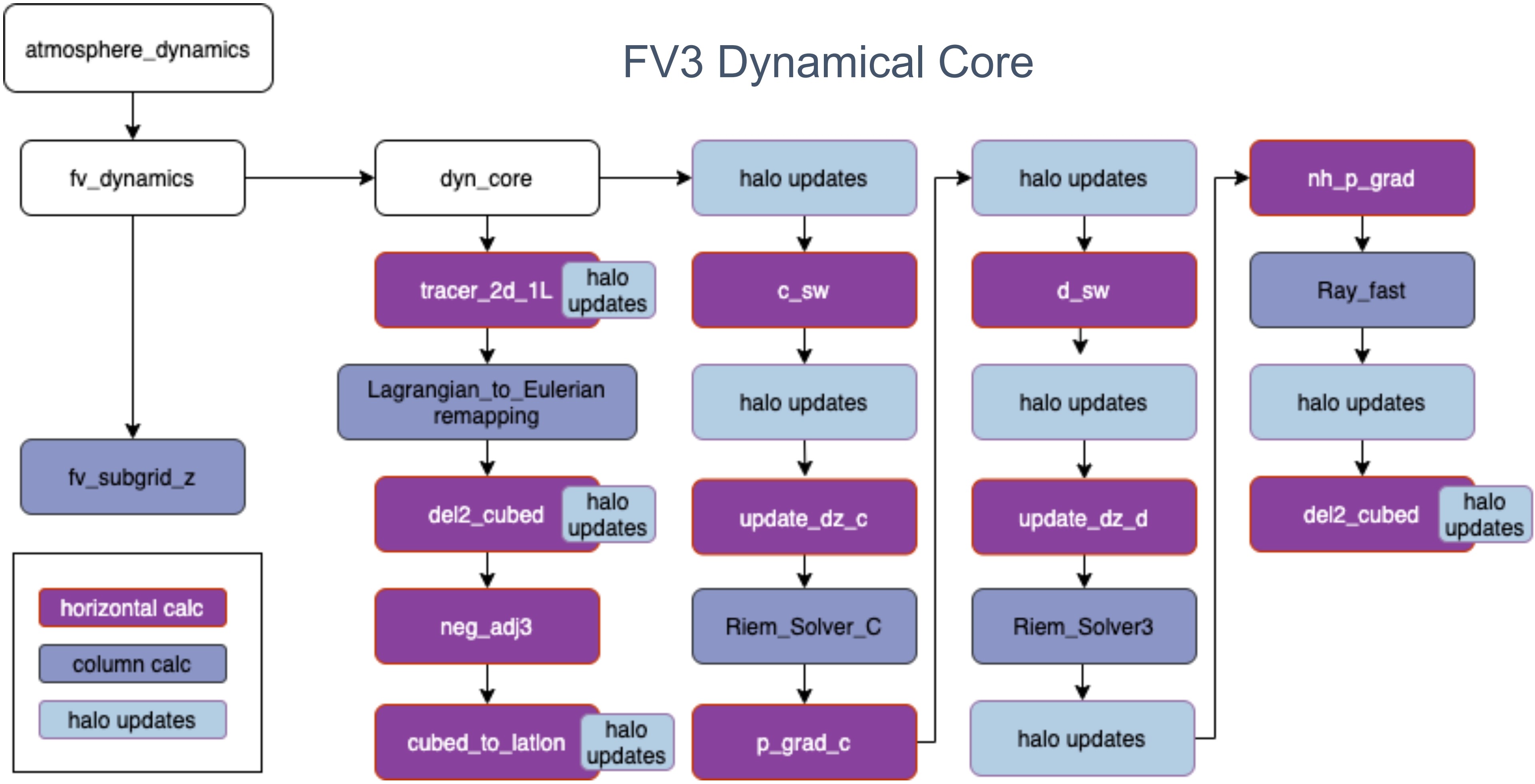
First port organized around test 'units'

Once computational units validate, added regression tests that call those functions

```
def fv_dynamics(state, comm):
    generate_temporary_variables(state)
    set_constants(state)
    last_step = False
    compute_preamble(state, comm)
    for n_map in range(namelist.k_split):
        if n_map == namelist.k_split - 1:
            last_step = True
        dyncore(state, comm)
        if grid.npz > 4:
            kord_tracer = [spec.namelist.kord_tr] * state.nq
            kord_tracer[6] = 9
            lagrangian_to_eulerian.compute(state)
            if last_step:
                post_remap(state, comm)
    wrapup(state, comm)
```

- Aggregated 'units' until we had a full port of the subroutine 'fv_dynamics' (and 'fv_subgridz'
- Tests and code with halo updates run in parallel with mpi
- Outer methods use a 'state' object to keep track of the many variables'
- Actively refactoring for performance, style and use of new gt4py features → more code inside of stencils

FV3 Dynamical Core


atmosphere_dynamics
fv_dynamics
fv_subgrid_z
dyn_core
tracer_2d_1L
halo updates
Lagrangian_to_Eulerian remapping
del2_cubed
halo updates
neg_adj3
cubed_to_latlon
halo updates
halo updates
c_sw
halo updates
update_dz_c
Riem_Solver_C
p_grad_c
halo updates
d_sw
halo updates
update_dz_d
Riem_Solver3
halo updates
nh_p_grad
Ray_fast
halo updates
del2_cubed
halo updates
horizontal calc
column calc
halo updates