psyclone Documentation

Release 1.0.0

Rupert Ford

July 16, 2015

CONTENTS

PSyclone, the PSy code generator, is being developed for use in finite element, finite volume and finite difference codes. PSyclone is being developed to support the emerging API in the GungHo project for a finite element dynamical core.

The GungHo project is designing and building the heart of the Met Office's next generation software (known as the dynamical core) using algorithms that will scale to millions of cores. The project is a collaboration between the Met Office, NERC (via NERC funded academics) and STFC, and the resultant software is expected to be operational in 2022.

The associated GungHo software infrastructure is being developed to support multiple meshes and element types thus allowing for future model development. GungHo is also proposing a novel separation of concerns for the software implementation of the dynamical core. This approach distinguishes between three layers: the Algorithm layer, the Kernel layer and the Parallelisation System (PSy) layer. Together this separation is termed PSyKAl.

The Algorithm layer specifies the algorithm that the scientist would like to run (in terms of calls to kernel and infrastructure routines) and logically operates on full fields.

The Kernel layer provides the implementation of the code kernels as subroutines. These subroutines operate on local fields (a set of elements, a vertical column, or a set of vertical columns, depending on the kernel).

The PSy layer sits in-between the algorithm and kernel layers and its primary role is to provide node-based parallel performance for the target architecture. The PSy layer can be optimised for a particular hardware architecture, such as multi-core, many-core, GPGPUs, or some combination thereof with no change to the algorithm or kernel layer code. This approach therefore offers the potential for portable performance.

Rather than writing the PSy layer manually, the GungHo project is developing the PSyclone code generation system which can help a user to optimise the code for a particular architecture (by providing optimisations such as blocking, loop merging, inlining etc), or alternatively, generate the PSy layer automatically.

PSyclone is also being extended to support an API being developed in the GOcean project for two finite difference ocean model benchmarks, one of which is based on the NEMO ocean model.

Contents:

CHAPTER

ONE

GETTING GOING

1.1 Download

PSyclone is available for download from the GungHo repository.

```
svn co https://puma.nerc.ac.uk/svn/GungHo_svn/PSyclone/trunk PSyclone
```

Hereon the location where you download PSyclone (including the PSyclone directory itself) will be refered to as <PSYCLONEHOME>

1.2 Dependencies

PSyclone is written in python so needs python to be installed on the target machine. PSyclone has been tested under python 2.6.5 and 2.7.3.

PSyclone immediately relies on two external libraries, f2py and pyparsing. To run the test suite you will require py.test.

1.2.1 f2py quick setup

The source code of f2py (revision 93) is provided with PSyclone in the sub-directory `f2py_93`.

To use f2py provided with PSyclone you can simply set up your PYTHONPATH variable to include this directory.

```
> export PYTHONPATH=<PSYCLONEHOME>/f2py_93:${PYTHONPATH}
```

If for some reason you need to install f2py yourself then see *f2py installation*.

1.2.2 pyparsing

PSyclone requires pyparsing, a library designed to allow parsers to be be built in Python. PSyclone uses pyparsing to parse fortran regular expressions as f2py does not fully parse these, (see http://pyparsing.wikispaces.com for more information).

PSyclone has been tested with pyparsing version 1.5.2 which is a relatively old version but is currently the version available in the Ubuntu software center.

You can test if pyparsing is already installed on your machine by typing `import pyparsing` from the python command line. If pyparsing is installed, this command will complete succesfully. If pyparsing is installed you can check its version by typing `pyparsing.__version__` after succesfully importing it. Versions higher than 1.5.2 should work but have not been tested.

If pyparsing is not installed on your system you can install it from within Ubuntu using the software center (search for the "python-pyparsing" module in the software center and install). If you do not run Ubuntu you could follow the instructions here http://pyparsing.wikispaces.com/Download+and+Installation.

1.2.3 py.test

The PSyclone test suite uses py.test. You can test whether it is already installed by simply typing `py.test` at a shell prompt. If it is present you will get output that begins with

```
======================== test session starts ==================
```

If you do not have it then py.test can be installed from here http://pytest.org/latest/ (or specifically here http://pytest.org/latest/getting-started.html).

1.3 Environment

In order to use PSyclone (including running the test suite) you will need to tell Python where to find the PSyclone source:

```
> export PYTHONPATH=<PSYCLONEHOME>/src:${PYTHONPATH}
```

1.4 Test

Once you have the necessary dependencies installed and your environment configured, you can test that things are working by using the PSyclone test suite:

```
> cd <PSYCLONEHOME>/src/tests
> py.test
```

If everything is working as expected then you should see output similar to:

```
============================= test session starts ========================
platform linux2 -- Python 2.7.5 -- py-1.4.26 -- pytest-2.6.4
collected 35 items

alggen_test.py xxxxx.x.x
f2pygen_test.py .x....
generator_test.py ..
parser_test.py .
psyGen_test.py ............
transformations_test.py xx.x

================== 24 passed, 11 xfailed in 1.21 seconds ==================
```

1.5 Run

Having checked things with the test suite you are ready to try running PSyclone on the examples. One way of doing this is to use the generator.py script:

```
> cd <PSYCLONEHOME>/src
> python ./generator.py
usage: generator.py [-h] [-oalg OALG] [-opsy OPSY]  [-api API] filename
generator.py: error: too few arguments
```

As indicated above, the generator.py script takes the name of the Fortran source file containing the algorithm specification (in terms of calls to invoke()). It parses this, finds the necessary kernel source files and produces two Fortran files. The first contains the PSy, middle layer and the second a re-write of the algorithm code to use that layer. These files are named according to the user-supplied arguments (options -oalg and -opsy). If those arguments are not supplied then the script writes the generated/re-written Fortran to the terminal.

Examples are provided in the examples directory. There are 3 subdirectories (dynamo, gocean and gunghoproto) corresponding to different API's that are supported by PSyclone. In this case we are going to use one of the dynamo examples

```
> cd <PSYCLONEHOME>/examples/dynamo/eg1
> python ../../../src/generator.py -oalg dynamo_alg.f90 -opsy dynamo_psy.f90 dynamo.F90
```

You should see two new files created called dynamo_alg.f90 (containing the re-written algorithm layer) and dynamo_psy.f90 (containing the generated PSy- or middle-layer). Since this is a dynamo example the Fortran source code has dependencies on the dynamo system and therefore cannot be compiled stand-alone.

You can also use the runme.py example to see the interactive API in action. This script contains:

```
from parse import parse
from psyGen import PSyFactory

# This example uses version 0.1 of the Dynamo API
api="dynamo0.1"

# Parse the file containing the algorithm specification and
# return the Abstract Syntax Tree and invokeInfo objects
ast,invokeInfo=parse("dynamo.F90",api=api)

# Create the PSy-layer object using the invokeInfo
psy=PSyFactory(api).create(invokeInfo)
# Generate the Fortran code for the PSy layer
print psy.gen

# List the invokes that the PSy layer has
print psy.invokes.names

# Examine the 'schedule' (e.g. loop structure) that each
# invoke has
schedule=psy.invokes.get('invoke_v3_kernel_type').schedule
schedule.view()

schedule=psy.invokes.get('invoke_v3_solver_kernel_type').schedule
schedule.view()
```

It can be run non-interactively as follows:

```
> cd <PSYCLONEHOME>/example/dynamo/eg1
> python runme.py
```

However, to understand this example in more depth it is instructive to cut-and-paste from the runme.py file into your own, interactive python session:

```
> cd <PSYCLONEHOME>/example/dynamo/eg1
> python
```

In addition to the runme.py script, there is also runme_openmp.py which illustrates how one applies an OpenMP transform to a loop schedule within the PSy layer. The initial part of this script is the same as that of runme.py (above) and is therefore omitted here:

```
# List the various invokes that the PSy layer contains
print psy.invokes.names

# Get the loop schedule associated with one of these
# invokes
schedule=psy.invokes.get('invoke_v3_kernel_type').schedule
schedule.view()

# Get the list of possible loop transformations
from psyGen import TransInfo
t=TransInfo()
print t.list

# Create an OpenMPLoop-transformation object
ol=t.get_trans_name('OpenMPLoop')

# Apply it to the loop schedule of the selected invoke
new_schedule,memento=ol.apply(schedule.children[0])
new_schedule.view()

# Replace the original loop schedule of the selected invoke
# with the new, transformed schedule
psy.invokes.get('invoke_v3_kernel_type')._schedule=new_schedule
# Generate the Fortran code for the new PSy layer
print psy.gen
```

CHAPTER

TWO

API

generator.py

-oalg <filename>

-opsy <filename>

<filename>

Command line version of the generator. If -oalg or -opsy or not provided then the generated code is printed to stdout. Uses the `generator.generate()` function to generate the code. Please see the run documentation for more details.

For example:

```
> python generator.py algspec.f90
```

`generator.`**`generate`**(*filename*, *api=''*, *kernel_path=''*)

Takes a GungHo algorithm specification as input and outputs the associated generated algorithm and psy codes suitable for compiling with the specified kernel(s) and GungHo infrastructure. Uses the `parse.parse()` function to parse the algorithm specification, the `psyGen.PSy` class to generate the PSy code and the `algGen.Alg` class to generate the modified algorithm code.

Parameters

- **filename** (*str*) – The file containing the algorithm specification.
- **kernel_path** (*str*) – The directory from which to recursively search for the files containing the kernel source (if different from the location of the algorithm specification)

Returns The algorithm code and the psy code.

Return type ast

Raises IOError if the filename or search path do not exist

For example:

```
>>> from generator import generate
>>> psy,alg=generate("algspec.f90")
```

`parse.`**`parse`**(*alg_filename*, *api=''*, *invoke_name='invoke'*, *inf_name='inf'*, *kernel_path=''*)

Takes a GungHo algorithm specification as input and outputs an AST of this specification and an object containing information about the invocation calls in the algorithm specification and any associated kernel implementations.

Parameters

- **alg_filename** (*str*) – The file containing the algorithm specification.

- **invoke_name** (*str*) – The expected name of the invocation calls in the algorithm specification
- **inf_name** (*str*) – The expected module name of any required infrastructure routines.
- **kernel_path** (*str*) – The path to search for kernel source files (if different from the location of the algorithm source).

Return type ast,invoke_info

Raises

- **IOError** – if the filename or search path does not exist
- **ParseError** – if there is an error in the parsing
- **RuntimeError** – if there is an error in the parsing

For example:

```
>>> from parse import parse
>>> ast,info=parse("argspec.F90")
```

class transformations.**LoopFuseTrans**

class transformations.**SwapTrans**

A test transformation. This swaps two entries in a schedule. These entries must be siblings and next to eachother in the schedule.

For example:

```
>>> schedule=[please see schedule class for information]
>>> print schedule
>>> loop1=schedule.children[0]
>>> loop2=schedule.children[1]
>>> trans=SwapTrans()
>>> newSchedule,memento=SwapTrans.apply(loop1,loop2)
>>> print newSchedule
```

class transformations.**OpenMPLoop**

Adds an OMP directive to a loop.

For example:

```
>>> from parse import parse
>>> from psyGen import PSyFactory
>>> ast,invokeInfo=parse("dynamo.F90")
>>> psy=PSyFactory("dynamo0.1").create(invokeInfo)
>>> schedule=psy.invokes.get('invoke_v3_kernel_type').schedule
>>> schedule.view()
>>>
>>> from transformations import OpenMPLoop
>>> trans=OpenMPLoop()
>>> new_schedule,memento=trans.apply(schedule.children[0])
>>> new_schedule.view()
```

class transformations.**ColourTrans**

This module provides generic support for PSyclone's PSy code optimisation and generation. The classes in this method need to be specialised for a particular API and implementation.

class psyGen.**PSy**(*invoke_info*)

Manage and generate PSy code for a single algorithm file. Takes the invocation information output from the function `parse.parse()` as it's input and stores this is a way suitable for optimisation and code generation.

Parameters **invoke_info** (*FileInfo*) – An object containing the required invocation information for code optimisation and generation. Produced by the function `parse.parse()`.

For example:

```
>>> from parse import parse
>>> ast, info = parse("argspec.F90")
>>> from psyGen import PSy
>>> psy = PSy(info)
>>> print(psy.gen)
```

class `algGen.`**`Alg`**(*ast*, *psy*)

Generate a modified algorithm code for a single algorithm specification. Takes the ast of the algorithm specification output from the function `parse.parse()` and an instance of the `psyGen.PSy` class as input.

Parameters

- **ast** (*ast*) – An object containing an ast of the algorithm specification which was produced by the function `parse.parse()`.
- **psy** (*PSy*) – An object (`psyGen.PSy`) containing information about the PSy layer.

For example:

```
>>> from parse import parse
>>> ast,info=parse("argspec.F90")
>>> from psyGen import PSy
>>> psy=PSy(info)
>>> from algGen import Alg
>>> alg=Alg(ast,psy)
>>> print(alg.gen)
```

gen

Generate modified algorithm code

Return type ast

CHAPTER

THREE

STUB GENERATION

PSyclone provides a kernel stub generator for the dynamo0.3 API. The kernel stub generator takes a kernel file as input and outputs the kernel subroutine arguments and declarations. The word "stub" is used to indicate that it is only the subroutine arguments and their declarations that are generated; the subroutine has no content.

The primary reason the stub generator is useful is that it generates the correct Kernel subroutine arguments and declarations for the dynamo0.3 API as specified by the Kernel metadata. As the number of arguments to Kernel subroutines can become large and the arguments have to follow a particular order, it can become burdensome, and potentially error prone, for the user to have to work out the appropriate argument list if written by hand.

The stub generator can be used when creating a new Kernel. A Kernel can first be written to specify the required metadata and then the generator can be used to create the appropriate (empty) Kernel subroutine. The user can then fill in the content of the subroutine.

The stub generator can also be used to check whether the arguments for an existing Kernel are correct i.e. whether the Kernel subroutine and Kernel metadata are consistent. One example would be where a Kernel is updated resulting in a change to the metadata and subroutine arguments.

The dynamo0.3 API requires Kernels to conform to a set of rules which determine the required arguments and types for a particular Kernel. These rules are required as the generated PSy layer needs to know exactly how to call a Kernel. These rules are outlined in Section *Rules*.

Therefore PSyclone has been coded with the dynamo0.3 API rules which are then applied when reading the Kernel metadata to produce the require Kernel call and its arguments in the generated PSy layer. These same rules are used by the Kernel stub generator to produce Kernel subroutine stubs, thereby guaranteeing that Kernel calls from the PSy layer and the associated Kernel subroutines are consistent.

3.1 Use

Before using the stub generator, PSyclone must be installed. If you have not already done so, please follow the instructions for setting up PSyclone in Section *Getting Going*.

PSyclone will be installed in a particular location on your machine. For the remainder of this section the location where PSyclone is installed (including the PSyclone directory itself) will be referred to as `<PSYCLONEHOME>`.

The easiest way to use the stub generator is to use the supplied script called `genkernelstub.py`, which is located in the `src` directory:

```
> cd <PSYCLONEHOME>/src
> python ./genkernelstub.py
usage: genkernelstub.py [-h] [-o OUTFILE] [-api API] filename
genkernelstub.py: error: too few arguments
```

You can get information about the `genkernelstub.py` arguments using `-h` or `--help`:

```
>  python genkernelstub.py -h
usage: genkernelstub.py [-h] [-o OUTFILE] [-api API] filename

Create Kernel stub code from Kernel metadata

positional arguments:
  filename              Kernel metadata

optional arguments:
  -h, --help            show this help message and exit
  -o OUTFILE, --outfile OUTFILE
                        filename of output
  -api API              choose a particular api from ['dynamo0.3'], default
                        dynamo0.3
```

As is indicated when using the `-h` option, the `-api` option only accepts `dynamo0.3` at the moment and is redundant if this option is also the default. However the number of supported API's is expected to expand in the future.

The `-o`, or `--outfile` option allows the user specify that the output should be written to a particular file. If `-o` is not specified then the python `print` statement is used. Typically the print statement results in the output being printed to the terminal.

3.2 Kernels

Any dynamo0.3 kernel can be used as input to the stub generator. Example Kernels can be found in the dynamo repository or, for more simple cases, in the `tests/test_files/dynamo0p3` directory. In the latter directory the majority start with `testkern`. The exceptions are: `simple.f90`, `ru_kernel_mod.f90` and `matrix_vector_mm_mod.F90`. The following test kernels can be used to generate kernel stub code:

```
tests/test_files/dynamo0p3/testkern_chi_2.F90
tests/test_files/dynamo0p3/testkern_chi.F90
tests/test_files/dynamo0p3/testkern_operator_mod.f90
tests/test_files/dynamo0p3/testkern_operator_nofield_mod.f90
tests/test_files/dynamo0p3/testkern_orientation.F90
tests/test_files/dynamo0p3/ru_kernel_mod.f90
tests/test_files/dynamo0p3/simple.f90
```

3.3 Example

A simple single field example of a kernel that can be used as input for the stub generator is found in `tests/test_files/dynamo0p3/simple.f90` and is shown below:

```
module simple_mod
type, extends(kernel_type) :: simple_type
    type(arg_type), dimension(1) :: meta_args =  &
        (/ arg_type(gh_field,gh_write,w1) /)
    integer, parameter :: iterates_over = cells
  contains
    procedure() :: code => simple_code
end type simple_type
contains
subroutine simple_code()
```

```
end subroutine
end module simple_mod
```

Note: The module name `simple_mod` and the type name `simple_type` share the same root `simple` and have the extensions `_mod` and `_type` respectively. This is a convention in dynamo0.3 and is required by the kernel stub generator as it needs to determine the name of the type containing the metadata and infers this by reading the module name. If this rule is not followed the kernel stub generator will return with an error message (see Section *Errors*).

Note: Whilst strictly the kernel stub generator only requires the Kernel metadata to generate the appropriate stub code, the parser that the generator relies on currently requires a dummy kernel subroutine to exist.

If we run the kernel stub generator on the `simple.f90` example:

```
> python genkernelstub.py tests/test_files/dynamo0p3/simple.f90
```

we get the following kernel stub output:

```
MODULE simple_code_mod
  IMPLICIT NONE
  CONTAINS
  SUBROUTINE simple_code(nlayers, field_1_w1, ndf_w1, undf_w1, map_w1)
    USE constants_mod, ONLY: r_def
    INTEGER, intent(in) :: nlayers
    INTEGER, intent(in) :: undf_w1
    REAL(KIND=r_def), intent(out), dimension(undf_w1) :: field_1_w1
    INTEGER, intent(in) :: ndf_w1
    INTEGER, intent(in), dimension(ndf_w1) :: map_w1
  END SUBROUTINE simple_code
END MODULE simple_code_mod
```

The subroutine content can then be copied into the required module, used as the basis for a new module, or checked with an existing subroutine for correctness.

Note: The output does not currently conform to Met Office coding standards so must be modified accordingly.

Note: The code will not compile without a) providing the constants_mod module in the compiler include path and b) adding in code that writes to any arguments declared as intent `out` or `inout`. For a quick check, the `USE` declaration and `KIND` declarations can be removed and the `field_1_w1` array can be initialised with some value in the subroutine. At this point the Kernel should compile successfully.

Note: Whilst there is only one field declared in the metadata there are 5 arguments to the Kernel. The first argument `nlayers` specifies the number of layers in a column for a field. The second argument is the array associated with the field. The field array is dimensioned as the number of unique degrees of freedom (undf) which is also passed into the kernel (the fourth argument). The naming convention is to call each field a field, followed by it's position in the (algorithm) argument list (which is reflected in the metadata ordering). The third argument is the number of degrees of freedom for the particular column and is used to dimension the final argument which is the degrees of freedom map (dofmap) which indicates the location of the required values in the field array. The naming convention for the `dofmap`, `undf` and `ndf` is to append the name with the space that it is associated with.

We now take a look at a more complicated example. The metadata in this example is the same as an actual dynamo kernel, however the subroutine content and various comments have been removed. The metadata specifies that there

are four fields passed by the algorithm layer, the fourth of which is a vector field of size three. All three of the spaces require a basis function and the w0 and w2 function spaces additionally require a differential basis function. The content of the Kernel is given below.

```
module ru_kernel_mod
type, public, extends(kernel_type) :: ru_kernel_type
  private
  type(arg_type) :: meta_args(4) = (/                                    &
       arg_type(GH_FIELD,   GH_INC,  W2),                                &
       arg_type(GH_FIELD,   GH_READ, W3),                                &
       arg_type(GH_FIELD,   GH_READ, W0),                                &
       arg_type(GH_FIELD*3, GH_READ, W0)                                 &
       /)
  type(func_type) :: meta_funcs(3) = (/                                  &
       func_type(W2, GH_BASIS, GH_DIFF_BASIS),                           &
       func_type(W3, GH_BASIS),                                          &
       func_type(W0, GH_BASIS, GH_DIFF_BASIS)                            &
       /)
  integer :: iterates_over = CELLS
contains
  procedure, nopass ::ru_code
end type
contains
subroutine ru_code()
end subroutine ru_code
end module ru_kernel_mod
```

If we run the kernel stub generator on this example:

```
> python genkernelstub.py tests/test_files/dynamo0p3/ru_kernel_mod.f90
```

we obtain the following output:

```
MODULE ru_code_mod
  IMPLICIT NONE
  CONTAINS
  SUBROUTINE ru_code_code(nlayers, field_1_w2, field_2_w3, field_3_w0, field_4_w0_v1, field_4_w0_v2,
    USE constants_mod, ONLY: r_def
    INTEGER, intent(in) :: nlayers
    INTEGER, intent(in) :: undf_w2
    INTEGER, intent(in) :: undf_w3
    INTEGER, intent(in) :: undf_w0
    REAL(KIND=r_def), intent(inout), dimension(undf_w2) :: field_1_w2
    REAL(KIND=r_def), intent(in), dimension(undf_w3) :: field_2_w3
    REAL(KIND=r_def), intent(in), dimension(undf_w0) :: field_3_w0
    REAL(KIND=r_def), intent(in), dimension(undf_w0) :: field_4_w0_v1
    REAL(KIND=r_def), intent(in), dimension(undf_w0) :: field_4_w0_v2
    REAL(KIND=r_def), intent(in), dimension(undf_w0) :: field_4_w0_v3
    INTEGER, intent(in) :: ndf_w2
    INTEGER, intent(in), dimension(ndf_w2) :: map_w2
    REAL, intent(in), dimension(3,ndf_w2,nqp_h,nqp_v) :: basis_w2
    REAL, intent(in), dimension(1,ndf_w2,nqp_h,nqp_v) :: diff_basis_w2
    INTEGER, intent(in), dimension(ndf_w2,2) :: boundary_dofs_w2
    INTEGER, intent(in) :: ndf_w3
    INTEGER, intent(in), dimension(ndf_w3) :: map_w3
    REAL, intent(in), dimension(1,ndf_w3,nqp_h,nqp_v) :: basis_w3
    INTEGER, intent(in) :: ndf_w0
    INTEGER, intent(in), dimension(ndf_w0) :: map_w0
    REAL, intent(in), dimension(1,ndf_w0,nqp_h,nqp_v) :: basis_w0
```

```
    REAL, intent(in), dimension(3,ndf_w0,nqp_h,nqp_v) :: diff_basis_w0
    INTEGER, intent(in) :: nqp_h, nqp_v
    REAL(KIND=r_def), intent(in), dimension(nqp_h) :: wh
    REAL(KIND=r_def), intent(in), dimension(nqp_v) :: wv
  END SUBROUTINE ru_code_code
END MODULE ru_code_mod
```

The above example demonstrates that the argument list can get quite complex. Rather than going through an explanation of each argument you are referred to Section *Rules* for more details on the rules for argument types and argument ordering. Regarding naming conventions for arguments you can see that the arrays associated with the fields are labelled as 1-4 depending on their position in the metadata. For a vector field, each vector results in a different array. These are distinguished by appending `_vx` where `x` is the number of the vector.

3.4 Errors

The stub generator has been written to provide useful errors if mistakes are found. If you run the generator and it does not produce a useful error - and in particular if it produces a stack trace - please contact the PSyclone developers.

The following tests do not produce stub kernel code either because they are invalid or because they contain functionality that is not supported in the stub generator.

```
tests/test_files/dynamo0p3/matrix_vector_mm_mod.f90
tests/test_files/dynamo0p3/testkern_any_space_1_mod.f90
tests/test_files/dynamo0p3/testkern_any_space_2_mod.f90
tests/test_files/dynamo0p3/testkern.F90
tests/test_files/dynamo0p3/testkern_invalid_fortran.F90
tests/test_files/dynamo0p3/testkern_no_datatype.F90
tests/test_files/dynamo0p3/testkern_operator_orient_mod.f90
tests/test_files/dynamo0p3/testkern_qr.F90
tests/test_files/dynamo0p3/testkern_short_name.F90
```

`testkern_invalid_fortran.F90`, `testkern_no_datatype.F90`, `testkern_short_name.F90`, `testkern.F90` and `matrix_vector_mm_mod.f90` are designed to be invalid for PSyclone testing purposes and should produce appropriate errors. For example:

```
> python genkernelstub.py tests/test_files/dynamo0p3/testkern_invalid_fortran.F90
Error: 'Parse Error: Code appears to be invalid Fortran'
```

`any_space` is not currently supported in the stub generator so `testkern_any_space_1_mod.f90` and `testkern_any_space_2_mod.f90` should fail with appropriate warnings because of that. For example:

```
> python genkernelstub.py tests/test_files/dynamo0p3/testkern_any_space_1_mod.f90
Error: "Generation Error: Unknown space, expecting one of 'W0,W1,W2,W3' but found 'any_space_1'"
```

`testkern_operator_orient_mod.f90` and `testkern_qr.F90` use basis functions on function spaces that have not been used by dynamo at this point so there is no example of correct output to base the stub generator on. Therefore these examples should fail with appropriate warnings. For example:

```
> python genkernelstub.py tests/test_files/dynamo0p3/testkern_qr.F90
Error: "Generation Error: I don't know what dimension to use for a basis function in w1 space"
```

3.5 Rules

Kernel arguments follow a set of rules which have been specified for the dynamo0.3 API. These rules are encoded in the `_create_arg_list()` method within the `DynKern` class in the `dynamo0p3.py` file. The rules, along with

PSyclone's naming conventions, are:

1. If an operator is passed then include the `cells` argument. `cells` is an integer and has intent `in`.
2. Include `nlayers`, the number of layers in a column. `nlayers` is an integer and has intent `in`.
3. For each field/vector_field/operator in the order specified by the meta_args metadata.
   (a) if the current entry is a field then include the field array. The field array name is currently specified as being `"field_"<argument_position>"_"<field_function_space>`. A field array is a real array of type `r_def` and dimensioned as the unique degrees of freedom for the space that the field operates on. This value is passed in separately. The intent is determined from the metadata (see later for an explanation).
   (b) if the current entry is a field vector then for each dimension of the vector, include a field array. The field array name is specified as being using `"field_"<argument_position>"_"<field_function_space>"_v"<vector_position>`. A field array in a field vector is declared in the same way as a field array (described in the previous step).
   (c) if the current entry is an operator then first include a dimension size. This is an integer. The name of this size is `<operator_name>"_ncell_3d"`. Next include the operator. This is a real array of type `r_def` and is 3 dimensional. The first two dimensions are the local degrees of freedom for the `from` and `to` function spaces (currently in an unknown order). The third dimension is the dimension size mentioned before. At the moment the `from` and `to` function spaces must be the same in the generator. The name of the operator is `"op_"<argument_position>`. Again the intent is determined from the metadata and is explained later.
4. For each function space in the order they appear in the metadata arguments
   (a) Include the number of local degrees of freedom for the function space. This is an integer and has intent `in`. The name of this argument is `"ndf_"<field_function_space>`.
   (b) If there is a field on this space
      i. Include the unique number of degrees of freedom for the function space. This is an integer and has intent `in`. The name of this argument is `"undf_"<field_function_space>`.
      ii. Include the dofmap for this function space. This is an integer array with intent `in`. It has one dimension sized by the local degrees of freedom for the function space.
   (c) For each operation on the function space (`basis`, `diff_basis`, `orientation`) in the order specified in the metadata
      i. If it is a basis function, include the associated argument. This is a real array with intent `in`. It has four dimensions. The first dimension is 1 or 3 depending on the function space (w0=1,w1=?,w2=3,w3=1). The second dimension is the local degrees of freedom for the function space. The third argument is the quadrature rule size which is currently named `nqp_h` and the fourth argument is the quadrature rule size which is currently named `nqp_v`. The name of the argument is `"basis_"<field_function_space>`
      ii. If it is a differential basis function, include the associated argument. The sizes and dimensions are the same as the basis function except for the size of the first dimension which is sized as 1 or 3 depending on different function space rules (w0=3,w1=?,w2=1,w3=?). The name of the argument is `"diff_basis_"<field_function_space>`.
      iii. If is an orientation array, include the associated argument. The argument is a real array with intent `in`. There is one dimension of size the local degrees of freedom for the function space. The name of the array is `"orientation_"<field_function_space>`.
   (d) If the Kernel name is `ru_code` and the function space is `w2` then include a two dimensional integer array with intent `in`. The first dimension of this array is the local degrees of freedom and the second dimension is 2. The name of this array is boundary_dofs_w2. This is a kernel hack to support ru_kernel as boundary conditions are not currently dealt with properly. This hack will be removed in the next revision of the API.

5. if Quadrature is required (this is the case if any of the function spaces require a basis or differential basis function)
   (a) include `nqp_h`. This is an integer scalar with intent `in`.
   (b) include `nqp_v`. This is an integer scalar with intent `in`.
   (c) include `wh`. This is a real array of kind r_def with intent `in`. It has one dimension of size `nqp_h`.
   (d) include `wv`. This is a real array of kind r_def with intent `in`. It has one dimension of size `nqp_v`.

CHAPTER

FOUR

F2PY INSTALLATION

PSyclone requires version 3 of f2py, a library designed to allow fortran to be called from python (see http://code.google.com/p/f2py/wiki/F2PYDevelopment for more information). PSyclone makes use of the fortran parser (fparser) contained within.

The source code of f2py (revision 93) is provided with PSyclone in the sub-directory `f2py_93`. If you would prefer to install f2py rather than simply use it as is (see the previous section) then the rest of this section explains how to do this.

f2py uses the numpy distutils package to install. In version 1.6.1 of distutils (currently the default in Ubuntu) distutils supports interactive setup. In this case to install f2py using gfortran and gcc (for example) you can perform the following (where "cgcc", "fgfortran", "1" and "2" are interactive commands to setup.py)

```
> cd f2py_93
> sudo ./setup.py
cgcc
fgfortran
1
> sudo ./setup.py
cgcc
fgfortran
2
```

For later versions of distutils (1.8.0 has been tested) where the interactive setup has been disabled you can perform the following (using g95 and gcc in this case):

```
> cd f2py_93
> sudo ./setup.py build -fcompiler=g95 -ccompiler=gcc
> sudo ./setup.py install
```

For more information about possible build options you can use the available help:

```
> ./setup.py --help
> ./setup.py build --help
> ./setup.py build --help-fcompiler
```

In particular, if you do not have root access then the python modules can be installed in your user account by specifying –user to the install command:

```
> ./setup.py install --user
```

This causes the software to be installed under ${HOME}/.local/

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

INDEX

Symbols

A

C

G

L

O

P

S

T