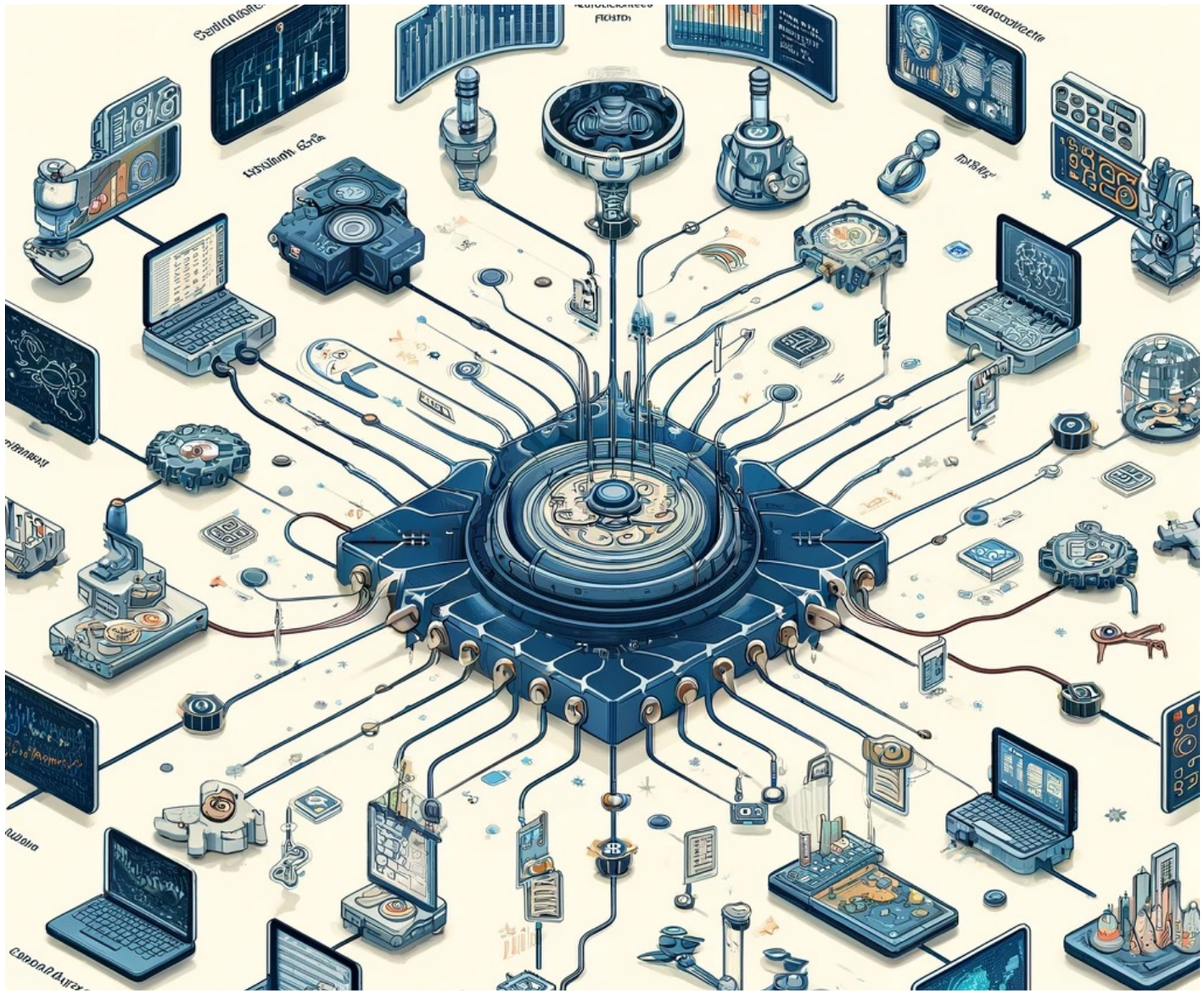
Exploring Scientific Workflows with CWL and dispel4py

Module 3.a

- Dr. Rosa Filgueira
- Lecturer at the School of Computer Science
- University of St Andrews
- rf208@st-andrews.ac.uk
- rosa.filgueira.vicente@gmail.com



Module 3.a – dispel4py Basic Concepts

- What is dispel4py
- What is a stream
- What is a processing element (PE)
- What I need for constructing a workflow
- Learning dispel4py by an example
- How to Implement a PE

What is dispel4py

- dispel4py for distributed **data-intensive applications**
- It enables **abstract description of methods**
- dispel4py maps to **multiple enactment** systems

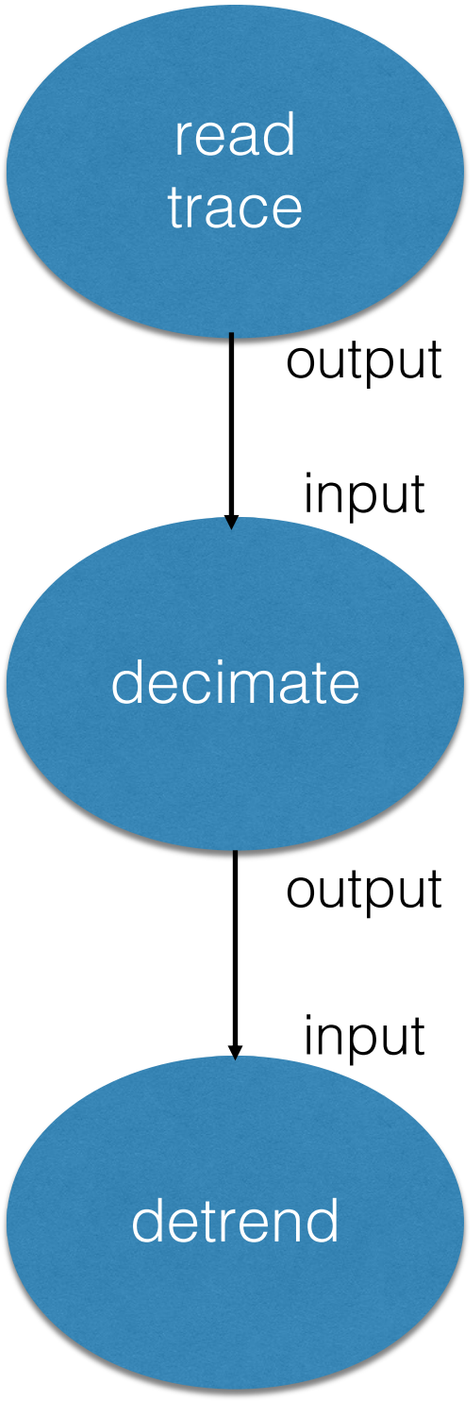
- Applications **scale automatically**
  - exploiting parallel processing, clusters, grids and clouds

- dispel4py is **dataflow-oriented**
  - rather than control-oriented
  - no explicit specification of data movement
  - light-weight composition of data operations

What is a data stream

- A **stream** is a sequence of data units:
  - from external source
  - between data operations - Processing Elements (PEs)
  - to external destination
- Flow of input or output data between PEs
- Processes data from a source and delivers data to one or more destinations

What is a processing element (PE)



- Computational activity encapsulates
  - algorithm
  - services
  - data transformation processes

- Basic computational elements of dispel4py workflows

- PEs have:
  - inputs & outputs
  - computational activity.
- PEs are connected by streams
  - saves computational costs

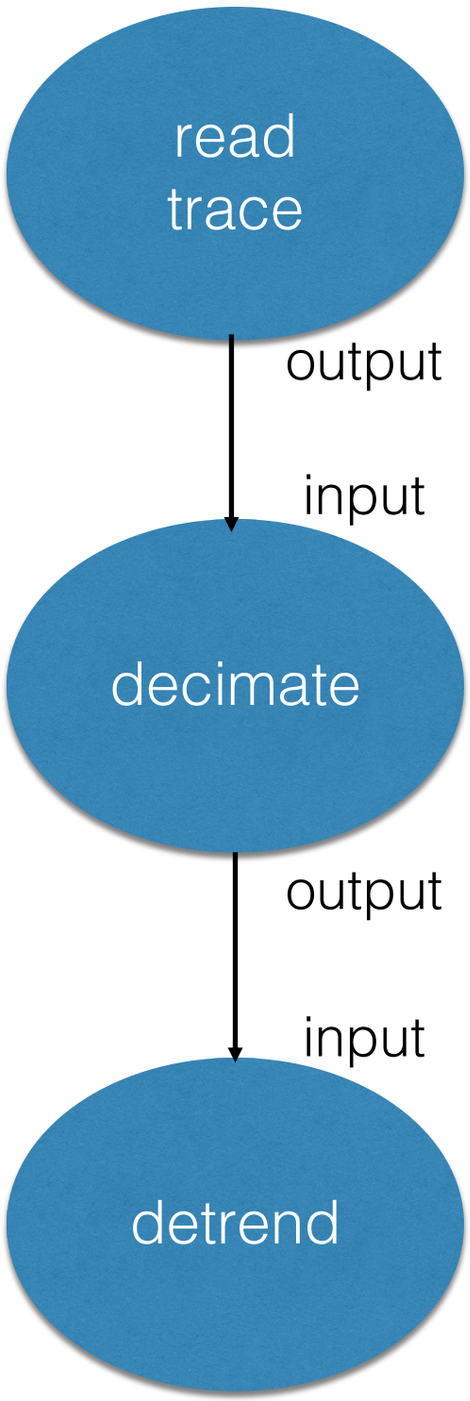
What is a graph

- How the PEs are **connected**
- How data is **streamed**
- The **topology** of the data flow
- **No limitations** on the type of graphs

What I need for constructing a dispel4py workflow

- You **only** have to **implement PEs** (in Python) and connect them:
  - Learn how to implement PEs.
  - Learn how to connect them.

Learning dispel4py by an example



- dispel4py workflow that reads a trace
- decimate the trace
- detrend the trace

How to implement a PE

- Each PE specifies:
  - input & output connections
  - computational activity for processing data units
  - implement the “_process” method.

Types of PEs

Type	Inputs	Outputs	When to use it
GenericPE	*n* inputs	*m* outputs	many inputs and/or many outputs
IterativePE	1 input named 'input'	1 output named 'output'	process one and produce one data unit in each iteration
ConsumerPE	1 input named 'input'	no output	no output and one input
ProducerPE	no input	1 input named 'output'	no inputs and one output; usually the root in a graph
SimpleFunctionPE	1 input named 'input'	1 output named 'output'	only implement _process method; it can not store state between calls
create_iterative_chain	1 input named 'input'	1 output named 'output'	pipeline of functions processing sequentially; creates a composite PE

GenericPE example

```
from dispel4py.core import GenericPE

class StreamAndStatsProducer(GenericPE):
    def __init__(self):
        GenericPE.__init__(self)
        self._add_input('input')
        self._add_output('output')
        self._add_output('output_stats')

    def process(self, inputs):
        data = inputs['input']
        filename = data
        st = read(filename)
       return {'output': st, 'output_stats': st[0].stats}
```

This PE also reads a file that contains seismological traces and returns two outputs: obspy stream and metadata.

What we have learnt:

- We can add several outputs with different names
- The process method gets values from the input streams
- The process method returns both streams

IterativePE example

```
from dispel4py.base import IterativePE
from obspy.core import read

class ReadTrace(IterativePE):
    def __init__(self):
        IterativePE.__init__(self)
    def _process(self, data):
        filename = data
        st = read(filename)
        return st
```

This PE receives a filename (‘input’), reads the obspy trace from the file, and writes it to the output (‘output’)

What we have learnt:

- We don’t need to specify the input and output
- The parameter to the _process method is the data (the filename)
- _process returns the value that is written to the output stream

SimpleFunctionPE example

```
from dispel4py.base import SimpleFunctionPE

def decimate(st, sps):
    st.decimate(int(st[0].stats.sampling_rate/sps))
    return st

decimate = SimpleFunctionPE(decimate, {‘sps’: 4})
```

This PE will emit a decimated trace.

What we have learnt:

- Only implement the processing function
- The easiest but the most restrictive way
- 1 input called ‘input’, 1 output called ‘output’.

ConsumerPE example

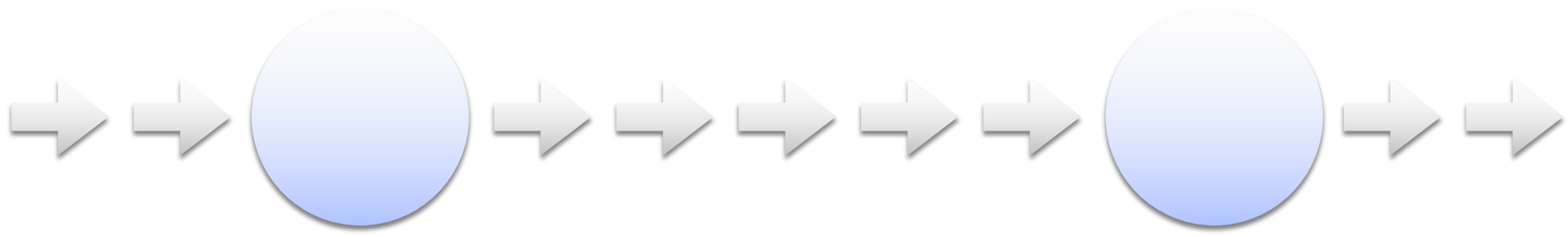
```
from dispel4py.base import ConsumerPE

class Detrend(ConsumerPE):
    def __init__(self):
        ConsumerPE.__init__(self)
    def _process(self, st):
        st.detrend('simple')
        st.write(st[0].getId() + '.mseed', 'MSEED')
```

This PE receives one decimated trace, applies detrend and writes it to a MSEED file.

What we have learnt:

- We don't need to specify the input
- It does not return any output.

How to connect PEs: What does it mean?



- PEs process a small amount of data at a time
- Data need not be explicitly stored
- PEs may store a small amount of result data (e.g. stacking) or big amount (if you have the resources)

How to connect PEs:

Create the PEs

```
readtrace = ReadTrace()
decimate = SimpleFunctionPE(decimate, {‘sps’: 4})
detrend = Detrend()
```

Create the graph and connect the PEs

```
from dispel4py.workflow_graph import WorkflowGraph

graph = WorkflowGraph()
graph.connect(readtrace, 'output', decimate, ‘input')
graph.connect(decimate, 'output', detrend, 'input')
```

Example- Summary

```
from dispel4py.base import IterativePE, ConsumerPE, SimpleFunctionPE
from dispel4py.workflow_graph import WorkflowGraph
from obspy.core import read

class ReadTrace(IterativePE):
    def __init__(self):
        IterativePE.__init__(self)
    def _process(self, data):
        filename = data
        st = read(filename)
        return st

def decimate(st, sps):
    st.decimate(int(st[0].stats.sampling_rate/sps))
    return st

class Detrend(ConsumerPE):
    def __init__(self):
        ConsumerPE.__init__(self)
    def _process(self, st):
        st.detrend('simple')
     st.write(st[0].getId() + '.mseed', 'MSEED')

readtrace = ReadTrace()
decimate = SimpleFunctionPE(decimate, {'sps': 4})
detrend = Detrend()

graph = WorkflowGraph()
graph.connect(readtrace, 'output', decimate, 'input')
graph.connect(decimate, 'output', detrend, 'input')
```