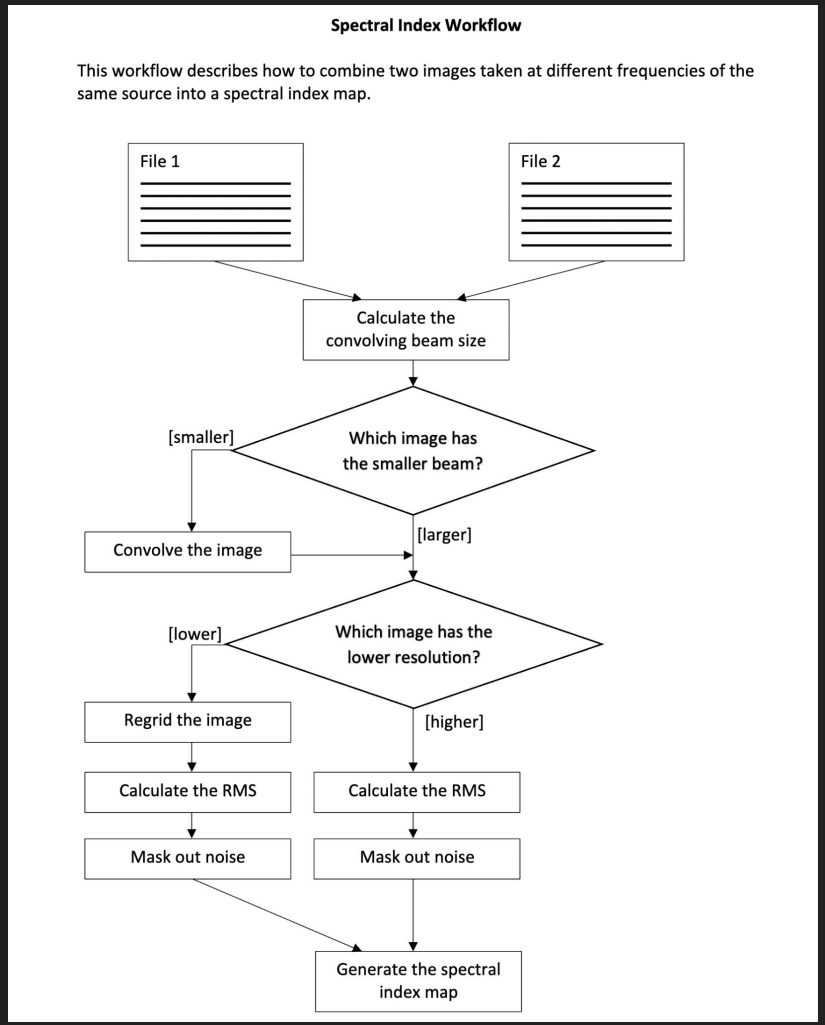
# Tangerine team update (SKA Science Platform Prototyping)

## Vision and Progress

Arpan Das(EPFL)
Rohit Sharma (FHNW)




- Brief overview of how existing workflow packages run and how the integration tools supplied by each workflow
- Not to give an in depth functional analysis of each workflow package
- This analysis is designed to give a more technical overview of how each workflow package could be used within the SRC Net

**The packages studied are:**

- Nextflow
- Snakemake
- Daliuge
- CWL
- WDL
- Ruffus
- Yadage
- Stimela

# Nextflow:

NextFlow uses a scripting language similar to CWL, based upon a language called 'groovy' (an extension of java), and provides out-of-the-box executors for GridEngine, SLURM, LSF, PBS, Moab and HTCondor batch schedulers, and for Kubernetes, Amazon AWS, Google Cloud, and MS Azure platforms.

The NextFlow application offers the following commands:

- clean - clean up a project cache and work directories
- clone - clone a project into a folder
- config - print a project configuration
- console - launch NextFlow interactive console
- drop - delete the local copy of a project
- help - print the usage help for a command
- info - print project and system runtime information
- kuberun - execute a workflow in a Kubernetes cluster (experimental)
- list - list all downloaded projects
- log - print executions log and runtime info
- pull - download or update a project
- run - execute a pipeline project
- self-update - update NextFlow runtime to the latest available version
- view - view project script file(s)

# Basic Concepts

- Designed around Linux Platform for data science applications
- Linux provide many simple but powerful CL and scripting tools, when chained together facilitate complex data manipulations
- Nextflow extends this approach, adding ability to define complex program and a high-level parallel computational environment based on dataflow programming model

**Nextflow Pipeline Script are made by  joining different processes**

- Processes are executed independently and isolated from one another
- Each process can be written in any scriptable language which is executable on Linux platform (Bash, Perl, Ruby, Python etc..)
- They do share a common writable state
- The processes can communicate via asynchronous queues, called channels
- Any Process can define one or more channels as input and output

# Features

Nextflow is built around the idea that Linux is the *lingua franca* of data science.

## Fast prototyping

Nextflow allows you to write a computational pipeline by making it simpler to put together many different tasks.

You may reuse your existing scripts and tools and you don't need to learn a new language or API to start using it.

## Reproducibility

Nextflow supports Docker and Singularity containers technology.

This, along with the integration of the GitHub code sharing platform, allows you to write self-contained pipelines, manage versions and to rapidly reproduce any former configuration.

## Portable

Nextflow provides an abstraction layer between your pipeline's logic and the execution layer, so that it can be executed on multiple platforms without it changing.

It provides out of the box executors for GridEngine, SLURM, LSF, PBS, Moab and HTCondor batch schedulers and for Kubernetes, Amazon AWS, Google Cloud and Microsoft Azure platforms.

## Unified parallelism

Nextflow is based on the *dataflow* programming model which greatly simplifies writing complex distributed pipelines.

Parallelisation is implicitly defined by the processes input and output declarations. The resulting applications are inherently parallel and can scale-up or scale-out, transparently, without having to adapt to a specific platform architecture.

## Continuous checkpoints

All the intermediate results produced during the pipeline execution are automatically tracked.

This allows you to resume its execution, from the last successfully executed step, no matter what the reason was for it stopping.

## Stream oriented

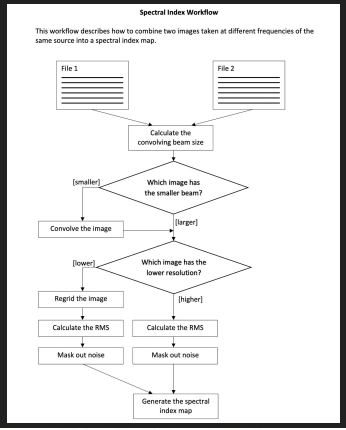
Nextflow extends the Unix pipes model with a fluent DSL, allowing you to handle complex stream interactions easily.

It promotes a programming approach, based on functional composition, that results in resilient and easily reproducible pipelines.

```
// Declare syntax version
nextflow.enable.dsl=2
// Script parameters
params.query = "/some/data/sample.fa"
params.db = "/some/path/pdb"

process processOne {
   input:
      path query
      path db
   output:
      path "pathOne.txt"

   // process is defined here.
   """
   """
}

process processTwo {
   input:
      path in_path
   output:
      path "pathTwo.txt"

   // process is defined here.
   """
   """
}

workflow {
   def query_ch = Channel.fromPath( params.query )
   processOne( query_ch, params.db ) | processTwo | view
}
```

- The pipe between the two processes forwards the output from one process to the inputs of the following one
- Creates two processes (one for each function) and a channel (query_ch)
- Both processes will be started at the same time and they will listen to their respective input channels
- When processOne emit a value, processTwo will receive it
- Inside the triple quotes the user writes a bash script, which can kick off a script in python, R, or other languages
- To use something other than bash, the first line of the script should be #!/usr/bin/perl, or #!/usr/bin/python, etc
- To submit jobs via SLURM the process.executor in the nextflow.config file should be 'slurm'
- NextFlow integrates with github, and if a pipeline is not found locally by the executor it will automatically look for a public github repository with the same name, download it, and execute it.

# Spectral Index Workflow




```
MEASUREMENT_SET = file( '/measurement-sets/3c391_ctm_mosaic_spw0.ms' )
SCRIPT_IMAGE_0 = file( '/scripts/make_image_channel.py' )
SCRIPT_IMAGE_63 = file( '/scripts/make_image_channel.py' )
SCRIPT_BEAM_SIZE_0 = file( '/scripts/get_beam_size.py' )
SCRIPT_BEAM_SIZE_63 = file( '/scripts/get_beam_size.py' )
SCRIPT_SPECTRAL_INDEX = file( '/scripts/make_spectral_index_map.py' )
SCRIPT_EXPORT_FITS = file( '/scripts/export_fits.py' )
SCRIPT_CONVOLVE = file( '/scripts/convolve.py' )

// --------------------------------
//
//      processes
//
// --------------------------------

//
//      make_image_channel_0
//
//      Modified:       26/04/2023
//
//      Make a casa image from channel 0.
//

process make_image_channel_0
{

        container 'amigahub/casa:6.5.2'

        input:
                val measurement_set

        // NOTE: the emit keywords doesn't appear to work for type file, so I have to use type path.
        output:
        //      file '*.image', emit: clean_image_0
                path '3c391_ctm_spw_0_ch_0_I.image', emit: clean_image

        """
        docker run -v /home/tangerine/scripts:/scripts -v /home/tangerine/measurement-sets:/measurement-sets -v
        """

} // make_image_channel_0

//
//      make_image_channel_63
//
//      Modified:       26/04/2023
//
//      Make a casa image from channel 63.
//
```

```
process get_convolution_beam_size
{

        // NOTE: for some reason I have to make these variables of type val instead of path or file, or otherwi
        //       does not pull in the files from the previous working directory.
        input:
                val beam_size_0_path
                val beam_size_1_path

        // NOTE: the emit keywords doesn't appear to work for type file, so I have to use type path.
        output:
//              path 'beam-size', emit: beam_size
                val beam_size

        // NOTE: the exec block runs groovy script. the command task.workDir.resolve() doesn't work inside a sc
        //       strangely, inside an exec or script block the working directory is the directory from which Ne
        //       bash script (bounded by three double quotes) the working directory is the task directory of th
        //       task.workDir.resolve() to add the full path to the task directory.
        exec:

                // get the beam sizes from the files.
                beam_size_0 = file( task.workDir.resolve( beam_size_0_path ) ).text.toFloat()
                beam_size_1 = file( task.workDir.resolve( beam_size_1_path ) ).text.toFloat()

                // which one is the larger? update the return value.
                beam_size = beam_size_0
                if (beam_size_1 > beam_size_0)
                {
                        beam_size = beam_size_1
                }

                // increase the beam size by 2% so that we can do the convolution.
                beam_size = beam_size * 1.02

                // NOTE: groovy has a max() function, but I couldn't get it to work.
                //beam_size = max( beam_size_0, beam_size_1 )

                // write to an output file.
                // NOTE: using a bash script that does 'echo ... >> beam-size' doesn't work for some reason. th
//                task.workDir.resolve( 'beam-size' ).text = beam_size.toString()

                println "convolving to a circular beam of size " + beam_size.toString() + " arcsec"

}
```

```
process export_fits
{

        container 'amigahub/casa:6.5.2'
        stageInMode 'copy'

        input:
                file casa_map

        output:
                file '3c391_spectral_index.fits'

        //docker run -v /home/tangerine/scripts:/scripts -v /home/tangerine/workflows/work/58/7153760ab74edc39d
        """
        docker run -v /home/tangerine/scripts:/scripts -v .:/work amigahub/casa:6.5.2 ./casa-6.5.2-26-py3.6/bin/
        """

} // export_fits

//
//      copy_required_images
//
//      Modified:       26/04/2023
//
//      Copy the casa and fits maps to the output folder (~/images/).
//

process copy_required_images
{

        input:
                file casa_map
                file fits_map

        """
        cp -r $casa_map /home/tangerine/images/
        cp $fits_map /home/tangerine/images/
        """

} // copy_required_images

// --------------------------------
//
//      workflows
//
// --------------------------------

//
//      Modified:       26/04/2023
//
//      Construct a spectral index map from channels 0 and 63, and export the casa image to fits format.
//      Both the casa and fits maps are written to the ~/images/ directory.
//
```

Nextflow Workflow

```
workflow
{

        // make images from channels 0 and 63 (in parallel)
        make_image_channel_0( MEASUREMENT_SET )
        make_image_channel_63( MEASUREMENT_SET )

        // get the major and minor axes of the psfs.
        //
        //      majorAxis = imhead( imagename = 'image.image', mode = 'get', hdkey = 'bmaj' )
        //      minorAxis = imhead( imagename = 'image.image', mode = 'get', hdkey = 'bmin' )
        //
        // the returned format is {'unit': 'arcsec', 'value': 18.57394027709961}, so to get the value and unit
        //
        //      value = qa.getvalue( majorAxis )
        //      unit = qa.getunit( majorAxis )

        // get the major axis restoring beam sizes for these images.
        get_beam_size_0( make_image_channel_0.out.clean_image )
        get_beam_size_63( make_image_channel_63.out.clean_image )

        // get the larger of the two beam sizes.
        beam_size = get_convolution_beam_size( get_beam_size_0.out.beam_size, get_beam_size_63.out.beam_size )

        // convolve image 0 with a Gaussian.
        convolve_0( make_image_channel_0.out.clean_image, beam_size )

        // convolve image 1 with a Gaussian.
        convolve_63( make_image_channel_63.out.clean_image, beam_size )

        // make a spectral index map from our two images.
        map_spectral_index = make_spectral_index_map( convolve_0.out.smooth_image, convolve_63.out.smooth_image
        map_fits = export_fits( map_spectral_index )

        // copy any files we need to the images directory.
        copy_required_images( map_spectral_index, map_fits )

} // workflow
```

# Stimela:

Stimela (the IsiZulu word for a train) is a platform independent radio interferometry scripting framework based on Python and the containerization technologies that now comes standard with all major Linux distributions.

Currently supports:

- Podman
- Docker
- Singularity
- uDocker

- In this framework, radio interferometry related tasks such as imaging, calibration and data synthesis are executed in containers.
- The packages that perform these tasks are Python modules.
- Stimela does not do any data processing, synthesis or analysis but offers a unified Pythonic interface to packages that perform these tasks.

The primary aims of Stimela is to provide the following services to the Radio Astronomy community:

- A user friendly and modular scripting environment that gives general users easy access to novel radio interferometry calibration, imaging, and synthesis packages.

- Simplified installation and production deployment. All the software available to the Stimela user is prebuild and available on Docker Hub.

The production environment is fixed. The versions of the interfaced software products is fixed for a particular release version of Stimela. This means one can roll back and forward with ease and ensures that the production environment is always verbatim with that used for the original reduction work.

Stimela is centred around two sets of images:

- Base images, which have the required software tools installed in them. The base images can either be built locally (on the host machine) or pulled from Docker hub.

- Very light weight executor images (a.k.a cab images) based on the base images, these perform radio interferometry related tasks like imaging, data synthesis, and calibration. The executor images are built locally.

Base images:

- stimela/meqtrees - MeqTrees calibration/simulation tool
- stimela/lwimager - Uses the casarest based lwimager tool for imaging and deconvolution
- stimela/wsclean - WSClean imaging tool
- stimela/simms - Uses CASA simulate tool to create a simulated (empty) MS
- stimela/tigger - Tools for managing and manupulating analytic sky models (Gaussian and point sources)
- stimela/aoflagger - Automated RFI flagging tool
- stimela/casa - CASA
- stimela/lofar - Lofar
- stimela/sourcery - Source finding and source characterisation tool
- stimela/msutils - Convenience functions for manipulating MSs

Executors (a.k.a 'cabs'):

These images are generally pre-loaded with Python scripts that perform a specified task (e.g calibrating a visibility dataset). A stimela cab image takes some input as well a set of instructions, performs some task, then returns the output. The following are examples of available tasks:

- cab/simms
- cab/simulator
- cab/calibrator
- cab/lwimager
- cab/wsclean
- cab/tigger_convert
- cab/tigger_restore
- cab/tigger_tag
- cab/specfit
- cab/sourcery
- cab/autoflagger (AOFlagger)
- cab/flagms
- cab/casa_{clean, gaincal, bandpass, etc.}
- cab/ddfacet
- cab/cubical
- cab/tricolour (Tricolour)

```python
# import stimela package
import stimela
import os

# Recipe I/O configuration
INPUT = "input"  # This folder must exist
OUTPUT = "output"
MSDIR = "msdir"
PREFIX = "stimela-example"  # Prefix for output images
try:
    SINGULARTITY_IMAGE_DIR = os.environ["STIMELA_SINGULARTITY_IMAGE_DIR"]
except KeyError:
    SINGULARTITY_IMAGE_DIR = None

# MS name
MS = "meerkat_simulation_example.ms"

# Use the NVSS skymodel. This is natively available
LSM = "nvss1deg.lsm.html"

# Start stimela Recipe instance
pipeline = stimela.Recipe("Simulation Example",     # Recipe name
                          ms_dir=MSDIR,
                          indir=INPUT,
                          outdir=OUTPUT,
                          singularity_image_dir=SINGULARTITY_IMAGE_DIR,
                          log_dir=os.path.join(OUTPUT, "logs"),
                          )

pipeline.JOB_TYPE = "singularity"

# 1: Make empty MS
pipeline.add("cab/simms",                   # Executor image to start container from
             "simms_example",               # Container name
             {  # Parameters to parse to executor container
                "msname":   MS,
                "telescope":   "meerkat",             # Telescope name
                "direction":   "J2000,0deg,-30deg",    # Phase tracking centre of observation
                "synthesis":   2,                 # Synthesis time of observation
                "dtime":   30,                     # Integration time in seconds
                "freq0":   "750MHz",              # Start frequency of observation
                "dfreq":   "1MHz",                # Channel width
                "nchan":   16                      # Number of channels
             },
             label="Creating MS",                       # Process label
             cpus=2.5,
             memory_limit="2gb")
# 2
pipeline.add("cab/casa_listobs", "obsinfo",
        {
            "vis" : MS,
            "listfile" : MS + "-obsinfo.txt",
            "overwrite": True,
            },
        label="obsinfo:: Observation information")
```

Hint: double-click to select code

```python
# 3: Simulate visibilities into it
pipeline.add("cab/simulator",
             "simulator_example",
             {
                 "msname":   MS,
                 "skymodel":   LSM,                    # Sky model to simulate into MS
                 "addnoise":   True,                   # Add thermal noise to visibilities
                 "column":   "DATA",
                 "Gjones": True, # Simulated data will be saved in this column
                 "sefd":   831,                        # Compute noise from this SEFD
                 # Recentre sky model to phase tracking centre of MS
                 "tile-size": 64,
                 "threads": 4,
             },
             label="Simulating visibilities")
# 4
pipeline.add("cab/calibrator",
             "cal_example",
             {
                 "msname":   MS,
                 "skymodel":   LSM,
                 "tile-size": 64,
                 "threads": 4,
                 "output-data" : 'CORR_DATA',
             },
             label="Calibrating visibilities")



# 5,6,7 : Image
# Make things a bit interesting by imaging with different weights
# Briggs robust values to use for each image
briggs_robust = [2, 0, -2]

for i, robust in enumerate(briggs_robust):

    pipeline.add("cab/wsclean",
                 "imager_example_robust_{:d}".format(i),
                 {
                     "msname":   MS,
                     "weight":   f"briggs {robust}",
                     "prefix":   "{:s}_robust-{:d}".format(PREFIX, robust),
                     "npix":   4096,                    # Image size in pixels
                     "cellsize":   2,                       # Size of each square pixel
                     # Perform 1000 iterarions of clean (Deconvolution)
                     "niter":   5000,
                     "mgain" : 0.85,
                     "pol" : "I",
                     "multiscale": True,
                     "multiscale-scales" : [0,2],
                 },
                 label="Imaging MS, robust={:d}".format(robust),
                 cpus=2,
                 memory_limit="2gb")
# 8
pipeline.add("cab/casa_rmtables", "delete_ms", {
    "tablenames": MS + ":msfile",
},
    label="Remove MS")

pipeline.run()
```

Hint: double-click to select code

# SRC Science Analysis Platform Vision Document

de Boer, J.[1], Cimpan, I.[2], Das, A.[3], Fabbro, S.[4], Grange, Y. G.[1*], Hardcastle, M. J.[5], Sharma, R.[6], Skipper, C. J.[2], Swinbank, J. D.[1], Webster, B.[5]

[1] *ASTRON, the Netherlands Institute for Radio Astronomy, Oude Hoogeveensedijk 4,7991 PD Dwingeloo, The Netherlands*
[2] *Jodrell Bank Centre for Astrophysics, Alan Turing Building, The University of Manchester, Manchester, M13 9PL, UK*
[3] *École polytechnique fédérale de Lausanne, Rte Cantonale, 1015 Lausanne, Switzerland*
[4] *NRC Herzberg Astronomy and Astrophysics, 5071 West Saanich Road, Victoria, BC V9E 2E7, Canada*
[5] *Centre for Astrophysics Research, University of Hertfordshire, College Lane, Hatfield, AL10 9AB, UK*
[6] *Fachhochschule Nordwestschweiz, Bahnhofstrasse 6, 5210 Windisch, Switzerland*

*corresponding author

**Abstract.** This document describes the vision for the Square Kilometer Array (SKA) Regional Centres (SRC) Science Analysis Platform. It is intended to set the broad terms of reference for the platform and to provide guidance for both development teams and other stakeholders. Among the features and services that are expected to be included are data querying and discovery tools, some form of notebook interface, user-managed software environments, workflow management, and a comprehensive set of application programming interfaces (APIs) enabling access to all low-level platform functionality. This document is not a design specification, and the features and services described herein will be further refined, or could be discarded, at a later stage of development.

- Implementing external feedbacks in progress
- ~ 100 comments