Hands-on SPECULA, a python simulation tool for Adaptive Optics in the ELT era

Fabio Rossi: fabio.rossi@inaf.it

Introduction

- Developed by: Fabio Rossi, Alfio Puglisi, and Guido Agapito,
  - Adaptive Optics (AO) group at the Arcetri Observatory (INAF)
  - more contributors joining, see our upcoming abstract for SPIE 2026
- Heritage: SPECULA is the modern, Python-based evolution of its predecessor, PASSATA (an IDL and CUDA-based object-oriented framework).
- Citation: Rossi, F., Puglisi, A., & Agapito, G. (2026). Introducing a new generation adaptive optics simulation framework: from PASSATA to SPECULA. *Journal of Astronomical Telescopes, Instruments, and Systems*, *12*(1), 019001. https://doi.org/10.1117/1.JATIS.12.1.019001
- Repository and documentation:
  - GitHub: https://github.com/ArcetriAdaptiveOptics/SPECULA
  - PyPI: https://pypi.org/project/specula/
  - Documentation: https://specula.readthedocs.io/

SPECULA: High-Fidelity AO Simulation for ELT Exoplanet Science

Bridging the gap between Atmospheric Physics and Instrument Simulators

- From Snapshots to **Time-Series** (GPU Accelerated)
  - Simulating ELT-scale systems on CPUs is often too slow for High Contrast Imaging (HCI).
  - SPECULA uses GPUs to generate long time-series (seconds/minutes) to characterize quasi-static speckles and test ADI/PCA techniques.
- The **"Ground Truth"** for Data Reduction
  - Provides physically accurate Residual Electric Fields (Phase & Amplitude).
  - Enables the injection of synthetic planets (in post-processing) into realistic residuals to validate detection limits and pipeline robustness.
- Enabling **PSF Reconstruction** (PSF-R)
  - Exports full, synchronized Telemetry (WFS slopes, DM commands, Actuator positions).
  - Ideal environment for developing and calibrating PSF-R algorithms without telescope time.
- Interface & Interoperability
  - Outputs complex fields ready to be ingested by instrument and detector simulators (e.g. ScopeSim).
  - Note: A library of several standard **coronagraph** classes is already included in the framework.

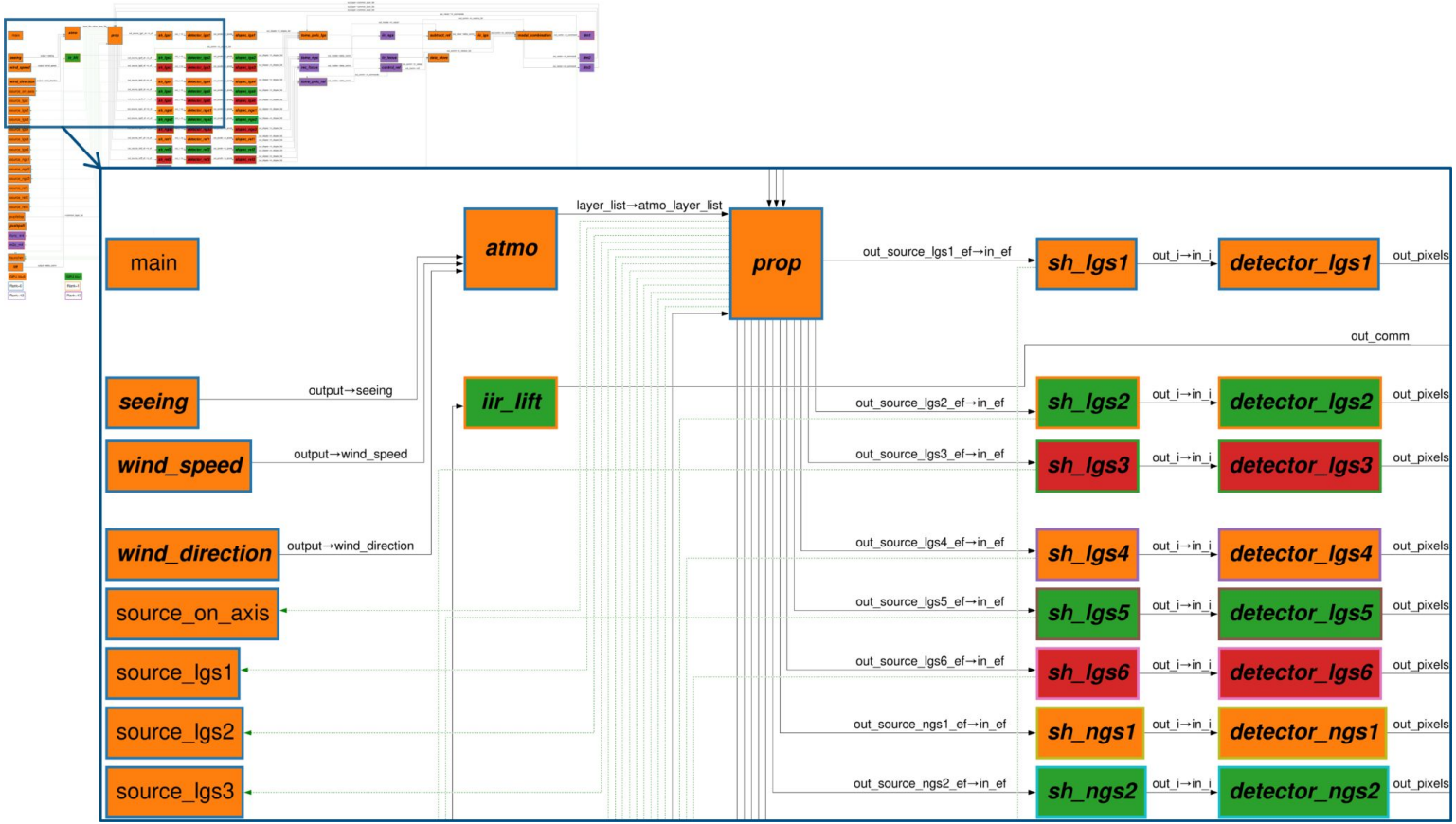
Key features

- Like PASSATA, it is an End-To-End simulation tool
  - what does this mean?
- Fully coded in Python, based on numpy/cupy for agnostic platform (CPU/GPU) coding
- Focus on GPU optimization for the exports: Single/double precision cuda graphs, streams (optional)
- Simulations are described rather than coded (yaml files)
  - objects configurations and interconnections
- Simulation results can be saved and later “replayed”
- HPC support
  - tested on Leonardo supercomputer, up to 7 nodes and 28 GPU used at once
- Hybrid simulations (hardware in the loop) support
  - tested at Arcetri AO labs

It’s an open source project, contributors are welcome

- Maybe your technological or scientific project needs something which is not there yet!
  - You are designing a novel Wavefront Sensor
  - You are developing a new algorithm for PSF reconstruction
  - You have a new device to experiment with
- Prerequisites for devs
  - Object oriented Python programming
  - Numpy, optionally Cupy
  - General understanding of Adaptive Optics loop concepts and components
- Fork the Github repo, start developing and issuing pull requests
- But again, you don’t need to develop Python code to use SPECULA!

What is a Simulation for us?

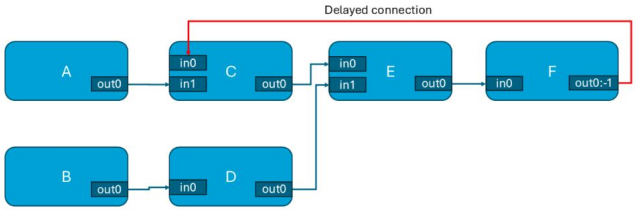


What is a Simulation for us?

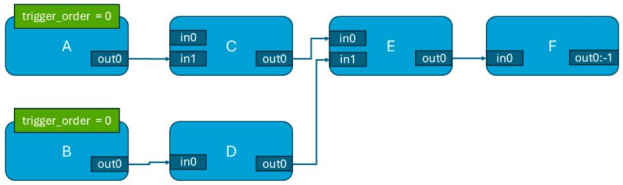
- A graph of blocks (Processing Objects) triggered (executed) at each time step, in a given order, to perform some computation, possibly reading the values of some inputs, possibly updating the values of some outputs
  - Our infrastructure could be used for simulations that have nothing to do with AO
- Inputs and outputs are instances of Data Objects
- Parameters can be python/numpy/cupy data types or Data Objects
- Inputs and outputs can change at each timestep, Parameters are static
- To close a loop in the graph avoiding circular dependencies, a “one step” delay must be introduced (more about this later)

Trigger order

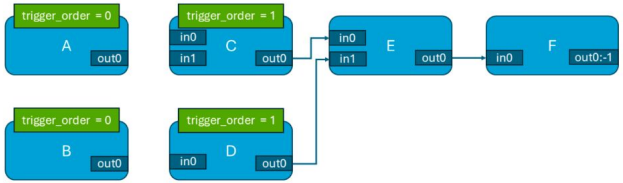
- In SPECULA the trigger order is determined automatically from the graph structure defined by the user in the config file



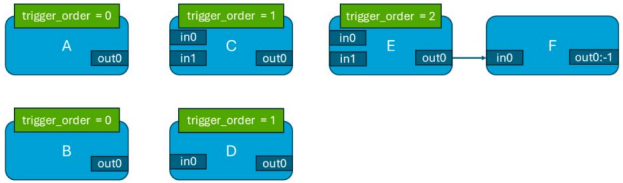
(a) Initial simulation graph.



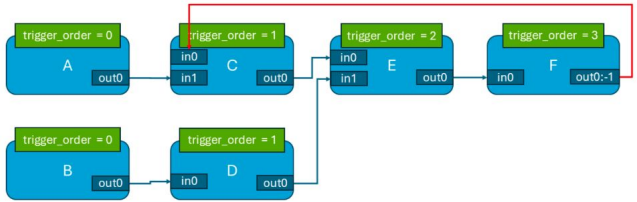
(b) Delayed connections are removed, leaves get the initial trigger order (0).



(c) Leaves are removed, trigger_order is incremented, new leaves get assigned the current trigger_order.



(d) Leaves are removed, trigger_order is incremented, new leaves get assigned the current trigger_order.



(e) Leaves are removed, trigger_order is incremented, new leaves get assigned the current trigger_order. All the connections are restored.

Objects categories: Data objects

- in the folder specula/data_objects
- classes used to represents the inputs, outputs or parameters of processing objects
- data can be read/written from disk (fits files, think about calibration data)
- base class: BaseDataObject, which is derived from BaseTimeObject (the info about generation time is attached!)
    - BaseValue
    - Source
    - IFunc
    - ElectricField
        - Layer
        - PupilStop
    - Intensity
    - Pixels
    - PupData
    - SubapData
    - Lenslet
    - ConvolutionKernel
        - GaussianConvolutionKernel
    - Slopes
    - IIRFilterData
    - M2C
    - Intmat
    - Recmat

Objects categories: Processing objects (1)

- in the folder specula/processing_objects
- some examples are:

- IFunc (Function Generator)
- BaseOperation (aritmetic, concatenation etc)
- Atmosphere simulation
  - AtmoEvolution (precomputed phase screens)
  - InfiniteAtmoEvolution
  - AtmoRandomPhase
- AtmoPropagation
- Ccd
- Saving and replaying of simulation data:
  - DataStore
  - DataSource
- DM
- Controllers
  - IntegratorController
  - IIRControl
- ModalAnalysis
- ModalRec
  - POLC or no
- WFSs
  - ModulatedPyramid
  - SH
- PSF
- Slope Computers
  - PyramidSlopeComputer
  - SHSlopeComputer
- Calibrators
  - ImRecCalibrator
  - ShSubapCalibrator
- Several Plots and images display

Objects categories: Processing objects (2)

- the computation part is written in the “trigger” method
- the trigger method is the computationally intensive part of the execution of the object
  - can be translated to a CUDA processing graph
- prepare_trigger and post_trigger methods for:
  - preparation of the inputs
  - preparation of the output
  - parts of the computation which is not suitable for the CUDA processing graph

Objects categories: Processing objects (3)

- how to write the yaml block for a specific processing object?

```
object_nameB:
  class:                    'ObjectClass'
  scalar_param:             23
  namedDataObject_object:   'filename1'
  namedRawData_data:        'filename2'
  namedDataObjects_ref:     ['object_name1', 'object_name2', …]
  inputs:
    in_i: 'object_nameA.out_name'
  outputs:  ['out_name1', 'out_name2', …]
```

A data object read from disk

Raw data read from disk

A reference to one or more data objects defined in the same yaml file

Example yaml blocks

```
detector:
  class:              'CCD'
  size:               [80,80]                 # Detector size in pixels
  dt:                 0.001                   # [s] Detector integration time
  bandw:              300                     # [nm] Sensor bandwidth
  photon_noise:       True                    # activate photon noise
  readout_noise:      True                    # activate readout noise
  readout_level:      1.0                     # readout noise in [e-/pix/frame]
  quantum_eff:        0.32                    # quantum efficiency * total transmission
  inputs:
    in_i: 'pyramid.out_i'
  outputs:  ['out_pixels']

slopec:
  class:              'PyrSlopec'
  pupdata_object:     'scao_pup'            # tag of the pyramid WFS pupils
  sn_object:          'scao_sn'             # tag of the slope reference vector
  inputs:
    in_pixels:         'detector.out_pixels'
  outputs:  ['out_slopes', 'out_pupdata', 'total_counts', 'subap_counts']
```

What you have to worry about

- Did I provide all the mandatory parameters to all of my processing objects?
  - Check API documentation or the python Class constructor (__init__) in the code
  - If I rely on the defaults, are these what I want in my case?
  - If the parameter data is read from disk, do I have the correct file for it?
- Did I connect compatible output and inputs?
- Which data do I want to save?
- How long should my simulation last?
- Do I want to visualize some data live, during the simulation?
- Possibly: How do I allocate the different objects to CPU processes/GPU/HPC nodes?

What you don’t need to worry (too much) about

- Somewhere a Simulation class…
  - builds all the objects you specified
  - creates the connections between inputs and outputs
  - finds out an order of execution which will be followed at each timestep
- Later, a Scheduler class, at each timestep, for each processing objects, will command its execution (read the inputs, perform the computation, update the outputs)
- Objects may reside on different memory spaces (different processes on the CPU, different GPUs, different nodes of a HPC cluster) → Data referencing/copy/communication will be handled for you

Processing objects: developers perspective

- suggestions to write your processing object:
  - look at the code of the existing ones
  - identify:
    - inputs
    - outputs
    - parameters (data that does not change during the simulation)
    - what is done in initialization phase ? (__init__ method)
    - what is done in the trigger/prepare_trigger/post_trigger methods?
  - A skeleton is provided in the file specula/template_processing_object.py
  - You are welcome to ask the dev team for help!

How to run a Simulation

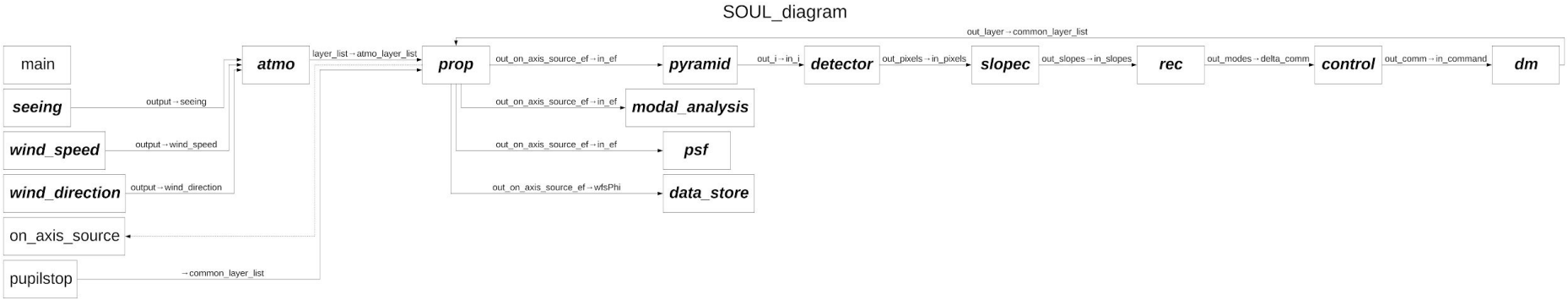
- pip install specula
- From command line
  - specula simul_name.yml [parameters]
- From python code (notebooks included):
  - `import specula`
  - `specula.init(target_device_idx, precision=1)`
  - `yml_files = [“simul_name.yml”]`
  - `specula.main_simul(yml_files) # other parameters might follow in the ()`

See notebook, cell 1.3

- On HPC nodes
  - For example, on Leonardo, you have to write a configuration file for the slurm scheduler,
    - examples are provided in the git repository
- Available parameters
  - parser.add_argument('--nsimul', type=int, default=1, help='Number of simulations to run')
  - parser.add_argument('--cpu', action='store_true', help='Force CPU execution (equivalent to --target=-1)')
  - parser.add_argument('--overrides', type=str, help='YAML string with parameter overrides')
  - parser.add_argument('--target', type=int, default=0, help='Target device ID for GPU execution')
  - parser.add_argument('--precision', type=int, default=1, choices=[0, 1] help='Floating point precision: 0=double (float64), 1=single (float32)')
  - parser.add_argument('--profile', action='store_true', help='Enable python profiler and print stats at the end')
  - parser.add_argument('--mpi', action='store_true', help='Use MPI for parallel execution')
  - parser.add_argument('--mpidbg', action='store_true', help='Activate MPI debug output')
  - parser.add_argument('--stepping', action='store_true', help='Allow simulation stepping')
  - parser.add_argument('--diagram', action='store_true', help='Save image block diagram')
  - parser.add_argument('--diagram-title', type=str, default=None, help='Block diagram title')
  - parser.add_argument('--diagram-filename', type=str, default=None, help='Block diagram filename')
  - parser.add_argument('--diagram-colors-on', action='store_true', help='Enable colors in block diagram')
  - parser.add_argument('yml_files', nargs='+', type=str, help='YAML parameter files')

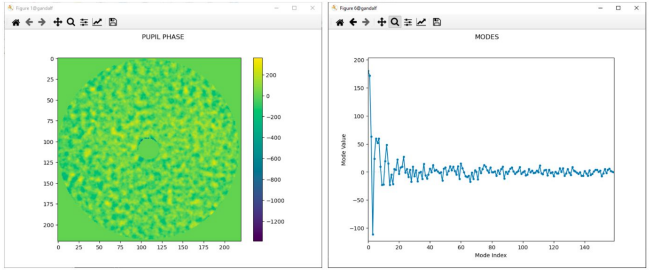
We like diagrams

- you can ask specula to produce a png image rendering a simple diagram of the simulation it is going to run

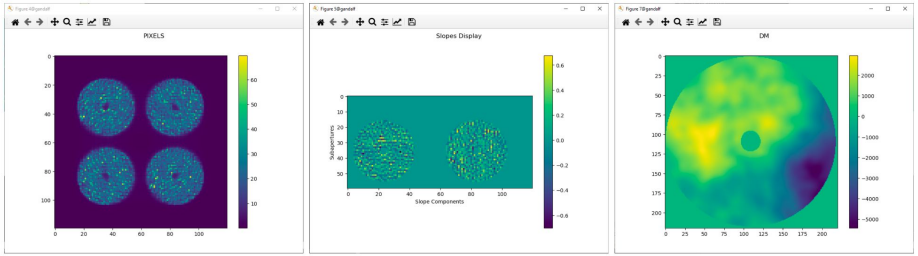


Live Data visualization

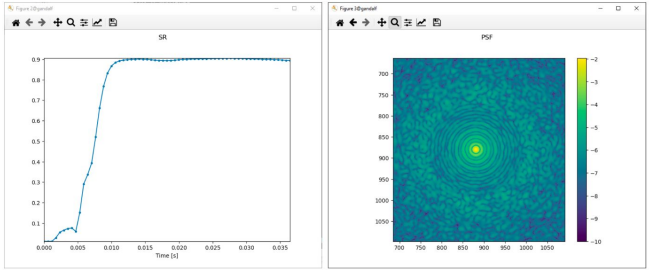
- On the same computer you run the simulation on
  - Live matplotlib plots, specialized by data object type
  - Compatible with notebooks, see notebook cell 1.5



(a) PhaseDisplay, at pupil (b) ModesDisplay



(c) PixelsDisplay (d) SlopecDisplay (e) PhaseDisplay, at Deformable Mirror output.



(f) PlotDisplay (g) PsfDisplay

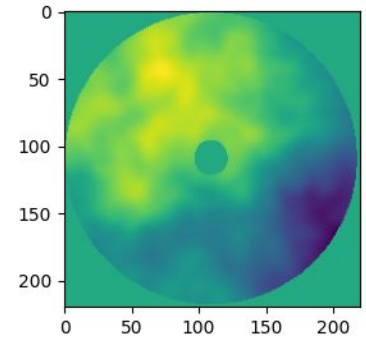
Live Data visualization

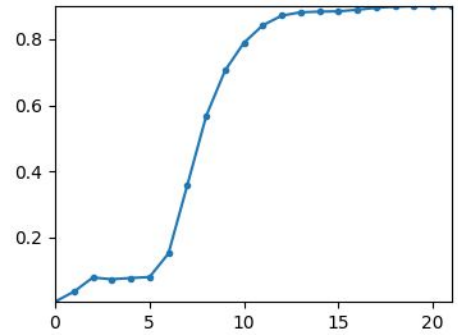
- From a remote server
  - Display Server + web page, a processing object you can define like this:

```
server:
  class:          'DisplayServer'
  host:           '0.0.0.0'
  port:           3000
```

  - Each one of the outputs can be selected
- Coming up next
  - Specula Studio

slopec	Input	in_pixels
	Output	out_slopes
rec	Input	in_slopes
	Output	out_modes
control	Input	delta_comm
dm	Input	in_command
	Output	out_layer
psf	Input	in_ef
	Output	out_sr





Data storage (1)

- To save data you have to define a data store object, in our example:

```
data_store:
  class:              'DataStore'
  store_dir:
'/content/drive/MyDrive/HandsOnSPECULA/output'
  data_format:        'fits'
  inputs:
    input_list: ['wfsPhi-prop.out_on_axis_source_ef',
                 'resMod-modal_analysis.out_modes',
                 'ccdframes-detector.out_pixels',
                 'slopes-slopec.out_slopes',
                 'deltaComm-rec.out_modes',
                 'comm-control.out_comm',
                 'srRes-psf.out_sr']
# the syntax for an element of the list here is
fitsfilename-objectName.outputname
```

- Note that the prop object output names are not in the config file! Here you need to know that the propagation object produces an output for each of the sources, named out_sourceName_ef

Data storage (2)

- Running the simulation has produced a Tracking Number sub-folder in output, named with date and time, something like:
  - 20260205_145654
- The folder will contain:
  - all the files and data specified in the data store object description
  - copy of the parameter file describing the simulation that produced the data, for future reference (renamed as params.yml)
  - a new parameter file (named replay_params.yml) that can be used to replay the simulation

Simulation replay

See notebook, cell 1.6

- the command:
  - `python main_simul.py output/20250217_162346/replay_params.yml`
  - will not produce another TN, but will produce this diagram (replay_params.png)
- data_store was replaced by data_source
- prop object is removed
- prop output (which was saved by data_store) is replaced by corresponding data_source output
- atmo and the objects of lower trigger order in the original simulation are now disconnected by the rest of the simulation, so will never be triggered
  - Note for devs: we can actually remove these too

Overrides: modifying your base simulation in clean way

- You have defined carefully your system (for example ANDES, SOUL etc) but you want to
  - Run the simulation using different atmospheric conditions
  - When you run it on your server: Allocate some objects on GPU0 and some other on GPU1
  - When you run it on your laptop: Allocate some objects on Process0 and some other on Process1
  - When you run it on HPC nodes: define complex allocation to processes and GPUs
  - Save more or less data
  - Test a different configuration of your Control strategy
  - Many more use cases
- How to avoid polluting your “base plant” file?
- How to avoid explosion of yaml files / versions?

See notebook cell 1.5 and 1.7

Overrides

- The simulation accept a sequence of yaml files, which are applied in order
- In each file, after the first one, you can:
- Add a new section/object:
  - just write the section defining the new object
- remove one or more objects:
  - write a list of the objects to be removed:
    - remove: ['objToRemove1', 'objToRemove2', ...]
- modify Parameters and Inputs of a previously defined object:
  - objName_override: { param_key: new_param_value }

Calibrations and overrides

- Another use case for overrides
- Where do the files we use to configure some object come from?
- As in the real systems, we can perform calibrations of the objects to define some of their properties
- The calibration procedure is just another simulation
    - some specific logic might be needed, so some new processing object might have to be developed

Calibration example: Interaction Matrix and Reconstructor

See Notebook cell 1.7

ANDES simulation

- Simplified ANDES simulation:
  - No telescope static aberrations (M1 surface/phasing errors)
  - No wind-shake nor vibrations on M1 and M2
  - Simplified control
  - Perfect WFS for “petals” correction
- All of the above features can be easily added

See Notebook cell 2.3/2.4

ANDES with Coronagraph

See notebook cell 2.5

Thanks to Matteo Menessini for the Lyot Coro implementation!

Display Server Demo

Specula Studio (Work in Progress)

- The yaml simulation file as the results of GUI interaction
- One application to handle simulation running, stepping, live display and results replay/analysis
- Technically it’s a node editor, similar to other domains where you define a graph of objects (graphics rendering tools, computation pipelines, other simulation engines)

Discussion Time

Running on HPC - Slurm configuration

```
#!/bin/bash

#SBATCH --job-name=spM4                          # Descriptive job name
#SBATCH --time=00:10:00                          # Maximum wall time (hh:mm:ss)
#SBATCH --nodes=7                                # Number of nodes to use
#SBATCH --ntasks-per-node=1                      # Number of MPI tasks per node (e.g., 1 per GPU)
#SBATCH --cpus-per-task=4                        # Number of CPU cores per task (adjust as
needed)
#SBATCH --gres=gpu:4                             # Number of GPUs per node (adjust to match
hardware)
#SBATCH --partition=boost_usr_prod               # GPU-enabled partition
#SBATCH --qos=normal                             # Quality of Service
#SBATCH --output=speculaJobMorfeoMpi4.out        # File for standard output
#SBATCH --error=speculaJobMorfeoMpi4.err         # File for standard error
#SBATCH --account=try25_rossi                    # Project account number

module load cuda/12.3                            # Load CUDA toolkit
module load openmpi/4.1.6--nvhpc--23.11
module load nvhpc/23.11

srun --mpi=none bash -c "specula --mpi --diagram-filename morfeo4.png --diagram-colors-on
$WORK/SPECULA/config/MORFEO/params_morfeo_full.yml
$WORK/SPECULA/config/leonardo/overrides/ov_MORFEO_leonardo_ds4.yml"
```

Running on HPC - Allocation

```
main_override:
  root_dir:         '/leonardo_work/try25_rossi/MORFEO_DATA0'         # Root directory for calibration manager
  total_time:       0.100
  display_server:   false

# Parallelization over GPUs on a multiple nodes

pupilstop_override: { target_rank: 0 }

ifunc_m4_override: { target_device_idx: 0, target_rank: 0 }
m2c_m4_override: { target_device_idx: 0, target_rank: 0 }
dm1_override: { target_device_idx: 0, target_rank: 0 }
dm2_override: { target_device_idx: 1, target_rank: 0 }
dm3_override: { target_device_idx: 2, target_rank: 0 }

sh_lgs1_override: { class: 'DistributedSH', n_slices: 4, target_device_idx: 0, target_rank: 1 }
sh_lgs2_override: { class: 'DistributedSH', n_slices: 4, target_device_idx: 0, target_rank: 2 }
sh_lgs3_override: { class: 'DistributedSH', n_slices: 4, target_device_idx: 0, target_rank: 3 }
sh_lgs4_override: { class: 'DistributedSH', n_slices: 4, target_device_idx: 0, target_rank: 4 }
sh_lgs5_override: { class: 'DistributedSH', n_slices: 4, target_device_idx: 0, target_rank: 5 }
sh_lgs6_override: { class: 'DistributedSH', n_slices: 4, target_device_idx: 0, target_rank: 6 }

detector_lgs1_override: { target_device_idx: 0, target_rank: 1 }
detector_lgs2_override: { target_device_idx: 0, target_rank: 2 }
detector_lgs3_override: { target_device_idx: 0, target_rank: 3 }
detector_lgs4_override: { target_device_idx: 0, target_rank: 4 }
detector_lgs5_override: { target_device_idx: 0, target_rank: 5 }
detector_lgs6_override: { target_device_idx: 0, target_rank: 6 }


...
```

Some Notes on performances

- Test runs on Leonardo
  - SCAO pyramid closed loop, VLT case
    - Single GPU: 320 Hz
  - Current Morfeo case performance
    - 22 Hz distributed over 25 GPUs!
    - Single GPU: 1.5 Hz+
    - Dual GPU, multiprocessing: 4.3 Hz (Our “Cheap” server with 2 nvidia L40S)
- Comparison with COMPASS
  - 10+ years of development
  - Also works on multi–GPU
  - Great performances
  - Python interface but C++ and CUDA implementation of the computation
    - very difficult to modify/extend for the average AO scientist compared to our python code
  - We have similar closed loop performances on pyramid WFS of the same size, ELT case!
  - Our SH considerably slower, but we don’t know many details about their implementation
    - Room for improvement implementing better “sampling” strategies