Details About Nipype Workflow Showcase Notebook:

The provided Jupyter notebook code demonstrates the use of **Nipype**, a Python-based framework for creating and executing neuroimaging workflows, to preprocess fMRI data. Below are the key points and steps covered in the notebook:

Overview

The notebook outlines how to:

1. **Set up a preprocessing workflow** for fMRI data using Nipype.
2. **Run the workflow in parallel** to leverage multi-core processing.
3. **Apply essential preprocessing steps** such as slice timing correction, motion correction, and smoothing.
4. **Visualize the workflow** and inspect its outputs.

1. Preprocessing Workflow Setup

- **Nipype Tools**: The main tools used are Node (to define individual processing steps) and Workflow (to connect these steps).

python

from nipype **import** Node, Workflow

from nipype.interfaces.fsl **import** SliceTimer, MCFLIRT, Smooth

- **Nodes for Preprocessing Steps**:
    - SliceTimer: Corrects for slice acquisition timing differences in fMRI data.
    - MCFLIRT: Performs motion correction.
    - Smooth: Applies Gaussian smoothing to the functional images.

python

slicetimer = Node(SliceTimer(index_dir=False, interleaved=True, time_repetition=2.5), name="slicetimer")

mcflirt = Node(MCFLIRT(mean_vol=True, save_plots=True), name="mcflirt")

smooth = Node(Smooth(fwhm=4), name="smooth")

- **Workflow Creation**: A preprocessing workflow named preproc01 is created and the nodes are connected to define the flow of data between them.

python

```
preproc01 = Workflow(name='preproc01', base_dir='.')

preproc01.connect([(slicetimer, mcflirt, [('slice_time_corrected_file', 'in_file')]),

                   (mcflirt, smooth, [('out_file', 'in_file')])])
```

2. Running the Workflow

- The workflow is run on a functional image with parallelization using 5 processors.

python

```
slicetimer.inputs.in_file = '/data/ds000114/sub-01/ses-test/func/sub-01_ses-test_task-fingerfootlips_bold.nii.gz'

%time preproc01.run('MultiProc', plugin_args={'n_procs': 5})
```

- **Parallel Execution**: Nipype leverages multiple processors to run tasks in parallel when possible. However, since each node depends on the output of the previous node in this example (sequential dependencies), parallelization doesn't significantly reduce execution time.

3. Workflow Visualization

- The workflow graph is generated and visualized within the notebook to understand how data flows between nodes.

python

```
preproc01.write_graph(graph2use='orig')

from IPython.display import Image

Image(filename="preproc01/graph_detailed.png")
```

4. Rerunning the Workflow

- The Gaussian smoothing kernel's full-width half-maximum (FWHM) value is changed from 4 to 2, and the workflow is rerun. Nipype uses caching, so only the modified node (smooth) is rerun.

python

```
smooth.inputs.fwhm = 2

%time preproc01.run('MultiProc', plugin_args={'n_procs': 5})
```

5. Parallel Execution on Multiple Functional Images

- The notebook demonstrates how to clone workflows (preproc02, preproc03, etc.) and run them in parallel using a "metaflow" that contains multiple workflows.

python

metaflow = Workflow(name='metaflow', base_dir='.')

metaflow.add_nodes([preproc01, preproc02, preproc03, preproc04, preproc05])

%time metaflow.run('MultiProc', plugin_args={'n_procs': 5})

6. Output Inspection

- After running the workflows, the outputs are stored in directories corresponding to each step (slicetimer, mcflirt, smooth). The outputs include:
    - Slice-timed corrected image.
    - Motion-corrected image.
    - Smoothed image.

bash

!tree metaflow -I '*js|*json|*pklz|_report|*.dot|*html'

Conclusion

This notebook showcases how Nipype simplifies fMRI preprocessing by:

1. Automating workflows with modular nodes.
2. Leveraging parallel computing for efficient execution.
3. Providing detailed logging and caching mechanisms to avoid redundant computations.