# Summary of OSE Scientific Computing Project on **“Pydsge”** (by Yuxin, Altaf and Philipp)

## Testing with pytest

- A testing suite for regression testing of the tutorial(s) was created
- A preliminary test against a closed form solution of a smaller model was created
- A basic test of the parser was created

## Multiprocessing on Windows

- A pull-request with the relevant changes was created for grglib
- Gregor’s simplifications were adopted
- [for documentation purposes and Prof. Eisenhauer, full details are provided below.]

## Estimation Tutorial

- An estimation tutorial for pydsge was created based on a script provided by Gregor
- Several issues that were discovered in the process were fixed, documentation was improved
- **To-do:** Parts where we were uncertain were highlighted for Gregor

## Pre-commit check of Code style

- A pre-commit hook was created which, among others, includes black, blacken and flake8 (with several adjustments)

## Continuous Integration Workflow

- To enable cross-platform and cross-version code control and maintenance, a github CI workflow for Unix, MacOs and Windows as well as Python 3.8 and 3.9 was created. (Having a specific badge for these environments would require Anaconda)
- The CI workflow runs the pytest framework and, if needed, can be set to update the pickle used for the tests (Ubuntu with python 3.8 is used as the stable version.)

# Multiprocessing on Windows (details)

## 1. Description of the issue and the reason for it.

The issue of this project is that multiprocessing doesn't work on Windows.

The reason for the issue is that Windows doesn't provide `fork`, which is used on the Unix operating system. The child process created by `fork` is effectively identical to the parent process, which is the reason why multiprocessing works fine on both MacOS and Linux. Windows uses `spawn` instead, which creates a fully independent new process. It leads to three bugs when using a multiprocessing pool in Python on Windows:

1) the child process will execute the same codes as its parent, causing every child process to try and create a multiprocessing pool of their own, which causes exceptions;
2) on the Windows system, the function and its called functions have to be passed to the child process after serialization. This is not necessary in Linux because the child process has the same environment and variables as its parent process;
3) if the attributes of the object are modified, the modification will be ignored. I found the true reason is that when two functions are serialized separately, these two functions make their own deep copy of the same object.

## 2. Different solutions developed to debug.

We have developed treatments for all of these three bugs.
**2.1** For the first bug, we can judge under the condition of the current process being the main process. I.e.

```
from multiprocessing import freeze_support
if __name__ == '__main__':
    freeze_support()
    main()
else:
    pass
```

**2.2** For the second bug, we can use `dill` to serialize the function with "param recurse=True". There are three solutions to serialize the function.
A) This solution is the original solution used by Gregor. It's used only on Unix.

```python
def serializer(func):
    """Dirty hack that transforms the non-serializable function to a
serializable one (when using dill)
    ...
    Don't try that at home!
    """

    fname = func.__name__
    exec("dark_%s = func" % fname, locals(), globals())

    def vodoo(*args, **kwargs):
        return eval("dark_%s(*args, **kwargs)" % fname)

    return vodoo
```

B) This solution dumps the function and returns `vodoo` which loads the function and calls it. We need to load the original function before we call it, no matter whether multiprocessing is enabled or disabled.

```python
def serializer(func):
    x = dill.dumps(func, recurse=True)
    def vodoo(*args, **kwargs):
        return dill.loads(x)(*args, **kwargs)
    return vodoo
```

C) This solution dumps the function and loads it immediately. When we pass it to the multiprocessing pool, it needs to be dumped and loaded again.

```python
def serializer(func):
    fstr = dill.dumps(func, recurse=True)
    return dill.loads(fstr)
```

You can refer to the speed of three solutions in the table below as a reference:

| | Linux(8 cores vm) | Windows(8 cores) | MacOS(2 cores) |
|---|---|---|---|
| A | 39.83s | / | 110.51s |
| B | 392.46s | 596.24s | 611.16s |
| C | 86.53s | 152.64s | 137.84s |

In addition, it has been found that solution B runs faster than solution C when parallelization is enabled. However, it runs much slower than solution C when parallelization is disabled. Because every time `vodoo` is called, `dill.load` will be run again. Parallelization is disabled in part of my test code, so it takes such a long time when using solution B.

After comparing the speed and drawbacks of these three solutions, we suggest using solution A on Unix. For Windows, both solution B and solution C are acceptable, but solution B is faster.

**2.3** For the third bug, the solution is serializing two functions together:

```
def serializer_unix(func):
    fname = func.__name__
    exec("dark_%s = func" % fname, locals(), globals())
    def vodoo(*args, **kwargs):
        return eval("dark_%s(*args, **kwargs)" % fname)
    return vodoo

def serializer(*functions):
    if platform == "darwin" or platform == "linux":
        rtn = []
        for func in functions:
            rtn.append(serializer_unix(func))
    else:
        fstr = dill.dumps(functions, recurse=True)
        rtn = dill.loads(fstr)
    if len(functions) == 1:
        return rtn[0]
    else:
        return rtn
```

Meanwhile, we need to change the codes like these

```
npas = serializer(self.filter.npas)
run_filter = serializer(self.run_filter)
```

to

```
run_filter, npas = serializer(self.run_filter, self.filter.npas)
```

To solve the third bug, we choose to use solution C mentioned before on Windows regardless of speed. Solution B can also pickle multiple functions, but the modifications are more complex.