pynetlogo Documentation

Release 0.5.3

J.H. Kwakkel

Feb 13, 2024

CONTENTS

sphinx-quickstart on Sat Mar 23 14:18:16 2013. You can adapt this file completely to your liking, but it should at least contain the root *toctree* directive.

Interface to use and access NetLogo (Wilensky 1999) from Python. One can interact with NetLogo in either headless (no GUI) or interactive GUI mode. The library provides functions to load models, execute commands, and get values from reporters. It is compatible with NetLogo 6.1 and newer. It is largely similar to the NetLogo Mathematica Link and RNetLogo (deprecated).

CHAPTER

ONE

DOCUMENTATION

1.1 Installation

pynetlogo requires the NumPy, SciPy and pandas packages, which are included in most scientific Python distributions.

In addition, pynetlogo depends on jpype. When installing pynetlogo, jpype will be installed as well. However, if you want to have full control over how jpype is installed, check their installation details and install jpype before installing pynetlogo.

pyNetLogo can be installed using the pip package manager, with the following command from a terminal:

```
pip install pynetlogo
```

By default, pynetlogo and jpype will attempt to automatically identify the NetLogo version and installation directory on Mac or Windows, as well as the Java home directory. On Linux, or in case of issues (e.g. if NetLogo was installed in a different directory, or if the Java path is not found on a Mac), these parameters can be passed directly to the NetLogoLink class as described in the module documentation.

1.1.1 Known bugs and limitations

- On a Mac, only headless mode (without GUI) is supported.
- pynetlogo can be used to control NetLogo from within Python. Calling Python from within NetLogo is not supported by this library. However, this can be achieved using the Python extension for NetLogo.
- See jpype limitations for additional limitations.
- Mixing 32-bit and 64-bit versions of Java, Python, and NetLogo will crash Python.
- on M1 macs, your java architecture must match your python architecture. So you cannot use AArch64 (ARM) java with an x64 python install or the other way around. Use *jvm_path* to control which jvm pynetlogo will use.

1.2 Example 1: NetLogo interaction through the pyNetLogo connector

This notebook provides a simple example of interaction between a NetLogo model and the Python environment, using the Wolf Sheep Predation model included in the NetLogo example library (Wilensky, 1999). This model is slightly modified to add additional agent properties and illustrate the exchange of different data types. All files used in the example are available from the pyNetLogo repository at https://github.com/quaquel/pyNetLogo.

We start by instantiating a link to NetLogo, loading the model, and executing the `setup` command in NetLogo.

[1]:
```
%matplotlib inline

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_style("white")
sns.set_context("talk")

import pynetlogo

netlogo = pynetlogo.NetLogoLink(
    gui=True,
    jvm_path="/Users/jhkwakkel/Downloads/jdk-19.0.2.jdk/Contents/MacOS/libjli.dylib",
)

netlogo.load_model("./models/Wolf Sheep Predation_v6.nlogo")
netlogo.command("setup")
```

We can use the `write_NetLogo_attriblist` method to pass properties to agents from a Pandas dataframe – for instance, initial values for given attributes. This improves performance by simultaneously setting multiple properties for multiple agents in a single function call.

As an example, we first load data from an Excel file into a dataframe. Each row corresponds to an agent, with columns for each attribute (including the `who` NetLogo identifier, which is required). In this case, we set coordinates for the agents using the `xcor` and `ycor` attributes.

[2]:
```
agent_xy = pd.read_excel("./data/xy_DataFrame.xlsx")
agent_xy[["who", "xcor", "ycor"]].head(5)
```

[2]:

	who	xcor	ycor
0	0	-24.000000	-24.000000
1	1	-23.666667	-23.666667
2	2	-23.333333	-23.333333
3	3	-23.000000	-23.000000
4	4	-22.666667	-22.666667

We can then pass the dataframe to NetLogo, specifying which attributes and which agent type we want to update:

[3]:
```
netlogo.write_NetLogo_attriblist(agent_xy[["who", "xcor", "ycor"]], "a-sheep")
```

We can check the data exchange by returning data from NetLogo to the Python workspace, using the report method. In the example below, this returns arrays for the `xcor` and `ycor` coordinates of the `sheep` agents, sorted by their `who` number. These are then plotted on a conventional scatter plot.

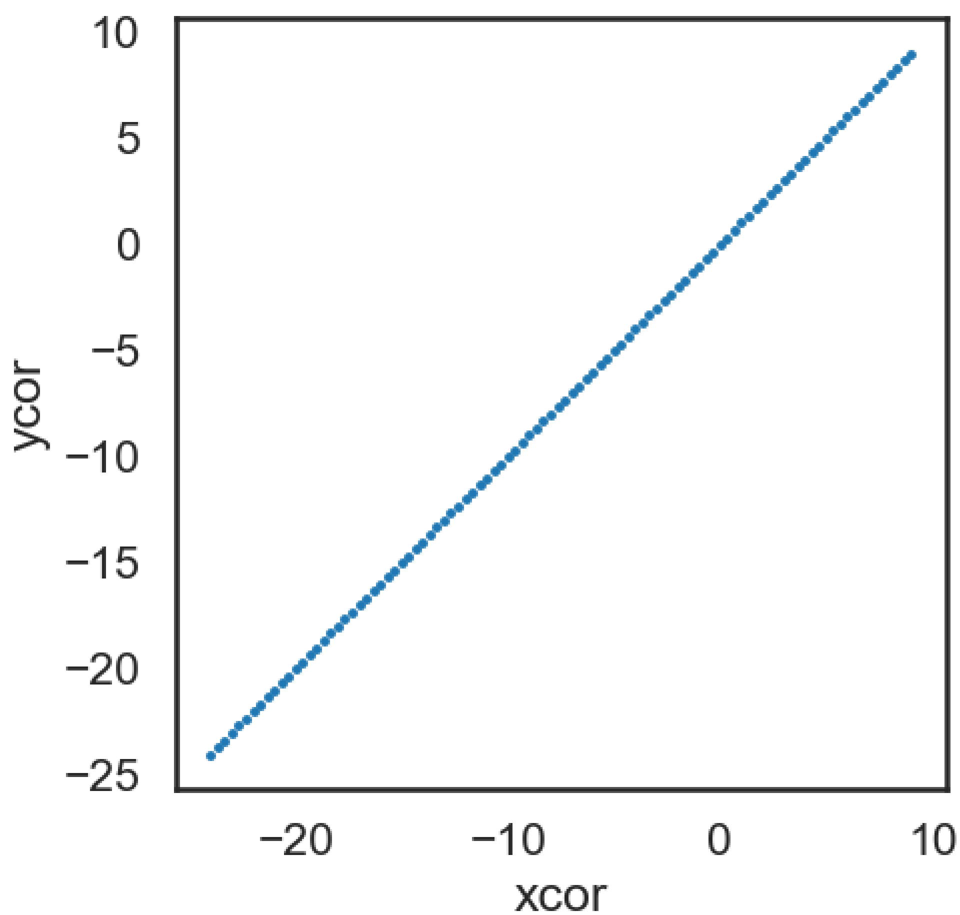
[4]:
```
x = netlogo.report("map [s -> [xcor] of s] sort sheep")
y = netlogo.report("map [s -> [ycor] of s] sort sheep")
```

[5]:
```
fig, ax = plt.subplots(1)

ax.scatter(x, y, s=4)
ax.set_xlabel("xcor")
ax.set_ylabel("ycor")
ax.set_aspect("equal")
```

(continues on next page)

(continued from previous page)

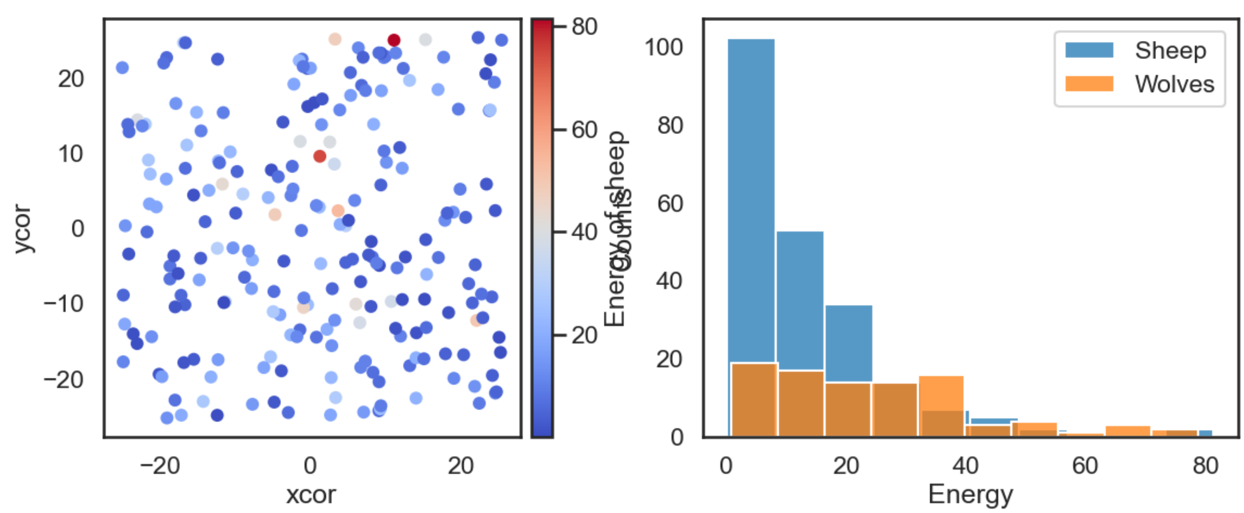
```
fig.set_size_inches(5, 5)

plt.show()
```



We can then run the model for 100 ticks and update the Python coordinate arrays for the sheep agents, and return an additional array for each agent's energy value. The latter is plotted on a histogram for each agent type.

[6]:
```
# We can use either of the following commands to run for 100 ticks:

netlogo.command("repeat 100 [go]")
# netlogo.repeat_command('go', 100)


# Return sorted arrays so that the x, y and energy properties of each agent are in the␣
↪same order
x = netlogo.report("map [s -> [xcor] of s] sort sheep")
y = netlogo.report("map [s -> [ycor] of s] sort sheep")
energy_sheep = netlogo.report("map [s -> [energy] of s] sort sheep")

energy_wolves = netlogo.report("[energy] of wolves")  # NetLogo returns these in random␣
↪order
```

[7]:
```
from mpl_toolkits.axes_grid1 import make_axes_locatable

fig, ax = plt.subplots(1, 2)

sc = ax[0].scatter(x, y, s=50, c=energy_sheep, cmap=plt.cm.coolwarm)
ax[0].set_xlabel("xcor")
ax[0].set_ylabel("ycor")
ax[0].set_aspect("equal")
divider = make_axes_locatable(ax[0])
cax = divider.append_axes("right", size="5%", pad=0.1)
cbar = plt.colorbar(sc, cax=cax, orientation="vertical")
cbar.set_label("Energy of sheep")

sns.histplot(energy_sheep, kde=False, bins=10, ax=ax[1], label="Sheep")
sns.histplot(energy_wolves, kde=False, bins=10, ax=ax[1], label="Wolves")
ax[1].set_xlabel("Energy")
ax[1].set_ylabel("Counts")
ax[1].legend()
fig.set_size_inches(14, 5)

plt.show()
```



The `repeat_report` method returns a dictionary with the reporter as key. The value is a list order by ticks. By default, this assumes the model is run with the "go" NetLogo command; this can be set by passing an optional `go` argument.

Often, the dictionary can easily be converted into a dataframe, for easy further analysis.In this case, we track the number of wolf and sheep agents over 200 ticks; the outcomes are first plotted as a function of time. The number of wolf agents is then plotted as a function of the number of sheep agents, to approximate a phase-space plot.

[8]:
```
counts = netlogo.repeat_report(["count wolves", "count sheep"], 200, go="go")
```

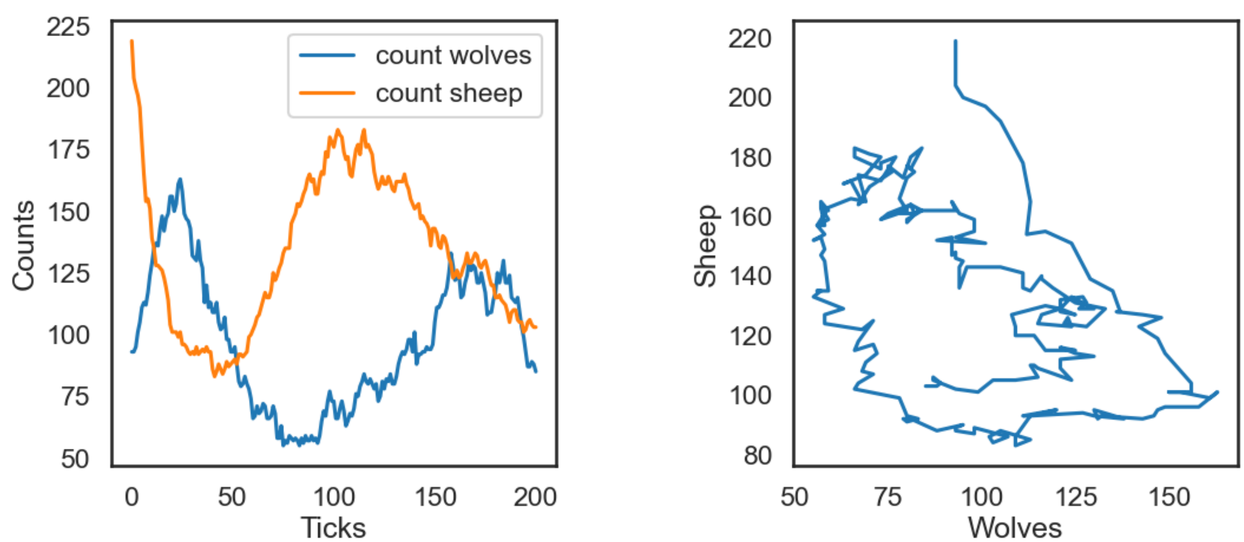
[12]:
```
counts = pd.DataFrame(counts)
```

[13]:
```
fig, (ax1, ax2) = plt.subplots(1, 2)

counts.plot(ax=ax1, use_index=True, legend=True)
```

(continues on next page)

(continued from previous page)

```
ax1.set_xlabel("Ticks")
ax1.set_ylabel("Counts")

ax2.plot(counts["count wolves"], counts["count sheep"])
ax2.set_xlabel("Wolves")
ax2.set_ylabel("Sheep")


for ax in [ax1, ax2]:
    ax.set_aspect(1 / ax.get_data_ratio())


fig.set_size_inches(12, 5)
plt.tight_layout()
plt.show()
```



The `repeat_report` method can also be used with reporters that return a NetLogo list. In this case, the list is converted to a numpy array. As an example, we track the energy of the wolf and sheep agents over 5 ticks, and plot the distribution of the wolves' energy at the final tick recorded in the dataframe. Note that the number of sheep and wolves vary over time. This means that for each tick, the size of the array will be different. So, we cannot straightforwardly convert these results into a dataframe.

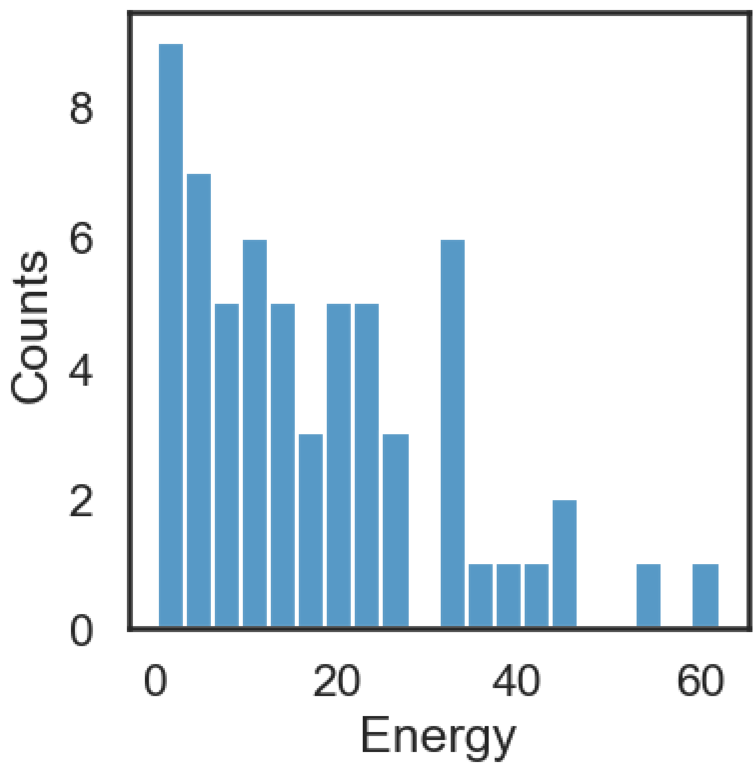
To illustrate different data types, we also track the `[sheep_str] of sheep` reporter (which returns a string property across the sheep agents, converted to a numpy object array), `count sheep` (returning a single numerical variable), and `glob_str` (returning a single string variable).

[16]:
```
results = netlogo.repeat_report(
    [
        "[energy] of wolves",
        "[energy] of sheep",
        "[sheep_str] of sheep",
        "count sheep",
        "glob_str",
    ],
    5,
```

(continues on next page)

(continued from previous page)

```
)

fig, ax = plt.subplots(1)

sns.histplot(results["[energy] of wolves"][-1], kde=False, bins=20, ax=ax)
ax.set_xlabel("Energy")
ax.set_ylabel("Counts")
fig.set_size_inches(4, 4)

plt.show()
```




```
[18]: list(results.keys())
```

```
[18]: ['[energy] of wolves',
 '[energy] of sheep',
 '[sheep_str] of sheep',
 'count sheep',
 'glob_str']
```

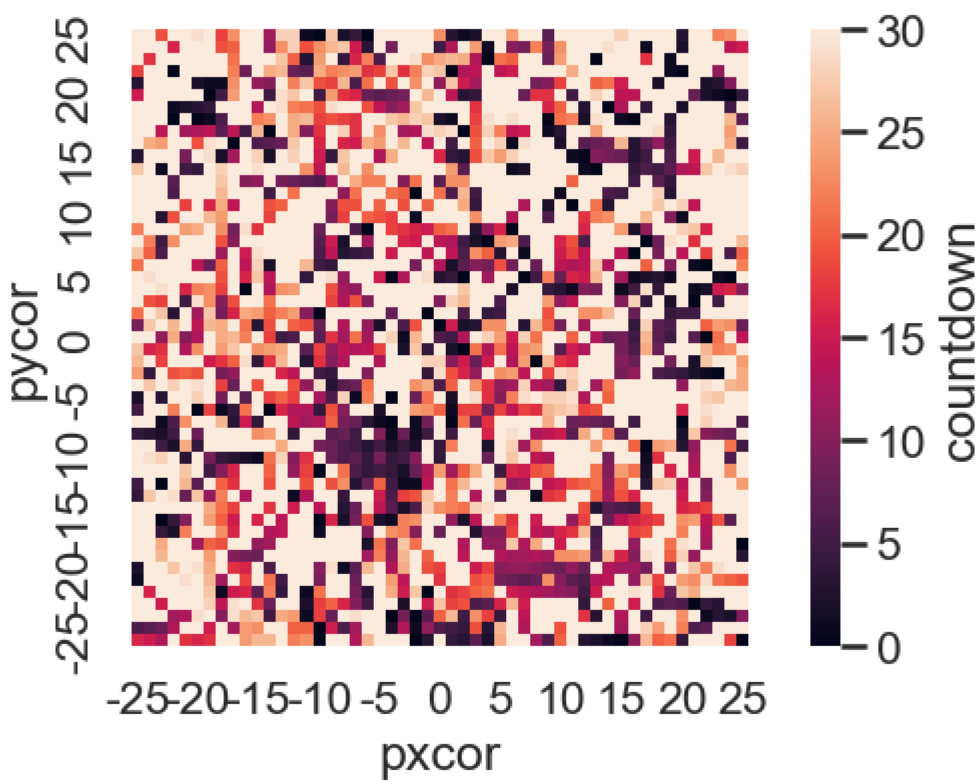
The `patch_report` method can be used to return a dataframe which (for this example) contains the `countdown` attribute of each NetLogo patch. This dataframe essentially replicates the NetLogo environment, with column labels corresponding to the xcor patch coordinates, and indices following the pycor coordinates.

```
[13]: countdown_df = netlogo.patch_report("countdown")

fig, ax = plt.subplots(1)
```

(continues on next page)

(continued from previous page)

```
patches = sns.heatmap(
    countdown_df, xticklabels=5, yticklabels=5, cbar_kws={"label": "countdown"}, ax=ax
)
ax.set_xlabel("pxcor")
ax.set_ylabel("pycor")
ax.set_aspect("equal")
fig.set_size_inches(8, 4)

plt.show()
```



The dataframes can be manipulated with any of the existing Pandas functions, for instance by exporting to an Excel file. The `patch_set` method provides the inverse functionality to `patch_report`, and updates the NetLogo environment from a dataframe.

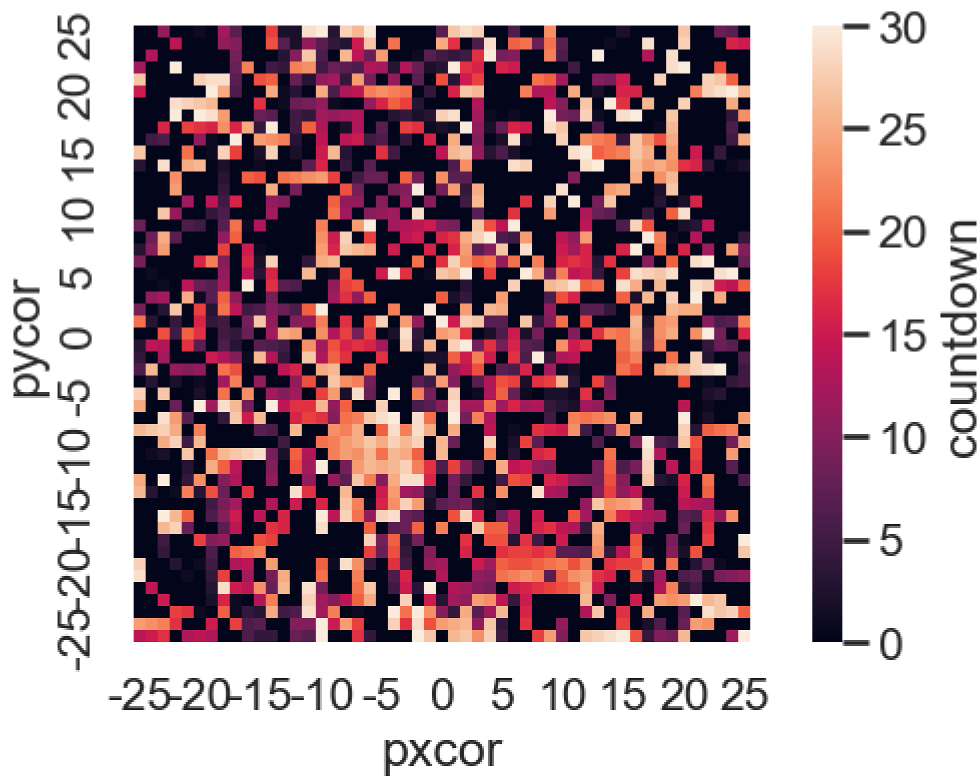
[14]:
```
countdown_df.to_excel("countdown.xlsx")
netlogo.patch_set("countdown", countdown_df.max() - countdown_df)
```

[15]:
```
countdown_update_df = netlogo.patch_report("countdown")

fig, ax = plt.subplots(1)

patches = sns.heatmap(
    countdown_update_df,
    xticklabels=5,
    yticklabels=5,
    cbar_kws={"label": "countdown"},
```

(continues on next page)

(continued from previous page)

```
    ax=ax,
)
ax.set_xlabel("pxcor")
ax.set_ylabel("pycor")
ax.set_aspect("equal")
fig.set_size_inches(8, 4)

plt.show()
```



Finally, the `kill_workspace()` method shuts down the NetLogo instance.

[16]:
```
netlogo.kill_workspace()
```

[ ]:

1.3 Example 2: Sensitivity analysis for a NetLogo model with SALib and ipyparallel

This provides a more advanced example of interaction between NetLogo and a Python environment, using the SALib library (Herman & Usher, 2017); available through the pip package manager) to sample and analyze a suitable experimental design for a Sobol global sensitivity analysis. Furthermore, the ipyparallel package (also available on pip) is used to parallelize the simulations.

All files used in the example are available from the pyNetLogo repository at https://github.com/quaquel/pyNetLogo.

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

sns.set_style("white")
sns.set_context("talk")

import pynetlogo

# Import the sampling and analysis modules for a Sobol variance-based
# sensitivity analysis
from SALib.sample import sobol as sobolsample
from SALib.analyze import sobol
```

SALib relies on a problem definition dictionary which contains the number of input parameters to sample, their names (which should here correspond to a NetLogo global variable), and the sampling bounds. Documentation for SALib can be found at https://salib.readthedocs.io/en/latest/.

[2]:
```
problem = {
    "num_vars": 6,
    "names": [
        "random-seed",
        "grass-regrowth-time",
        "sheep-gain-from-food",
        "wolf-gain-from-food",
        "sheep-reproduce",
        "wolf-reproduce",
    ],
    "bounds": [
        [1, 100000],
        [20.0, 40.0],
        [2.0, 8.0],
        [16.0, 32.0],
        [2.0, 8.0],
        [2.0, 8.0],
    ],
}
```

The SALib sampler will automatically generate an appropriate number of samples for Sobol analysis, using a revised Saltelli sampling sequence. To calculate first-order, second-order and total sensitivity indices, this gives a sample size of $n(2p+2)$, where p is the number of input parameters, and n is a baseline sample size which should be large enough to stabilize the estimation of the indices. For this example, we use n = 1000, for a total of 14000 experiments.

[3]:
```
n = 1024
param_values = sobolsample.sample(problem, n, calc_second_order=True)
```

The sampler generates an input array of shape ($n(2p+2)$, p) with rows for each experiment and columns for each input parameter.

[4]:
```
param_values.shape
```

[4]:
```
(14336, 6)
```

1.3.1 Running the experiments in parallel using ipyparallel

Ipyparallel is a standalone package (available through the pip package manager) which can be used to interactively run parallel tasks from IPython on a single PC, but also on multiple computers. On machines with multiple cores, this can significantly improve performance: for instance, the multiple simulations required for a sensitivity analysis are easy to run in parallel. Documentation for Ipyparallel is available at http://ipyparallel.readthedocs.io/en/latest/intro.html.

Ipyparallel first requires starting a controller and multiple engines, which can be done from a terminal or command prompt, or conveniently from within a notebook.

[5]:
```
import ipyparallel as ipp

cluster = ipp.Cluster(n=4)
cluster.start_cluster_sync();
```

```
Starting 4 engines with <class 'ipyparallel.cluster.launcher.LocalEngineSetLauncher'>
```

Next, we can connect the interactive notebook to the started cluster by instantiating a client, and checking that client.ids returns a list of 4 available engines.

[7]:
```
rc = cluster.connect_client_sync()
rc.ids
```

[7]:
```
[0, 1, 2, 3]
```

With the client setup, we can now interact with the cluster. We can for example get a direct view of all engines in the cluster.

[8]:
```
direct_view = rc[:]
```

[9]:
```
import os

# Push the current working directory of the notebook to a "cwd" variable on the engines␣
↪that can be accessed later
direct_view.push(dict(cwd=os.getcwd()), block=True)
```

[9]:
```
[None, None, None, None]
```

[10]:
```
# Push the "problem" variable from the notebook to a corresponding variable on the␣
↪engines
direct_view.push(dict(problem=problem), block=True)
```

[10]:
```
[None, None, None, None]
```

The %%px command can be added to a notebook cell to run it in parallel on each of the engines. Here the code first involves some imports and a change of the working directory. We then start a link to NetLogo, and load the example model on each of the engines.

[12]:
```
%%px
import os
os.chdir(cwd)

import pynetlogo
import pandas as pd
```

(continues on next page)

(continued from previous page)

```
netlogo = pynetlogo.NetLogoLink(gui=False)
netlogo.load_model('./models/Wolf Sheep Predation_v6.nlogo')
```

```
%px:   0%|          | 0/4 [00:00<?, ?tasks/s]
```

We can then use the IPyparallel map functionality to run the sampled experiments, now using a "load balanced" view to automatically handle the scheduling and distribution of the simulations across the engines. This is for instance useful when simulations may take different amounts of time.

We first set up a simulation function that takes a single experiment (i.e. a vector of input parameters) as an argument, and returns the outcomes of interest in a pandas Series.

[13]:
```
def simulation(experiment):

    # Set the input parameters
    for i, name in enumerate(problem["names"]):
        if name == "random-seed":
            # The NetLogo random seed requires a different syntax
            netlogo.command("random-seed {}".format(experiment[i]))
        else:
            # Otherwise, assume the input parameters are global variables
            netlogo.command("set {0} {1}".format(name, experiment[i]))

    netlogo.command("setup")
    # Run for 100 ticks and return the number of sheep and wolf agents at each time step
    counts = netlogo.repeat_report(["count sheep", "count wolves"], 100)

    results = pd.Series(
        [counts["count sheep"].values.mean(), counts["count wolves"].values.mean()],
        index=["Avg. sheep", "Avg. wolves"],
    )

    return results
```

We then create a load balanced view and run the simulation with the `map_sync` method. This method takes a function and a Python sequence as arguments, applies the function to each element of the sequence, and returns results once all computations are finished.

In this case, we pass the simulation function and the array of experiments (param_values), so that the function will be executed for each row of the array.

The DataFrame constructor is then used to immediately build a DataFrame from the results (which are returned as a list of Series). The `to_csv` method provides a simple way of saving the results to disk; pandas supports several more advanced storage options, such as serialization with msgpack, or hierarchical HDF5 storage.

[15]:
```
lview = rc.load_balanced_view()

results = pd.DataFrame(lview.map_sync(simulation, param_values))
```

[16]:
```
results.to_csv("./data/Sobol_parallel.csv")
```

[17]:
```
results.head(5)
```

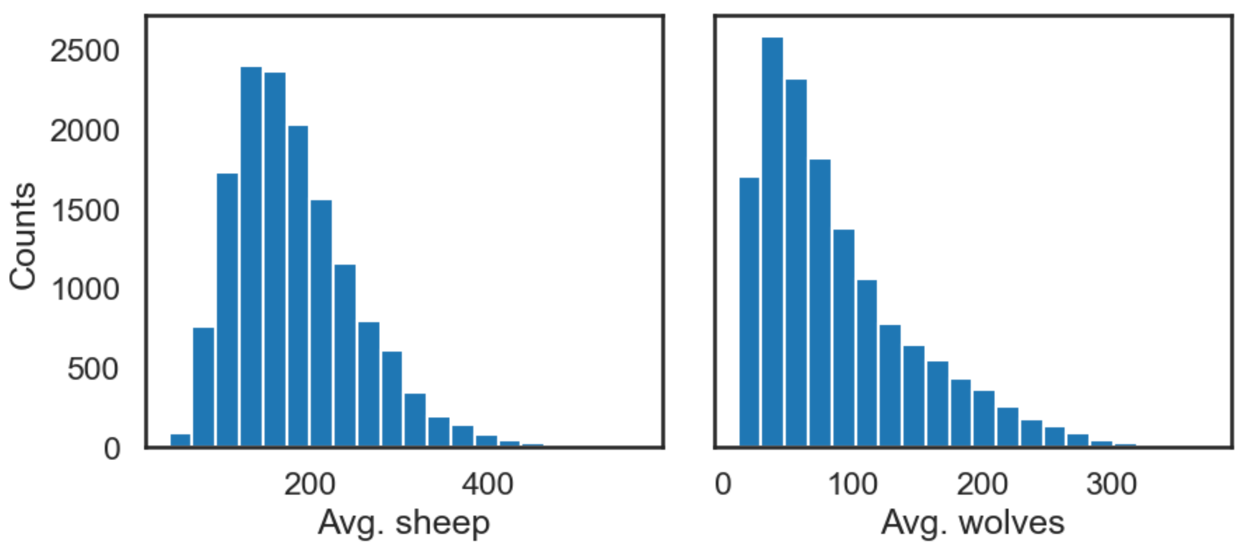
[17]:

	Avg. sheep	Avg. wolves
0	106.861386	82.128713
1	109.465347	65.158416
2	106.861386	82.128713
3	133.267327	154.594059
4	129.297030	45.990099

1.3.2 Using SALib for sensitivity analysis

We can then proceed with the analysis, first using a histogram to visualize output distributions for each outcome:

[18]:
```
fig, ax = plt.subplots(1, len(results.columns), sharey=True)

for i, n in enumerate(results.columns):
    ax[i].hist(results[n], 20)
    ax[i].set_xlabel(n)
ax[0].set_ylabel("Counts")

fig.set_size_inches(10, 4)
fig.subplots_adjust(wspace=0.1)

plt.show()
```



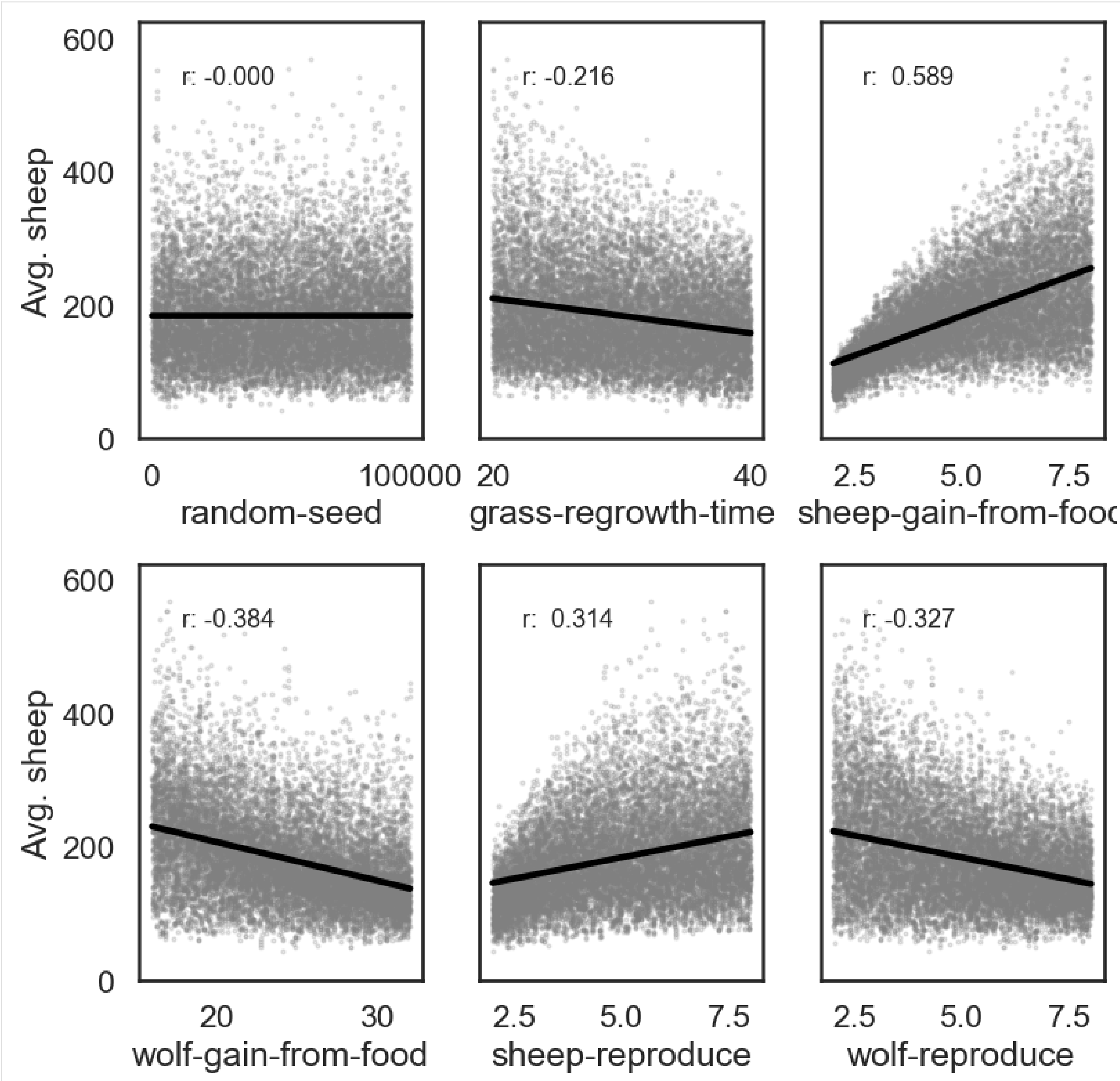
Bivariate scatter plots can be useful to visualize relationships between each input parameter and the outputs. Taking the outcome for the average sheep count as an example, we obtain the following, using the scipy library to calculate the Pearson correlation coefficient (r) for each parameter, and the seaborn library to plot a linear trend fit.

[21]:
```
import scipy

nrow = 2
ncol = 3

fig, ax = plt.subplots(nrow, ncol, sharey=True)

```

(continues on next page)

(continued from previous page)

```
y = results["Avg. sheep"]

for i, a in enumerate(ax.flatten()):
    x = param_values[:, i]
    sns.regplot(
        x=x,
        y=y,
        ax=a,
        ci=None,
        color="k",
        scatter_kws={"alpha": 0.2, "s": 4, "color": "gray"},
    )
    pearson = scipy.stats.pearsonr(x, y)
    a.annotate(
        "r: {:6.3f}".format(pearson[0]),
        xy=(0.15, 0.85),
        xycoords="axes fraction",
        fontsize=13,
    )
    if divmod(i, ncol)[1] > 0:
        a.get_yaxis().set_visible(False)
    a.set_xlabel(problem["names"][i])
    a.set_ylim([0, 1.1 * np.max(y)])

fig.set_size_inches(9, 9, forward=True)
fig.subplots_adjust(wspace=0.2, hspace=0.3)

plt.show()
```



This indicates a positive relationship between the "sheep-gain-from-food" parameter and the mean sheep count, and negative relationships for the "wolf-gain-from-food" and "wolf-reproduce" parameters.

We can then use SALib to calculate first-order (S1), second-order (S2) and total (ST) Sobol indices, to estimate each input's contribution to output variance as well as input interactions (again using the mean sheep count). By default, 95% confidence intervals are estimated for each index.

[22]:
```
Si = sobol.analyze(
    problem,
    results["Avg. sheep"].values,
    calc_second_order=True,
    print_to_console=False,
)
```

As a simple example, we first select and visualize the total and first-order indices for each input, converting the dictionary returned by SALib to a DataFrame. The default pandas plotting method is then used to plot these indices along with their estimated confidence intervals (shown as error bars).

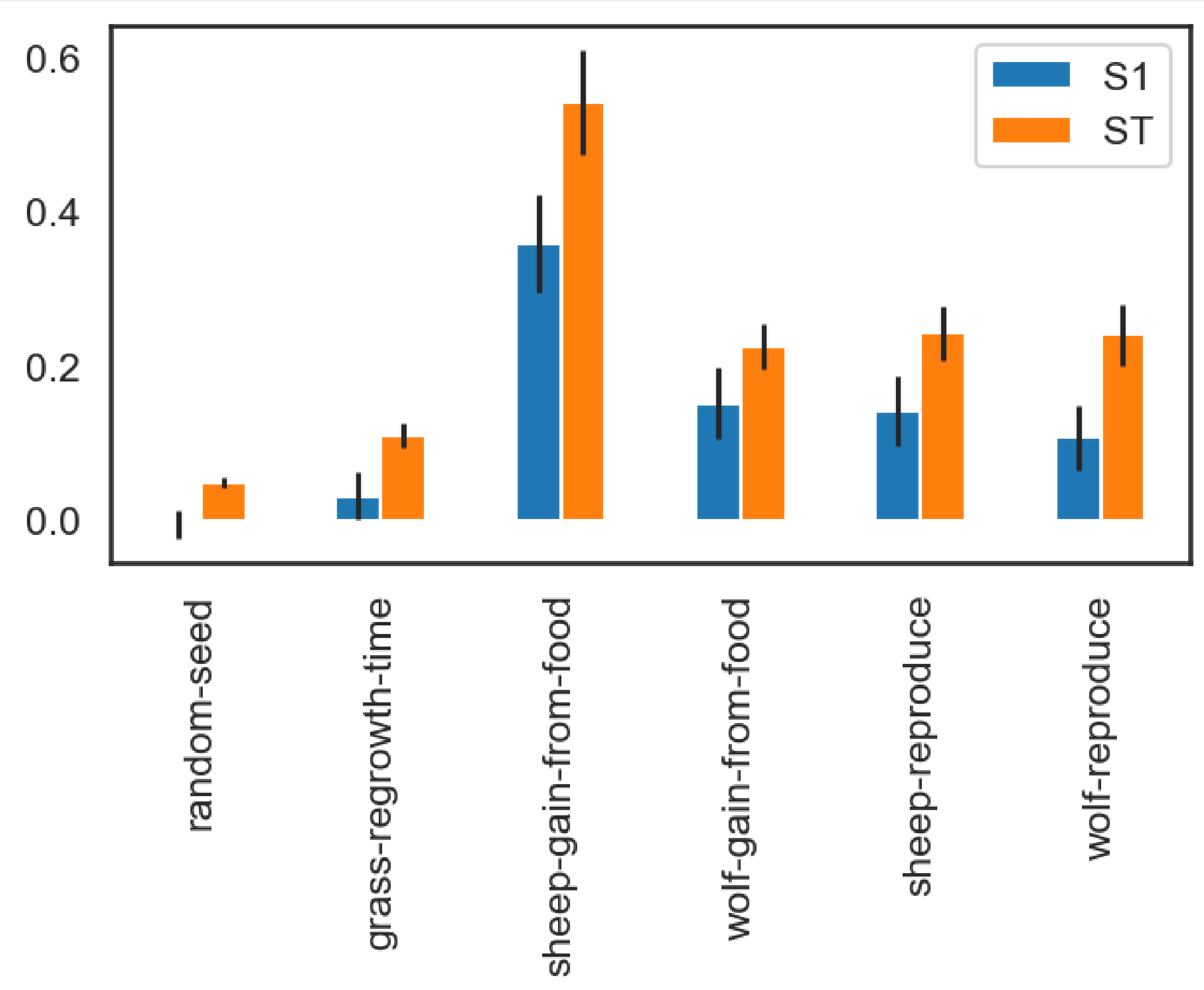
[23]:
```
Si_filter = {k: Si[k] for k in ["ST", "ST_conf", "S1", "S1_conf"]}
Si_df = pd.DataFrame(Si_filter, index=problem["names"])
```

[24]:
```
Si_df
```

[24]:

	ST	ST_conf	S1	S1_conf
random-seed	0.050776	0.006568	-0.004114	0.017626
grass-regrowth-time	0.111551	0.016164	0.032316	0.030256
sheep-gain-from-food	0.543193	0.067641	0.359385	0.062889
wolf-gain-from-food	0.225840	0.028651	0.152584	0.045471
sheep-reproduce	0.243577	0.034768	0.142665	0.044973
wolf-reproduce	0.240973	0.039485	0.108223	0.042038

[25]:
```
fig, ax = plt.subplots(1)

indices = Si_df[["S1", "ST"]]
err = Si_df[["S1_conf", "ST_conf"]]

indices.plot.bar(yerr=err.values.T, ax=ax)
fig.set_size_inches(8, 4)

plt.show()
```



The "sheep-gain-from-food" parameter has the highest ST index, indicating that it contributes over 50% of output variance when accounting for interactions with other parameters. However, it can be noted that confidence bounds are still quite broad with this sample size, particularly for the S1 index (which indicates each input's individual contribution to variance).

We can use a more sophisticated visualization to include the second-order interactions between inputs estimated from the S2 values.

[30]:
```
%matplotlib inline
import itertools
from math import pi


def normalize(x, xmin, xmax):
    return (x - xmin) / (xmax - xmin)


def plot_circles(ax, locs, names, max_s, stats, smax, smin, fc, ec, lw, zorder):
    s = np.asarray([stats[name] for name in names])
    s = 0.01 + max_s * np.sqrt(normalize(s, smin, smax))
```

(continues on next page)

(continued from previous page)

```
    fill = True
    for loc, name, si in zip(locs, names, s):
        if fc == "w":
            fill = False
        else:
            ec = "none"

        x = np.cos(loc)
        y = np.sin(loc)

        circle = plt.Circle(
            (x, y),
            radius=si,
            ec=ec,
            fc=fc,
            transform=ax.transData._b,
            zorder=zorder,
            lw=lw,
            fill=True,
        )
        ax.add_artist(circle)


def filter(sobol_indices, names, locs, criterion, threshold):
    if criterion in ["ST", "S1", "S2"]:
        data = sobol_indices[criterion]
        data = np.abs(data)
        data = data.flatten()  # flatten in case of S2
        # TODO:: remove nans

        filtered = [(name, locs[i]) for i, name in enumerate(names) if data[i] >␣
↪threshold]
        filtered_names, filtered_locs = zip(*filtered)
    elif criterion in ["ST_conf", "S1_conf", "S2_conf"]:
        raise NotImplementedError
    else:
        raise ValueError("unknown value for criterion")

    return filtered_names, filtered_locs


def plot_sobol_indices(sobol_indices, criterion="ST", threshold=0.01):
    """plot sobol indices on a radial plot

    Parameters
    ----------
    sobol_indices : dict
                    the return from SAlib
    criterion : {'ST', 'S1', 'S2', 'ST_conf', 'S1_conf', 'S2_conf'}, optional
    threshold : float
                only visualize variables with criterion larger than cutoff
```

(continues on next page)

(continued from previous page)

```
    """
    max_linewidth_s2 = 15  # 25*1.8
    max_s_radius = 0.3

    # prepare data
    # use the absolute values of all the indices
    # sobol_indices = {key:np.abs(stats) for key, stats in sobol_indices.items()}

    # dataframe with ST and S1
    sobol_stats = {key: sobol_indices[key] for key in ["ST", "S1"]}
    sobol_stats = pd.DataFrame(sobol_stats, index=problem["names"])

    smax = sobol_stats.max().max()
    smin = sobol_stats.min().min()

    # dataframe with s2
    s2 = pd.DataFrame(sobol_indices["S2"], index=problem["names"], columns=problem["names
↪"])
    s2[s2 < 0.0] = 0.0  # Set negative values to 0 (artifact from small sample sizes)
    s2max = s2.max().max()
    s2min = s2.min().min()

    names = problem["names"]
    n = len(names)
    ticklocs = np.linspace(0, 2 * pi, n + 1)
    locs = ticklocs[0:-1]

    filtered_names, filtered_locs = filter(sobol_indices, names, locs, criterion,␣
↪threshold)

    # setup figure
    fig = plt.figure()
    ax = fig.add_subplot(111, polar=True)
    ax.grid(False)
    ax.spines["polar"].set_visible(False)

    ax.set_xticks(locs)
    ax.set_xticklabels(names)

    ax.set_yticklabels([])
    ax.set_ylim(top=1.4)
    legend(ax)

    # plot ST
    plot_circles(
        ax,
        filtered_locs,
        filtered_names,
        max_s_radius,
        sobol_stats["ST"],
        smax,
```

(continues on next page)

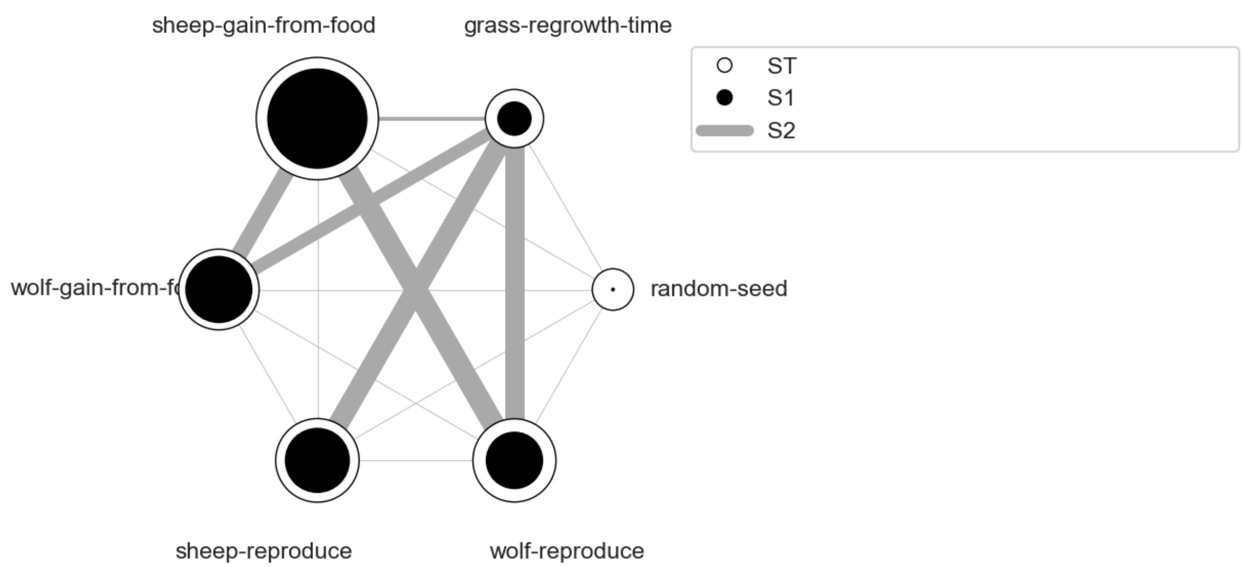
(continued from previous page)

```
        smin,
        "w",
        "k",
        1,
        9,
    )

    # plot S1
    plot_circles(
        ax,
        filtered_locs,
        filtered_names,
        max_s_radius,
        sobol_stats["S1"],
        smax,
        smin,
        "k",
        "k",
        1,
        10,
    )

    # plot S2
    for name1, name2 in itertools.combinations(zip(filtered_names, filtered_locs), 2):
        name1, loc1 = name1
        name2, loc2 = name2

        weight = s2.loc[name1, name2]
        lw = 0.5 + max_linewidth_s2 * normalize(weight, s2min, s2max)
        ax.plot([loc1, loc2], [1, 1], c="darkgray", lw=lw, zorder=1)

    return fig


from matplotlib.legend_handler import HandlerPatch


class HandlerCircle(HandlerPatch):
    def create_artists(
        self, legend, orig_handle, xdescent, ydescent, width, height, fontsize, trans
    ):
        center = 0.5 * width - 0.5 * xdescent, 0.5 * height - 0.5 * ydescent
        p = plt.Circle(xy=center, radius=orig_handle.radius)
        self.update_prop(p, orig_handle, legend)
        p.set_transform(trans)
        return [p]


def legend(ax):
    some_identifiers = [
        plt.Circle((0, 0), radius=5, color="k", fill=False, lw=1),
        plt.Circle((0, 0), radius=5, color="k", fill=True),
```

(continues on next page)

(continued from previous page)

```
        plt.Line2D([0, 0.5], [0, 0.5], lw=8, color="darkgray"),
    ]
    ax.legend(
        some_identifiers,
        ["ST", "S1", "S2"],
        loc=(1, 0.75),
        borderaxespad=0.1,
        mode="expand",
        handler_map={plt.Circle: HandlerCircle()},
    )


sns.set_style("whitegrid")
fig = plot_sobol_indices(Si, criterion="ST", threshold=0.005)
fig.set_size_inches(7, 7)
plt.show()
```



In this case, the "sheep-gain-from-food" variable has strong interactions with the "wolf-gain-from-food" and "wolf-reproduce" inputs in particular. The size of the ST and S1 circles correspond to the normalized variable importances.

[ ]:

1.4 Example 3: Sensitivity analysis for a NetLogo model with SALib and Multiprocessing

This is a short demo similar to example two but using the multiprocessing Pool All files used in the example are available from the pyNetLogo repository at https://github.com/quaquel/pyNetLogo. This code requires python3.

For in depth discussion, please see example 2.

1.4.1 Running the experiments in parallel using a Process Pool

There are multiple libraries available in the python ecosystem for performing tasks in parallel. One of the default libraries that ships with Python is concurrent.futures. This is in fact a high level interface around several other libraries. See the documentation for details. One of the libraries wrapped by concurrent.futures is multiprocessing. Below we use multiprocessing, anyone on python 3.8 or newer can use the either code below or use the ProcessPoolExecuturor from concurrent.futures (recommended).

Here we are going to use the ProcessPoolExecutor, which uses the multiprocessing library. Parallelization is an advanced topic and the exact way in which it is to be done depends at least in part on the operating system one is using. It is recommended to carefully read the documentation provided by both concurrent.futures and mulitprocessing. This example is ran on a mac, linux is expected to be similar but Windows is likely to be slightly different

```
from multiprocessing import Pool
import os
import pandas as pd

import pynetlogo

from SALib.sample import sobol as sobolsample

def initializer(modelfile):
    """initialize a subprocess

    Parameters
    ----------
    modelfile : str

    """

    # we need to set the instantiated netlogo
    # link as a global so run_simulation can
    # use it
    global netlogo

    netlogo = pynetlogo.NetLogoLink(gui=False)
    netlogo.load_model(modelfile)


def run_simulation(experiment):
    """run a netlogo model

    Parameters
    ----------
    experiments : dict

    """

    # Set the input parameters
    for key, value in experiment.items():
        if key == "random-seed":
            # The NetLogo random seed requires a different syntax
            netlogo.command("random-seed {}".format(value))
        else:
```

(continues on next page)

(continued from previous page)

```
            # Otherwise, assume the input parameters are global variables
            netlogo.command("set {0} {1}".format(key, value))

    netlogo.command("setup")
    # Run for 100 ticks and return the number of sheep and
    # wolf agents at each time step
    counts = netlogo.repeat_report(["count sheep", "count wolves"], 100)

    results = pd.Series(
        [counts["count sheep"].values.mean(), counts["count wolves"].values.mean()],
        index=["Avg. sheep", "Avg. wolves"],
    )
    return results


if __name__ == "__main__":
    modelfile = os.path.abspath("./models/Wolf Sheep Predation_v6.nlogo")

    problem = {
        "num_vars": 6,
        "names": [
            "random-seed",
            "grass-regrowth-time",
            "sheep-gain-from-food",
            "wolf-gain-from-food",
            "sheep-reproduce",
            "wolf-reproduce",
        ],
        "bounds": [[1, 100000], [20.0, 40.0], [2.0, 8.0], [16.0, 32.0], [2.0, 8.0], [2.0,
↪ 8.0]],
    }

    n = 1024
    param_values = sobolsample.sample(problem, n, calc_second_order=True)

    # cast the param_values to a dataframe to
    # include the column labels
    experiments = pd.DataFrame(param_values, columns=problem["names"])

    with Pool(4, initializer=initializer, initargs=(modelfile,)) as executor:
        results = []
        for entry in executor.map(run_simulation, experiments.to_dict("records")):
            results.append(entry)
        results = pd.DataFrame(results)
```

[ ]:

```

```

1.5 core

exception pynetlogo.core.**NetLogoException**

Base project exception

class pynetlogo.core.**NetLogoLink**(*gui=False*, *thd=False*, *netlogo_home=None*, *jvm_path=None*, *jvmargs=[]*)

Create a link with NetLogo. Underneath, the NetLogo JVM is started through Jpype.

If *netlogo_home*, *netlogo_version*, or *jvm_home* are not provided, the link will try to identify the correct parameters automatically on Mac or Windows. *netlogo_home* and *netlogo_version* are required on Linux.

Parameters

- **gui** (*bool, optional*) – If true, displays the NetLogo GUI (not supported on Mac)
- **thd** (*bool, optional*) – If true, use NetLogo 3D
- **netlogo_home** (*str, optional*) – Path to the NetLogo installation directory (required on Linux)
- **jvm_path** (*str, optional*) – path of the jvm
- **jvmargs** (*list of str, optional*) – additional arguments that should be used when starting the jvm

command(*netlogo_command*)

Execute the supplied command in NetLogo

Parameters

netlogo_command (*str*) – Valid NetLogo command

Raises

NetLogoException – If a LogoException or CompilerException is raised by NetLogo

kill_workspace()

Close NetLogo and shut down the JVM.

load_model(*path*)

Load a NetLogo model.

Parameters

path (*str*) – Path to the NetLogo model

Raises

- **FileNotFoundError** – in case path does not exist
- *NetLogoException* – In case of a NetLogo exception

patch_report(*attribute*)

Return patch attributes from NetLogo

Returns a pandas DataFrame with same dimensions as the NetLogo world, with column labels and row indices following pxcor and pycor patch coordinates. Values of the dataframe correspond to patch attributes.

Parameters

attribute (*str*) – Valid NetLogo patch attribute

Returns

DataFrame containing patch attributes

Return type
pandas DataFrame

Raises
NetLogoException – If a LogoException or CompilerException is raised by NetLogo

patch_set(*attribute*, *data*)

Set patch attributes in NetLogo

Inverse of the *patch_report* method. Sets a patch attribute using values from a pandas DataFrame of same dimensions as the NetLogo world.

Parameters

- **attribute** (*str*) – Valid NetLogo patch attribute
- **data** (*Pandas DataFrame*) – DataFrame with same dimensions as NetLogo world

Raises
NetLogoException – If a LogoException or CompilerException is raised by NetLogo

repeat_command(*netlogo_command*, *reps*)

Execute the supplied command in NetLogo a given number of times

Parameters

- **netlogo_command** (*str*) – Valid NetLogo command
- **reps** (*int*) – Number of repetitions for which to repeat commands

Raises
NetLogoException – If a LogoException or CompilerException is raised by NetLogo

repeat_report(*netlogo_reporter*, *reps*, *go='go'*, *include_t0=True*)

Return values from a NetLogo reporter over a number of ticks.

Can be used with multiple reporters by passing a list of strings. The values of the returned DataFrame are formatted following the data type returned by the reporters (numerical or string data, with single or multiple values). If the reporter returns multiple values, the results are converted to a numpy array.

netlogo_reporter
[str or list of str] Valid NetLogo reporter(s)

reps
[int] Number of NetLogo ticks for which to return values

go
[str, optional] NetLogo command for running the model ('go' by default)

include_t0
[boolean, optional] include the value of the reporter at t0, prior to running the go command

dict

key is the reporter, and the value is a list order by ticks

NetLogoException
If reporters are not in a valid format, or if a LogoException or CompilerException is raised by NetLogo

This method relies on files to send results from netlogo back to Python. This is slow and can break when used at scale. For such use cases, you are better of using a model specific way of interfacing. For example, have a go routine which accumulates the relevant reporters into lists. First run the model for the required time steps using command, and next retrieve the lists through report.

report(*netlogo_reporter*)

Return values from a NetLogo reporter

Any reporter (command which returns a value) that can be called in the NetLogo Command Center can be called with this method.

Parameters
`netlogo_reporter` (*str*) – Valid NetLogo reporter

Raises
NetLogoException – If a LogoException or CompilerException is raised by NetLogo

report_while(*netlogo_reporter*, *condition*, *command='go'*, *max_seconds=10*)

Return values from a NetLogo reporter while a condition is true in the NetLogo model

Parameters

- `netlogo_reporter` (*str*) – Valid NetLogo reporter
- `condition` (*str*) – Valid boolean NetLogo reporter
- `command` (*str*) – NetLogo command used to execute the model
- `max_seconds` (*int, optional*) – Time limit used to break execution

Raises
NetLogoException – If a LogoException or CompilerException is raised by NetLogo

write_NetLogo_attriblist(*agent_data*, *agent_name*)

Update attributes of a set of NetLogo agents from a DataFrame

Assumes a set of NetLogo agents of the same type. Attribute values can be numerical or strings.

Parameters

- `agent_data` (*pandas DataFrame*) – DataFrame indexed with a row for each agent, and columns for each attribute to update. Requires a 'who' column for the NetLogo agent ID
- `agent_name` (*str*) – Name of the NetLogo agent type to update (singular, e.g. a-sheep)

Raises
NetLogoException – If a LogoException or CompilerException is raised by NetLogo

1.6 Changelog

1.6.1 Version 0.5

- support for netlogo 6.3
- dropped support for netlogo 5.x and 6.0
- renamed library from pyNetLogo to pynetlogo to abide with pep8 naming conventions
- minor changes in names of keyword arguments of various methods
- shift from setup.py to pyproject.toml

- removal of python 2 support

1.6.2 Version 0.4

support for NetLogo 6.1 and 6.2

1.6.3 Version 0.3

- new repeat_report method
- load_model now raises a FileNotFoundError if the model can't be found
- use temporary folders created by tempfile module in repeat_report (contributed by tfrench)
- extensions now no longer need to be copied to the model directory (contributed by tfrench)
- addition keyword argument on init of PyNetLogo link for passing additional arguments to jvm
- additional documentation

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX