desdeo-vis Documentation

Release v0.1.5-39-gb12b96d

Frankie Robertson

Aug 28, 2018

Contents:

CHAPTER 1

desdeo-vis

Visualisations and Jupyter Notebook enabled preference selection widgets for the DESDEO interactive multiobjective optimization library.

Currently features:

- Parallel coordinate plots based on Vega v3.
- Preference selection for NIMBUS (first stage only).

1.1 Installation / Usage

Typically you should install this at the same time as DESDEO, by following the instructions there.

1.2 Development

1.2.1 Compilation

Run:

```
npm run watch
```

You will need to refresh your browser, and possibly reload the Jupyter kernel to see some changes.

1.2.2 Adding/modifying a notebook

Each notebook has two versions, one in the `desdeo_notebooks` directory and one in the `desdeo_notebooks/output` directory. The version in the prior directory should be scrubbed of all output and is the canonical copy. Currently they have have to be kept in sync manually, e.g. by regenerating the output notebook from the canonical version. When regenerating, make sure to save widget state with `Widgets > Save Notebook Widget State`.

1.2.3 Known issues

If you are developing using a virtualenv (recommended) you may have trouble widget Javascript from outside the virtualenv getting picked up, meaning you can't test your changes. This is a problem when you have installed desdeo-vis outside the virtualenv either system-wide or per-user. Currently, there's no clear way to isolate this Javascript, so the solution is to uninstall the other versions of desdeo-vis and make sure to only install it inside virtualenvs.

1.2.4 Release process

1. Update the version number in `desdeo_vis/_version.py` and `package.json`, and run `npm install` to update it in `package-lock.json` too.
2. Add an entry to HISTORY.md.
3. Make a release commit.
4. Make a git tag of this commit with `git tag v$VERSION`
5. Push – including the tags with `git push && git push --tags`
6. Upload to PyPI with `rm -rf build/ && python setup.py sdist bdist_wheel` and `twine upload dist/*`

CHAPTER 2

Example notebooks

2.1 Solving the river pollution problem using NIMBUS

Welcome to this notebook demonstrating the DESDEO interactive multi-objective optimisation framework.

In this notebook we will consider a toy problem dealing with water quality management, and the pollution produced by a fishery. We want to maximise the water quality as measured in two locations: the fishery and a downstream city (the measurements are based on dissolved oxygen concentration). Additionally, we want to maximise the ROI (Return On Investment) of the fishery. Simultaneously we want to minimise the additional tax which residents of the city must pay.

To run this example, you need to run each individual code block (the ones saying `In [ ]:` to their left) by clicking on each one and then clicking "Run". Please refer to the background section of the documentation for information about NIMBUS.

2.1.1 Imports

This first code snippet simply imports the parts of DESDEO we need for this notebook. We import optimisation methods, NIMBUS and the RiverPollution problem definition from the desdeo module, and visualisation and preference selection tools and widgets from the desdeo_vis module.

```
In [1]: from desdeo.method.NIMBUS import NIMBUS
        from desdeo.optimization import SciPyDE
        from desdeo.problem.toy import RiverPollution

        from desdeo_vis.widget import NimbusPrefWidget, ParplotWidget
```

2.1.2 Initial iteration and preference selection

First we initialise the RiverPollution problem and the NIMBUS solution method. Then we get an initial result. We can plot solutions at any time using `parplot`.

```
In [2]: problem = RiverPollution()
        method = NIMBUS(problem, SciPyDE)
        results = method.init_iteration()

        ParplotWidget(results.objective_vars, problem)
```

```
ParplotWidget(cur_max_as_min=True, maximized=[True, True, True, False], orig_max_as_min=True, spec={
```

Next, we will give our first preference based on NIMBUS. We construct a `NimbusPrefWidget`, save it to a variable and display it. You can now specify your preferences using the displayed widget. If you're not sure how to use it, read the documentation on classification in NIMBUS.

```
In [3]: pref = NimbusPrefWidget(results.objective_vars, problem)
        pref
```

```
NimbusPrefWidget(cur_max_as_min=True, maximized=[True, True, True, False], orig_max_as_min=True, spec
```

2.1.3 Solutions based on preference

We can now generate a new set of results based on this preference. Note that this will raise an `InvalidNimbusPreferencesException` if you run it while the above preferences are invalid.

```
In [4]: results2_all = method.next_iteration(preference=pref.nimbus_clf(method))

        ParplotWidget(results2_all.objective_vars, problem)
```

```
ParplotWidget(cur_max_as_min=True, maximized=[True, True, True, False], orig_max_as_min=True, spec={
```

We might choose to generate less extra solutions...

```
In [5]: results2_less = method.next_iteration(preference=pref.nimbus_clf(method), num_scalars=2)

        ParplotWidget(results2_less.objective_vars, problem)
```

```
ParplotWidget(cur_max_as_min=True, maximized=[True, True, True, False], orig_max_as_min=True, spec={
```

We can also choose a subset of scalarization functions from NIM, ACH, GUESS, STOM. These are the NIMBUS scalarization function and the NIMBUS version of the achievement, guess and satisficing trade-off functions respectively.

```
In [6]: results2_spec = method.next_iteration(preference=pref.nimbus_clf(method), scalars=['NIM', 'GU

        ParplotWidget(results2_spec.objective_vars, problem)
```

```
ParplotWidget(cur_max_as_min=True, maximized=[True, True, True, False], orig_max_as_min=True, spec={
```

2.1.4 Generating intermediate solutions

If none of these solutions exactly satisfy us, we can generate and view solutions between two solutions we've generated so far. Here we generate 4 solutions between the solutions generated by the NIMBUS and GUESS scalarisation functions.

```
In [7]: results3 = method.between(results2_spec.objective_vars[0], results2_spec.objective_vars[1], 4

        ParplotWidget(results3.objective_vars, problem)
```

```
ParplotWidget(cur_max_as_min=True, maximized=[True, True, True, False], orig_max_as_min=True, spec={
```

2.2 Solving the cylinder problem using NIMBUS

Let's consider a cell shaped like a cylinder, that is, a circular cross-sectional prism. The shape of the cell is here determined by two quantities, its radius x_1 and its height x_2. We want to maximize the volume of the cylinder and minimize the surface area. In addition to this, cylinder's height should be close to 15 units.

The volume of a cylinder is the product of its base area and height. A cylinder can be cut and unrolled into a rectangle and the surface area of this rectangle is the product of its height and the perimeter of the circle $2\pi x_1 x_2$. The sum of the cylinder's two flat circular caps is $2\pi x_1^2$. The total surface area of the cylinder with flat circular ends is then $2\pi x_1^2 + 2\pi x_1 x_2$.

Three functions can be made from the above information: the one describing the volume of the cylinder, the other telling the surface area and the last measuring the height difference.

So the problem is:

maximize Volume = $\pi x_1^2 x_2$
minimize SurfaceArea = $2\pi x_1^2 + 2\pi x_1 x_2$
minimize HeightDiff = $|x2 - 15.0|$

Let's assume that the cylinder's height must be greater or equal to its width. This information gives us the following constraint:

$g(x) = 2x_1 - x_2 \leq 0$

To run this example, you need to run each individual code block (the ones saying `In [ ]:` to their left) by clicking on each one and then clicking "Run". Please refer to the background section of the documentation for information about NIMBUS.

2.2.1 Imports

This first code snippet simply imports the parts of DESDEO we need for this notebook. We import optimisation methods, NIMBUS and the CylinderProblem problem definition from the desdeo module, and visualisation and preference selection tools and widgets from the desdeo_vis module.

```
In [1]: from desdeo.method.NIMBUS import NIMBUS
        from desdeo.optimization import SciPyDE
        from desdeo.problem.toy import CylinderProblem

        from desdeo_vis.widget import NimbusPrefWidget, ParplotWidget
```

2.2.2 Getting a solutions based on a preference

First we initialise the `CylinderProblem` problem and the `NIMBUS` solution method. Then we get an initial result. We can plot solutions at any time using `ParplotWidget`.

```
In [2]: problem = CylinderProblem()
        method = NIMBUS(problem, SciPyDE)
        results = method.init_iteration()

        ParplotWidget(results.objective_vars, problem)
```

```
ParplotWidget(cur_max_as_min=True, maximized=[True, False], orig_max_as_min=True, spec={'$schema': 'h
```

The first solution we get from NIMBUS is reasonable. However, we want to increase the cylinder's volume as much as possible, still keeping the surface area and height difference low.

To do this, first display a `NimbusPrefWidget`. After executing the cell, a widget will display which you can use as follows: * For now we let the volume vary freely by selecting <> from the leftmost dropdown. * The next column has the preferences for the surface area function. We want to know how much the volume will be when the surface area is less than 1900, so we set the dropdown to >= and type `1900` into the textbox next to it. * For height difference, we won't accept a worse solution than the current one, so we set the dropdown to <=.

```
In [3]: pref = NimbusPrefWidget(results.objective_vars, problem)
        pref
```

```
NimbusPrefWidget(cur_max_as_min=True, maximized=[True, False], orig_max_as_min=True, spec={'$schema'
```

Now we can plot the solutions given by NIMBUS.

```
In [5]: results2 = method.next_iteration(preference=pref.nimbus_clf(method))

        ParplotWidget(results2.objective_vars, problem)
```

```
INFEASIBLE 277.360873
```

```
ParplotWidget(cur_max_as_min=True, maximized=[True, False], orig_max_as_min=True, spec={'$schema': 'h
```

CHAPTER 3

API documentation

desdeo_vis.plot	This module contains the machinery for doing non-interactive plotting of multidimensional data.
desdeo_vis.widget	This module contains Jupyter widgets for interactively specifying preferences as well as interactively and non-interactively displaying solutions.

3.1 desdeo_vis.plot

This module contains the machinery for doing non-interactive plotting of multidimensional data. Currently it can generate VEGA specifications for parallel coordinates plots.

3.2 desdeo_vis.widget

This module contains Jupyter widgets for interactively specifying preferences as well as interactively and non-interactively displaying solutions.

Python Module Index

d

Index

D