PyAnalog

The Analog People

Apr 04, 2022

Introduction

Abstract and Summary of PyAnalog

`pyanalog` is a Python3[1] package and demonstrator/research code for a software stack for analog computers. It empowers users to

- manipulate ordinary differential equations in a way suitable for procession with the exemplary Analog Paradigm Model-1[2] analog computer or similar *high-level* architectures in terms of OpAmp-Level circuit description (*not* SPICE-level circuit description).
- simulate abstract computing circuits made of elements such as *summers, integrators, differentiator and multipliers*. The heart of the code is a *custom ODE solver (based on C++ codegeneration)* (page 38), but also interfaces to *SciPy* (page 41) and *SymPy* (page 44) solvers exist. Notably, the C++ solver can also solve integro-differential equations.
- generate and manipulate *netlists and VHDL-like circuit descriptions* (page 81)
- compile against machine architectures on a macrocell-level, set digital potentiometer arrays, cross bar switches, digital switches, etc.
- *interface a Model-1 hybrid controller* (page 99) in order to run a program, steer the operation, gain and visualize/postprocess output data.

The codes are supposed to work well in the Scientific Python[3] ecosystem. The target audience are clearly software engineers and scientists. The user interface is either command line (bash or python shell) or scripting (Python or C/YAML-like domain specific languages). This code can interface with remote analog computers (AAAS – *analog computing as a service*).

The `pyanalog` code is open-source and a *research code which is worked actively on*, ie. it is "work in progress". It is authored Anabrid GmbH[4]. The code is currently dual-licensed by GPL-3[5] and a proprietary/commercial use license[6]. See http://www.anabrid.com/licensing for further details.

[1] https://www.python.org/
[2] http://analogparadigm.com/
[3] https://www.scipy.org/
[4] http://www.anabrid.com
[5] https://github.com/anabrid/pyanalog/blob/master/LICENSE.GPL3
[6] https://github.com/anabrid/pyanalog/blob/master/LICENSE.ANABRID

1 Introduction

1.1 Overview of PyAnalog

1.1.1 The big picture

```
Input: Ordinary Differential Equation in mathematical notation
       (for instance in SymPy, Sage, Matlab, Latex, etc.)
     |
     v

PyDDA: Generation of a circuit
     |
  +--+---+   Several backends to deal with the
  |  |   |   circuits, for instance:
  | ...  |
  |      +--> Simulation with numerics (C++ or SciPy/Netlib ODE solver)
  |
  v

PyFPAA:  Circuit synthesis (=mapping) with based on architecture
         description.
  |
  |
  v

PyHyCon: Interaction with Analog-as-a-service
         (SSH to the small analog machine in Bernds basement)
  |
  v

Output: Measurement data from all relevant fields
```

1.1.2 The particular python modules

First and foremost, this documentation is an API documentation. Therefore, it mostly covers the functions and classes exposed by the following three python modules:

- *PyDDA* (page 15) is a tool for simple algebraic transformations of equations and circuits as well as their numerical simulation. DDA stands for *Digital Differential Analyzer*.
- *PyFPAA* (page 81) is a tool for compiling an electrical circuit to a machine architecture. This especially programs digital potentiometers and cross bar switches and relies on a minimalistic hardware description language (HDL) written in YAML. FPAA stands for *Field Programmable Analog Array*.
- *PyHyCon* (page 99) is an interface to a Model-1 *Hybrid Controller* in order to run a program, steer the operation and gain output data.

You can read more about the goals of this software at the page *A rationale about DDA* (page 18).

Note: Currently, the DDA code is by far the biggest part of the pyanalog package. The Hybrid Controller client is also usable. The FPAA part is working on its own and there are interfaces to the PyHyCon, however the interface between DDA and FPAA is basically nonexistent in the moment.

1.2 Obtaining and installing PyAnalog

PyAnalog can be used on a "vanilla" Python installation without any dependencies. Only make sure you use a *recent* Python 3 installation: We use f-strings[7], so at least *Python 3.6* is required.

In order to use some of the advanced features, we recommend to install the following Python packages:

- PyYAML[8], for the intended usage of *PyFPAA* (page 81)
- SymPy[9], for the *SymPy/Computer Algebra interface* (page 44) of PyDDA
- SciPy[10], for the *SciPy interface* (page 41) of PyDDA. Furthermore, some additioanl postprocessing tools, as well as unit tests and examples of PyDDA require NumPy[11] and Matplotlib[12] to be installed.

The code is primarily developed on Linux, but has also been used successfully on Mac OS X and even Microsoft Windows. If you run one of these platforms and never got in contact to Python coding there yet, the following guides and tips can be helpful for getting a PyAnalog running on Mac or Windows.

[7] https://docs.python.org/3.6/whatsnew/3.6.html#whatsnew36-pep498
[8] https://pyyaml.org/
[9] https://www.sympy.org/
[10] http://www.scipy.org/
[11] https://numpy.org/
[12] https://matplotlib.org/

1.2.1 Obtaining the code

The PyAnalog code is public available at https://github.com/anabrid/pyanalog. Furthermore, we maintain an in-house (private) mirror at https://lab.analogparadigm.com/software/pyanalog. You don't need access to the later repository right now.

Note: We try to keep the master branch[13] in a working condition. If things do not work, you might want to checkout the last working commit in the master. You can recognize it at the github commit history[14] where there is a green checkmark at the tests (and not a red crossmark). I promise there will be releases/versions/tags soon :-)

1.2.2 Mac OS X-specific installation notes

First, make sure you have some (recent) Python 3 installed on your machine. You can check by opening a terminal and executing `python --version`. If still have installed a very old version such as Python 2.7, you need to upgrade it. There are many guides in the web[15]. It is likely that the manual installation of a newer Python version will give you two Pythons installed on your system. In many situations, you have to call the more recent version by invoking a command such as `python3` in place of `python`. The same aplies with the Python package manager `pip3` instead of `pip`. There is nothing wrong with this, and similar situations exist on some older Linux distributions where Python2 and Python3 live next to each other on the same system.

Mac OS X quickstart guide

The following code block shows a minimal way how quickly to start on Mac OS X. This assumes you have some C++ compiler available, e.g. from Xcode (Clang).

```
$ pip3 install scipy numpy matplotlib # the only dependencies you really␣
↪need
$ git clone https://github.com/anabrid/pyanalog.git
$ cd pyanalog
$ python3 setup.py develop --user # this way you can import the python␣
↪module from anwhere
$ cd examples/traditional-dda-circuits
$ python3 -m dda chua.dda c > chua.cpp
$ c++ --std=c++1z chua.cpp     # c++1z-Standard is specific to Mac / clang
$ ./a.out > chua.dat
$ gnuplot etc...
```

Note that with the Clang/LLVM compiler, you have to set `--std=c++1z` instead of `--std=c++17` (as with GCC).

[13] https://github.com/anabrid/pyanalog/commits/master
[14] https://github.com/anabrid/pyanalog/commits/master
[15] https://docs.python-guide.org/starting/install3/osx/

1.2.3 Microsoft Windows-specific installation notes

For Windows, there are several Python distributions available. The most prominent and mature is certainly Anaconda[16]. Conda is much into Python enviroments[17], which is somewhat orthogonal to the standard Python package manager pip[18]. However, there are straightforwards ways to install the requirements.txt in conda[19] (see also here how to use pip in an environment[20]).

Other options exist. For instance, you can install Python also directly from the Windows store (and use pip as usual). To do so, just open any Terminal (Powershell or `cmd.exe`) and type `python`. This will guide you straight to Windows store. You can also install the popular IDE Spyder[21] which ships a Python installation with itself. This is particularly handy because it provides also a lean way to setup the developer machine. For instance, there is a menu item *Tools/Current user environment variables...* which allows you to set the system wide `PATH`.

Regarding C++ compilers, we have made best experiences with MinGW[22], the GCC port for Windows. Make sure you add the installation directory to your `PATH` in order to be able to access the compiler from everywhere (i.e. every terminal).

Regarding Windows Environment variables

The `HKEY_CURRENT_USER\Environment` should, as a list, probably contain paths such as (in this example assumung you have installed MinGW and Spyder, system-wide):

```
C:\MinGW\bin
C:\Program Files\Spyder\Python
```

This way, you are fully flexible to use both the C compiler as well as the Python binaries from everywhere. If you still experience that Windows wants you to visit the Microsoft store in order to do a *second and independent* installation of Python (which means you also have two sets of completely independent module installations, individually managed by *pip*), you might want to carry out these steps[23] in order to get rid of the python stub pointers to the store:

```
cd C:\Users\<you name>\AppData\Local\Microsoft\WindowsApps
del python.exe
del python3.exe
```

In case you use Anaconda and do not want (or can) add Python to your path, then you can also start write some wrapper `BAT` script for your actual Python script:

Listing 1: run-script.bat

```
REM conda activate base
REM This activates (base) in  current scope
REM Probably have to change %HOMEPATH% to where conda is installed (user␣
↪or system wide)
```

(continues on next page)

[16] https://www.anaconda.com/
[17] https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html
[18] https://pypi.org/project/pip/
[19] https://datumorphism.leima.is/til/programming/python/python-anaconda-install-requirements/
[20] https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html#using-pip-in-an-environment
[21] https://www.spyder-ide.org/
[22] https://www.mingw-w64.org/
[23] https://stackoverflow.com/a/63203720

(continued from previous page)

```
CALL %HOMEPATH%\anaconda3\Scripts\activate.bat

python.exe your-actual-script.py

PAUSE
```

Here, the Windows Batch call[24] works the same way as source does in the unix bash[25], allowing you to invoke `python.exe` or `pip.exe` afterwards. Pause[26] prevents the Windows terminal from closing after invoking, so you can still see the output when launching the script by double-clicking the Bat file in Windows explorer.

Further software recommendations for Windows

If you do not have a comfortable IDE, you might want to look into Microsoft Powershell[27] as well as the new Windows Terminal[28] which can be obtained in the Microsoft store for free.

While it is attractive to download the PyAnalog software as a ZIP file from Github, we recommend you instead to install a proper Git client, such as the comprehensible Github Desktop GUI[29]. It can also be used independently from the Windows Git Command Line Interface[30], which itself can be installed with winget[31]. With git, it is as easy as a single click on *update* or an invocation of `git pull` to obtain a more recent version of the PyAnalog code.

Note: At Windows, we have experienced some hazzles with *UTF-16* encoded files. While we are working on improving the compatibility with the PyAnalog tools, you can convert any file to *UTF-8* by using such a Powershell command:

```
Get-Content nameOfYourFile.txt | Out-File -Encoding UTF8 nameOfYourFile-
↪fixed.txt
```

Also don't forget that Windows generally does not allow you to *open files for writing while they are opened for reading*. If you are used to the unix kind of dealing with files, this can make some steps more cumbersome and result in more copies of files.

---

[24] https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/call
[25] https://superuser.com/questions/46139/what-does-source-do
[26] https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/pause
[27] https://docs.microsoft.com/en-us/powershell/scripting/overview?view=powershell-7.2
[28] https://www.microsoft.com/de-de/p/windows-terminal/9n0dx20hk701#activetab=pivot:overviewtab
[29] https://desktop.github.com
[30] https://git-scm.com/download/win
[31] https://docs.microsoft.com/en-us/windows/package-manager/winget/

1.2.4 Recommended way of installation (developer machine setup)

We recommend the following way of obtaining and installing the PyAnalog code on your developer machine using *setuptools* by using the development mode[32] (which creastes a user-wide softlink to the working copy):

```
$ cd prefered/directory/for/code/of/the/analogians
$ git clone https://github.com/anabrid/pyanalog.git # this works always
$ git clone git@github.com:anabrid/pyanalog.git     # use this if you are␣
↪experienced at github
$ cd pyanalog
$ python3 setup.py develop --user
```

After these steps you are ready to use import the pyanalog package modules from any Python3 script anywhere on your system, but only *as your current user*. The installation is successful when you can for instance `import dda` somewhere:

```
$ python3
Python 3.8.6 (default, Sep 30 2020, 04:00:38)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import dda
>>> dda.Symbol("foo")
foo
```

1.2.5 User (non-developer) installation

If you want to install the package, just call `pip install` or `python3 setup.py install` in the repository root directory after cloning or without any cloning by just running:

```
$ pip install git+ssh://git@github.com:anabrid/pyanalog.git
```

This will automatically install all the dependencies from the `requirements.txt` file.

You can also add the flags `--user` for a per-user (instead of system wide) installation. If you want to perform an upgrade on an existing installation, add the `--upgrade` flag.

Note that by using this installation method, you cannot easily make edits to the pyanalog code itself. This is perfectly fine if you only want to use the library.

Note: Since the overall code is still subject to heavy changes, you should also opt in for the developer machine setup installation method if you do not indend to work on the PyAnalog code. This is for easier updating or changing versions with `git` without having to perform a fresh installation. Basically, with the development installation way, you can change/update/downgrade the PyAnalog code and immediately profit in your applications.

---

[32] https://setuptools.readthedocs.io/en/latest/setuptools.html#development-mode

1.2.6 Using without installation

We differentiate between *installing dependencies* for using PyAnalog and *installing PyAnalog itself*. While you should ensure having all the dependencies (which are not a lot, but see above), when it comes to installing PyAnalog, you can gratefully skip this step if you don't bother or don't like to do so.

That is, you can just checkout the repository, navigate to the root directory and open a python script from there. This way, you *don't have to install anything* but have the modules right in your `PYTHONPATH`. This is really the easiest way of getting started if you don't want to mess around with `pip` or `setup.py`.

However, it is usually good practice to use *virtualenv*. Using *virtualenv* was not covered in this document at all, but standard routines apply, given the presence of `requirements.txt` and `setup.py`.

1.3 Some general recommendations in the Python ecosystem

If you are new to Python, here are some tools you should look into:

- The Interactive Python shell[33] (IPython). It enriches the REPL[34] interaction with syntax highlighting, tab completion, comprehensive object introspection, input and output history, much more readable stack traces, etc. (see list of features[35]).
- Jupyter[36] and JupyterLab[37]. They bring a Matlab-like notebook-oriented interface which allows for writing high-quality documents with mixture of code, output and Latex/Rich text documentation. These notebooks can be rendered as websites and shared easily. Most noteworthly, they allow interactivity in cells, such as sliders and animations. There is a whole universe to explore once you look for Jupyter notebooks. And you can easily host your own notebooks in the cloud[38].
- The Python debugger[39] can come in handy in case of errors. With IPython, it's just the four letters `%pdb` away.
- If you look for plotting, Matplotlib[40] is the defacto standard. Being part of Scipy, it depends on Numpy[41], which provides N-dimensional arrays, linear algebra and input/output. When it comes to scientific computing, Numpy got some kind of hub and it's website lists dozens of related projects within all sciences.

[33] https://ipython.org/
[34] https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop
[35] https://ipython.readthedocs.io/en/stable/
[36] https://jupyter.org/
[37] https://jupyter.org/try
[38] https://mybinder.org/
[39] https://docs.python.org/3/library/pdb.html
[40] https://matplotlib.org/
[41] https://numpy.org/

1.4 Software testing in PyAnalog

The PyAnalog code is still quite alpha, but we do a number of software tests without much extra work:

Doctests[42] are super simple to add, they are just embedded in the documentation. It's a quite pythonic way to demonstrate (and test at the same time) the functionality of small pieces of code.

We have doctests all over the place. The code snippets are of course also included in the documentation. You can copy & paste them into your python console to play with the API.

Acceptance/Integration tests We can provide some DDA files where we know the solution analytically. It should be a useful exercise both for readers and testers to run these examples. We use the pytest[43] third party library for (unit) testing.

These tests are located in the `tests/` directory. They can be executed by running `pytest tests` or just `make test` from the parent directory.

Note: Tests are special because they can be evaluated for success or failure automatically. This differs them from other code contributions, such as the example codes in the `examples/` directory, which cannot be evaluated for correctnes.

1.4.1 How to run the tests

Just run `make test` in the root directory to run all of the tests. The output should look somewhat like this one:

```
$make test
make doctest unittests
make[1]: Verzeichnis ..../dda wird betreten
python3 -m pytest --doctest-modules --pyargs dda --ignore=dda/__main__.py␣
↪-v
============================================= test session starts␣
↪=============================================
platform linux -- Python 3.9.7, pytest-6.2.5, py-1.10.0, pluggy-0.13.1 -- /
↪usr/bin/python3
cachedir: .pytest_cache
rootdir: /home/sven/Analog/Software/dda
collected 22 items

dda/__init__.py::dda.clean PASSED                                          ␣
↪                                [  4%]
dda/ast.py::dda.ast.BreveState PASSED                                      ␣
↪                                [  9%]
dda/ast.py::dda.ast.State PASSED                                           ␣
↪                                [ 13%]
dda/ast.py::dda.ast.State.dependency_graph PASSED                          ␣
↪                                [ 18%]
dda/ast.py::dda.ast.State.equation_adder PASSED                            ␣
↪                                [ 22%]
dda/ast.py::dda.ast.State.name_computing_elements PASSED                   ␣
↪                                [ 27%]
```

(continues on next page)

[42] https://docs.python.org/3/library/doctest.html
[43] https://pytest.org

(continued from previous page)

```
dda/ast.py::dda.ast.State.variable_ordering PASSED                              ␣
↪                                [ 31%]
dda/ast.py::dda.ast.Symbol PASSED                                               ␣
↪                                [ 36%]
dda/ast.py::dda.ast.Symbol.draw_graph PASSED                                    ␣
↪                                [ 40%]
dda/ast.py::dda.ast.Symbol.map_heads PASSED                                     ␣
↪                                [ 45%]
dda/ast.py::dda.ast.Symbol.map_tails PASSED                                     ␣
↪                                [ 50%]
dda/ast.py::dda.ast.Symbol.map_terms PASSED                                     ␣
↪                                [ 54%]
dda/ast.py::dda.ast.Symbol.map_variables PASSED                                 ␣
↪                                [ 59%]
dda/ast.py::dda.ast.symbols PASSED                                              ␣
↪                                [ 63%]
dda/computing_elements.py::dda.computing_elements PASSED                        ␣
↪                                [ 68%]
dda/cpp_exporter.py::dda.cpp_exporter.run PASSED                                ␣
↪                                [ 72%]
dda/dsl.py::dda.dsl PASSED                                                      ␣
↪                                [ 77%]
dda/dsl.py::dda.dsl.read_traditional_dda SKIPPED (all tests skipped by␣
↪+SKIP option)                      [ 81%]
dda/scipy.py::dda.scipy.to_scipy PASSED                                         ␣
↪                                [ 86%]
dda/scipy.py::dda.scipy.to_scipy.rhs PASSED                                     ␣
↪                                [ 90%]
dda/sympy.py::dda.sympy.to_latex PASSED                                         ␣
↪                                [ 95%]
dda/sympy.py::dda.sympy.to_sympy PASSED                                         ␣
↪                                [100%]

======================================== 21 passed, 1 skipped in 1.82s␣
↪=========================================
python3 -m pytest --doctest-modules --pyargs hycon  -v
============================================= test session starts␣
↪=============================================
platform linux -- Python 3.9.7, pytest-6.2.5, py-1.10.0, pluggy-0.13.1 -- /
↪usr/bin/python3
cachedir: .pytest_cache
rootdir: /home/sven/Analog/Software/dda
collected 12 items

hycon/HyCon.py::hycon.HyCon PASSED                                              ␣
↪                                [  8%]
hycon/HyCon.py::hycon.HyCon.ensure PASSED                                       ␣
↪                                [ 16%]
hycon/HyCon.py::hycon.HyCon.expect PASSED                                       ␣
↪                                [ 25%]
hycon/autosetup.py::hycon.autosetup.DotDict PASSED                              ␣
↪                                [ 33%]
hycon/autosetup.py::hycon.autosetup.PotentiometerAddress PASSED                 ␣
↪                                [ 41%]
hycon/connections.py::hycon.connections PASSED                                  ␣
↪                                [ 50%]
```

(continues on next page)

(continued from previous page)

```
hycon/replay.py::hycon.replay.HyConRequestReader PASSED                  ␣
 ↪                               [ 58%]
hycon/replay.py::hycon.replay.consume PASSED                             ␣
 ↪                               [ 66%]
hycon/replay.py::hycon.replay.consume.list PASSED                        ␣
 ↪                               [ 75%]
hycon/replay.py::hycon.replay.consume.number PASSED                      ␣
 ↪                               [ 83%]
hycon/replay.py::hycon.replay.delayed PASSED                             ␣
 ↪                               [ 91%]
hycon/replay.py::hycon.replay.replay PASSED                              ␣
 ↪                               [100%]

============================================= 12 passed in 0.03s␣
 ↪=============================================
# all other modules don't have useful tests anyway
python3 -m pytest -v
============================================= test session starts␣
 ↪=============================================
platform linux -- Python 3.9.7, pytest-6.2.5, py-1.10.0, pluggy-0.13.1 -- /
 ↪usr/bin/python3
cachedir: .pytest_cache
rootdir: /home/sven/Analog/Software/dda
collected 15 items

tests/test_cpp_interface.py::test_similar_dtypes PASSED                  ␣
 ↪                               [  6%]
tests/test_differentiation.py::test_polynomial_diff PASSED               ␣
 ↪                               [ 13%]
tests/test_differentiation.py::test_sinusodial_diff PASSED               ␣
 ↪                               [ 20%]
tests/test_exponential_solution.py::test_run_simulation PASSED           ␣
 ↪                               [ 26%]
tests/test_latex_symbols.py::test_state PASSED                           ␣
 ↪                               [ 33%]
tests/test_latex_symbols.py::test_c_code PASSED                          ␣
 ↪                               [ 40%]
tests/test_latex_symbols.py::test_dda_code PASSED                        ␣
 ↪                               [ 46%]
tests/test_simulation_time.py::test_setup_state PASSED                   ␣
 ↪                               [ 53%]
tests/test_simulation_time.py::test_run_simulation PASSED                ␣
 ↪                               [ 60%]
tests/test_symbol_mappings.py::test_map_variable PASSED                  ␣
 ↪                               [ 66%]
tests/test_traditional_ddas.py::test_if_double_pendulum_is_scaled PASSED ␣
 ↪                               [ 73%]
tests/test_traditional_ddas.py::test_if_double_pendulum_is_working PASSED␣
 ↪                               [ 80%]
tests/test_traditional_ddas.py::test_if_chua_is_scaled PASSED            ␣
 ↪                               [ 86%]
tests/test_traditional_ddas.py::test_notch_is_scaled PASSED              ␣
 ↪                               [ 93%]
tests/test_traditional_ddas.py::test_nose PASSED                         ␣
 ↪                               [100%]
```

(continues on next page)

(continued from previous page)

```
============================================== 15 passed in 11.23s␣
↪==============================================
make[1]: Verzeichnis ..../dda wird verlassen
```

Test scripts can also be run and inspected with python interactively, i.e.

```
you@yourcomputer$ python -i test_exponential_solution.py
>>> from pylab import *
>>> ion()
>>> time, ysim = test_run_simulation()
generated.cc: In Elementfunktion »void csv_writer::write_header() const«:
generated.cc:171:43: Warnung: Operation auf »i« könnte undefiniert sein [-
↪Wsequence-point]
171 |             std::cout << query_variables[i++] << sep(i);
    |                                          ~^~
generated.cc: In Funktion »int main(int, char**)«:
generated.cc:275:90: Warnung: Operation auf »i« könnte undefiniert sein [-
↪Wsequence-point]
275 |                 for(size_t j=0;j<5 && i<all_variables.size();j++)␣
↪cerr << all_variables[i++] << (i!=all_variables.size() ? ", " : ""); }
    |                                                                   ␣
↪                    ~^~
Running: ./a.out --max_iterations=60 --modulo_write=1 --always_compute_aux_
↪before_printing=1 --write_initial_conditions=0
>>> print(time)
[0.05 0.1  0.15 0.2  0.25 0.3  0.35 0.4  0.45 0.5  0.55 0.6  0.65 0.7
0.75 0.8  0.85 0.9  0.95 1.   1.05 1.1  1.15 1.2  1.25 1.3  1.35 1.4
1.45 1.5  1.55 1.6  1.65 1.7  1.75 1.8  1.85 1.9  1.95 2.   2.05 2.1
2.15 2.2  2.25 2.3  2.35 2.4  2.45 2.5  2.55 2.6  2.65 2.7  2.75 2.8
2.85 2.9  2.95 3.  ]
>>> plot(time, ysim, "o")
[<matplotlib.lines.Line2D object at 0x7fe9a99c7520>]
>>> plot(time, -exp(-time))
[<matplotlib.lines.Line2D object at 0x7fe9982f43a0>]
>>> savefig("exponential_solution.png")
```

1.4.2 How to run doctests on the whole package

Use pytest[44] as a slim frontend for the python doctest builtin, for instance:

```
$ cd $(git rev-parse --show-toplevel)            # execute from PyAnalog␣
↪root directory
$ pytest-3 --doctest-modules --pyargs dda  -v
```

See also the `Makefile` provided in the root directory.

[44] https://docs.pytest.org/

1.4.3 Where are the tests running?

Tests are run by our *Gitlab Continous Integration* whenever the code is committed. You can view the file `.gitlab-ci.yml` in the root of the repository in order to see what is happening, which is at the moment something like

- Make the docs (run sphinx)
- Deploy the docs (upload them somewhere)
- Run all the tests (as above)

The finished/running pipelines can be seen at https://lab.analogparadigm.com/software/dda/-/pipelines We also run these tests at our Gitlab CI when pushes happen to the Github repository https://github.com/anabrid/pyanalog thanks to mirroring at https://lab.analogparadigm.com/software/pyanalog-mirror-from-github

1.4.4 What about analog hardware tests

This would require having dedicated testing hardware somewhere. This is out of scope for the moment.

2 PyDDA

2.1 Introduction to PyDDA

PyDDA is a small library to write and generate DDA code in Python. DDA stands for *digital differential analyzer*. In this context, it is a code for solving ordinary differential equations (ODEs) given in a domain specific language description (i.e. an electrical circuit).

The main advantage of this implementation in contrast to the older Perl implementation is the *abstract syntax tree* level of circuit representation. The syntax tree representation allows for fine-grained manipulations of terms where the older Perl code could only apply regular expressions.

This code can replace the old `dda2c.pl` Perl implementation (see `misc/` directory for it's code, or also here[45]). It is a standalone Python 3 code with no third party dependencies. It generates standalone C++ code with no dependencies (not even on the old `dda.h`).

2.1.1 Usage: As a library or from the command line

The *dda* (page 46) module can either be used from a DDA file written in Python or directly from old-style traditional DDA plaintext files. While pythonic dda files have the advantage to be able to use all the flexibility of Python scripting (such as using numpy for linear algebra computations and matplotlib for postprocessing of results), plaintext DDA files are in general shorter and more precise to read. See *dda.dsl* (page 36) for further details on the *traditional* DDA file format.

The module can also be used from the command line as a utility. The behaviour is similar to the `simulate.pl` and `dda2c.pl` utilities but also covers a few more features. Usage example (implementation provided by *dda.dsl.cli_exporter()* (page 37)):

```
me@localhost $ python -m dda --help
usage: dda.py [-h] [-o [OUTPUT]] [circuit_file] {c,dda,dot,latex}

PyDDA, the AST-based DDA compiler

positional arguments:
circuit_file          DDA setup (traditional file). Default is stdin.
{c,dda,dot,latex}     File formats which can be generated

```

(continues on next page)

[45] https://github.com/anabrid/pyanalog/tree/master/misc/DDA.pl

(continued from previous page)

```
optional arguments:
-h, --help            show this help message and exit
-o [OUTPUT], --output [OUTPUT]
                        Where to write exported code to. Default is stdout.

A Command Line Interface (CLI) for PyDDA...
```

Here is a full bash script which demonstrates how to use PyDDA as a drop-in replacement for the traditional Perl-based DDA code. It allows using the PyDDA C++ code generator without writing a single line of Python:

```
#!/usr/bin/bash

# given the DDA file "notch_simplified.dda", which you can find in the
# directory ./examples/traditional-dda-circuits, we simulate the system
# for 2000 timesteps and plot the time evolution of the fields "cn", "cd"␣
↪and "cnr"
# which are part of the DDA file (in terms of "cn = int(...)")

python -m dda notch_simplified.dda C --output notch_simplified.cc
g++ --std=c++17  notch_simplified.cc -onotch_simplified.exe
./notch_simplified.exe --max_iterations=2000 --skip_header=1  cn cd cnr  >␣
↪scratch.dat

cat <<GNUPLOT_FILE > gnuplot.dat

set terminal pdf
set output "notch_simplified_gnuplot.pdf"
set key autotitle columnheader
set title "Notch simplified (with PyDDA/Gnuplot)"

plot "scratch.dat" using 1 with lines title "cn", \
     "scratch.dat" using 2 with lines title "cd", \
     "scratch.dat" using 3 with lines title "cnr"

GNUPLOT_FILE

gnuplot gnuplot.dat
open notch_simplified_gnuplot.pdf
```

2.1.2 Usage without a C++ compiler

The PyDDA code grew out of its predecessor (`dda2c.pl`) as a *code generator* for an ODE solver. Having C++ has target language, it requires a C++ compiler to work. However, during the time, PyDDA got mature as a toolkit for exporting the code also in different formats. In fact, using the *dda.scipy* (page 41) module, one can avoid C++ and use PyDDA solely within the Python environment. This can be handy for anyone who cannot or does not want to deal with C++ or all the fundamentals.

Therefore, instead of following all the tedious way of C++ code generation, compilation and running (which also requires that you have all the build tools installed for compiling C++ on your system), you can also take the short track by making use of SciPy, which is also much easier to install on many systems. This also works from the command line:

```
me@localhost $ python -m dda.scipy --help
usage: scipy.py [-h] [-o [OUTPUT]] -t TFINAL [--method [METHOD]] [circuit_
↪file]

PyDDA's scipy interface simulation runner

positional arguments:
circuit_file          DDA setup (traditional file). Default is stdin.

optional arguments:
-h, --help            show this help message and exit
-o [OUTPUT], --output [OUTPUT]
                        Where to write output CSV to. Default is stdout.
-q [QUERY_FIELDS ...], --query-fields [QUERY_FIELDS ...]
                        List of fields to plot. Just pass whitespace␣
↪seperated (i.e. -q a b c). Also add 't' if you want
                        to have the solution time (recommended).

Arguments passed to scipy.integrate.solve_ivp:
-t TFINAL, --tfinal TFINAL
                        Time (in simulation units) to run up to. Do not␣
↪confuse this with some iteration counter.
-m [METHOD], --method [METHOD]
                        Integration method to use
-d, --dense-output    Dense Output (default is not dense)

A Command Line Interface (CLI) for :mod:`dda.scipy`. This CLI API␣
↪basically solves a DDA file ...
```

Here is a shell script example, again with the notch DDA file, of using SciPy instead of the C++ based solver:

```
#!/usr/bin/bash

# given the DDA file "notch_simplified.dda", which you can find in the
# directory ./examples/traditional-dda-circuits, we simulate the system
# for 2000 timesteps and plot the time evolution of the fields "cn", "cd"␣
↪and "cnr"
# which are part of the DDA file (in terms of "cn = int(...)")

python -m dda.scipy -d -t 20 -q cn_minus cd_minus cnr_minus --method RK45␣
↪notch_simplified.dda > scratch.dat

cat <<GNUPLOT_FILE > gnuplot.dat

set terminal pdf
set output "notch_simplified_gnuplot.pdf"
set key autotitle columnheader
set title "Notch simplified (with PyDDA/Gnuplot)"

plot "scratch.dat" using 1 with lines title "cn", \
     "scratch.dat" using 2 with lines title "cd", \
     "scratch.dat" using 3 with lines title "cnr"

GNUPLOT_FILE
```

(continues on next page)

(continued from previous page)

```
gnuplot gnuplot.dat
open notch_simplified_gnuplot.pdf
```

Note that the naming of the columns is different to the top example, since you can only access the *evolution quantities*, whcih are called `cn_minus` here, while `cn` is a deviated quantity. More details on these limitations can be found in the description of the *dda.scipy* (page 41) module.

2.1.3 Known Bugs and limitations

Please see the issue list at https://github.com/anabrid/pyanalog/issues for a list of bugs.

We also have an internal bug tracker at https://lab.analogparadigm.com/software/pyanalog/-/issues which is subject to be merged into the public one.

2.2 A rationale about DDA

The following text is an introduction into the *DDA* concepts. It is envisaged for readers who want to trace the connections between analog computing and it's digital simulation. If you only want to work with *PyDDA* but do not care about the fundamentals, you can skip this text.

2.2.1 The digital number flow machine

DDA is short for *digital differential analyzer*. This term describes a certain way of building an algorithmic-logical unit which is programmed with a dataflow paradigm. It can be imagined as an analog computer but with digital computing elements in place of analog computing elements. Such a DDA machine could feature n bit integer adders (for instance a ripple-carry adder), binary multipliers, and even discrete integrators for adding integers (i.e. a stateful computing element). In general, it is straightforward to design such a machine for some fixed width binary number representation. It is worth emphasizing that such a machine neither features continous time nor continous number representation as a real analog computer would have. Nevertheless, it is an interesting computing architecture as it is half-way between an analog and a digital computer.

In the age of FPGAs (field programmable gate arrays), it is straightforward to generate digital computing circuits by software descriptions. Furthermore, being a digital computer, the DDA architecture can be simulated by any Turing machine. This makes it straightforward to write a simulator for contemporary register machines (that is, regular and widespread computers/processors).

2.2.2 Describing and simulating a DDA machine

The DDA code proposed in this document consists of several parts:

- An easy description language for the computational network (circuit)
- A compiler from that language to an iterative imperative code (Perl/C)
- Tools for running such a code and evaluating the results

Loosely speaking, this translation works as follows: A circuit file is an ASCII line based text file which looks like

```
dt = const(0.0005)
t = int(1, dt)
y0  = const(-1)
minus_dy0 = const(-1)

minus_dy = int(y, dt, minus_dy0)
y        = int(minus_dy, dt, y0)
```

That is, each line is an assignment of some variable to some expression, which is either a constant (`const`) or a compound expression of computing elements. These expressions are written in some standard C-like notation `f(x,y,...)` where `f` is the identifier for the function and `x,y,...` are comma seperated arguments. The following *basic arithmetic* (from the perspective of an analog computer) computing elements are defined:

- $neg(x) = -x$, the inverse
- $div(x, b) = a/b$, the standard division
- $mult(a_0, a_1, \dots) = \prod_i a_i$, the standard multiplication
- $sum(a_0, a_1, \dots) = -\sum_i a_i$, the summation in analog-computer typical *negating* convention.
- $int(a_0, a_1, \dots, \Delta t, I_0) \approx -\int \sum_i a_i \, \Delta t + I_0$, the time integration (again in analog-computer typical negating convention). The digital integrator is discussed in detail in the following text.

Furthermore, a couple of case-discreminating computing elements are defined. Here, they are given in C-like notation `x ? y : z` which evaluates to `if(x) then y else z`.

- `lt(a,b,c,d) = (a < b) ? c : d`
- `le(a,b,c,d) = (a <= b) ? c : d`
- `gt(a,b,c,d) = lt(b,a,c,d)`
- `ge(a,b,c,d) = le(b,a,c,d)`
- `dead_lower(a,b) = (a<b) ? (a-b) : 0 = gt(a,b,a-b,0)`
- `dead_upper(a,b) = (a>b) ? (a-b) : 0 = lt(b,a,a-b,0)`
- `min(a,b) = (a<b) ? a : b = lt(a,b,a,b)`
- `max(a,b) = (a>b) ? a : b = gt(a,b,a,b)`
- `abs(a) = (a<0) ? -a : a = lt(a,0,-a,a)`
- `floor(a) = (int)a` rounds `a` to the next lower integer.

Note that this is only a subset of the overall computing elements defined. It is easy to introduce new computing elements, they are defined in the *dda.computing_elements* (page 37).

2.2.3 Linearization of a circuit

The `dda2c.pl` Perl script translates a DDA circuit file to a valid *C* program. To do so, all quantities are treated as *real valued* and are associated with the *double* floating point data type. As C is a strongly typed language, all defined quantities are collected and introduced as stack variables before use.

The actual imperative program then just takes the DDA circuit line-by-line. This introduces a bias, as the computing network by itself is executed *synchronously* while a simulation with a single arithmetic logical unit (ALU) on an ordinary processor can only execute one operation at a time.

Note: It would be interesting to think a bit whether we could not write an DDA-level exact simulator, since the DDA machine is clocked. We should be able to correctly simulate this clock.

Since the DDA is subject to a discrete computing cycle, a register machine can simulate the DDA architecture cycle by cycle, computing the value of each computing element input and output. For the sake of extraordinary introspection and debugging facilities, the DDA to C compiler dismantles compound expressions `f(g(x))` or `f(a,b(c),d(e))` and entitles all intermediate expressions such as `gx=g(x)` in `f(gx)` or `g=b(c)` and `h=d(e)` in `f(a,g,h)`. This is especially handy when the DDA is seen as an approximation of the analog computer, as it allows for checking the boundness (correct scaling) of all variables during the cycles (time evolution).

Having said that, the DDA simulator allows for simulating a DDA circuit iteration by iteration and dumping (outputting) values every `n` iterations. Therefore, while the input of a circuit is always fixed by the constants (`const` statements, no focus has given to the point of interfacing other codes, which is left as an exercise for the reader), the output is always a time series for a given set of quantities. We refer to theses quantities as *observables*, which are *queried* for at code generation time. One can thus understand the output as a fully discrete table of numbers, where the columns hold the time series for a given variable and the each row stands for one time iteration (or some average or surrogate for a larger number of iterations, if requested). These numbers are represented as ASCII column seperated values (CSV) in the output of the compiled C program.

2.2.4 Applicability for solving differential equations

The usability for this software-based DDA implemenetation for solving ordinary differential equations highly depends on the internals of the integrator component. From all computing elements described above, the integrator is the only one with an *internal state*. That is, it has to remember from iteration to iteration the current integration value.

The most easy integrator component will internally look like the following imperative dummy code:

```
double integrate(double integrand, double dx, double initial_value) {
    static double internal_state = initial_value;
    internal_state += integrand * dx;
    return internal_state;
}
```

Here, the `internal_state` is declared as a *static* variable, which you can think of a global variable (with a lifetime longer then the function evaluation) if you don't know C. In fact, this dummy code comes quite close to the actual implementation of the integrator in the DDA C code. We refer to the above numerical scheme as the *Euler time integration*, since it approximates the time-continous integral by it's Riemann sum.

Within the DDA code, higher order explicit integration schemes can be chosen, such as Runge-Kutta. However, given the nature of the problem description in a circuit, implicit methods can not be applied by the compiler without an actual analysis of the differential equation. Howver, on can imagine a DDA circuit which itself describes a numerical scheme on a digital-circuit level.

2.2.5 On PyDDA, the successor of the DDA Perl code

The first DDA code was written by Bernd. It's job was to simulate circuits, and this was performed by a small Perl script which threw a few regexes onto the DDA file to convert it to an executable C numeric simulation.

As described above, we found out that even with slightly more challenging circuits (kind of *border cases*, such as the depicted one above) the simple ideology of looping over numeric equations breaks down.

2.2.6 Lexical sorting of variable dependencies

Instead, was has to be applied for a stable integration of an electric circuit, i.e. an ordinary differential equaiton, is the correct sorting of equation ordering. To do so, we must study the dependencies of equations. This requires a memory representation of equations, and there we enter the domain of *computer algebra systems* (CAS). Their central piece of information are algebraic equations, which are typically represented as (abstract) *syntax trees*.

PyDDA was an effort to rewrite the Perl-based DDA with a minimal amount of work. Exploiting that DDA looks almost like Python, the idea was to bring a number of archivements with a single code:

- Allow to write high-level DDA codes, which probably involve indexing, n-dimensional arrays, etc.
- Allow for easy interoperation with various codes and tools, such as other CAS, (evventually generated) numerical simulation codes or reprogrammable analog computers.
- Enable the user for a Read-eval-print loop interface (REPL) in order to encourage explorative programming.
- Meshing literate programming, generation of documentation and reports out of the equations without much work
- Picking the community where it is: Scientific Python is a thing, and so we choose python. Thus we also can stick to python when it comes to simulation analysis and postprocessing.
- Avoid dependencies if not neccessary. Don't reinvent the wheel, but try out how far we can get without employing a large computer algebra system.

2.2.7 I want to simulate electronics or solve an ODE. Do I really need DDA?

DDA is great if you want to learn about and work with analog computers. Expressing your equations in terms of this domain specific hardware description language (which includes all the "quirks" such as *negating* operational amplifiers used for summation/integration) can be helpful when it comes to implementing it on real analog computers. PyDDA is also a helpful tool if you want to learn about numerical vs. analog computing, or experiment with more challenging systems such as partial differential equations.

If your goal is to simulate electronics on an advanced level, you might want to look into the SPICE class[46] of tools, such as ngspice[47].

If your goal is to solve a differential equation or to study a small system of interest, you better do so by using tools particularly made for doing this job very well, such as Matlab, GNU Octave or some computer algebra system. These are typically mature systems with decades of development, while PyAnalog is a small research code developed by a single person over a single year.

Especially small systems with only a few unknowns, it is rather straightforward to transform them into analog circuits once required, and you can safely postpone this task up to when it gets relevant. The only thing to keep in mind is that when developing your applications within another (assumably numerical) toolkit, be careful about implementing your own algorithmic features. Ideally, you straightforwardly implement a closed set of mathematical equations in order to maintain the option to implement this system on an analog computer, without touching the solution vector with algorithms during the time evolution.

In fact, you even can do a scaling study without PyAnalog. Most time evolution codes solve some ODE equation $\dot{y} = f(y)$ by allowing the user to implement the function `f(y)` freely. You could easily implement any checks on scaling within `f(y)` in your favourite programming language/tool .

2.3 PyDDA API reference

2.3.1 Abstract Syntax tree

The minimalistic pythonic standalone abstract syntax tree (**AST**) representation in this module is the heart of the PyDDA package. The code has no external dependencies, especially it does not rely on a Computer Algebra System or even on SymPy.

The *Symbol* (page 22) object represents a node in a AST and the edges to it's children. In order to simplify mass symbol generation, *symbols()* (page 28) can be used.

The *State* (page 28) object represents a list (set) of equations. It basically maps variables to their expressions. The *State* (page 28) represents a (traditional) DDA file. From a python perspective, a *State* (page 28) is not much more then a dictionary on stereoids.

class dda.ast.**Symbol**(*head*, **tail*)

A symbol is similar to a LISP atom which has a Head and a tail, where tail is a list. Common notations for such a type are

- `head[tail]` in Mathematica,
- `(head, tail)` in Lisp

[46] https://en.wikipedia.org/wiki/SPICE
[47] http://ngspice.sourceforge.net/

- `head(tail)` in C-like languages like Python, Perl, Fortran, C
- Actually `[head, *tail]` in Python, but we don't use that.

A symbol also represents a vertex (node) and it's childs in an ordered tree. Think of head being the vertex and tail the (edge) list of children. We use the Symbol class to represent the abstract syntax tree (AST) of the DDA language for describing ODEs and circuitery.

When you call `str()` or similar on instances of this class, it will print its representation in the C-like notation. This notation is identical to the "classical" DDA language.

There are two types of Symbols: **Variables** have no tail, they just consist of a head:

```
>>> x = Symbol("x")
>>> print(x)
x
>>> x.head
'x'
>>> x.tail
()
```

In contrast, **Terms** have a tail:

```
>>> f = Symbol("f", Symbol("x"), Symbol("y"))
>>> print(f)
f(x, y)
>>> f.head
'f'
>>> f.tail
(x, y)
```

Variables can be used to create complex expressions for which they then serve for as a head:

```
>>> f,x,y,z = Symbol("f"), Symbol("x"), Symbol("y"), Symbol("z")
>>> f(x,y)
f(x, y)
>>> # example for kind of nonsensical terms:
>>> x(x,f,x)
x(x, f, x)
```

Calling a symbol will always replace it's tail:

```
>>> f(x)(y)
f(y)
```

Symbols are equal to each other if their head and tail equals:

```
>>> a1, a2 = Symbol("a"), Symbol("a")
>>> a1 == a2
True
>>> f(x) == f(x)
True
>>> f(x) == f(x,x)
False
```

Symbols can be used as dictionary keys, since they hash trivially due to their unique canonical (pythonic) string interpretation.

Note: In order to avoid confusion between Python Strings and Symbols, you should

- *always* use strings as Symbol heads but
- *never* use strings in Symbol tails. Instead, use there Symbols only.

Think of Symbol implementing the following type (hint): `Tuple[str, List[Symbol]]`.

The DDA code helps you to follow this guide. For instance, the representation of `f1` shows that it is a symbol with two string arguments, while `f2` has symbol arguments:

```
>>> f1 = Symbol("f", "x", "y")
>>> f2 = Symbol("f", Symbol("x"), Symbol("y"))
>>> f1
f('x', 'y')
>>> f2
f(x, y)
```

And DDA prevents you from shooting in your foot:

```
>>> f, x, y = symbols("f,x,y")
>>> f3 = Symbol(f,x,y)
Traceback (most recent call last):
...
TypeError: Trying to initialize Symbol f(x, y) but head f is a Symbol,
↪ not a String.
```

In previous versions of DDA, the thin line between strings and symbols hasn't been made so clear and tracing errors was harder.

Summing up, it is a good convention to *only* have Symbols and floats/integers being part of the Symbol tails.

---

is_variable()
A variable is a symbol without a tail.

is_term()
A term is a symbol with a tail.

variables()
Compute the direct dependencies of this symbol, i.e. other variables directly occuring in the tail.

all_variables()
Like :meth:variables, but also find variables in all children.

all_terms()
Like :meth:all_variables, but for terms: Returns a list of all terms in all children of this node.

map_heads(*mapping*)
Call a mapping function on all heads in all (nested) subexpressions. The mapping is effectively carried out on the head (ie. maps strings) Returns a new mapped Symbol. This routine is suitable for renaming variables and terms within an AST. Example usage:

```
>>> Symbol("x", Symbol("y"), 2).map_heads(lambda head: head+"foo")
xfoo(yfoo, 2)
```

The mapping is unaware of the AST context, so you have to distinguish between variables and terms yourself if you need to. See also *map_variables()* (page 25) for context-aware head mapping. Compare these examples to the ones given for *map_variables()* (page 25):

```
>>> x, map, y = Symbol("x"), lambda _: "y", Symbol("y")
>>> x.map_heads(map) == x.map_variables(map)  # == y
True
>>> x(x,x).map_heads(map) == y(y,y)
True
>>> x(x, x(x)).map_heads(map) == y(y, y(y))
True
```

map_variables(*mapping*, *returns_symbol=False*)

Calls a mapping function on all variables within the (nested) subexpressions. The mapping is effectively carried out on the head (ie. maps strings). This is a mixture between *map_heads()* (page 24) and *map_tails()* (page 25).

Returns a new mapped Symbol. This routine is suitable for renaming variables but not terms within the AST. Examples:

```
>>> x, map, y = Symbol("x"), lambda _: "y", Symbol("y")
>>> x.map_variables(map) == y
True
>>> x(x,x).map_variables(map) == x(y,y)
True
>>> x(x, x(x)).map_variables(map) == x(y, x(y))
True
```

This function ignores non-symbols as they cannot be variables. This is the same as *map_tails()* (page 25) does and is handy when you have numbers within your expressions:

```
>>> x = Symbol("x")
>>> expr = x(123, x(9.1), x, x(x, 0.1, x))
>>> res1 = expr.map_variables(lambda xx: "y")
>>> res2 = expr.map_variables(lambda xx: Symbol("y"), returns_
↪symbol=True)
>>> res1 == res2
True
>>> res1
x(123, x(9.1), y, x(y, 0.1, y))
```

If you want to use `map_variables` to change a variable to a term, and/or if your mapping function does not return strings but Symbols, use `returns_symbol=True`:

```
>>> Symbol("x").map_variables(lambda x: Symbol("y", 123), returns_
↪symbol=True)
y(123)
>>> Symbol("x").map_variables(lambda x: Symbol("y", 123)) # this␣
↪won't work
Traceback (most recent call last):
...
TypeError: Trying to initialize Symbol y(123) but head y(123) is␣
↪a Symbol, not a String.
```

map_tails(*mapping*, *map_root=False*)

Calls a mapping function on all tails in all (nested) subexpressions. The mapping is carried out on the tail symbols (ie. maps Symbols). Returns a new mapped Symbol. The routine is suitable for AST walking, adding/removing stuff in the tails while preserving the root symbol. This could also be called `map_symbols`, c.f. *map_terms()* (page 26).

Example for recursively wrapping all function calls:

```
>>> x,y,z = symbols("x,y,z")
>>> x(y, z(x), x(y)).map_tails(lambda smb: Symbol("foo")(smb))
x(foo(y), foo(z(foo(x))), foo(x(foo(y))))
>>> x(y, z(x), x(y)).map_tails(lambda smb: Symbol("foo")(smb),␣
↪map_root=True)
foo(x(foo(y), foo(z(foo(x))), foo(x(foo(y)))))
```

Example for recursively removing certain unary functions `z(x)` for any `x`:

```
>>> remover = lambda head: lambda x: x.tail[0] if isinstance(x,
↪Symbol) and x.head==head else x
>>> x,y,z = symbols("x,y,z")
>>> x(y, z(x), x(z(y),x)).map_tails(remover("z"))
x(y, x, x(y, x))
```

The argument `map_root` decides whether the map is run on the root node or not. It will be `map_root=False` in any recursive use. In former instances of this code, it was always `map_root=False`. Example:

```
>>> (a, b), flip = symbols("a,b"), lambda smb: b if smb.head==a.
↪head else a
>>> a(b,a,b).map_tails(flip, map_root=True)
b
>>> a(b,a,b).map_tails(flip, map_root=False)
a(a, b, a)
```

Note how the `flip` function cuts every tail and returns variables only. Here is a variant which perserves any tail:

```
>>> (a, b) = symbols("a,b")
>>> flipper = lambda smb: (b if smb.head==a.head else a)(*smb.
↪tail)
>>> a(b,a,b).map_tails(flipper, map_root=True)
b(a, b, a)
```

Here is another example which highlights how `map_tails` can convert terms to variables:

```
>>> x, map, y = Symbol("x"), lambda _: Symbol("y"), Symbol("y")
>>> x(x,x).map_tails(map, map_root=False)
x(y, y)
>>> x(x,x(x,x)).map_tails(map, map_root=False)
x(y, y)
```

For real-life examples, study for instance the source code of `cpp_exporter` or grep any DDA code for `map_tails`.

See also *map_heads()* (page 24) and *map_variables()* (page 25) for variants.

map_terms (*mapping*, *returns_symbol=False*)

Calls a mapping function on all terms within the (nested) subexpressions. The mapping

is effectively carried out on the term head (ie. maps strings). See *map_variables()* (page 25) for the similar-minded antoganist as well as *map_heads()* (page 24) and *map_tails()* (page 25) for more *low level* minded variants.

Returns a new mapped Symbol. This routine is suitable for renaming terms but not variables within the AST. Examples:

```
>>> x, map, y = Symbol("x"), lambda _: "y", Symbol("y")
>>> x.map_terms(map) == x
True
>>> x(x,x).map_terms(map) == y(x,x)
True
>>> x(x, x(x)).map_terms(map) == y(x, y(x))
True
```

This function ignores non-symbols as they cannot be variables, similar to *map_variables()* (page 25).

It is basically `map_terms(map) = map_tails(lambda smb: Symbol(map(smb.head))(smbl.tail) if symb.is_variable() else smb)`.

The argument `returns_symbol` allows to discriminate between mapping functions which return strings (for symbol heads) or symbols. The later allows for manipulating expressions.

draw_graph(*graph=None*)

Uses graphviz to draw the AST down from this symbol.

See also **:method:`State.draw_dependency_graph`** for similar draph drawing code and notes on python library dependencies.

Note: This method constructs the graph by drawing edges between similar named symbols. This will *not* represent the abstract syntax tree if a single symbol head, regardless of whether variable or term, appears twice.

If you want to draw the actual AST with this function, you have to make each symbol (head) unique by giving them distinct names.

Simple usage example:

```
>>> x,y,z = symbols("x,y,z")
>>> expression = x(1,y,2,z(3,4))
>>> graph = expression.draw_graph()
>>> print(graph)
digraph "DDA-Symbol" {
   node [shape=doublecircle]
   x
   node [shape=circle]
   x -> 1
   node [shape=doublecircle]
   y
   node [shape=circle]
   x -> y
   x -> 2
```

(continues on next page)

(continued from previous page)

```
    node [shape=doublecircle]
    z
    node [shape=circle]
    z -> 3
    z -> 4
    x -> z
}
>>> graph.view() # Call this to draw the graph
```

dda.ast.**is_symbol**(*smbl*)
Convenience function

dda.ast.**symbols**(**query*)
Quickly make symbol objects. Usage similar to sympy's symbol function:

```
>>> a, b = symbols("a", "b")
>>> x, y, z = symbols("x, y, z")
```

dda.ast.**topological_sort**(*dependency_pairs*)
Sort a graph (given as edge list) subject to dependency constraints. The result are two lists: One for the sorted nodes, one for the unsortable (cyclically dependent) nodes.

Implementation shamelessly stolen from https://stackoverflow.com/a/42359401

class dda.ast.**State**(*initialdata={}*, *type_peacemaking=True*, *default_symbol=True*)
A state is a dictionary which is by convention a mapping from variable names (as *strings*) to their symbolic meaning, i.e. a `Symbol()`. We refer to the keys in the dictionary as the *Left Hand Side* (LHS) and the values in the dictionary as the *Right Hand Side* (RHS), in analogy to an Equation.

Note: Since `Symbol()` spawns an AST, a state is a list of variable definitions. A DDA file is a collection of equations. Therefore, a state holds the content of a DDA file.

This class collects a number of basic helper routines for dealing with states.

In order to simplify writing DDA files in Python, this class extends the dictionary idiom with the following optional features, which are turned on by default (but can be disabled by constructor arguments `type_peacemaking` and `default_symbol`).

- **Type peacemaking**: Query a `Symbol()`, get translated to `str()`:

```
>>> State({ "foo": Symbol("bar")})[Symbol("foo")] == Symbol("bar")
True
```

- **Default Symbol**: Automatically add an entry when unknown:

```
>>> State()["foo"] == Symbol("foo")
True
```

Note: By intention, the keys of the State are always strings, never Symbols. This also should make sure you don't use complex ASTs for keys, such as `Symbol("foo", "bar")`.

As `State` extends `collections.UserDict`, you can access the underlying dictionary:

```
>>> x,y = symbols("x,y")
>>> add, integrate = symbols("add", "integrate")
>>> eqs = { x: add(y,y), y: integrate(x) }
>>> state = State(eqs); print(state)
State({'x': add(y, y), 'y': integrate(x)})
>>> state.data
{'x': add(y, y), 'y': integrate(x)}
```

Warning: Don't be fooled by refering to the state while constructing the state. This will end up in overly complex expressions. By rule of thumb, only use `Symbols` at the state definition (or in particular on the right hand side). For instance, you do want to construct a state like

```
>>> state = State()
>>> state["x"] = Symbol("add", Symbol("y"), Symbol("y"))
>>> state["y"] = Symbol("int", Symbol("x"))
>>> state
State({'x': add(y, y), 'y': int(x)})
```

In contrast, this is most likely not what you want:

```
>>> state = State()
>>> state["x"] = Symbol("add", Symbol("y"), Symbol("y"))
>>> state["y"] = Symbol("int", state["x"])
>>> state
State({'x': add(y, y), 'y': int(add(y, y))})
```

This time, you did not exploit the definition of `state["x"]` by referencing on DDA level but instead inserted the expression by referencing on Python level. This is like *compile-time evaluation* versus *runtime evaluation*, when *compile-time* is at python and *runtime* is when evaluating the DDA expressions in some time evolution code.

Summing up, the mistake above is to reference to `state` while constructing7 the `state`. You should not do that. You go best by defining the `Symbol` instances before and then only using them all over the place:

```
>>> x, y, add, int = symbols("x, y, add, int")
>>> state = State()
>>> state[x] = add(y,y)
>>> state[y] = int(x)
>>> state
State({'x': add(y, y), 'y': int(x)})
```

Note: Why the name? The class name `State` seems arbitrary and quirky, `System` may be a better choice (given that the class instances hold an equation system). However, one could also argue that the class instances hold the definition for a system in a particular state. especially, `State.keys()` are the state variables which undergo a definition by their corresponding `State.values()`. Most CAS do not have a special class for collections of equations. Instead, they typically have some *equation* type and equation systems are sets or lists of equations. In PyDDA, we don't have an equation type because the DDA domain specific language (see `dsl`) doesn't provide advanced treatments of equations but is basically only a lengthy definition of a set of equations, which you could understand as a mapping/dictionary data type defining the state of the system. That's why *State* is actually an enriched *dict*.

classmethod from_string(**string_or_list_of_strings*)
Shorthand for `dsl.read_traditional_dda()`. Returns new instance.

to_string()
Shorthand for `dsl.to_traditional_dda()`. Returns a string (representation).

update(*[E]*, ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

equation_adder()
Syntactic sugar for adding new equations to the system. Usage:

```
>>> state = State()
>>> x,y,z,add,int = symbols("x,y,z,add,int")
>>> eq = state.equation_adder()
>>> eq(y=int(x))
>>> eq(x=add(y,z), z=int(x,0,0.1))
>>> state
State({'x': add(y, z), 'y': int(x), 'z': int(x, 0, 0.1)})
```

Known limitations: This doesn't work any better then the `BreveState` below because keywords must not be variables, they will always resolve to their string representation.

```
>>> foo = Symbol("bar")
>>> s1, s2 = State(), State()
>>> eq1 = s1.equation_adder()
>>> eq1(foo=42)
>>> s2[foo] = 42
>>> s1
State({'foo': 42})
>>> s2
State({'bar': 42})
```

map_tails(*mapper*, *map_root=True*)
Apply *Symbol.map_tails()* (page 25) on all right hand sides.

map_heads(*mapper*)
This function is suitable for renaming variables. mapper is always executed on the string variable names (Symbol heads)

symbols(**query*)
Same as *symbols()* (page 28) above, but register at self (state)

constant_validity()
Check validity of numeric constants in the state. Depending on context, values -1 < t < +1 are illegal.

(Not yet implemented!)

dependency_graph()
Returns the edge list of the variable dependency graph of this state. We can call *topological_sort()* (page 28) on the result of this method.

A weird example including some corner cases:

```
>>> s1 = State.from_string("foo = const(0.7)", "bar=mult(bar,baz)
↪", "baz=f(bar)")
>>> s1.dependency_graph()
[('bar', 'bar'), ('bar', 'baz'), ('baz', 'bar')]
```

Another even more weird example which exploits raw value assignment, something which is not following the `foo=call(bar)` requirement for DDA files:

```
>>> a,b,c,d,f = symbols("a,b,c,d,f")
>>> s2 = State({ a: f(0.7), b: c(b,b), c: 42, d: c })
>>> s2.dependency_graph()
[('b', 'b'), ('d', 'c')]
```

Note that this function always returns list of tuples of strings. No more symbols. See also *draw_dependency_graph()* (page 31) for a quick way of exporting or plotting this graph.

draw_dependency_graph (*export_dot=True*, *dot_filename='test.dot'*)

If you have `networkx` and `pyGraphViz` installed, you can use this method to draw the *variable dependency graph* (see method *dependency_graph()* (page 30)) with `Dot/GraphViz`. This method will return the `nx.DiGraph()` instance. If `export_dot` is set, it will also write a dotfile, call `dot` to render it to a bitmap and open that bitmap.

Note: Your distribution package `python-graphviz` is probably not `pygraphviz`. You are on the safe side if you run: `pip install pygraphviz`

---

name_computing_elements (*strict=False*)

Name all computing elements / intermediate expressions. Returns a new State which is *linearized* in a way that the numbering proposes a computing order.

Linearization is an idempotent operation, i.e. for any `lin = state.name_computing_elements()` it is `lin == lin.name_computing_elements()`. Mathematically, it is a projection of the state on its linearized one.

Linearization means to define a evaluation order and to give unique names to all terms occuring. Note that *all* depends on the strictness (*strict=True* vs. the default *strict=False*):

```
>>> x, y, sum, mult = symbols("x, y, sum, mult")
>>> ns = State({ x: sum(x,y, sum(y, mult(y,x))), y: mult(x) })
>>> print(ns.name_computing_elements().to_string())
mult_1 = mult(y, x)
sum_1 = sum(y, mult_1)
x = sum(x, y, sum_1)
y = mult(x)
>>> print(ns.name_computing_elements(strict=True).to_string())
mult_1 = mult(y, x)
mult_2 = mult(x)
sum_1 = sum(y, mult_1)
sum_2 = sum(x, y, sum_1)
x = sum_2
y = mult_2
```

Here, *strict* means that really *every* term is labeled, even if this yields in "dumb" assignments

such as `x = sum_2`. You want a strict naming when enumerating computing elements, while a non-strict naming is preferable for brief evaluation. Also note that

```
>>> x,const = symbols("x,const")
>>> State({ x: const(42) }).name_computing_elements(strict=False)
State({'x': const(42)})
>>> State({ x: const(42) }).name_computing_elements(strict=True)
State({'const_1': const(42), 'x': const_1})
>>> State({ x: const(42) }).name_computing_elements(strict=True).
↪name_computing_elements(strict=True)
/.../ast.py:819: UserWarning: State.named_computing_elements():␣
↪While counting const, I notice that  const_1 is already part of␣
↪the state. Maybe you want to run name_computing_
↪elements(strict=False) for idempotence.
  warnings.warn(...)
State({'const_1': const_1_, 'const_1_': const(42), 'x': const_1})
```

On this mini example, one especially sees that idempotence is only given when `strict=False`. It is `state.name_computing_elements(True) == state.name_computing_elements(s[0]).name_computing_elements(s[1])...name_computing_elements(s[n])` when `s` is a boolean array of `len(s)==n` and `sum(s) == 1`, i.e. only one occurence of `strict=True` and all other `False`.

The linearized state only has entries of a *normal form* `state[f_i] = f(v1,v2,...)` for a function (term) `f` and some variables `v_j`. Furthermore note how even `x = sum_2` in the above example indirects the former assignment of `x = sum(x,y...)`. Again, for any value in the linearized state, the tail only contains variables, no terms. This is handy for many things, such as circuit drawing, imperative evaluation (in combination with *variable_ordering()* (page 33), cf. `cpp_exporter`) and determination of integrands/actual variables. For instance

```
>>> s = State.from_string("foo = const(0.7)", "baz=mult(bar,bar)",
↪ "bar = neg(int(neg(baz), foo, 0.3))")
>>> s
State({'bar': neg(int(neg(baz), foo, 0.3)), 'baz': mult(bar, bar),
↪ 'foo': const(0.7)})
>>> print(s.name_computing_elements(strict=True).to_string())
bar = neg_2
baz = mult_1
const_1 = const(0.7)
foo = const_1
int_1 = int(neg_1, foo, 0.3)
mult_1 = mult(bar, bar)
neg_1 = neg(baz)
neg_2 = neg(int_1)
>>> print(s.name_computing_elements(strict=False).to_string())
bar = neg(int_1)
baz = mult(bar, bar)
foo = const(0.7)
int_1 = int(neg_1, foo, 0.3)
neg_1 = neg(baz)
```

Here one sees immediately that `int_1` is the actual integral solution while `bar` is only a derived quantity. Calls like `const(float)` remain unchanged since they are already in the normal form `f(v1,v2,...)`.

Here is a more complex example:

```
>>> from dda.computing_elements import neg,int,mult
>>> dda_state = State({"x": neg(int(neg(int(neg(mult(1, Symbol("x
↪")), 0.005, 1)), 0.005, 0))) })
>>> dda_state.name_computing_elements().variable_ordering().where_
↪is
{'x': 'vars.aux.sorted',
'mult_1': 'vars.aux.sorted',
'neg_2': 'vars.aux.sorted',
'neg_1': 'vars.aux.sorted',
'int_1': 'vars.evolved',
'int_2': 'vars.evolved'}
```

variable_ordering()

Will perform an analysis of all variables occuring in this state (especially in the RHS). This is based on the linarized variant of this state (see *name_computing_elements()* (page 31)).

The return value is an object (actually a types.SimpleNamespace instance) which contains lists of variable names (as strings). The properties (categories) are primarly

- explicit constants: Any entry `state["foo"] = const(1.234)`
- State variables/evolved variables: Any outcome of a time integration, i.e. `int(...)`, i.e. `Symbol("int")`. This can be as simple as `state["foo"] = int(Symbol("foo"),...)`. Complex terms such as `state["foo"] = mult(int(foo), int(bar))` will result in intermediate variables called like `int_0`, ``int_1``(see *name_computing_elements()* (page 31) for the code which invents these names), which are the actual evolved variables.
- Auxilliary variables: Any other variables which are required to compute evolved variables.

By intention, we **sort only the aux. variables**. One should check that they DO NOT have any cyclic dependency, because feedback loops are only useful on integrators at this level of circuit modeling.

We differntiate the auxilliaries further into:

- `sorted_aux_vars`: Auxilliaries required to compute the state variable changes
- `cyclic_aux_vars`: Auxilliaries which have a cyclic dependency on each other (this should not happen as it won't lead to a stable circuit)
- `unneeded_auxers`: Auxilliaries which are not required to compute the state. These are probably used in postprocessing. If they depend on the state variables, further work is neccessary.

Given an ODE problem `dq/dt = f(q)`, an imperative code for evolving the state `q` in time should compute all auxillairy variables in the respective order before computing the actual `dq/dt`. The dependency is basically, in pseudo code:

```
>>> aux  = function_of(aux, state)
>>> dqdt = function_of(aux, state)
```

and in the numerical integration schema step

```
>>> state = function_of(dqdt)
```

This method returns a namespace object, which is basically a fancy dictionary. It is used over a simple dictionary just for shorter syntax.

The following examples demonstrate a deeply nested corner case, i.e. a compute graph consisting of a single "long" Euler cycle. By breaking up this cycle at the integrations, `variable_ordering()` can linearize these cycles correctly. This works both for non-strict and strict element naming.

```
>>> from dda.computing_elements import neg,int,mult
>>> dda_state = State({"x": neg(int(neg(int(neg(mult(1, Symbol("x
↪")), 0.005, 1)), 0.005, 0))) })
>>> # variable ordering is made based on non-strict naming:
>>> dda_state.name_computing_elements(strict=False)
State({'int_1': int(neg_1),
       'int_2': int(neg_2),
       'mult_1': mult(1, x),
       'neg_1': neg(mult_1, 0.005, 1),
       'neg_2': neg(int_1, 0.005, 0),
       'x': neg(int_2)})
>>> dda_state.name_computing_elements(strict=True)
State({'int_1': int(neg_1),
'int_2': int(neg_2),
'mult_1': mult(1, x),
'neg_1': neg(mult_1, 0.005, 1),
'neg_2': neg(int_1, 0.005, 0),
'neg_3': neg(int_2),
'x': neg_3})
>>> # This is how the full output looks like
>>> dda_state.variable_ordering()
namespace(aux=namespace(all=['mult_1', 'neg_1', 'neg_2', 'x'],
                        sorted=['x', 'mult_1', 'neg_2', 'neg_1'],
                        cyclic=[],
                        unneeded=set()),
          evolved=['int_1', 'int_2'],
          explicit_constants=[],
          all=['int_1', 'int_2', 'mult_1', 'neg_1', 'neg_2', 'x'],
          ordering=OrderedDict([('vars.explicit_constants', []),
                                ('vars.aux.sorted',
                                 ['x', 'mult_1', 'neg_2', 'neg_1
↪']),
                                ('vars.aux.cyclic', []),
                                ('vars.evolved', ['int_1', 'int_2
↪']),
                                ('vars.aux.unneeded', set())]),
          where_is={'int_1': 'vars.evolved',
                    'int_2': 'vars.evolved',
                    'mult_1': 'vars.aux.sorted',
                    'neg_1': 'vars.aux.sorted',
                    'neg_2': 'vars.aux.sorted',
                    'x': 'vars.aux.sorted'})
>>> # Compare the strict and nonstrict orderings:
>>> for k, v in dda_state.variable_ordering().ordering.items():␣
↪print(f"{k:25s}: {v}")
vars.explicit_constants  : []
vars.aux.sorted          : ['x', 'mult_1', 'neg_2', 'neg_1']
vars.aux.cyclic          : []
vars.evolved             : ['int_1', 'int_2']
```

(continues on next page)

(continued from previous page)

```
vars.aux.unneeded        : set()
>>> for k, v in dda_state.name_computing_elements(strict=True).
↪variable_ordering().ordering.items(): print(f"{k:25s}: {v}")
vars.explicit_constants  : []
vars.aux.sorted          : ['neg_3', 'x', 'mult_1', 'neg_2', 'neg_
↪1']
vars.aux.cyclic          : []
vars.evolved             : ['int_1', 'int_2']
vars.aux.unneeded        : set()
```

remove_duplicates()
Assuming a linearized state, this function simplifies the system by removing/resolving duplicate entries. No further renaming takes place: Always the first encounter of a term determines the name for all equivalent terms.

Returns a new state.

term_statistics()
Assuming a linearized state, tells how much each term occurs within this state. Returns instance of `collections.Counter`. For simplicity, the term heads are returned as strings.

Typical usage is like `state.name_computing_elements().remove_duplicates().term_statistics()`

export(*to*, ***passed_args*)
Syntactic sugar for dda.export(), for convenience

class dda.ast.**BreveState**(*initialdata={}*, *type_peacemaking=True*, *default_symbol=True*)
This subclass of a state adds *syntactic sugar* by allowing attribute/member access notation. Instead of `state["foo"]` you can write `state.foo` on instances of this class. Example usage:

```
>>> x,y,z = symbols("x,y,z")
>>> state = BreveState()
>>> state.x = x(y,z)
>>> state.y = y(x,z)
>>> state.z = z(x,y)
>>> print(state)
BreveState({'x': x(y, z), 'y': y(x, z), 'z': z(x, y)})
```

Warning: Known limitations:

- Breaks Python class introspection (for instance tab completion in *iPython*)
- Of course users cannot add any non-data related attribute (or method)

See also *State.equation_adder()* (page 30) for similar sugar which might have unexpected effects:

```
>>> s, b = State(), BreveState()
>>> foo = Symbol("bar") # in this context, similar to a string foo =
↪"bar"
>>> s[foo] = 42          # foo resolves to string represntation "bar"
>>> b.foo = 42           # equals b["foo"], thus has nothing to do␣
↪with variable foo
```

(continues on next page)

(continued from previous page)

```
>>> s
State({'bar': 42})
>>> b
BreveState({'foo': 42})
```

If you find this class useful, you also might like `types.SimpleNamespace` or `collections.namedtuple`. Both are basically immutable, while this object type is mutable by intention.

2.3.2 The DDA Domain Specific Language

The DDA domain specific language, also refered to as "traditional dda", is the C-like language invented by Bernd for his Perl'ish dda code.

It basically reads as the following snippet:

```
# Single-line comments are written like this

dt = const(0.5)  # constants are defined like this
y0 = const(1)

z = mult(y, y)
y = int(y, dt, y0)
```

As you see, variables do not even have to be introduced and can be used in any order. There is only one data type, the *analog line signal* which is basically a real number within a fixed interval.

Note: Interestingly, traditional DDA files are a python subset and thus can be easily parsed and generated by python syntax. That's why the code of this PyDDA module is so short. That's also a primary reason why DDA was rewritten in Python.

Demonstration usage of this module:

```
>>> traditional_dda_document='''
... # Single-line comments are written like this
...
... dt = const(0.5)  # constants are defined like this
... y0 = const(1)
...
... z = mult(y, y)
... y = int(y, dt, y0)
... '''
>>> state = read_traditional_dda(traditional_dda_document)
>>> state
State({'dt': const(0.5), 'y': int(y, dt, y0), 'y0': const(1), 'z': mult(y,␣
↪y)})
>>> print(to_traditional_dda(state))
# Canonical DDA file generated by PyDDA

dt = const(0.5)
y = int(y, dt, y0)
y0 = const(1)
z = mult(y, y)
```

dda.dsl.**to_traditional_dda**(*state*, *cleanup=True*, *prefix='# Canonical DDA file generated by PyDDA\n'*, *suffix='\n'*)
Export state to canonical dda file format (i.e. without all the python).

Returns the generated DDA file as string

dda.dsl.**read_traditional_dda**(*content*, *return_ordered_dict=False*)
Read some traditional dda file. We use the Python parser (`ast` builtin) for this job. This is possible because the DDA syntax is a python subset and the parser doesn't care about semantics, only syntax.

Thanks to the `ast` builtin package, we can just transform the python AST to the Symbolic/State class data structures used in this module.

Note: If some of the assertions fail, you can debug your DDA file by inspecting the output of ast.parse(content) on iPython. You can also run the Python debugger (pdb) on this function, for instance in iPython:

```
>>> %pdb
>>> read_traditional_dda(file("foo.dda").read())
```

Returns a state instance or OrderedDict, on preference.

dda.dsl.**read_traditional_dda_file**(*filename*, ***kwargs*)
Syntactic sugar for *read_traditional_dda()* (page 37), so users can directly pass a filename if the have their DDA code in a file.

dda.dsl.**cli_exporter**()
A Command Line Interface (CLI) for PyDDA.

This CLI API does mainly what the old dda2c.pl script did, i.e. translating a (traditional) DDA file to C code. There are fewer options, because –iterations, –modulus and –variables are now runtime options for the generated C program.

However, we can generate much more then C. Output is always text.

Invocation is either `python -m dda --help` or `python -m dda.dsl --help` anywhere from the system. `setup.py` probably also installed a `pydda` binary somewhere calling the same. You can also just call `./dsl.py --help`.

2.3.3 DDA Computing elements

The DDA language is built around the analog computing elements (or primitives). These are basically electrical block circuits implementing basic arithmetics such as summation and multiplication, but also integration. These are also special elements for clipping, exponentials, square roots, and many more.

The DDA domain specific language is agnostic for function names. Host languages such as Python or C are not. Many of the function names are reserved words in these languages. Examples are:

- `const`: Reserved word in C/C++ for constant variables
- `int`: Type name in C, overwritable in python
- `sum`: Builtin in python
- `div`: Function in `stdlib.h` in C, which can cause clashes in slightly more complex C codes.

When it comes to exporting to languages such as C, we rewrite these keywords. In Python, we don't have to, because none of the well known DDA function names is really *reserved*. The primitive builtins can always be recovered by `from builtins import int, sum`, etc.

Different ways to access the well-known DDA computing elements in Python

If you write

```
>>> from dda.computing_elements import *
>>> int(int,sum)   # Make use of the imported Symbols
int(int, sum)
```

you will load a bunch of names such as `int` and `sum` in your local namespace.

You can also just call

```
>>> from dda.computing_elements import dda_functions, dda_symbols
>>> print(dda_symbols["floor"]) # will print a Symbol()
floor
```

You can use the symbols dictionary to populate your namespace at whish:

```
>>> globals().update(dda_symbols)
```

Note that Python globals are module-local, so we cannot provide this line as a function.

Last but not least, you can also just use the namespaced version with prefixes, which leaves you on a safe footing:

```
>>> dda.floor(dda.sum)
floor(sum)
```

Definition/Implementation of the primitives

Once we have a pure-Python DDA evolution code (probably using scipy), we will have a python implementation of the DDA functions. Otherwise I could avoid that. This module also contains a C++ implementation of the primitives, which resides as a string (`cpp_impl`).

2.3.4 DDA C++ code generator

C++ code generation is a major feature of the PyDDA code. The generated code is built from a string template and has the following features:

- Standalone code: No further dependencies (beyond standard libc, lm and STL).
- Lightweight object oriented: Uses classes (structures) to hold the different variables (basically AoS instead of SoA approach). Little C++ templating.
- Organization in a few functions which allows to edit the generated C++ code manually afterwards without going mad.
- CSV or binary output, or no output at all. Output is always made to stdout. Information messages are always sent to stderr.

- Debugging facilities built right into the code for setting NaNs and abortion in case of floating point exceptions.
- Runtime arguments via the commandline (argv): Parsing and passing.

Note: **C++17** is required for building the C++ code. This is because we use variadic templates.

For the C++ runtime arguments, we support so far:

- *Simulation steering*: Selection of number of integration iterations and frequency of dumping the solution.
- *Query based plotting*: Selection which variables shall be outputted at runtime.
- Further *Flags* and *Numeric arguments* as well as a useful `--help` message.
- *Initial data* and *time step sizes* can also be chosen at run time.
- *Introspection capabilities*, for instance one can ask the binary about the evolution quantities built in.

Basically the equation structure is the only thing left hardcoded at C++ code generation time.

`dda.cpp_exporter.`**`to_cpp`**(*state*, *number_precision=inf*, *constexpr_consts=False*)
Given a state, returns standalone C++ code as string.

This code can be written to a file, compiled with a recent C++ compiler and then solves the differential equation system when executed.

The algorithm is basically:

1. linearize the state (this can raise)
2. determine all the C++ template fields
3. Return the filled out template

We plan to add logging for non-fatal information about the C++ code quality (see TODOs in the code).

The argument `number_precision` currently has no effect.

`dda.cpp_exporter.`**`compile`**(*code*, *c_filename='generated.cc'*, *compiler='g++'*, *compiler_output='./a.out'*, *options='--std=c++17 -Wall'*)
Small helper function to compile C++ code from python.

Write string *code* to *c_filename* and run the *compiler* on that, afterwards. Will raise an error if compilation fails.

`dda.cpp_exporter.`**`runproc`**(*command*, *decode=False*)
Helper to run external command and slurp its output to a binary array

`dda.cpp_exporter.`**`list_all_variables`**(*command='./a.out'*)
in order to know which fields have been read, slurp all variables

`dda.cpp_exporter.`**`run`**(*command='./a.out'*, *binary=False*, *arguments={}*, *fields_to_export=[]*)
Small helper function to execute a code generated by this module.

Runs `command` on the command line, with given dict `arguments` in `--foo=bar` fashion and `fields_to_export` just as a sequential argument list. If no `fields_to_export` is given, `command --list_all_variables` will be run to query all default fields.

Pipes stdout to a string, which is returned. Stderr will just be passed. The function will return once the binary finished or raise in case of error.

If you set `binary=True`, raw data instead of CSV files will be passed between the spawned command and this python program. This decreases the runtime significantly if you write a lot of data (since CSV generating and parsing overload is gone).

Example usage:

```
>>> from dda import *
>>> state = State()
>>> state["x"] = Symbol("int", Symbol("neg", state["x"]), 0.2, 1)
>>> state
State({'x': int(neg(x), 0.2, 1)})
>>> cpp_code = to_cpp(state)
>>> print(cpp_code)
// This code was generated by PyDDA.
#include <cmath> /* don't forget -lm for linking */
#include <cfenv> /* for feraisexcept and friends */
#include <limits> /* for signaling NAN */
#include <vector>
....
>>> compile(cpp_code, compiler_output="foo.exe")
>>> res = run("./foo.exe", arguments={'max_iterations':10}, fields_to_
↪export=['x'])
Running: ./foo.exe --max_iterations=10 x
TODO: Doctesting this doesn't work good due to stderr (cf https://
↪stackoverflow.com/a/61533524)
>>> print(res)
x
1.2
1.44
1.728
2.0736
2.48832
2.98598
3.58318
4.29982
5.15978
6.19174
```

dda.cpp_exporter.**numpy_read**(*stdout*, *binary=False*, *return_ndarray=True*, *return_recarray=False*, *fields_to_export=[]*)

Postprocessing to fill the gap between the C++ output and a suitable numpy array. In order to so, this function has to know whether your output was binary or text. Furthermore, you need to tell him how many fields you had. You can use *list_all_variables()* (page 39)

Old Text:

This option only makes real sense if you set (the default) `return_ndarray=True`. Note that if you don't pass the `fields_to_export` option but set `binary=True`, in the moment the returned array is one-dimensional (a warning will be printed). If you like even more structured data be returned, turn on `return_recarray=True`. It will return a numpy.recarray, the same data type which you get when you read CSV data with column headers. `return_recarray=True`

implies `return_ndarray=True`.

class `dda.cpp_exporter.`**Solver**(*dda_state_or_code*, **runtime_fields_to_export*, *constexpr_consts=True*, ***runtime_arguments*)

Syntactic sugar for a more concise OOP feeling. Instead of calling `export(to="C")`, `compile()` and `run()` you can just write `Solver(state, runtime_arguments)`. This object will even clean up after running.

run(**runtime_fields_to_export*, *binary=False*, *cleanup=True*, ***runtime_arguments*)

Chaining and Syntactic sugar for delayed argument setting/overwriting

as_ndarray()

Return run results as a `np.ndarray` (i.e. like a table without headers, typically 2D data)

as_recarray()

Return run results as a `np.recarray` (i.e. like CSV table with named headers)

2.3.5 DDA SciPy interface (to generic ODE solvers)

The `dda.scipy` module allows in-python evaluation of DDA systems as well as their solution with ODE Integrators in Python, such as scipy.integrate[48]. For the usage and examples, see the main class *to_scipy* (page 41).

Note: In order to run this code, you need, obviously, SciPy[49] next to NumPy[50].

Warning: This module exposes a solver of a DDA system which is quite different to the `cpp_exporter`. In particular,

- The solver is required to be told the solution time span or final time in time units, not iteration indices.
- The solver only spills out the evolved (integration) quantities and not any derived quantities. You can recompute them at any timestep, but there is currently no code helping you in order to archieve this result. This can result in confusion when you cannot query the fields you asked for (in particular in the CLI frontend).
- The SciPy time integrator tries to find an optimal (and minimal) time step, yielding in a quite "rough" solution. You can turn on *dense output* in order to tell the SciPy solver to integrate between these time steps, yielding a more smooth output with more datapoints.

`dda.scipy.`**evaluate_values**(*smbl*, *values*)

Evaluate a symbol within the context of an already evaluated values dictionary given.

class `dda.scipy.`**to_scipy**(*state*)

The SciPy exporter. When initializing this class with your DDA system, it will setup a function `f(y)` which can be evaluated as any right hand side in the ordinary differential equation (ODE) system `dy/dt = f(y)`. Here, `y` are the evolution quantities, i.e. a vector which is composed automatically from the linearized DDA system (see `ast.State`.

[48] https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html
[49] https://www.scipy.org/
[50] https://numpy.org/

`name_computing_elements()` and `ast.State.variable_ordering()`). Furthermore, this class prepares the initial values `y0` for the integration.

You can evaluate these quantities in any python context, i.e. with any scientific python ODE solver library. For the time being, this class provides a convenience method *`solve()`* (page 44) which calls scipy.integrate.solve_ivp[51]. There is no other scipy dependence in this code.

Usage example:

```
>>> from dda.computing_elements import neg,int,mult
>>> from dda import State, symbols
>>> x, y, z = symbols("x, y, z")
>>> dda_state = State({x: int(y, 1, 1), y: mult(2,x), z: neg(int(z, 1,
↪ -1)) })
>>> py_state = to_scipy(dda_state)
>>> py_state.state # state has been linearized before processing
State({'int_1': Int(z), 'x': Int(y), 'y': mult(2, x), 'z': neg(int_1)}
↪)
>>> py_state.vars.evolved # evolved variables are therefore not [x,z]␣
↪but [int_1,x]
['int_1', 'x']
>>> py_state.y0  # initial values
array([-1,  1])
>>> py_state.dt # same timestep for all integrals (but see note below)
1
>>> py_state.rhs(py_state.y0)  # evaluation of f(y) on y0
array([-1, -2])
>>> y1 = py_state.rhs(py_state.y0) * py_state.dt  # a single Euler␣
↪integration timestep, for instance
>>> y1
array([-1, -2])
>>> sol = py_state.solve(10)  # ODE integration with SciPy
>>> sol.t  # integration went from 0->10 with 17 timesteps
array([ 0.        , ..., 10.        ])
>>> sol.y[:,-1] # the first solution is ~exp(+t)->inf, the second␣
↪exp(-t)->0
array([-2.20269685e+04,  1.94334984e-08])
>>> from pylab import plot, legend, show  # plotting example
>>> for i,fieldname in enumerate(py_state.vars.evolved):
        plot(sol.t, sol.y[i], label=fieldname)
>>> legend(); show()
```

Warning: Due to the way how widespread ODE integrators work, the per-integral step size `dt` is required to be the same for every integration which is part of the DDA system. That is, the following generates an error:

```
>>> from dda import dda, symbols
>>> a,b=symbols("a,b")
>>> state = State({ a: dda.int(a, 0.2, 0), b: dda.int(b, 0.5, 0) })
>>> to_scipy(state)
Traceback (most recent call last):
...
ValueError: Scipy requires all timesteps to be the same, however dt_
↪(['a', 'b']) = [0.2 0.5]
```

[51] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html

Most high-level integrators available in scientific Python toolkits (such as scipy) assume the overall system to have a single timestep size Δt (which is also quite natural from a mathematical perspective). The signature `dda.int(f,dt,ic)` is thus quirky from a mathematical or numerical viewpoint. It is written in such a way because analog computing integrators have a tunable time scale `k_0 ~= 1/dt` which however can also be consumed in the integrand itself: `dda.int(f,1/k_0,ic) == dda.int(f/k_0,1,ic)`.

Furthermore, most high-level integrators do adaptive timestepping anyway. The fine-tuning of timestep sizes is something which is only paid respect to in the `cpp_exporter` module.

evaluate_const(*var*)

Translate `const(foo)` by stripping foo or some bar by looking up if it is an explicit constant. Dynamical variables are not allowed here. This is somewhat similar but different to `cpp_exporter.lookup_const(var)`.

evaluate_state(*evolution_vector*, *copy=False*)

Recomputes the full state from the evolution state vector. Returns a dictionary with same keys as `self.state` and scalars (floats) as values.

This will especially compute the aux variables, while for the evaluation variables the RHS of `dy / dt = f(y)` is computed.

Note: As a user, you most likely want to call *reconstruct_state()* (page 43) or *rhs()* (page 44) instead of this function.

For optimization purpose, numerical state evaluation is always carried out on the `evaluation_default_values` member (which also hold the initial values for the first `rhs` evaluation). If you set `copy=True`, a shallow copy (which is equal to a deep copy for a dict holding floats) is returned. In external calls, you should probably always set `copy=True`.

Note: The implementation of this function currently evaluates the (prepared) DDA sytem by recursive calls with the help of a variable assignment directory. This is basically a *run-time compilation* (JIT/VM) in pure python. Needless to say, this won't give a great performance!

There are plenty of low-hanging fruits to provide optimized versions of this code: One could call the efficient (but still scalar) C++ implementation which `cpp_exporter` provides by methods provided by Cython[52]. One could also map the DDA abstract syntax tree (AST) to the python one and use some unparser code to evaluate the DDA code as pure python (see for instance Python: Modify AST and write back python code[53]).

For the time being, this code remains as a pure demonstration code. Thanks to using the linearized state, there should be no troubles with call stack overflows, however cyclic dependencies may not be properly resolved and can result in infinite recursions (stack overflow).

---

reconstruct_state(*evolution_vector*, *copy=True*)

Given the evolution vector sizes, this computes the full state. That is, this function differs

[52] https://cython.readthedocs.io/en/latest/src/userguide/external_C_code.html

[53] https://stackoverflow.com/questions/768634/parse-a-py-file-read-the-ast-modify-it-then-write-back-the-modified-source-c

from `evaluate()` at all evaluation quantities where the values of the evolution vector itself are put in place.

rhs (*evolution_vector*)
The ODE *right hand side* in `dy / dt = f(y)`. `y` is a numpy vector, and `f(y)` returns a similarly sized (numpy) vector which we call `rhs` here:

```
>>> ode = to_scipy(State({ Symbol("x"): Symbol("int")(Symbol("x"),
↪0.1,1) }))
>>> y1 = ode.y0 + ode.rhs(ode.y0) * ode.dt    # perform some euler␣
↪integration step
>>> y1
array([0.9])
```

Usually, you want to pass this function to some scipy integrator. See also `ft()`.

rhst (*t*, *evolution_vector*)
Syntactic sugar for scipy integrators who want a signature `rhst(t,y)`. Will just call `rhs(y)` instead.

solve (*tfinal*, ***kwargs*)
Basically passes all arguments to `scipy.integrate.solve_ivp`. See documentation for *to_scipy* (page 41) for usage example.

Currently, it is hardcodedly `tspan=[0,tfinal]`. All other (keyword) arguments are passed to `solve_ivp`.

`dda.scipy.`**cli_scipy**`()`
A Command Line Interface (CLI) for *dda.scipy* (page 41).

This CLI API basically solves a DDA file (see *dda.dsl* (page 36) for the syntax). This is a different approach then using the *dda.cpp_exporter* (page 38) Instead of code generation (and the need for a C++ compiler), this evaluates the DDA file within python. The disadvantage is that this is damned slow, the advantage is that the time integrator is much better then the selfmade one in the *dda.cpp_exporter* (page 38) module. And there is no need for a C++ compiler at all, all is (more or less) pure python.

Invocation is like `python -m dda.scipy --help` anywhere from the system. Run this to explore the usage of this command line interface.

The output will be CSV (file or stdout), in terms of one line per integration step (called *dense solution* in scipy ODESolver language).

2.3.6 Computer Algebra Interfaces

DDA implements a few parts of computer algebra system (CAS), especially with it's *Abstract Syntax tree* (page 22). Since we don't want to reinvent the wheel, we interface with common computer algebra systems. There are at least two popular for the Python ecosystem available:

- SymPy[54], bundled within the SciPy[55] package, can be easily used as a pure python library.

[54] https://www.sympy.org/
[55] https://www.scipy.org/

- Sagemath[56], which is more of a monolithic software. The symbolic foundation of sage is provided by Ginac[57] and Pynac[58], respectively. Many open source computer algebra systems are bundled with sage, such as Maxima[59] and Octave[60]. Furthermore, interfaces to many others such as Maple[61], Mupad[62] or Mathematica[63] are part of sage.

So far, we had quick success with adopting SymPy (see next section).

SymPy module API reference

This module provides interplay with the SymPy package. SymPy is a lightweight pure-python computer algebra system which is bundled with SciPy. An adapter to/from SymPy allows to use powerful Computer Algebra basic functions such as expression simplification.

We use this currently to provide a lean latex representation for the cumbersome DDA expressions.

`dda.sympy.`**`from_sympy`**(*sympy_equation_list*)

Import a state from a set of equations from SymPy.

This function expects a python list of sympy equations where there is a single sympy symbol on one hand and an expression on the other hand (see examples below).

The mapping basically follows the SymPy key invariant[64]: "Every well-formed SymPy expression must either have empty `args` or satisfy `expr == expr.func(*expr.args)`".

Therefore we basically map a sympy expression `(expr.func, expr.args)` to the DDA `(head, tail)` notation. While the heads are easy to map (for instance, `sympy.Mul` equals `dda.mult`), special attention must be given to the tails, for instance SymPys `Mul(a, b, c)` translates to DDAs `mult(mult(a,b),c)` (in DDA we always assume commutative real-valued variables). Also in DDA there is `neg(x)` or `div(x,y)` which is represented in SymPy as `Mul(Integer(-1), x)` and `Mul(Symbol('x'), Pow(Symbol('y'), Integer(-1)))`, respectively.

`dda.sympy.`**`to_sympy`**(*state*, *symbol_mapper=<function <lambda>>*, *round_n=15*)

Export a state to a set of equations for SymPy. Returns a list of `sympy.Eq` objects. Of course it requires Sympy installed/available.

Note: The heart of this function is a mapping from `ast.Symbol` terms (functions) to Sympy functions, for instance by mapping `Symbol("int")(...)` to `-sympy.Integral(sympy.Add(...), t)`.

Thanks to the ease of the computing elements, this mapping does not require pattern matching but can be performed on a basic level. However, not all terms are yet supported.

The argument *symbol_mapper* allows to apply another mapping on the DDA Symbol heads. By default, it is the identity function.

[56] https://www.sagemath.org/
[57] https://ginac.de/
[58] http://pynac.org/
[59] http://maxima.sourceforge.net/
[60] https://www.gnu.org/software/octave/
[61] https://maplesoft.com/
[62] https://www.mathworks.com/discovery/mupad.html
[63] https://www.wolfram.com/mathematica/
[64] https://docs.sympy.org/latest/tutorial/manipulation.html#args

With Sympy, you can do all funny things, such as:

```
>>> from dda import *
>>> x,int,neg=symbols("x,int,neg")
>>> state = State({'x': int(neg(x), 0.2, 1)})
>>> to_sympy(state)
[Eq(x, -Integral(1.2 - x, t))]
```

dda.sympy.**to_latex**(*state*, *chunk_n=None*)

Export to latex, using sympy.

This mostly differs from `sympy.latex` for large equation systems where we use the align latex environment instead of a single equation. For the above example:

```
>>> import sympy, dda
>>> x,int,neg=dda.symbols("x,int,neg")
>>> state = dda.State({'x': int(neg(x), 0.2, 1)})
>>> print(sympy.latex(to_sympy(state)))
\left[ x = - \int \left(1.2 - x\right)\, dt\right]
>>> print(to_latex(state))
\begin{align}
x &= - \int \left(1.2 - x\right)\, dt
\end{align}
```

2.3.7 The DDA module

(The following documentation is about the module itself and not very useful in the moment)

PyDDA is a small library to write and generate DDA code in Python. DDA stands for *digital differential analyzer*. In this context, it is a code for solving ordinary differential equations given in a domain specific language description (i.e. an electrical circuit).

For further details, please see the `doc/` directory (Sphinx documentation).

dda.**export**(*state*, *to*, ***kw*)

Convenience function to export (transform) a state to some other programming language.

Possible formats (allowed values for `to`) supported so far are:

- C/C++ (via *dda.cpp_exporter* (page 38))
- DDA (via *dda.dsl* (page 36))
- SymPy (via *dda.sympy* (page 45))
- Latex (via *dda.sympy* (page 45))

This function shall be nice, so it accepts many spelling/notation of these language names.

The return value are typically strings or tuples, dicts. There should be no side effects.

dda.**clean**(*thing*, *target='C'*)

Cleans an identifier for being compatible with the *target* language. This can be something like *C*, *python* or *dda* (cf. languages supported by *dda.export()* (page 46)) or also *latex*.

It will basically try to transliterate all Unicode to ASCII and then try to ensure that the identifier is a valid C variable name (i.e. don't start with numbers, etc.).

This function is nice, if you pass a `dda.State` or `dda.Symbol`, it will map the whole State/Symbol. Otherwise, it expects a string.

Examples:

```
>>> clean("\frac{x}{y}")  # backslashes are just removed
'fracxy'
>>> clean(r'a^{-1}')
'a__1'
>>> clean('a^b_c^{ef}')
'a_b_c_ef'
>>> clean(u'μ²')   # only if python package "unidecode" is installed
'u2'
>>> clean('77%alc')   # well, you can use numbers at the beginning of␣
↪strings
'_77alc'
```

2.4 Example circuits for DDA

We have a number of example circuits written in either

- traditional DDA language (can be run with any DDA compiler)
- pythonic DDA (requires the PyDDA code but can be translated also to traditional DDA files)
- Jupyter/IPython notebook files (i.e. pythonic DDA but within a rich-text document)

These files are located in the *examples/* directory and are there for being explored by interested users. A couple of files are gone into detail in the following.

2.4.1 Traditional DDA circuits

`chua.dda` A traditional DDA code, implementing the Chua attractor from chapter 6.15 from Bernds new book (ap2.pdf). We have both a scaled and unscaled version along with a plotting code available in this repository. For plotting, we also have capabilities to do a high-quality "phase space" histogram based on massive binary output of the C++ integrator.

`double-pendulum.dda` Another traditional DDA code, implementing a planar coupled gravity pendulum in classical small-angle approximation and in a formulation with two angles as degrees of freedom. The example stems from section 6.25 in Bernds new book (ap2.pdf). Again, we have a small Python code for plotting the results.

2.4.2 Command line DDA usage

An example on how to use PyDDA from command line: Gradiend descent

This example is a go-throught of how using PyDDA from the Unix (Linux or Mac OS X) command line. We use the Rosenbrock function[65] as an example for Gradient descent[66].

The file `rosenbrock.dda` is given with the following content:

[65] https://en.wikipedia.org/wiki/Rosenbrock_function
[66] https://en.wikipedia.org/wiki/Gradient_descent

```
# Gradient descent on Rosenbrock problem with continous-time analog␣
↪computing
# f(x,y) = (a-x)**2 + b*(y - x**2)**2

a = const(1)
b = const(100)

# Initial conditions are starting points for optimization

ic_x  = const(2)  # x(t0)
ic_my = const(2)  # -y(t0)

# timestep size is more for the numerical integrator
dt = const(0.01)

# next lines are the gradient
# D[f,x] = -2 (a - x) - 4 b x (-x^2 + y)
# D[f,y] = 2 b (-x^2 + y)
# written in DDA notation.
# Keep the negating sums and integrals in mind! Sum[x,y] == neg(sum(x,y))

x = neg(neg(int(mult(2,sum(a,neg(x))), mult(4,mult(b,mult(x,sum(neg(mult(x,
↪x)),y))))), dt, ic_x)))
y = neg(int(mult(2,mult(b,sum(neg(mult(x,x)),y))), dt, ic_my))
```

It can now be run on the shell by running the following commands:

```
python -m dda rosenbrock.dda c > rosenbrock.cpp
g++ -std=c++17 -o rosenbrock.o rosenbrock.cpp
./rosenbrock.o --dt:int_1=0.0001 --dt:int_2=0.0001  --max_
↪iterations=100000 x y | tee output.txt
```

You can now plot the output for instance with gnuplot:

```
set key top left autotitle columnhead
set logscale x
plot "output.txt" using 0:1 w l
replot "output.txt" using 0:2 w l
```

Or do anything else on the CSV data.

Inspecting the command line interface of the C++ ODE solver

Run the executable with `--help` to see all possible options:

```
Usage: ./rosenbrock.o [arguments] <variables_to_print>
This is an ODE integrator generated by PyDDA.

* Boolean arguments (Usage --foo or --foo=1 or --foo=0)
  always_compute_aux_before_printing (default value: 1)
  binary_output (default value: 0)
  debug (default value: 0)
  list_all_variables (default value: 0)
  skip_header (default value: 0)
  write_initial_conditions (default value: 0)
```

(continues on next page)

(continued from previous page)

```
* Numeric arguments: (Usage --foo=123)
  max_iterations (default value: 100)
  modulo_progress (default value: -1)
  modulo_write (default value: 1)
  number_precision (default value: 5)
  rk_order (default value: 1)
* Overwrite initial conditions (initial data): (Usage --initial:foo=1.23)
  int_1 (default value: 2)
  int_2 (default value: 2)
* Overwrite (per-variable) time step size: (Usage --dt:foo=0.001 or even --
↪dt:foo=1e-6)
  int_1 (default value: 0.01)
  int_2 (default value: 0.01)
* Overwrite constants: Not possible since compiled as compile time␣
↪constants.
* Query fields: (if none given, all are dumped)
  a, b, dt, ic_my, ic_x,
  int_1, int_2, mult_1, mult_2, mult_3,
  mult_4, mult_5, mult_6, mult_7, mult_8,
  neg_1, neg_2, neg_3, neg_4, sum_1,
  sum_2, sum_3, x, y

Exemplaric usage:
  ./rosenbrock.o --foo=1 --bar=0 --baz=7 --ic:var1=0.5 --dt:var2=0.01 --
↪const:something=42 var1 var2 var3

For more options and help, see the PyDDA code documentation at https://
↪pyanalog.readthedocs.io/
```

But I want a better ODE Solver!

The C++ based solver is really very basic. It allows to introspect all the quantities in a dense output, but for instance does not have adaptive time step size. Time to use something more mature, such as the mature ODE/IVP solvers by scipy[67] (which used to be a frontend for ODEPACK). There is a DDA interface to Scipy (see package `dda.scipy`) and it can also be used from command line:

```
python -m dda.scipy  -t 10 rosenbrock.dda | tee output2.txt
```

Note that in this example, `output.txt` and `output2.txt` hold similar data, since in the C++ solver case, the solver was run until `t_final = dt * N = 0.0001 * 100000 = 10`, the same as in the scipy solver case. However, by convention, the C++ solver does not include a time column (you have to generate it by yourself with `t=int(const(1), dt, 0)` if you need one).

[67] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.solve_ivp.html

2.4.3 Python DDA circuits

N-body.py is an example python file which generates traditional DDA code. It implements N-body physics with an inverse square law force (Coulomb/Newton like); the initial data show two particles in a two dimensional simulation domain on a circular orbit.

The code is written having in mind to compare the old dda2c.pl and the novel PyDDA implementation, therefore it has no coupling to the PyDDA code. It is in general not recommended to write PyDDA code like that.

md_water_toy.py A minimal molecular dynamics water toy simulation, basically an application of the N-body paradigm. This is a good and running example of a prototypical PyDDA application in science. It requires `numpy` because it compiles vector/matrix like quantities. It is also equipped with a plotting/visualization of the simulation after being run with a generated C++ code.

2.4.4 Jupyter/IPython Notebooks

The notebooks are embedded within this documentation.

DDA walkthrought example: Chua attractor

What follows is an example of the Chua attractor. It is described in the Analog Paradigm Application Note 3[68] as well as in section 6.15 in Bernd's new book (Analog Programming II). The attractor is described by a coupled set of three ordinary differential equations,

$$\begin{aligned} \dot{x} &= c_1(y - x - f(x)) \\ \dot{y} &= c_2(x - y + z) \\ \dot{z} &= -c_3 y \end{aligned}$$

with $f(x)$ a simple function decribing the Chua diode (given algebraically) and a number of parameters $c_{1,2,3}$. What follows is the scaling of these equations. The resulting set of equations is slightly more verbose. It's implementation is given in the following *traditional DDA* file:

[1]:
```
!cat chua.dda
```

```
#
# Copyright (c) 2020 anabrid GmbH
# Contact: https://www.anabrid.com/licensing/
#
# This file is part of the examples of the PyAnalog toolkit.
#
# ANABRID_BEGIN_LICENSE:GPL
# Commercial License Usage
# Licensees holding valid commercial anabrid licenses may use this file in
# accordance with the commercial license agreement provided with the
# Software or, alternatively, in accordance with the terms contained in
# a written agreement between you and Anabrid GmbH. For licensing terms
# and conditions see https://www.anabrid.com/licensing. For further
# information use the contact form at https://www.anabrid.com/contact.
#
# GNU General Public License Usage
# Alternatively, this file may be used under the terms of the GNU
```

(continues on next page)

[68] http://analogparadigm.com/downloads/alpaca_3.pdf

(continued from previous page)

```
# General Public License version 3 as published by the Free Software
# Foundation and appearing in the file LICENSE.GPL3 included in the
# packaging of this file. Please review the following information to
# ensure the GNU General Public License version 3 requirements
# will be met: https://www.gnu.org/licenses/gpl-3.0.html.
# ANABRID_END_LICENSE
#


# Chua attractor, chapter 6.15 from Bernds book ap2.pdf
# Below is the scaled version (equations 6.40-6.51)

x0 = const(0.1)
x1 = mult(-10, neg(sum(x, fx)))
x2 = neg(sum(y, mult(0.5, x1)))
x  = neg(sum(mult(3.12, neg(int(x2, dt, 0))), x0))

y1 = neg(sum(z, neg(mult(0.125, y))))
y2 = neg(sum(mult(1.25, x), mult(2, y1)))
y  = mult(4, neg(int(y2, dt, 0)))

z  = int(mult(3.5, y), dt, 0)

f1 = abs(sum(mult(0.7143,x),  0.2857))
f2 = abs(sum(mult(0.7143,x), -0.2857))
f3 = neg(sum(f1, neg(f2)))
fx = sum(mult(0.714, x), mult(0.3003, f3))

dt = const(0.001)
```

In the following, we use the PyDDA library to read in this DDA file and demonstrate the internal representation.

[2]:
```
from dda.dsl import read_traditional_dda
chua_text = open("chua.dda").read()
state = read_traditional_dda(chua_text)
state
```

[2]:
```
State({'dt': const(0.001),
 'f1': abs(sum(mult(0.7143, x), 0.2857)),
 'f2': abs(sum(mult(0.7143, x), -0.2857)),
 'f3': neg(sum(f1, neg(f2))),
 'fx': sum(mult(0.714, x), mult(0.3003, f3)),
 'x': neg(sum(mult(3.12, neg(int(x2, dt, 0))), x0)),
 'x0': const(0.1),
 'x1': mult(-10, neg(sum(x, fx))),
 'x2': neg(sum(y, mult(0.5, x1))),
 'y': mult(4, neg(int(y2, dt, 0))),
 'y1': neg(sum(z, neg(mult(0.125, y)))),
 'y2': neg(sum(mult(1.25, x), mult(2, y1))),
 'z': int(mult(3.5, y), dt, 0)})
```

Syntax trees: Down into the rabbit hole

Obviously, the output of the internal data structure *state* and the DDA file itself does not differ so much. That is by intention, both look quite pythonic. The state itself is basically a dictionary (mapping) from strings (the left hand sides in the DDA file) to the expressions (their right hand sides). Let's inspect such an expression.

[3]:
```
state["z"]
```

[3]:
```
int(mult(3.5, y), dt, 0)
```

[4]:
```
type(state["z"])
```

[4]:
```
dda.ast.Symbol
```

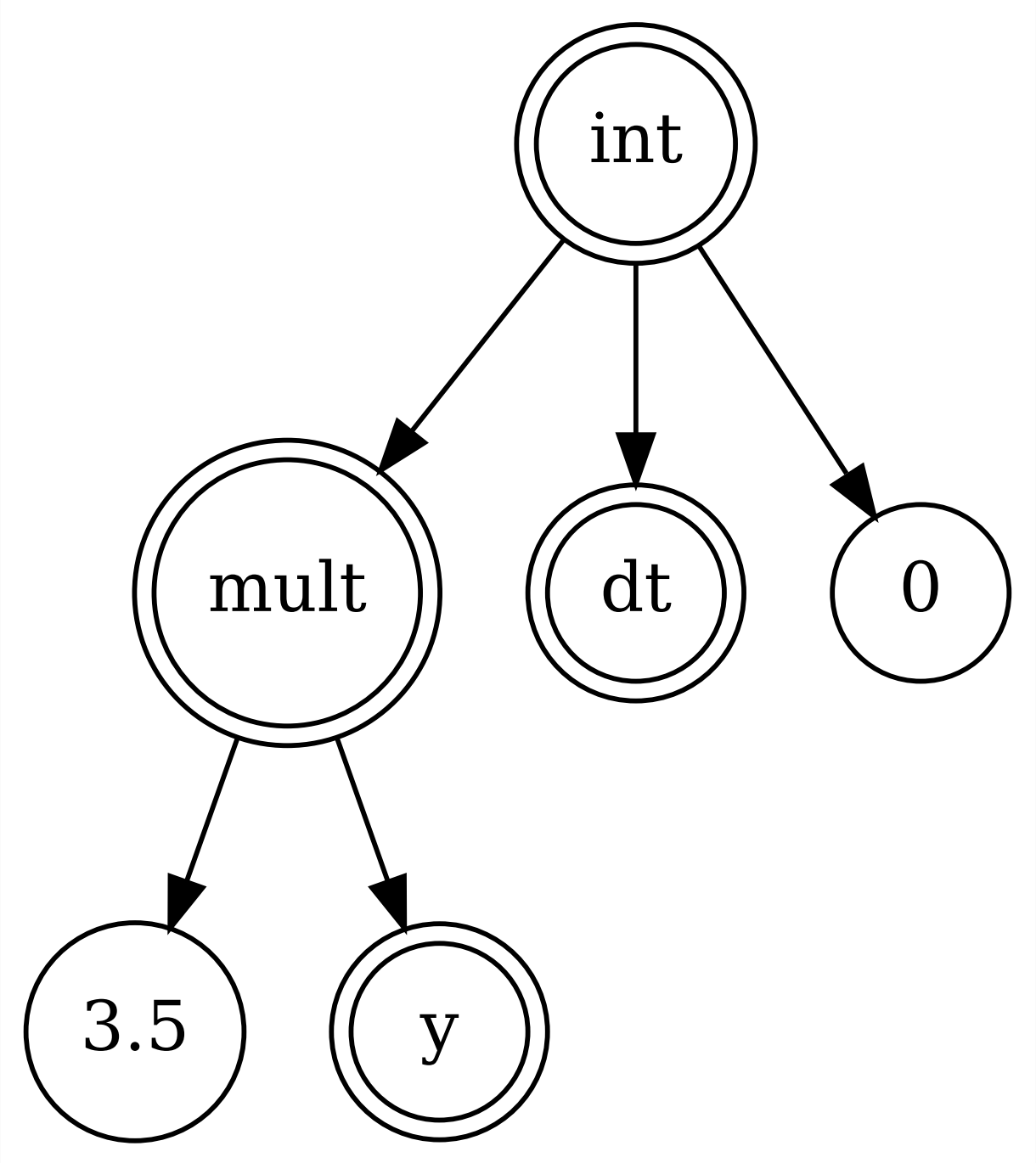
[5]:
```
print(state["z"].head)
print(state["z"].tail)
```

```
int
(mult(3.5, y), dt, 0)
```

What we are actually looking at is the PyDDA-representation of a mathematical expression tree. We can visualize this tree:

[6]:
```
state["z"].draw_graph()
```

[6]:

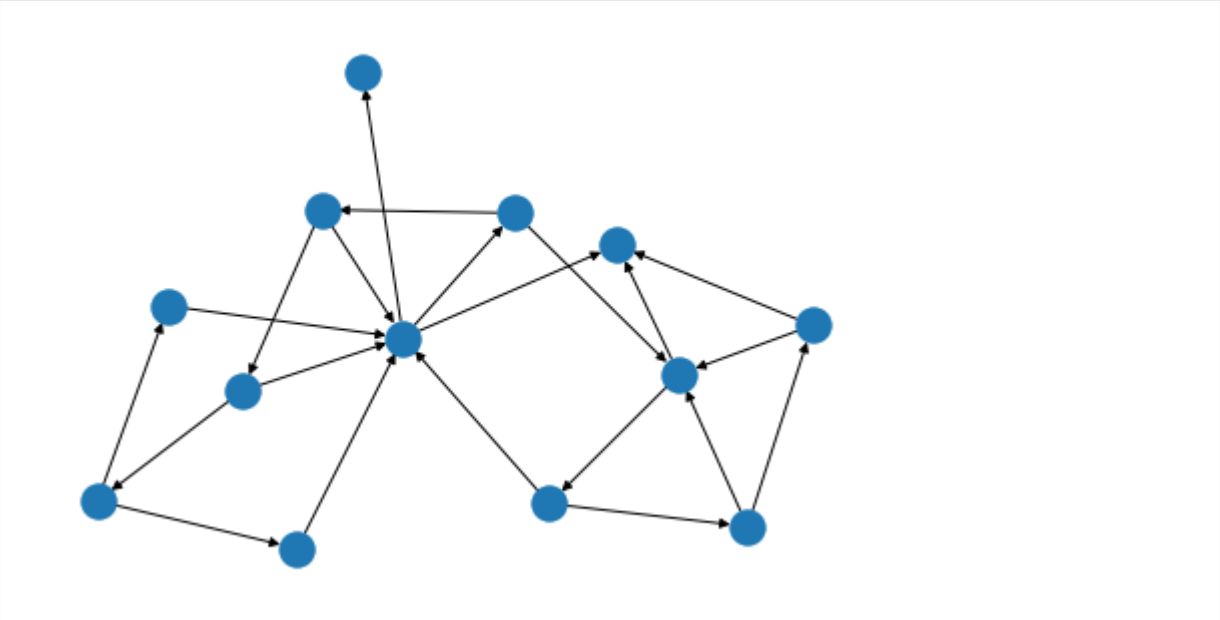


Given this, we can compute the dependencies of all variables in the sytem:

[7]:
```
graph = state.draw_dependency_graph(export_dot=False)
graph
```

[7]:
```
<networkx.classes.digraph.DiGraph at 0x7f431c5df3d0>
```

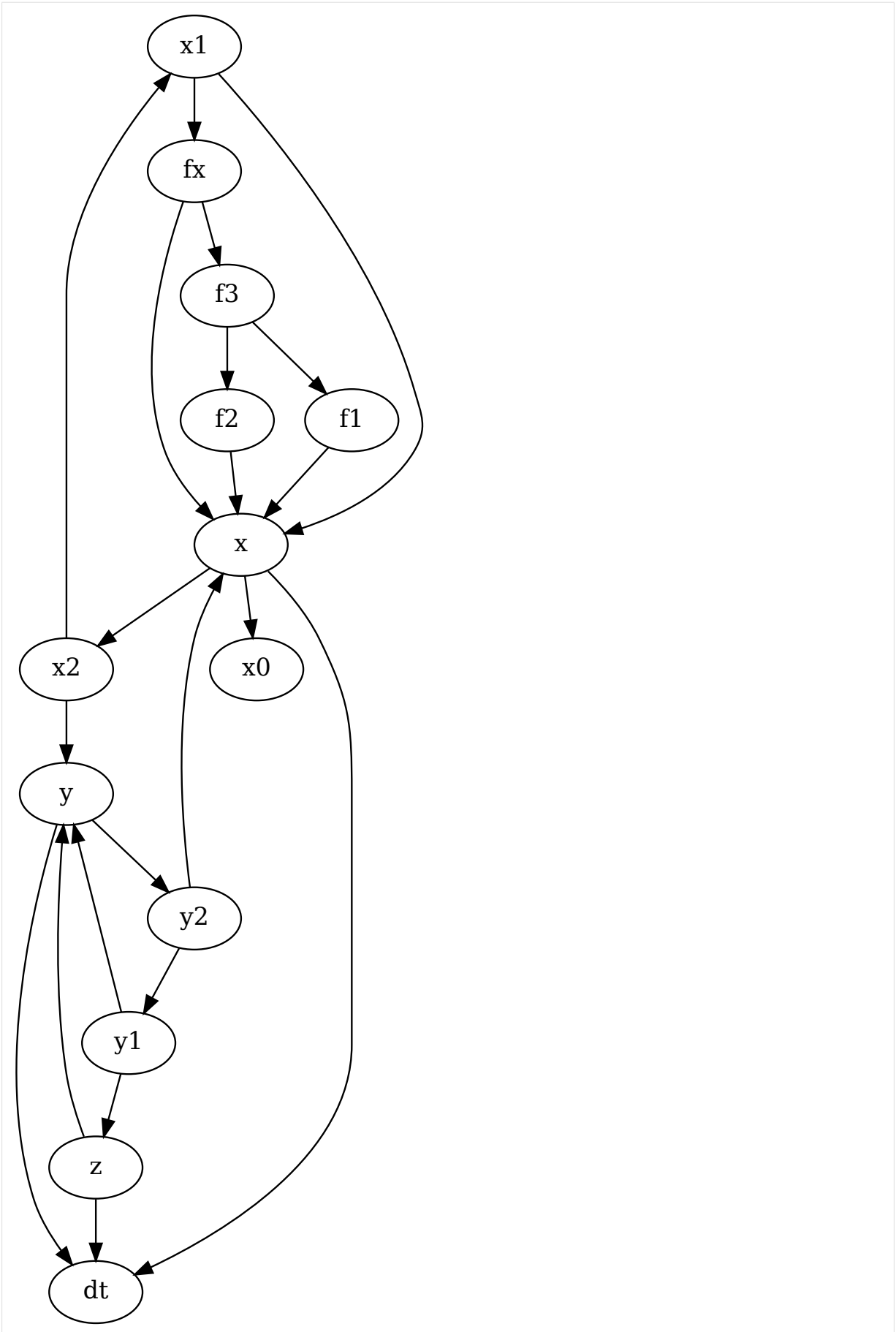
[9]:
```
# Draw the graph with matlotlib
import networkx as nx
from networkx.drawing.nx_agraph import graphviz_layout
pos = graphviz_layout(graph)
nx.draw(graph, pos=pos)
```



[10]:
```
# draw the graph with graphviz (requires pydot,graphviz)
from networkx.drawing.nx_pydot import to_pydot
from graphviz import Source
nx2dot = lambda graph: Source(to_pydot(graph).to_string())
```

[11]:
```
nx2dot(graph)
```

[11]:



Based on this dependency analysis, one can *linearize* the state, that is, define an ordering how to compute the state numerically:

[12]:
```
vars = state.variable_ordering()
```

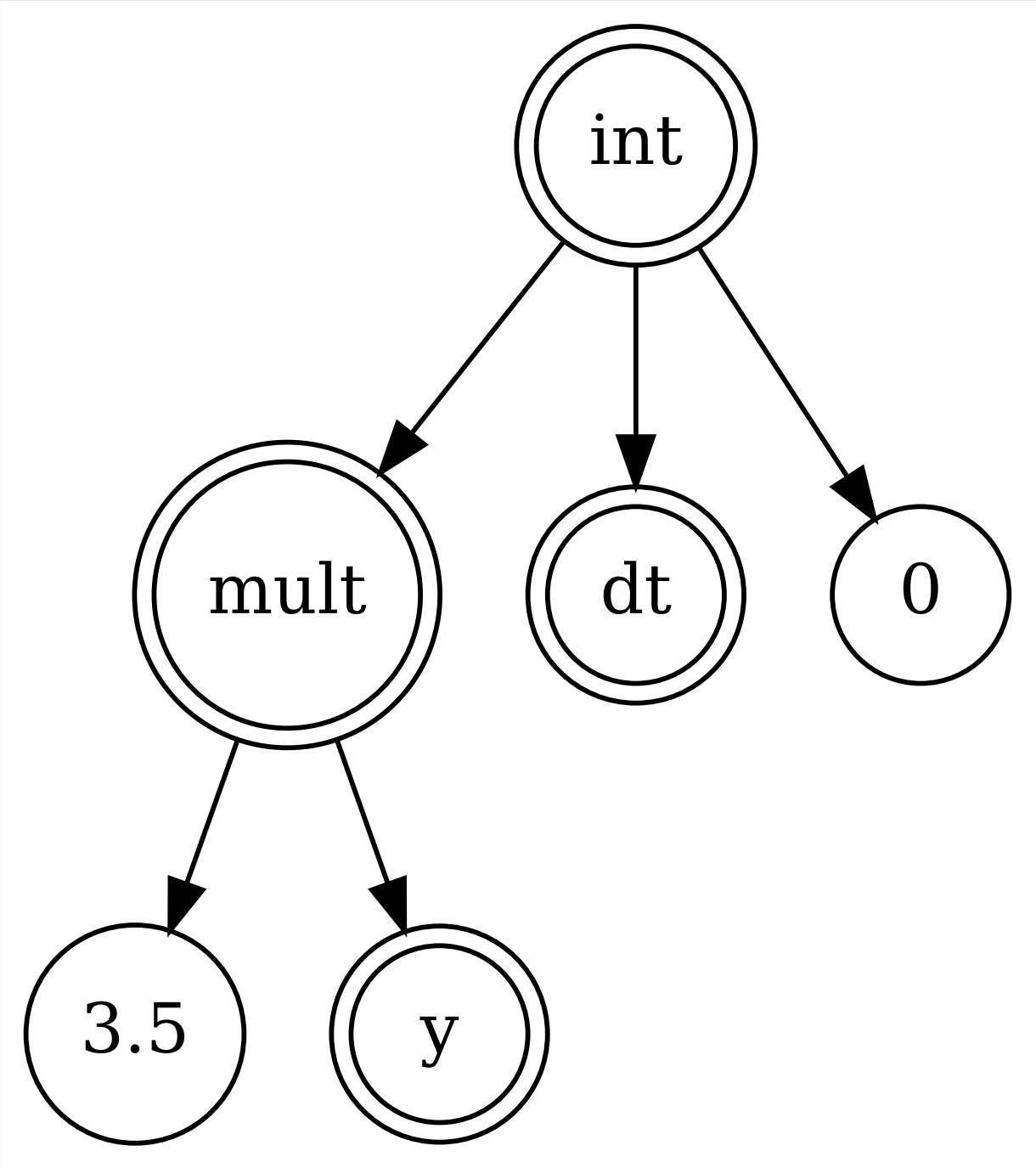
[13]:
```
print("The evolved variables are:", vars.evolved)
print("Auxilliary variables are:", vars.aux.all)
```

```
The evolved variables are: ['int_1', 'int_2', 'z']
Auxilliary variables are: ['f1', 'f2', 'f3', 'fx', 'mult_10', 'mult_6',
↪'mult_9', 'sum_1', 'sum_2', 'sum_3', 'sum_4', 'sum_5', 'sum_6', 'sum_7',
↪'sum_8', 'x', 'x1', 'x2', 'y', 'y1', 'y2']
```

One sees a number of new variables. They were introduced by *naming* all *intermediate* expressions. What are these intermediates? Let's review again the variable z:

[14]:
```
state["z"].draw_graph()
```

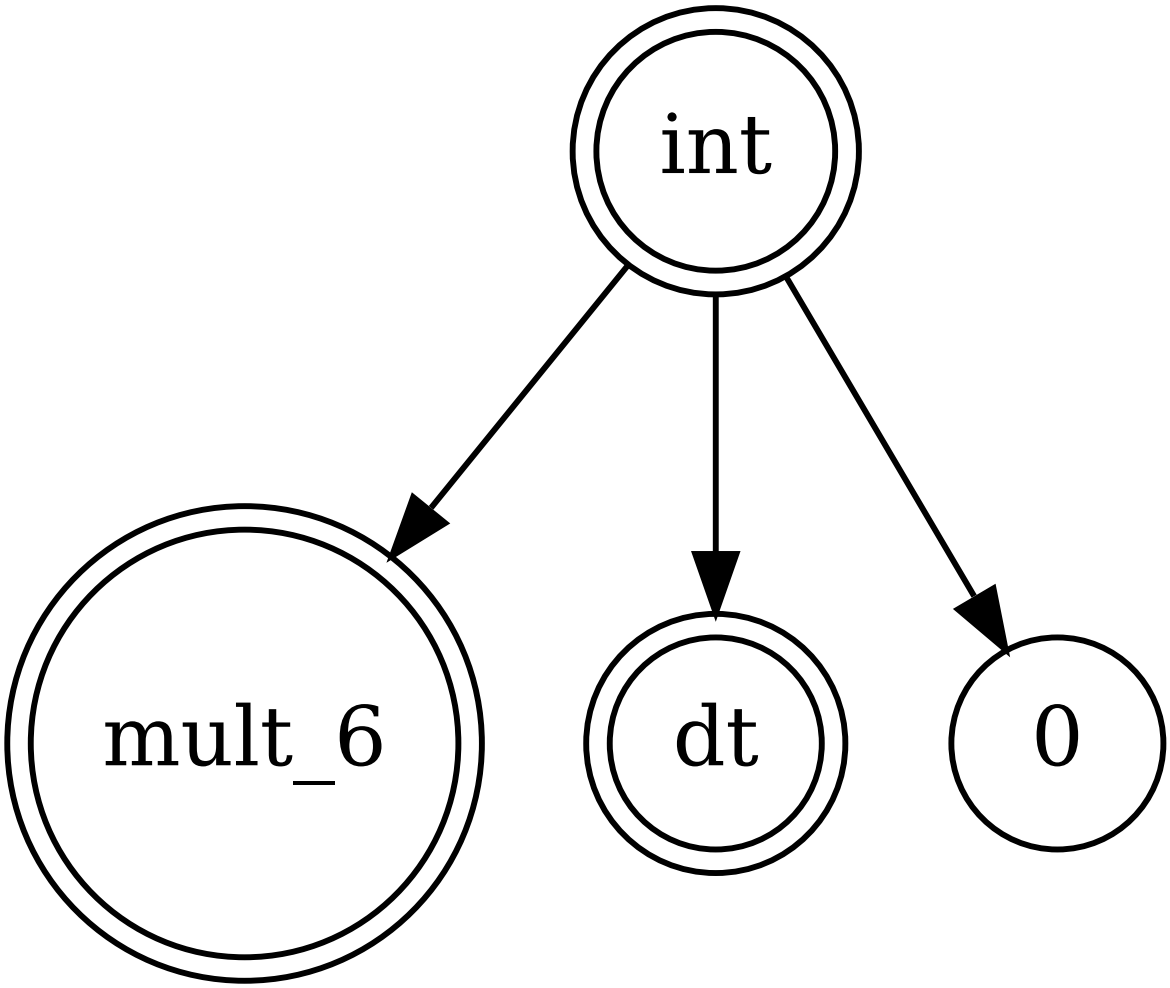
[14]:



Here, the left most child is an intermediate expression, since it computes `mult(3.5, y)`. We can give this intermediate result a concrete name and replace the whole subtree with this name:

[15]:
```
state.name_computing_elements()["z"].draw_graph()
```

[15]:



Simulating a circuit

In the following, we use the C++ code generator to simulate this simple ordinary differential equation:

[29]:
```
cpp_code = state.export(to="C")
print(cpp_code[:1000])
print("// ... (in total ", cpp_code.count("\n"), " lines of C/C++ code) ...
↪")
```

```
// This code was generated by PyDDA.

#include <cmath> /* don't forget -lm for linking */
#include <cfenv> /* for feraisexcept and friends */
#include <limits> /* for signaling NAN */
#include <vector>
#include <string>
#include <sstream>
#include <algorithm>
#include <map>
#include <cstdio>
#include <iostream>
#include <fstream>
```

(continues on next page)

(continued from previous page)

```
bool debug;
constexpr double _nan_ = std::numeric_limits<double>::signaling_NaN();

namespace dda {

/* if you use an old C++ compiler, just remove the newer features */
#define A constexpr double            /* constexpr requires C++11 */
#define D template<typename... T> A   /* Variadic templates require C++17
↪*/

// Known limitations for div(int, double): If certain arguments appear as
↪integer
// in the code (i.e. 1 instead of 1.0), there is div(int,int) kicking in
↪from
// cstdlib. TODO: Should rename div to Div; following int->Int.

A neg(double a) { return -a; }
A div(double a, double b) { return a/b; }
D Int(T... a) { return -(a + ...); } // int() is res
// ... (in total  481  lines of C/C++ code) ...
```

We printed the generated C++ code, which is in fact just a string in python, in the cell above. Next come some shortcut functions which call the system C++ compiler and run the binary, all externally on the system shell. The return value is slurped in as CSV data with numpy, so we readily have it for plotting.

[30]:
```
from dda.cpp_exporter import compile, run
compile(cpp_code)
```

[31]:
```
# This shows the stdout of the binary generated by the above C++ code:
print(run(arguments={"max_iterations":10}, fields_to_export=list("xyz"),
↪return_ndarray=False))
```

```
Running: ./a.out --max_iterations=10 x y z
x       y       z
0.100223        0.0005  0
0.100448        0.00100062      -1.75e-06
0.100675        0.00150184      -5.25215e-06
0.100905        0.00200368      -1.05086e-05
0.101136        0.00250611      -1.75215e-05
0.10137 0.00300915      -2.62929e-05
0.101605        0.00351277      -3.68249e-05
0.101843        0.00401699      -4.91196e-05
0.102082        0.0045218       -6.3179e-05
0.102324        0.00502718      -7.90053e-05
```

[32]:
```
# We can slurp in the CSV data directly to a numpy recarray:
result = run(arguments={"max_iterations":1000, "modulo_write":10})
```

```
Running: ./a.out --max_iterations=1000 --modulo_write=10
```

[33]:
```
result["x"]
```

[33]:
```
array([0.100223, 0.102568, 0.105126, 0.107902, 0.110904, 0.114137,
       0.11761 , 0.121327, 0.125297, 0.129525, 0.134019, 0.138785,
```

(continues on next page)

(continued from previous page)

```
       0.143831, 0.149164, 0.154792, 0.16072 , 0.166958, 0.173512,
       0.180391, 0.187603, 0.195156, 0.203058, 0.211319, 0.219946,
       0.22895 , 0.23834 , 0.248125, 0.258317, 0.268924, 0.279959,
       0.291432, 0.303354, 0.315739, 0.328597, 0.341943, 0.355789,
       0.370149, 0.385039, 0.400471, 0.415958, 0.430975, 0.445545,
       0.459687, 0.473419, 0.486757, 0.499714, 0.512304, 0.524535,
       0.536418, 0.54796 , 0.559167, 0.570045, 0.580597, 0.590828,
       0.600738, 0.61033 , 0.619603, 0.628559, 0.637196, 0.645513,
       0.653509, 0.661182, 0.668528, 0.675547, 0.682235, 0.688589,
       0.694606, 0.700283, 0.705617, 0.710605, 0.715244, 0.719531,
       0.723464, 0.727041, 0.730258, 0.733115, 0.73561 , 0.737741,
       0.739508, 0.74091 , 0.741947, 0.742621, 0.742931, 0.742879,
       0.742467, 0.741697, 0.740573, 0.739097, 0.737275, 0.735109,
       0.732606, 0.729772, 0.726611, 0.723132, 0.719342, 0.715248,
       0.710859, 0.706184, 0.701233, 0.696016])
```

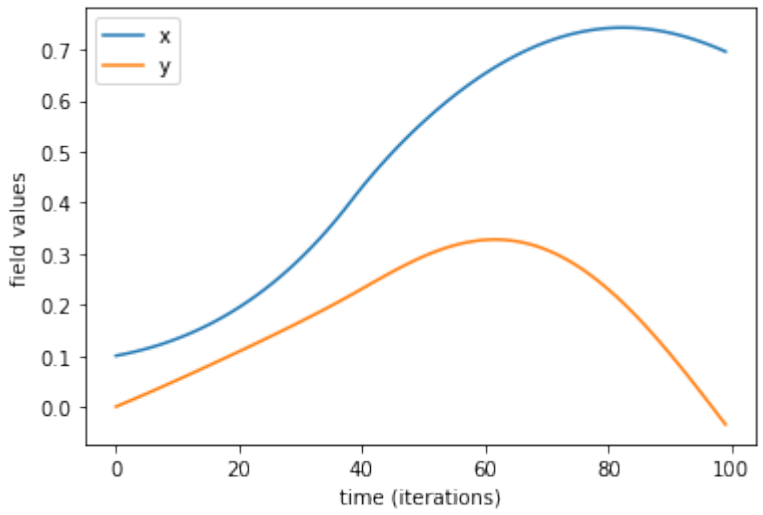
The above cell output shows a NumPy array, which is a Python-internal representation of the CSV file printed in the cell above. NumPy arrays are suitable for plotting, as we do next.

What is actually called by `cpp_exporter.run()` is np.genfromtxt()[69]. As our CSV file has a header row with column names, it creates a numpy recarray[70]. Therefore, we can address the column `x` by writing `result["x"]`.

[34]:
```
import matplotlib.pyplot as plt
```

[35]:
```
plt.plot(result["x"], label="x")
plt.plot(result["y"], label="y")
plt.xlabel("time (iterations)")
plt.ylabel("field values")
plt.legend()
```

[35]:
```
<matplotlib.legend.Legend at 0x7f42f5dd2820>
```



The above plot shows the time evolution of the quantity x and y. The plot is not very meaningful, but at least we see that the values are well within the analog computer bounds $[-1, 1]$.

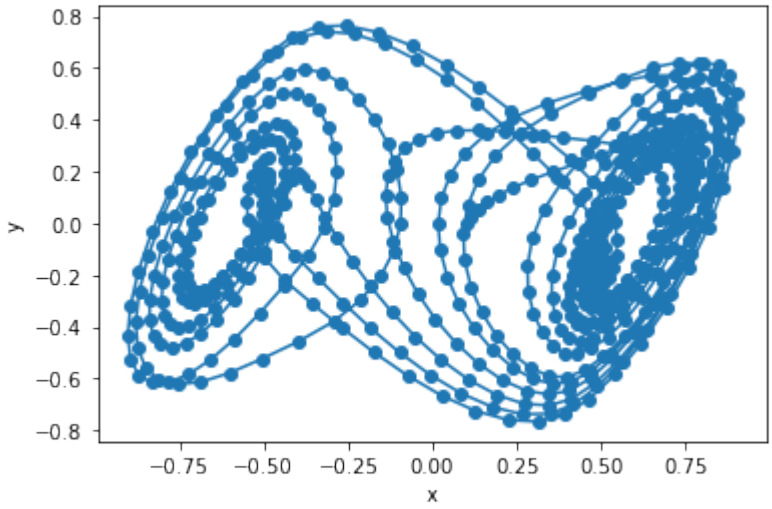
Let's run the simulation a bit longer and display a *phase space* plot of x and y:

[69] https://numpy.org/doc/stable/reference/generated/numpy.genfromtxt.html
[70] https://numpy.org/doc/stable/reference/generated/numpy.recarray.html

[78]:
```
result = run(arguments={"max_iterations":30000, "modulo_write":50})
plt.xlabel("x"); plt.ylabel("y")
plt.plot(result["x"], result["y"], "o-")
```

```
Running: ./a.out --max_iterations=30000 --modulo_write=50
```

[78]:
```
[<matplotlib.lines.Line2D at 0x7f12dd23d580>]
```



That's it for the moment. If you want to see even more advanced plotting, inspect the `run-chua.py` file in the directory of this notebook file (i.a. in the `experiments/` directory).

Heat-equation in 1D and 2D with DDA

The heat equation[71] is a partial differential equation (PDE). Being an elliptical equation with a Laplace operator, so second order in space, first order in time, it is one of the typical text-book introductions. It describes the fate of a given field $u = u(\vec{x}, t)$, and is given by

$$\frac{\partial u}{\partial t} = \alpha \nabla^2 u$$

where ∇^2 is the Laplace operator $\nabla^2 = \sum_{i=1}^{d} \partial^2 / \partial^2 x_i$ in d spatial dimensions and α is a constant for driving the coupling/gain.

The aim of this notebook is to demonstrate how to solve the heat-equation in one and two spatial dimensions with DDA or an analog computer, respectively. To do so, we use the method of lines[72] to convert the PDE in a set of ordinary differential equations (ODEs), which itself are spatially discretized with central finite differences[73].

This example follows the Analog Paradigm Application Notes 24[74], where a circuit and further comments on symmetries can be found.

For initial data, we will always model a singular peak $u_0(\vec{x}) = \delta(\vec{x}_0)$ at some position $\vec{x}_0$. Discretized, this translates to having all $u_i = 0$ for all i except one where $u_i = 1$.

[71] https://en.wikipedia.org/wiki/Heat_equation
[72] https://en.wikipedia.org/wiki/Method_of_lines
[73] https://en.wikipedia.org/wiki/Finite_difference
[74] http://analogparadigm.com/downloads/alpaca_24.pdf

[1]:
```
from numpy import *
from dda import *
from dda.computing_elements import *
from dda.cpp_exporter import compile, run
from matplotlib.pyplot import *
```

[2]:
```
alpha = const(4)
```

1D Heat equation

For the beginning, we solve the heat equation in one temporal and one spatial dimension.

First, we define the symbols u and $u0$ and the initial data:

[3]:
```
state = State() # start with a fresh state

N = 10 # supporting points
u = [ Symbol(f"u{i}") for i in range(N)  ]

# Initial conditions:
u0 = [ Symbol(f"ic_u{i}") for i in range(N) ]

for i in range(N):
    state[u0[i]] = const(0)

state[ u0[5] ] = const(1) # our candle at the boundary :-D

# Time step
dt = Symbol("dt")
state[dt] = const(0.01)

print("We have defined:")
print(f"{u = }")
print(f"{u0 = }")
print(f"{state = }")
```

```
We have defined:
u = [u0, u1, u2, u3, u4, u5, u6, u7, u8, u9]
u0 = [ic_u0, ic_u1, ic_u2, ic_u3, ic_u4, ic_u5, ic_u6, ic_u7, ic_u8, ic_u9]
state = State({'dt': const(0.01),
 'ic_u0': const(0),
 'ic_u1': const(0),
 'ic_u2': const(0),
 'ic_u3': const(0),
 'ic_u4': const(0),
 'ic_u5': const(1),
 'ic_u6': const(0),
 'ic_u7': const(0),
 'ic_u8': const(0),
 'ic_u9': const(0)})
```

In 1D, we approximate the Laplace operator by the central finite difference

$$\nabla^2 u = u_{i-1} + u_{i+1} - 2u_i$$

We implement periodic boundary conditions[75], so $u_N = u_0$.

[4]:
```
for i in range(N):
    # compute i-1 and i+1 with proper boundary conditions:
    im1 = i-1 if i>0 else N-1
    ip1 = i+1 if i!=N-1 else 0
    helper = Symbol(f"u_intermediate_{i}")
    state[helper] = neg(mult(2,u[i]))
    state[ u[i] ] = int(mult(alpha,sum(u[im1], u[ip1], helper)), dt, u0[i])
```

Let's inspect our state built so far:

[5]:
```
state
```

[5]:
```
State({'dt': const(0.01),
 'ic_u0': const(0),
 'ic_u1': const(0),
 'ic_u2': const(0),
 'ic_u3': const(0),
 'ic_u4': const(0),
 'ic_u5': const(1),
 'ic_u6': const(0),
 'ic_u7': const(0),
 'ic_u8': const(0),
 'ic_u9': const(0),
 'u0': int(mult(const(4), sum(u9, u1, u_intermediate_0)), dt, ic_u0),
 'u1': int(mult(const(4), sum(u0, u2, u_intermediate_1)), dt, ic_u1),
 'u2': int(mult(const(4), sum(u1, u3, u_intermediate_2)), dt, ic_u2),
 'u3': int(mult(const(4), sum(u2, u4, u_intermediate_3)), dt, ic_u3),
 'u4': int(mult(const(4), sum(u3, u5, u_intermediate_4)), dt, ic_u4),
 'u5': int(mult(const(4), sum(u4, u6, u_intermediate_5)), dt, ic_u5),
 'u6': int(mult(const(4), sum(u5, u7, u_intermediate_6)), dt, ic_u6),
 'u7': int(mult(const(4), sum(u6, u8, u_intermediate_7)), dt, ic_u7),
 'u8': int(mult(const(4), sum(u7, u9, u_intermediate_8)), dt, ic_u8),
 'u9': int(mult(const(4), sum(u8, u0, u_intermediate_9)), dt, ic_u9),
 'u_intermediate_0': neg(mult(2, u0)),
 'u_intermediate_1': neg(mult(2, u1)),
 'u_intermediate_2': neg(mult(2, u2)),
 'u_intermediate_3': neg(mult(2, u3)),
 'u_intermediate_4': neg(mult(2, u4)),
 'u_intermediate_5': neg(mult(2, u5)),
 'u_intermediate_6': neg(mult(2, u6)),
 'u_intermediate_7': neg(mult(2, u7)),
 'u_intermediate_8': neg(mult(2, u8)),
 'u_intermediate_9': neg(mult(2, u9))})
```

or as latex:

[28]:
```
from IPython.display import display, Markdown, Latex
display(Latex(state.export(to="latex")))
```

$$dt = 0.01 \tag{2.1}$$

$$ic_{u0} = 0 \tag{2.2}$$

$$ic_{u1} = 0 \tag{2.3}$$

(continues on next page)

[75] https://en.wikipedia.org/wiki/Periodic_boundary_conditions

(continued from previous page)

$$ic_{u2} = 0 \tag{2.4}$$

$$ic_{u3} = 0 \tag{2.5}$$

$$ic_{u4} = 0 \tag{2.6}$$

$$ic_{u5} = 1.0 \tag{2.7}$$

$$ic_{u6} = 0 \tag{2.8}$$

$$ic_{u7} = 0 \tag{2.9}$$

$$ic_{u8} = 0 \tag{2.10}$$

$$ic_{u9} = 0 \tag{2.11}$$

$$u_0 = -\int \left(dt + ic_{u0} - 4u_1 - 4u_9 - 4u_{intermediate0}\right)\, dt \tag{2.12}$$

$$u_1 = -\int \left(dt + ic_{u1} - 4u_0 - 4u_2 - 4u_{intermediate1}\right)\, dt \tag{2.13}$$

$$u_2 = -\int \left(dt + ic_{u2} - 4u_1 - 4u_3 - 4u_{intermediate2}\right)\, dt \tag{2.14}$$

$$u_3 = -\int \left(dt + ic_{u3} - 4u_2 - 4u_4 - 4u_{intermediate3}\right)\, dt \tag{2.15}$$

$$u_4 = -\int \left(dt + ic_{u4} - 4u_3 - 4u_5 - 4u_{intermediate4}\right)\, dt \tag{2.16}$$

$$u_5 = -\int \left(dt + ic_{u5} - 4u_4 - 4u_6 - 4u_{intermediate5}\right)\, dt \tag{2.17}$$

$$u_6 = -\int \left(dt + ic_{u6} - 4u_5 - 4u_7 - 4u_{intermediate6}\right)\, dt \tag{2.18}$$

$$u_7 = -\int \left(dt + ic_{u7} - 4u_6 - 4u_8 - 4u_{intermediate7}\right)\, dt \tag{2.19}$$

$$u_8 = -\int \left(dt + ic_{u8} - 4u_7 - 4u_9 - 4u_{intermediate8}\right)\, dt \tag{2.20}$$

$$u_9 = -\int \left(dt + ic_{u9} - 4u_0 - 4u_8 - 4u_{intermediate9}\right)\, dt \tag{2.21}$$

$$u_{intermediate0} = -2.0u_0 \tag{2.22}$$

$$u_{intermediate1} = -2.0u_1 \tag{2.23}$$

$$u_{intermediate2} = -2.0u_2 \tag{2.24}$$

$$u_{intermediate3} = -2.0u_3 \tag{2.25}$$

$$u_{intermediate4} = -2.0u_4 \tag{2.26}$$

$$u_{intermediate5} = -2.0u_5 \tag{2.27}$$

$$u_{intermediate6} = -2.0u_6 \tag{2.28}$$

$$u_{intermediate7} = -2.0u_7 \tag{2.29}$$

(continues on next page)

(continued from previous page)

$$u_{intermediate8} = -2.0u_8 \tag{2.30}$$

$$u_{intermediate9} = -2.0u_9 \tag{2.31}$$

[6]:
```
c_code = state.export(to="C")
compile(c_code, "heateq.cc")
```

If you want, you can print the generated c_code or inspect it with a text editor.

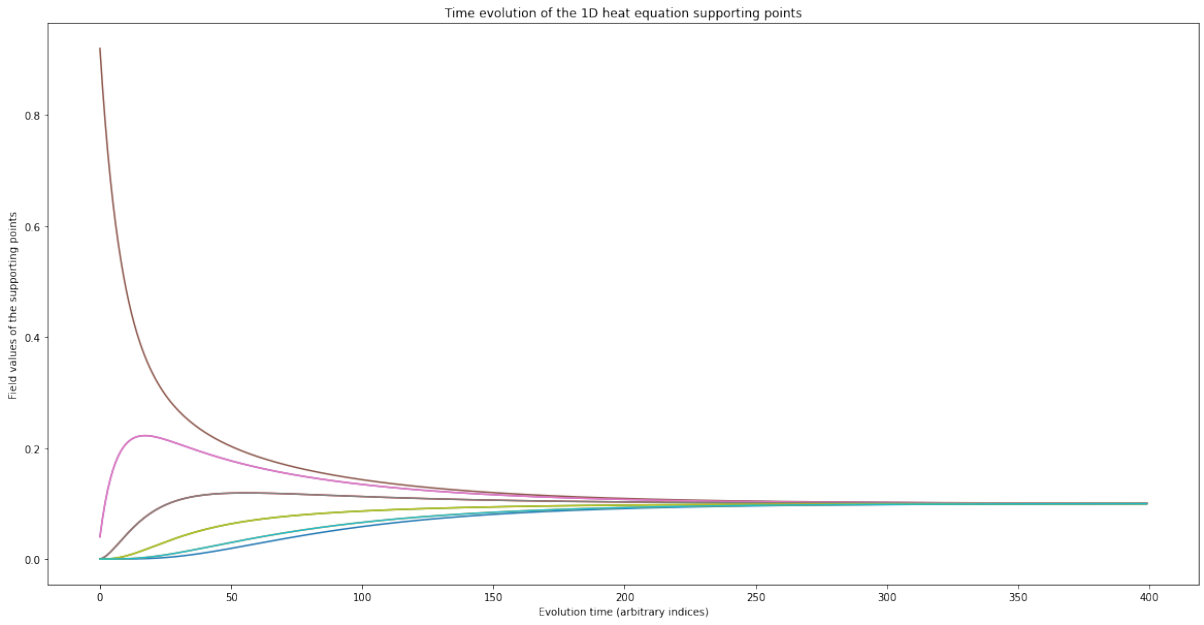
[7]:
```
data = run(arguments={'max_iterations':400})
```

```
Running: ./a.out --max_iterations=400
```

[12]:
```
rcParams["figure.figsize"] = (20,10)
for i in range(N):
    plot(data[f"u{i}"])

xlabel("Evolution time (arbitrary indices)")
ylabel("Field values of the supporting points")
title("Time evolution of the 1D heat equation supporting points")
```

[12]:
```
Text(0.5, 1.0, 'Time evolution of the 1D heat equation supporting points')
```

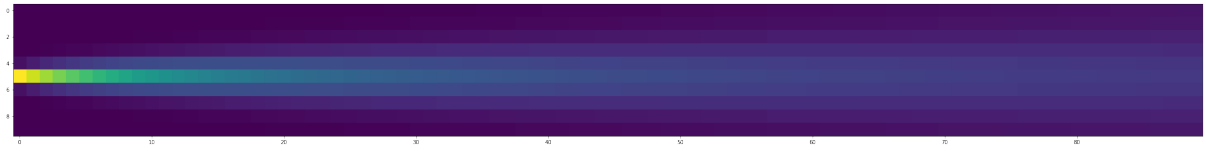


In the following, we reconstruct the higher-dimensional data and display it color encoded:

[17]:
```
heat = array([data[f"u{i}"] for i in range(N)])
```

[27]:
```
rcParams["figure.figsize"] = (40,20)
imshow(heat[:,0:90])
```

[27]:
```
<matplotlib.image.AxesImage at 0x7f3f68cbc7f0>
```



What you can see in the above plot is on the x-axis again the evolution time and on the y-axis the physical coordinate. Of course the whole image is pretty discretized. You see the initial data (one cell has value 1, all others value 0) and the equilibration of energy amongst all cells.

2D Heat Equation

We now solve the heat equation in two spatial and one temporal dimension, which is more similar to what is done in http://analogparadigm.com/downloads/alpaca_24.pdf

[30]:
```
state = State()  # restart with a clean state

N = 10 # supporting points
NN = (N, N) # shape in two dimensions

u = array([ Symbol("u_%d_%d"%(i,j)) for i,j in ndindex(NN) ]).reshape(NN)
u
```

[30]:
```
array([[u_0_0, u_0_1, u_0_2, u_0_3, u_0_4, u_0_5, u_0_6, u_0_7, u_0_8,
        u_0_9],
       [u_1_0, u_1_1, u_1_2, u_1_3, u_1_4, u_1_5, u_1_6, u_1_7, u_1_8,
        u_1_9],
       [u_2_0, u_2_1, u_2_2, u_2_3, u_2_4, u_2_5, u_2_6, u_2_7, u_2_8,
        u_2_9],
       [u_3_0, u_3_1, u_3_2, u_3_3, u_3_4, u_3_5, u_3_6, u_3_7, u_3_8,
        u_3_9],
       [u_4_0, u_4_1, u_4_2, u_4_3, u_4_4, u_4_5, u_4_6, u_4_7, u_4_8,
        u_4_9],
       [u_5_0, u_5_1, u_5_2, u_5_3, u_5_4, u_5_5, u_5_6, u_5_7, u_5_8,
        u_5_9],
       [u_6_0, u_6_1, u_6_2, u_6_3, u_6_4, u_6_5, u_6_6, u_6_7, u_6_8,
        u_6_9],
       [u_7_0, u_7_1, u_7_2, u_7_3, u_7_4, u_7_5, u_7_6, u_7_7, u_7_8,
        u_7_9],
       [u_8_0, u_8_1, u_8_2, u_8_3, u_8_4, u_8_5, u_8_6, u_8_7, u_8_8,
        u_8_9],
       [u_9_0, u_9_1, u_9_2, u_9_3, u_9_4, u_9_5, u_9_6, u_9_7, u_9_8,
        u_9_9]], dtype=object)
```

[31]:
```
# Symbols for initial conditions:
u0 = array([ Symbol("uinitial_%d_%d"%(i,j)) for i,j in ndindex(NN) ]).
↪reshape(NN)
u0
```

[31]:
```
array([[uinitial_0_0, uinitial_0_1, uinitial_0_2, uinitial_0_3,
        uinitial_0_4, uinitial_0_5, uinitial_0_6, uinitial_0_7,
        uinitial_0_8, uinitial_0_9],
       [uinitial_1_0, uinitial_1_1, uinitial_1_2, uinitial_1_3,
        uinitial_1_4, uinitial_1_5, uinitial_1_6, uinitial_1_7,
        uinitial_1_8, uinitial_1_9],
       [uinitial_2_0, uinitial_2_1, uinitial_2_2, uinitial_2_3,
        uinitial_2_4, uinitial_2_5, uinitial_2_6, uinitial_2_7,
        uinitial_2_8, uinitial_2_9],
       [uinitial_3_0, uinitial_3_1, uinitial_3_2, uinitial_3_3,
        uinitial_3_4, uinitial_3_5, uinitial_3_6, uinitial_3_7,
        uinitial_3_8, uinitial_3_9],
       [uinitial_4_0, uinitial_4_1, uinitial_4_2, uinitial_4_3,
```

(continues on next page)

(continued from previous page)

```
        uinitial_4_4, uinitial_4_5, uinitial_4_6, uinitial_4_7,
        uinitial_4_8, uinitial_4_9],
       [uinitial_5_0, uinitial_5_1, uinitial_5_2, uinitial_5_3,
        uinitial_5_4, uinitial_5_5, uinitial_5_6, uinitial_5_7,
        uinitial_5_8, uinitial_5_9],
       [uinitial_6_0, uinitial_6_1, uinitial_6_2, uinitial_6_3,
        uinitial_6_4, uinitial_6_5, uinitial_6_6, uinitial_6_7,
        uinitial_6_8, uinitial_6_9],
       [uinitial_7_0, uinitial_7_1, uinitial_7_2, uinitial_7_3,
        uinitial_7_4, uinitial_7_5, uinitial_7_6, uinitial_7_7,
        uinitial_7_8, uinitial_7_9],
       [uinitial_8_0, uinitial_8_1, uinitial_8_2, uinitial_8_3,
        uinitial_8_4, uinitial_8_5, uinitial_8_6, uinitial_8_7,
        uinitial_8_8, uinitial_8_9],
       [uinitial_9_0, uinitial_9_1, uinitial_9_2, uinitial_9_3,
        uinitial_9_4, uinitial_9_5, uinitial_9_6, uinitial_9_7,
        uinitial_9_8, uinitial_9_9]], dtype=object)
```

[32]:
```
# imprint the initial conditions:

for i,j in ndindex(NN):
    state[u0[i,j]] = const(0)

state[ u0[5,5] ] = const(1) # our candle in the very center

# Time step
dt = Symbol("dt")
state[dt] = const(0.01)

# Let's see what we made
state
```

[32]:
```
State({'dt': const(0.01),
 'uinitial_0_0': const(0),
 'uinitial_0_1': const(0),
 'uinitial_0_2': const(0),
 'uinitial_0_3': const(0),
 'uinitial_0_4': const(0),
 'uinitial_0_5': const(0),
 'uinitial_0_6': const(0),
 'uinitial_0_7': const(0),
 'uinitial_0_8': const(0),
 'uinitial_0_9': const(0),
 'uinitial_1_0': const(0),
 'uinitial_1_1': const(0),
 'uinitial_1_2': const(0),
 'uinitial_1_3': const(0),
 'uinitial_1_4': const(0),
 'uinitial_1_5': const(0),
 'uinitial_1_6': const(0),
 'uinitial_1_7': const(0),
 'uinitial_1_8': const(0),
 'uinitial_1_9': const(0),
 'uinitial_2_0': const(0),
 'uinitial_2_1': const(0),
 'uinitial_2_2': const(0),
```

(continues on next page)

(continued from previous page)

```
 'uinitial_2_3': const(0),
 'uinitial_2_4': const(0),
 'uinitial_2_5': const(0),
 'uinitial_2_6': const(0),
 'uinitial_2_7': const(0),
 'uinitial_2_8': const(0),
 'uinitial_2_9': const(0),
 'uinitial_3_0': const(0),
 'uinitial_3_1': const(0),
 'uinitial_3_2': const(0),
 'uinitial_3_3': const(0),
 'uinitial_3_4': const(0),
 'uinitial_3_5': const(0),
 'uinitial_3_6': const(0),
 'uinitial_3_7': const(0),
 'uinitial_3_8': const(0),
 'uinitial_3_9': const(0),
 'uinitial_4_0': const(0),
 'uinitial_4_1': const(0),
 'uinitial_4_2': const(0),
 'uinitial_4_3': const(0),
 'uinitial_4_4': const(0),
 'uinitial_4_5': const(0),
 'uinitial_4_6': const(0),
 'uinitial_4_7': const(0),
 'uinitial_4_8': const(0),
 'uinitial_4_9': const(0),
 'uinitial_5_0': const(0),
 'uinitial_5_1': const(0),
 'uinitial_5_2': const(0),
 'uinitial_5_3': const(0),
 'uinitial_5_4': const(0),
 'uinitial_5_5': const(1),
 'uinitial_5_6': const(0),
 'uinitial_5_7': const(0),
 'uinitial_5_8': const(0),
 'uinitial_5_9': const(0),
 'uinitial_6_0': const(0),
 'uinitial_6_1': const(0),
 'uinitial_6_2': const(0),
 'uinitial_6_3': const(0),
 'uinitial_6_4': const(0),
 'uinitial_6_5': const(0),
 'uinitial_6_6': const(0),
 'uinitial_6_7': const(0),
 'uinitial_6_8': const(0),
 'uinitial_6_9': const(0),
 'uinitial_7_0': const(0),
 'uinitial_7_1': const(0),
 'uinitial_7_2': const(0),
 'uinitial_7_3': const(0),
 'uinitial_7_4': const(0),
 'uinitial_7_5': const(0),
 'uinitial_7_6': const(0),
 'uinitial_7_7': const(0),
 'uinitial_7_8': const(0),
```

(continues on next page)

(continued from previous page)

```
 'uinitial_7_9': const(0),
 'uinitial_8_0': const(0),
 'uinitial_8_1': const(0),
 'uinitial_8_2': const(0),
 'uinitial_8_3': const(0),
 'uinitial_8_4': const(0),
 'uinitial_8_5': const(0),
 'uinitial_8_6': const(0),
 'uinitial_8_7': const(0),
 'uinitial_8_8': const(0),
 'uinitial_8_9': const(0),
 'uinitial_9_0': const(0),
 'uinitial_9_1': const(0),
 'uinitial_9_2': const(0),
 'uinitial_9_3': const(0),
 'uinitial_9_4': const(0),
 'uinitial_9_5': const(0),
 'uinitial_9_6': const(0),
 'uinitial_9_7': const(0),
 'uinitial_9_8': const(0),
 'uinitial_9_9': const(0)})
```

The Laplace operator is discretized as the following:

$$\dot{u}_{i,j} = \alpha(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} - 4u_{i,j})$$

[34]:
```
for i,j in ndindex(NN):
    # compute i-1 and i+1 with proper boundary conditions:
    im1 = i-1 if i>0 else N-1
    ip1 = i+1 if i!=N-1 else 0

    # compute j-1 and j+1 with proper boundary conditions:
    jm1 = j-1 if j>0 else N-1
    jp1 = j+1 if j!=N-1 else 0

    helper = Symbol(f"u_intermediate_{i}_{j}")
    state[helper] = neg(mult(4,u[i,j]))
    state[ u[i,j] ] = int(mult(alpha,sum(
        u[im1,j], u[ip1,j],
        u[i,jm1], u[i,jp1],
        helper)), dt, u0[i,j])

len(state)
```

[34]:
```
301
```

The state is already enormous. Realizing this circuit requires more then 300 wires or computing elements, respectively:

[35]:
```
state
```

[35]:
```
State({'dt': const(0.01),
 'u_0_0': int(mult(const(4), sum(u_9_0, u_1_0, u_0_9, u_0_1, u_
↪intermediate_0_0)), dt, uinitial_0_0),
 'u_0_1': int(mult(const(4), sum(u_9_1, u_1_1, u_0_0, u_0_2, u_
↪intermediate_0_1)), dt, uinitial_0_1),
```

(continues on next page)

(continued from previous page)

```
 'u_0_2': int(mult(const(4), sum(u_9_2, u_1_2, u_0_1, u_0_3, u_
↪intermediate_0_2)), dt, uinitial_0_2),
 'u_0_3': int(mult(const(4), sum(u_9_3, u_1_3, u_0_2, u_0_4, u_
↪intermediate_0_3)), dt, uinitial_0_3),
 'u_0_4': int(mult(const(4), sum(u_9_4, u_1_4, u_0_3, u_0_5, u_
↪intermediate_0_4)), dt, uinitial_0_4),
 'u_0_5': int(mult(const(4), sum(u_9_5, u_1_5, u_0_4, u_0_6, u_
↪intermediate_0_5)), dt, uinitial_0_5),
 'u_0_6': int(mult(const(4), sum(u_9_6, u_1_6, u_0_5, u_0_7, u_
↪intermediate_0_6)), dt, uinitial_0_6),
 'u_0_7': int(mult(const(4), sum(u_9_7, u_1_7, u_0_6, u_0_8, u_
↪intermediate_0_7)), dt, uinitial_0_7),
 'u_0_8': int(mult(const(4), sum(u_9_8, u_1_8, u_0_7, u_0_9, u_
↪intermediate_0_8)), dt, uinitial_0_8),
 'u_0_9': int(mult(const(4), sum(u_9_9, u_1_9, u_0_8, u_0_0, u_
↪intermediate_0_9)), dt, uinitial_0_9),
 'u_1_0': int(mult(const(4), sum(u_0_0, u_2_0, u_1_9, u_1_1, u_
↪intermediate_1_0)), dt, uinitial_1_0),
 'u_1_1': int(mult(const(4), sum(u_0_1, u_2_1, u_1_0, u_1_2, u_
↪intermediate_1_1)), dt, uinitial_1_1),
 'u_1_2': int(mult(const(4), sum(u_0_2, u_2_2, u_1_1, u_1_3, u_
↪intermediate_1_2)), dt, uinitial_1_2),
 'u_1_3': int(mult(const(4), sum(u_0_3, u_2_3, u_1_2, u_1_4, u_
↪intermediate_1_3)), dt, uinitial_1_3),
 'u_1_4': int(mult(const(4), sum(u_0_4, u_2_4, u_1_3, u_1_5, u_
↪intermediate_1_4)), dt, uinitial_1_4),
 'u_1_5': int(mult(const(4), sum(u_0_5, u_2_5, u_1_4, u_1_6, u_
↪intermediate_1_5)), dt, uinitial_1_5),
 'u_1_6': int(mult(const(4), sum(u_0_6, u_2_6, u_1_5, u_1_7, u_
↪intermediate_1_6)), dt, uinitial_1_6),
 'u_1_7': int(mult(const(4), sum(u_0_7, u_2_7, u_1_6, u_1_8, u_
↪intermediate_1_7)), dt, uinitial_1_7),
 'u_1_8': int(mult(const(4), sum(u_0_8, u_2_8, u_1_7, u_1_9, u_
↪intermediate_1_8)), dt, uinitial_1_8),
 'u_1_9': int(mult(const(4), sum(u_0_9, u_2_9, u_1_8, u_1_0, u_
↪intermediate_1_9)), dt, uinitial_1_9),
 'u_2_0': int(mult(const(4), sum(u_1_0, u_3_0, u_2_9, u_2_1, u_
↪intermediate_2_0)), dt, uinitial_2_0),
 'u_2_1': int(mult(const(4), sum(u_1_1, u_3_1, u_2_0, u_2_2, u_
↪intermediate_2_1)), dt, uinitial_2_1),
 'u_2_2': int(mult(const(4), sum(u_1_2, u_3_2, u_2_1, u_2_3, u_
↪intermediate_2_2)), dt, uinitial_2_2),
 'u_2_3': int(mult(const(4), sum(u_1_3, u_3_3, u_2_2, u_2_4, u_
↪intermediate_2_3)), dt, uinitial_2_3),
 'u_2_4': int(mult(const(4), sum(u_1_4, u_3_4, u_2_3, u_2_5, u_
↪intermediate_2_4)), dt, uinitial_2_4),
 'u_2_5': int(mult(const(4), sum(u_1_5, u_3_5, u_2_4, u_2_6, u_
↪intermediate_2_5)), dt, uinitial_2_5),
 'u_2_6': int(mult(const(4), sum(u_1_6, u_3_6, u_2_5, u_2_7, u_
↪intermediate_2_6)), dt, uinitial_2_6),
 'u_2_7': int(mult(const(4), sum(u_1_7, u_3_7, u_2_6, u_2_8, u_
↪intermediate_2_7)), dt, uinitial_2_7),
 'u_2_8': int(mult(const(4), sum(u_1_8, u_3_8, u_2_7, u_2_9, u_
↪intermediate_2_8)), dt, uinitial_2_8),
 'u_2_9': int(mult(const(4), sum(u_1_9, u_3_9, u_2_8, u_2_0, u_
↪intermediate_2_9)), dt, uinitial_2_9),
```

(continues on next page)

(continued from previous page)

```
 'u_3_0': int(mult(const(4), sum(u_2_0, u_4_0, u_3_9, u_3_1, u_
↪intermediate_3_0)), dt, uinitial_3_0),
 'u_3_1': int(mult(const(4), sum(u_2_1, u_4_1, u_3_0, u_3_2, u_
↪intermediate_3_1)), dt, uinitial_3_1),
 'u_3_2': int(mult(const(4), sum(u_2_2, u_4_2, u_3_1, u_3_3, u_
↪intermediate_3_2)), dt, uinitial_3_2),
 'u_3_3': int(mult(const(4), sum(u_2_3, u_4_3, u_3_2, u_3_4, u_
↪intermediate_3_3)), dt, uinitial_3_3),
 'u_3_4': int(mult(const(4), sum(u_2_4, u_4_4, u_3_3, u_3_5, u_
↪intermediate_3_4)), dt, uinitial_3_4),
 'u_3_5': int(mult(const(4), sum(u_2_5, u_4_5, u_3_4, u_3_6, u_
↪intermediate_3_5)), dt, uinitial_3_5),
 'u_3_6': int(mult(const(4), sum(u_2_6, u_4_6, u_3_5, u_3_7, u_
↪intermediate_3_6)), dt, uinitial_3_6),
 'u_3_7': int(mult(const(4), sum(u_2_7, u_4_7, u_3_6, u_3_8, u_
↪intermediate_3_7)), dt, uinitial_3_7),
 'u_3_8': int(mult(const(4), sum(u_2_8, u_4_8, u_3_7, u_3_9, u_
↪intermediate_3_8)), dt, uinitial_3_8),
 'u_3_9': int(mult(const(4), sum(u_2_9, u_4_9, u_3_8, u_3_0, u_
↪intermediate_3_9)), dt, uinitial_3_9),
 'u_4_0': int(mult(const(4), sum(u_3_0, u_5_0, u_4_9, u_4_1, u_
↪intermediate_4_0)), dt, uinitial_4_0),
 'u_4_1': int(mult(const(4), sum(u_3_1, u_5_1, u_4_0, u_4_2, u_
↪intermediate_4_1)), dt, uinitial_4_1),
 'u_4_2': int(mult(const(4), sum(u_3_2, u_5_2, u_4_1, u_4_3, u_
↪intermediate_4_2)), dt, uinitial_4_2),
 'u_4_3': int(mult(const(4), sum(u_3_3, u_5_3, u_4_2, u_4_4, u_
↪intermediate_4_3)), dt, uinitial_4_3),
 'u_4_4': int(mult(const(4), sum(u_3_4, u_5_4, u_4_3, u_4_5, u_
↪intermediate_4_4)), dt, uinitial_4_4),
 'u_4_5': int(mult(const(4), sum(u_3_5, u_5_5, u_4_4, u_4_6, u_
↪intermediate_4_5)), dt, uinitial_4_5),
 'u_4_6': int(mult(const(4), sum(u_3_6, u_5_6, u_4_5, u_4_7, u_
↪intermediate_4_6)), dt, uinitial_4_6),
 'u_4_7': int(mult(const(4), sum(u_3_7, u_5_7, u_4_6, u_4_8, u_
↪intermediate_4_7)), dt, uinitial_4_7),
 'u_4_8': int(mult(const(4), sum(u_3_8, u_5_8, u_4_7, u_4_9, u_
↪intermediate_4_8)), dt, uinitial_4_8),
 'u_4_9': int(mult(const(4), sum(u_3_9, u_5_9, u_4_8, u_4_0, u_
↪intermediate_4_9)), dt, uinitial_4_9),
 'u_5_0': int(mult(const(4), sum(u_4_0, u_6_0, u_5_9, u_5_1, u_
↪intermediate_5_0)), dt, uinitial_5_0),
 'u_5_1': int(mult(const(4), sum(u_4_1, u_6_1, u_5_0, u_5_2, u_
↪intermediate_5_1)), dt, uinitial_5_1),
 'u_5_2': int(mult(const(4), sum(u_4_2, u_6_2, u_5_1, u_5_3, u_
↪intermediate_5_2)), dt, uinitial_5_2),
 'u_5_3': int(mult(const(4), sum(u_4_3, u_6_3, u_5_2, u_5_4, u_
↪intermediate_5_3)), dt, uinitial_5_3),
 'u_5_4': int(mult(const(4), sum(u_4_4, u_6_4, u_5_3, u_5_5, u_
↪intermediate_5_4)), dt, uinitial_5_4),
 'u_5_5': int(mult(const(4), sum(u_4_5, u_6_5, u_5_4, u_5_6, u_
↪intermediate_5_5)), dt, uinitial_5_5),
 'u_5_6': int(mult(const(4), sum(u_4_6, u_6_6, u_5_5, u_5_7, u_
↪intermediate_5_6)), dt, uinitial_5_6),
 'u_5_7': int(mult(const(4), sum(u_4_7, u_6_7, u_5_6, u_5_8, u_
↪intermediate_5_7)), dt, uinitial_5_7),
```

(continues on next page)

(continued from previous page)

```
 'u_5_8': int(mult(const(4), sum(u_4_8, u_6_8, u_5_7, u_5_9, u_
↪intermediate_5_8)), dt, uinitial_5_8),
 'u_5_9': int(mult(const(4), sum(u_4_9, u_6_9, u_5_8, u_5_0, u_
↪intermediate_5_9)), dt, uinitial_5_9),
 'u_6_0': int(mult(const(4), sum(u_5_0, u_7_0, u_6_9, u_6_1, u_
↪intermediate_6_0)), dt, uinitial_6_0),
 'u_6_1': int(mult(const(4), sum(u_5_1, u_7_1, u_6_0, u_6_2, u_
↪intermediate_6_1)), dt, uinitial_6_1),
 'u_6_2': int(mult(const(4), sum(u_5_2, u_7_2, u_6_1, u_6_3, u_
↪intermediate_6_2)), dt, uinitial_6_2),
 'u_6_3': int(mult(const(4), sum(u_5_3, u_7_3, u_6_2, u_6_4, u_
↪intermediate_6_3)), dt, uinitial_6_3),
 'u_6_4': int(mult(const(4), sum(u_5_4, u_7_4, u_6_3, u_6_5, u_
↪intermediate_6_4)), dt, uinitial_6_4),
 'u_6_5': int(mult(const(4), sum(u_5_5, u_7_5, u_6_4, u_6_6, u_
↪intermediate_6_5)), dt, uinitial_6_5),
 'u_6_6': int(mult(const(4), sum(u_5_6, u_7_6, u_6_5, u_6_7, u_
↪intermediate_6_6)), dt, uinitial_6_6),
 'u_6_7': int(mult(const(4), sum(u_5_7, u_7_7, u_6_6, u_6_8, u_
↪intermediate_6_7)), dt, uinitial_6_7),
 'u_6_8': int(mult(const(4), sum(u_5_8, u_7_8, u_6_7, u_6_9, u_
↪intermediate_6_8)), dt, uinitial_6_8),
 'u_6_9': int(mult(const(4), sum(u_5_9, u_7_9, u_6_8, u_6_0, u_
↪intermediate_6_9)), dt, uinitial_6_9),
 'u_7_0': int(mult(const(4), sum(u_6_0, u_8_0, u_7_9, u_7_1, u_
↪intermediate_7_0)), dt, uinitial_7_0),
 'u_7_1': int(mult(const(4), sum(u_6_1, u_8_1, u_7_0, u_7_2, u_
↪intermediate_7_1)), dt, uinitial_7_1),
 'u_7_2': int(mult(const(4), sum(u_6_2, u_8_2, u_7_1, u_7_3, u_
↪intermediate_7_2)), dt, uinitial_7_2),
 'u_7_3': int(mult(const(4), sum(u_6_3, u_8_3, u_7_2, u_7_4, u_
↪intermediate_7_3)), dt, uinitial_7_3),
 'u_7_4': int(mult(const(4), sum(u_6_4, u_8_4, u_7_3, u_7_5, u_
↪intermediate_7_4)), dt, uinitial_7_4),
 'u_7_5': int(mult(const(4), sum(u_6_5, u_8_5, u_7_4, u_7_6, u_
↪intermediate_7_5)), dt, uinitial_7_5),
 'u_7_6': int(mult(const(4), sum(u_6_6, u_8_6, u_7_5, u_7_7, u_
↪intermediate_7_6)), dt, uinitial_7_6),
 'u_7_7': int(mult(const(4), sum(u_6_7, u_8_7, u_7_6, u_7_8, u_
↪intermediate_7_7)), dt, uinitial_7_7),
 'u_7_8': int(mult(const(4), sum(u_6_8, u_8_8, u_7_7, u_7_9, u_
↪intermediate_7_8)), dt, uinitial_7_8),
 'u_7_9': int(mult(const(4), sum(u_6_9, u_8_9, u_7_8, u_7_0, u_
↪intermediate_7_9)), dt, uinitial_7_9),
 'u_8_0': int(mult(const(4), sum(u_7_0, u_9_0, u_8_9, u_8_1, u_
↪intermediate_8_0)), dt, uinitial_8_0),
 'u_8_1': int(mult(const(4), sum(u_7_1, u_9_1, u_8_0, u_8_2, u_
↪intermediate_8_1)), dt, uinitial_8_1),
 'u_8_2': int(mult(const(4), sum(u_7_2, u_9_2, u_8_1, u_8_3, u_
↪intermediate_8_2)), dt, uinitial_8_2),
 'u_8_3': int(mult(const(4), sum(u_7_3, u_9_3, u_8_2, u_8_4, u_
↪intermediate_8_3)), dt, uinitial_8_3),
 'u_8_4': int(mult(const(4), sum(u_7_4, u_9_4, u_8_3, u_8_5, u_
↪intermediate_8_4)), dt, uinitial_8_4),
 'u_8_5': int(mult(const(4), sum(u_7_5, u_9_5, u_8_4, u_8_6, u_
↪intermediate_8_5)), dt, uinitial_8_5),
```

(continues on next page)

(continued from previous page)

```
 'u_8_6': int(mult(const(4), sum(u_7_6, u_9_6, u_8_5, u_8_7, u_
↪intermediate_8_6)), dt, uinitial_8_6),
 'u_8_7': int(mult(const(4), sum(u_7_7, u_9_7, u_8_6, u_8_8, u_
↪intermediate_8_7)), dt, uinitial_8_7),
 'u_8_8': int(mult(const(4), sum(u_7_8, u_9_8, u_8_7, u_8_9, u_
↪intermediate_8_8)), dt, uinitial_8_8),
 'u_8_9': int(mult(const(4), sum(u_7_9, u_9_9, u_8_8, u_8_0, u_
↪intermediate_8_9)), dt, uinitial_8_9),
 'u_9_0': int(mult(const(4), sum(u_8_0, u_0_0, u_9_9, u_9_1, u_
↪intermediate_9_0)), dt, uinitial_9_0),
 'u_9_1': int(mult(const(4), sum(u_8_1, u_0_1, u_9_0, u_9_2, u_
↪intermediate_9_1)), dt, uinitial_9_1),
 'u_9_2': int(mult(const(4), sum(u_8_2, u_0_2, u_9_1, u_9_3, u_
↪intermediate_9_2)), dt, uinitial_9_2),
 'u_9_3': int(mult(const(4), sum(u_8_3, u_0_3, u_9_2, u_9_4, u_
↪intermediate_9_3)), dt, uinitial_9_3),
 'u_9_4': int(mult(const(4), sum(u_8_4, u_0_4, u_9_3, u_9_5, u_
↪intermediate_9_4)), dt, uinitial_9_4),
 'u_9_5': int(mult(const(4), sum(u_8_5, u_0_5, u_9_4, u_9_6, u_
↪intermediate_9_5)), dt, uinitial_9_5),
 'u_9_6': int(mult(const(4), sum(u_8_6, u_0_6, u_9_5, u_9_7, u_
↪intermediate_9_6)), dt, uinitial_9_6),
 'u_9_7': int(mult(const(4), sum(u_8_7, u_0_7, u_9_6, u_9_8, u_
↪intermediate_9_7)), dt, uinitial_9_7),
 'u_9_8': int(mult(const(4), sum(u_8_8, u_0_8, u_9_7, u_9_9, u_
↪intermediate_9_8)), dt, uinitial_9_8),
 'u_9_9': int(mult(const(4), sum(u_8_9, u_0_9, u_9_8, u_9_0, u_
↪intermediate_9_9)), dt, uinitial_9_9),
 'u_intermediate_0_0': neg(mult(4, u_0_0)),
 'u_intermediate_0_1': neg(mult(4, u_0_1)),
 'u_intermediate_0_2': neg(mult(4, u_0_2)),
 'u_intermediate_0_3': neg(mult(4, u_0_3)),
 'u_intermediate_0_4': neg(mult(4, u_0_4)),
 'u_intermediate_0_5': neg(mult(4, u_0_5)),
 'u_intermediate_0_6': neg(mult(4, u_0_6)),
 'u_intermediate_0_7': neg(mult(4, u_0_7)),
 'u_intermediate_0_8': neg(mult(4, u_0_8)),
 'u_intermediate_0_9': neg(mult(4, u_0_9)),
 'u_intermediate_1_0': neg(mult(4, u_1_0)),
 'u_intermediate_1_1': neg(mult(4, u_1_1)),
 'u_intermediate_1_2': neg(mult(4, u_1_2)),
 'u_intermediate_1_3': neg(mult(4, u_1_3)),
 'u_intermediate_1_4': neg(mult(4, u_1_4)),
 'u_intermediate_1_5': neg(mult(4, u_1_5)),
 'u_intermediate_1_6': neg(mult(4, u_1_6)),
 'u_intermediate_1_7': neg(mult(4, u_1_7)),
 'u_intermediate_1_8': neg(mult(4, u_1_8)),
 'u_intermediate_1_9': neg(mult(4, u_1_9)),
 'u_intermediate_2_0': neg(mult(4, u_2_0)),
 'u_intermediate_2_1': neg(mult(4, u_2_1)),
 'u_intermediate_2_2': neg(mult(4, u_2_2)),
 'u_intermediate_2_3': neg(mult(4, u_2_3)),
 'u_intermediate_2_4': neg(mult(4, u_2_4)),
 'u_intermediate_2_5': neg(mult(4, u_2_5)),
 'u_intermediate_2_6': neg(mult(4, u_2_6)),
 'u_intermediate_2_7': neg(mult(4, u_2_7)),
```

(continues on next page)

(continued from previous page)

```
 'u_intermediate_2_8': neg(mult(4, u_2_8)),
 'u_intermediate_2_9': neg(mult(4, u_2_9)),
 'u_intermediate_3_0': neg(mult(4, u_3_0)),
 'u_intermediate_3_1': neg(mult(4, u_3_1)),
 'u_intermediate_3_2': neg(mult(4, u_3_2)),
 'u_intermediate_3_3': neg(mult(4, u_3_3)),
 'u_intermediate_3_4': neg(mult(4, u_3_4)),
 'u_intermediate_3_5': neg(mult(4, u_3_5)),
 'u_intermediate_3_6': neg(mult(4, u_3_6)),
 'u_intermediate_3_7': neg(mult(4, u_3_7)),
 'u_intermediate_3_8': neg(mult(4, u_3_8)),
 'u_intermediate_3_9': neg(mult(4, u_3_9)),
 'u_intermediate_4_0': neg(mult(4, u_4_0)),
 'u_intermediate_4_1': neg(mult(4, u_4_1)),
 'u_intermediate_4_2': neg(mult(4, u_4_2)),
 'u_intermediate_4_3': neg(mult(4, u_4_3)),
 'u_intermediate_4_4': neg(mult(4, u_4_4)),
 'u_intermediate_4_5': neg(mult(4, u_4_5)),
 'u_intermediate_4_6': neg(mult(4, u_4_6)),
 'u_intermediate_4_7': neg(mult(4, u_4_7)),
 'u_intermediate_4_8': neg(mult(4, u_4_8)),
 'u_intermediate_4_9': neg(mult(4, u_4_9)),
 'u_intermediate_5_0': neg(mult(4, u_5_0)),
 'u_intermediate_5_1': neg(mult(4, u_5_1)),
 'u_intermediate_5_2': neg(mult(4, u_5_2)),
 'u_intermediate_5_3': neg(mult(4, u_5_3)),
 'u_intermediate_5_4': neg(mult(4, u_5_4)),
 'u_intermediate_5_5': neg(mult(4, u_5_5)),
 'u_intermediate_5_6': neg(mult(4, u_5_6)),
 'u_intermediate_5_7': neg(mult(4, u_5_7)),
 'u_intermediate_5_8': neg(mult(4, u_5_8)),
 'u_intermediate_5_9': neg(mult(4, u_5_9)),
 'u_intermediate_6_0': neg(mult(4, u_6_0)),
 'u_intermediate_6_1': neg(mult(4, u_6_1)),
 'u_intermediate_6_2': neg(mult(4, u_6_2)),
 'u_intermediate_6_3': neg(mult(4, u_6_3)),
 'u_intermediate_6_4': neg(mult(4, u_6_4)),
 'u_intermediate_6_5': neg(mult(4, u_6_5)),
 'u_intermediate_6_6': neg(mult(4, u_6_6)),
 'u_intermediate_6_7': neg(mult(4, u_6_7)),
 'u_intermediate_6_8': neg(mult(4, u_6_8)),
 'u_intermediate_6_9': neg(mult(4, u_6_9)),
 'u_intermediate_7_0': neg(mult(4, u_7_0)),
 'u_intermediate_7_1': neg(mult(4, u_7_1)),
 'u_intermediate_7_2': neg(mult(4, u_7_2)),
 'u_intermediate_7_3': neg(mult(4, u_7_3)),
 'u_intermediate_7_4': neg(mult(4, u_7_4)),
 'u_intermediate_7_5': neg(mult(4, u_7_5)),
 'u_intermediate_7_6': neg(mult(4, u_7_6)),
 'u_intermediate_7_7': neg(mult(4, u_7_7)),
 'u_intermediate_7_8': neg(mult(4, u_7_8)),
 'u_intermediate_7_9': neg(mult(4, u_7_9)),
 'u_intermediate_8_0': neg(mult(4, u_8_0)),
 'u_intermediate_8_1': neg(mult(4, u_8_1)),
 'u_intermediate_8_2': neg(mult(4, u_8_2)),
 'u_intermediate_8_3': neg(mult(4, u_8_3)),
```

(continues on next page)

(continued from previous page)

```
'u_intermediate_8_4': neg(mult(4, u_8_4)),
'u_intermediate_8_5': neg(mult(4, u_8_5)),
'u_intermediate_8_6': neg(mult(4, u_8_6)),
'u_intermediate_8_7': neg(mult(4, u_8_7)),
'u_intermediate_8_8': neg(mult(4, u_8_8)),
'u_intermediate_8_9': neg(mult(4, u_8_9)),
'u_intermediate_9_0': neg(mult(4, u_9_0)),
'u_intermediate_9_1': neg(mult(4, u_9_1)),
'u_intermediate_9_2': neg(mult(4, u_9_2)),
'u_intermediate_9_3': neg(mult(4, u_9_3)),
'u_intermediate_9_4': neg(mult(4, u_9_4)),
'u_intermediate_9_5': neg(mult(4, u_9_5)),
'u_intermediate_9_6': neg(mult(4, u_9_6)),
'u_intermediate_9_7': neg(mult(4, u_9_7)),
'u_intermediate_9_8': neg(mult(4, u_9_8)),
'u_intermediate_9_9': neg(mult(4, u_9_9)),
'uinitial_0_0': const(0),
'uinitial_0_1': const(0),
'uinitial_0_2': const(0),
'uinitial_0_3': const(0),
'uinitial_0_4': const(0),
'uinitial_0_5': const(0),
'uinitial_0_6': const(0),
'uinitial_0_7': const(0),
'uinitial_0_8': const(0),
'uinitial_0_9': const(0),
'uinitial_1_0': const(0),
'uinitial_1_1': const(0),
'uinitial_1_2': const(0),
'uinitial_1_3': const(0),
'uinitial_1_4': const(0),
'uinitial_1_5': const(0),
'uinitial_1_6': const(0),
'uinitial_1_7': const(0),
'uinitial_1_8': const(0),
'uinitial_1_9': const(0),
'uinitial_2_0': const(0),
'uinitial_2_1': const(0),
'uinitial_2_2': const(0),
'uinitial_2_3': const(0),
'uinitial_2_4': const(0),
'uinitial_2_5': const(0),
'uinitial_2_6': const(0),
'uinitial_2_7': const(0),
'uinitial_2_8': const(0),
'uinitial_2_9': const(0),
'uinitial_3_0': const(0),
'uinitial_3_1': const(0),
'uinitial_3_2': const(0),
'uinitial_3_3': const(0),
'uinitial_3_4': const(0),
'uinitial_3_5': const(0),
'uinitial_3_6': const(0),
'uinitial_3_7': const(0),
'uinitial_3_8': const(0),
'uinitial_3_9': const(0),
```

(continues on next page)

(continued from previous page)

```
 'uinitial_4_0': const(0),
 'uinitial_4_1': const(0),
 'uinitial_4_2': const(0),
 'uinitial_4_3': const(0),
 'uinitial_4_4': const(0),
 'uinitial_4_5': const(0),
 'uinitial_4_6': const(0),
 'uinitial_4_7': const(0),
 'uinitial_4_8': const(0),
 'uinitial_4_9': const(0),
 'uinitial_5_0': const(0),
 'uinitial_5_1': const(0),
 'uinitial_5_2': const(0),
 'uinitial_5_3': const(0),
 'uinitial_5_4': const(0),
 'uinitial_5_5': const(1),
 'uinitial_5_6': const(0),
 'uinitial_5_7': const(0),
 'uinitial_5_8': const(0),
 'uinitial_5_9': const(0),
 'uinitial_6_0': const(0),
 'uinitial_6_1': const(0),
 'uinitial_6_2': const(0),
 'uinitial_6_3': const(0),
 'uinitial_6_4': const(0),
 'uinitial_6_5': const(0),
 'uinitial_6_6': const(0),
 'uinitial_6_7': const(0),
 'uinitial_6_8': const(0),
 'uinitial_6_9': const(0),
 'uinitial_7_0': const(0),
 'uinitial_7_1': const(0),
 'uinitial_7_2': const(0),
 'uinitial_7_3': const(0),
 'uinitial_7_4': const(0),
 'uinitial_7_5': const(0),
 'uinitial_7_6': const(0),
 'uinitial_7_7': const(0),
 'uinitial_7_8': const(0),
 'uinitial_7_9': const(0),
 'uinitial_8_0': const(0),
 'uinitial_8_1': const(0),
 'uinitial_8_2': const(0),
 'uinitial_8_3': const(0),
 'uinitial_8_4': const(0),
 'uinitial_8_5': const(0),
 'uinitial_8_6': const(0),
 'uinitial_8_7': const(0),
 'uinitial_8_8': const(0),
 'uinitial_8_9': const(0),
 'uinitial_9_0': const(0),
 'uinitial_9_1': const(0),
 'uinitial_9_2': const(0),
 'uinitial_9_3': const(0),
 'uinitial_9_4': const(0),
 'uinitial_9_5': const(0),
```

(continues on next page)

(continued from previous page)

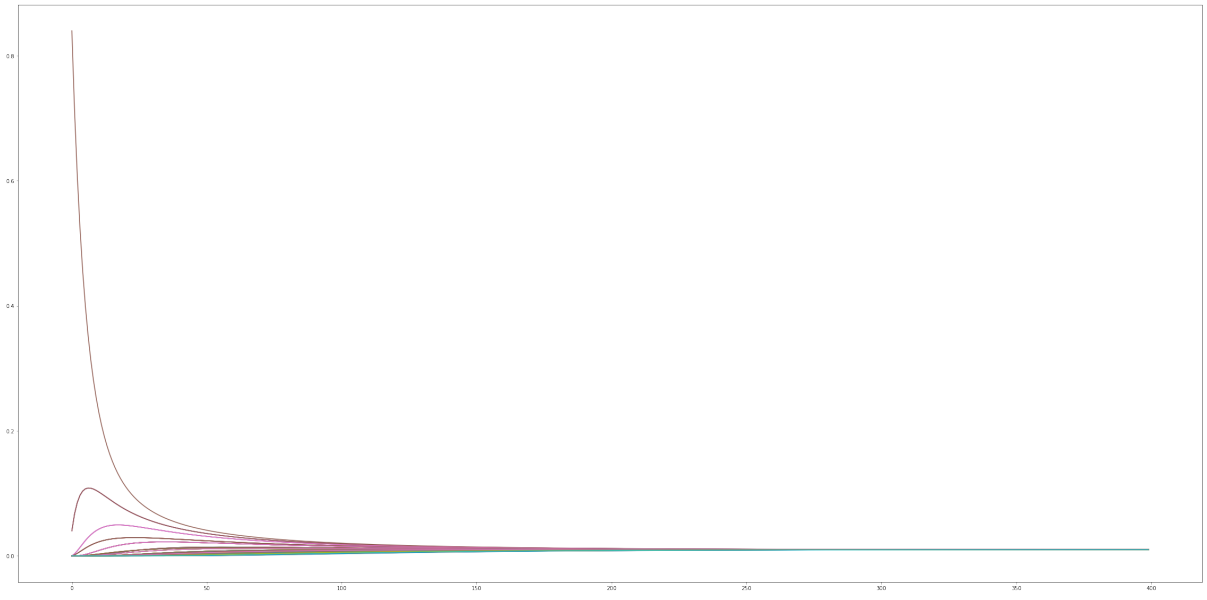
```
 'uinitial_9_6': const(0),
 'uinitial_9_7': const(0),
 'uinitial_9_8': const(0),
 'uinitial_9_9': const(0)})
```

[36]:
```
c_code = state.export(to="C")
compile(c_code, "heateq2d.cpp")
```

[37]:
```
data = run(arguments={'max_iterations':400})
```

```
Running: ./a.out --max_iterations=400
```

[38]:
```
for i,j in ndindex(NN):
    plot(data[f"u_{i}_{j}"])
```



[39]:
```
heat = array([data[f"u_{i}_{j}"] for i,j in ndindex(NN)]).reshape((N,N,
↪len(data)))
heat.shape
```

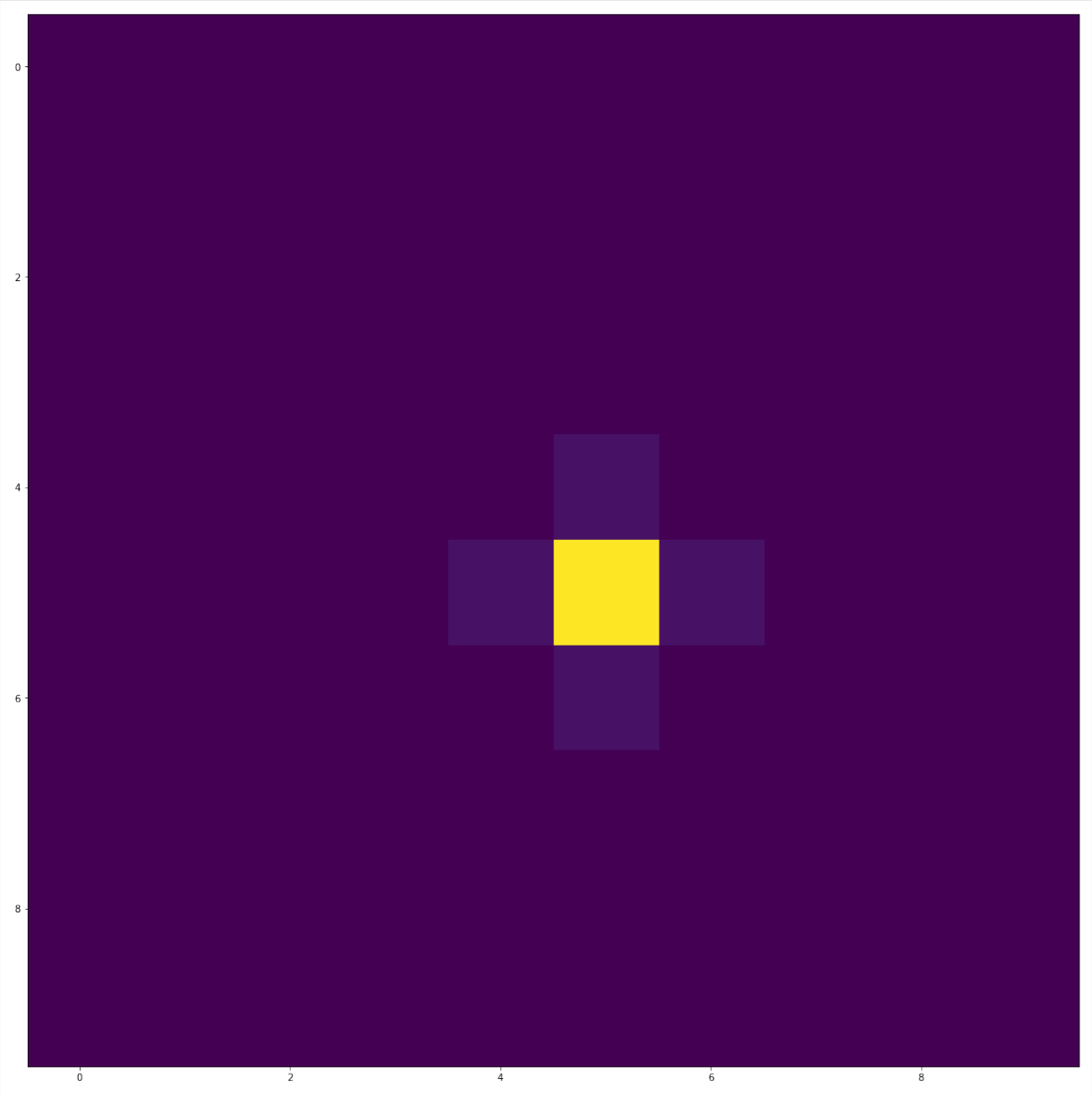
[39]:
```
(10, 10, 400)
```

The physical recovery of the digital data is not so easy to inspect, since the two spatial dimensions would require to make a movie in order to display the time evolution. What we do instead here is to show the first snapshot and the last one.

In the following two figures, the x and y axis correspond to their physical coordinates. These are *still images* for $t = 0$ and $t = t_{\text{final}}$.

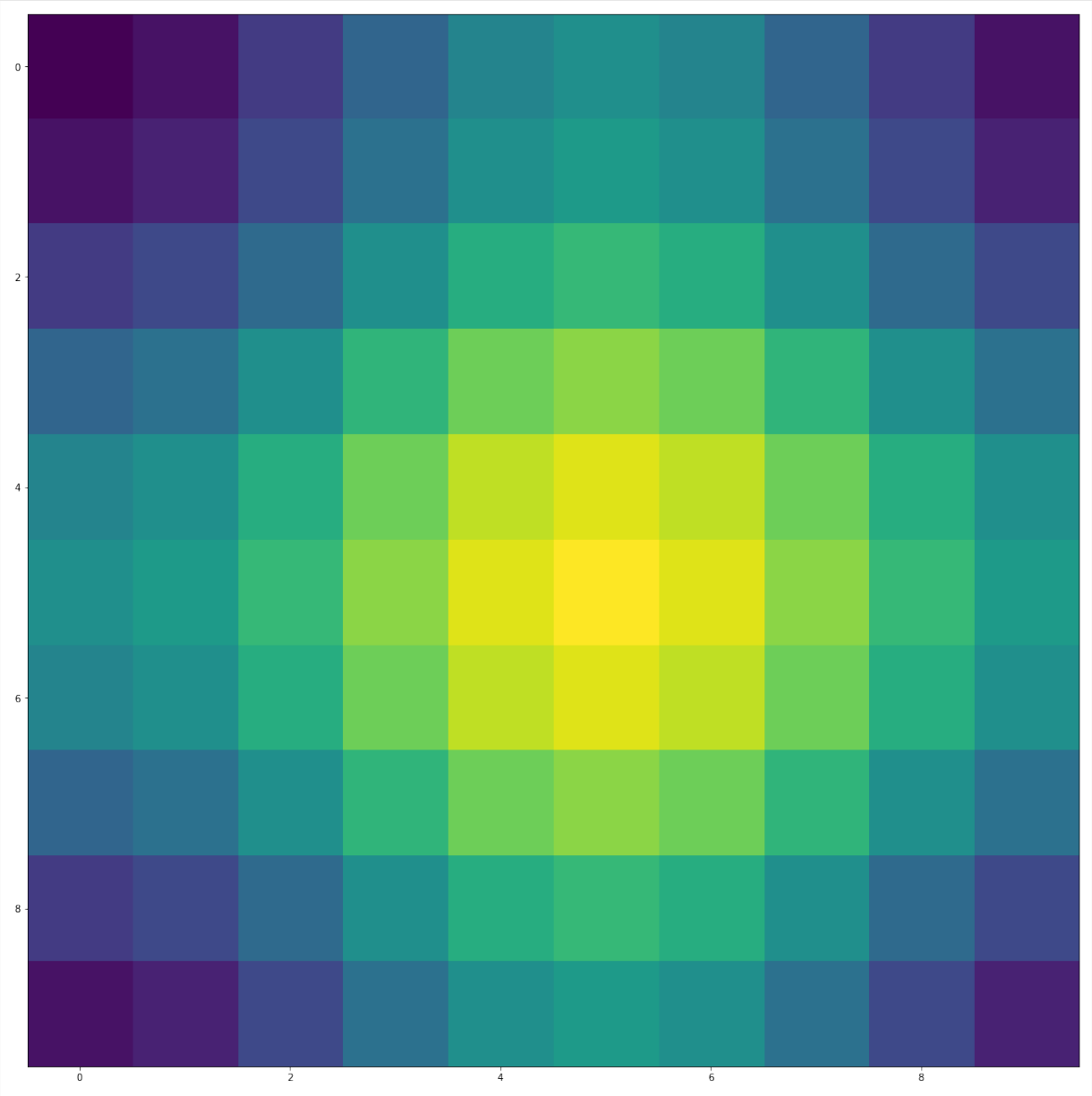
[22]:
```
rcParams["figure.figsize"] = (40,20)
imshow(heat[:,:,0])
```

[22]:
```
<matplotlib.image.AxesImage at 0x7f7fd5956ac0>
```



[23]:
```
imshow(heat[:,:,-1])
```

[23]:
```
<matplotlib.image.AxesImage at 0x7f7fd59675e0>
```



In order to provide a final discussion, what is visible here is the dissipation of heat from a single maximum to the whole simulation domain.

With this, we end the demonstration notebook about the heat equation in DDA.

About this notebook

This notebook was written during a two hour phone call as an interactive demonstration how to do high-level programming with DDA.

3 PyFPAA

3.1 About FPAA

PyFPAA is a work bench to gain experience with hybrid computer programming. It can be used to create examples for the digital/hybrid extensions of the Analog Paradigm Model-1 analog computer (AP/M-1).

pyFPAA is a small Python3 code for compiling instructions for the AP/M-1 Hybrid Controller (HC), especially for the DPT (digital potentiometers), XBAR (crossbar switch) and the HC itself. The input is a quite extensive machine description together with a program (basically an analog circuit), both written in a minimalistic HDL in YAML representation. See the *README* file there as well as the *doc* directory for more details.

See also *examples/fpaa-circuits* for a number of example input for the pyFPAA compiler. There also exist basic test scripts within *tests*.

PyFPAA is a compiler for programmable analog computers (FPAAs). It was written by SvenK in Dec 2019 for quickly approaching a testing infrastructure for the XBAR module for the *Analog Paradigm M-1* analog computer.

The script requires a (lengthy) machine description, which encodes the computational parts available and is quite similar to a machine library in VHDL. That file encodes especially the hard-wired vs. configurable parts of the machine. The actual program is then rather short and describes how the configurable computational parts are connected with each other. It also specifies constant coefficients which are set with digital potentiometers or other digital steering parameters.

The output of the code is the analog computer configuration, as required by `hycon`. This can be either

- a single line of text, which are mostly hexadecimal encoded instructions together with command characters, all following the serial console protocol which the HybridController of the machine expects (http://analogparadigm.com/downloads/hc_handbook.pdf).
- configuration tuples for `hycon`
- or a direct API to `hycon`

In order to run this program, all you need is `PyYAML` to read the YAML files. If you want to draw circuit plots, you need `matplotlib`.

3.1.1 Command line interface

```
% python -m fpaa --help
usage: fpaa.py [-h] [-v] [-o OUTPUT.txt] [-p OUTPUT.pdf] (-A {XBAR-Only,AP-
↪M1-Mini} | -a path/to/my/MACHINE.yml)
            CIRCUIT.yml

A circuit synthesizer for the HyConAVR.

positional arguments:
CIRCUIT.yml           The YAML file holding the circuit description

optional arguments:
-h, --help            show this help message and exit
-v, --verbose         increases log verbosity for each occurence.
-o OUTPUT.txt, --output OUTPUT.txt
                      Put output string into file (default is '-' and␣
↪means stdout)
-p OUTPUT.pdf, --plot OUTPUT.pdf
                      Plot crossbar switch
-A {XBAR-Only,AP-M1-Mini}, --registered-arch {XBAR-Only,AP-M1-Mini}
                      Target machine architecture description: Any YAML␣
↪file in directory
                      /home/sven/Analog/Forschungsauftrag/dda/fpaa is␣
↪available as machine
-a path/to/my/MACHINE.yml, --arch path/to/my/MACHINE.yml
                      Target machine architecture description (any valid␣
↪filename)
```

See *fpaa.cli()* (page 83) for further details.

class fpaa.fpaa.**Target**(*part*, *pin*)

part
Alias for field number 0

pin
Alias for field number 1

fpaa.fpaa.**load_from_yaml**(*circuit*, *arch*)
Expects arch and circuit to be strings.

fpaa.fpaa.**synthesize**(*circuit*, *arch*)
Translate a circuit to a netlist for a given target architecture.

This routine is the heart of the FPAA compiler. It mainly

- Allocates available hardware to the requested ones by the user circuits and allows for book keeping between the user named and architecture named computing components.
- Ensures consistency of the resulting circuit (there are no dangling wires, no over-allocation, etc.)

There is a lot of *info* and *debug* output available if turned on via Python logging.

Expects `arch` and `circuit` to be nested data structures (dicts and lists holding strings and numbers), similar to their YAML representation. The documentation does not yet cover an in-depth description of these data structures, but there are tons of example YAML files which are

straightforward to understand.

Returns `wired_circuit`, a list of computing components (which itself are again "PODs", i.e. dicts with nested data structures).

`fpaa.fpaa.`**`normalize_potentiometer`**(*value*, *resolution_bits=10*)
Map a real value [0..1] to Potentiometer value [0..1023]

`fpaa.fpaa.`**`compile_instructions`**(*wired_circuit*, *arch*)
Compile a netlist (`wired_circuit`) to configuration *instructions* for setting up the analog computer hybrid controller.

This routine basically loops over the *hardwired parts* of the given architecture, i.e. built-in

- potentiometers (DPT-24 and HC)
- cross bar arrays (XBAR)

and configures them according to the given wired_circuit. This means that relevant allocated potentiometers will be set and the XBAR configuration bitmask will be computed from the hardware description provided by the circuit and the architecture.

Currently returns a list of instructions (tuples) which could be directly be written out to serial or passed to PyHyCon.

Note: This method will change in near time and talk directly to a PyHyCon instance.

`fpaa.fpaa.`**`plot_xbar`**(*target_file*, *circuit_title*, *xbar_config=None*, *interactive_plotting=False*)
Draw an the allocation of a crossbar switch array (xbar) matrix.

xbar_config is a tuple with (cols,rows,boolean_matrix), and by default the last one from a global registry (last_seen_xbars) is taken, which is what you want.

`fpaa.fpaa.`**`cli`**()
This module is callable via `python -m fpaa` or `./fpaa.py`. It exposes the main functions on the command line which is especially helpful for debugging or interactively programming an analog computer from the command line.

Call `--help` for all possible command line options.

3.2 Example codes for FPAA

We have a number of circuits implemented from the Analog Paradigm application notes[76] website (*Alpaca* in short).

These are for instance:

- Rössler attractor[77] (Alpaca 1)
- Lorentz attractor[78] (Alpaca 2)
- Lotka-Volterra equations[79] (Alpaca 6)

[76] http://analogparadigm.com/documentation.html
[77] http://analogparadigm.com/downloads/alpaca_1.pdf
[78] http://analogparadigm.com/downloads/alpaca_2.pdf
[79] http://analogparadigm.com/downloads/alpaca_6.pdf

- Nose-Hoover oscillator, Jerk oscillator[80] (Alpaca 15)
- Biochemistry enzyme kinetics (based on n1.pdf)
- Different toy codes for solving `exp(t)`, `sin(t)`, etc.

3.3 Example: A reprogrammable small Model-1

This is work in progress and here comes some more pictures and description.

Listing 1: Well there

```
# This is a YAML description file for a machine.
# A poor man's Analog HDL (should use Verilog HDLs instead)
#
# Copyright (c) 2020 anabrid GmbH
# Contact: https://www.anabrid.com/licensing/
#
# This file is part of the FPAA module of the PyAnalog toolkit.
#
# ANABRID_BEGIN_LICENSE:GPL
# Commercial License Usage
# Licensees holding valid commercial anabrid licenses may use this file in
# accordance with the commercial license agreement provided with the
# Software or, alternatively, in accordance with the terms contained in
# a written agreement between you and Anabrid GmbH. For licensing terms
# and conditions see https://www.anabrid.com/licensing. For further
# information use the contact form at https://www.anabrid.com/contact.
#
# GNU General Public License Usage
# Alternatively, this file may be used under the terms of the GNU
# General Public License version 3 as published by the Free Software
# Foundation and appearing in the file LICENSE.GPL3 included in the
# packaging of this file. Please review the following information to
# ensure the GNU General Public License version 3 requirements
# will be met: https://www.gnu.org/licenses/gpl-3.0.html.
# For Germany, additional rules exist. Please consult /LICENSE.DE
# for further agreements.
# ANABRID_END_LICENSE
#

title: AP/M-1 Mini

description: |
   This file describes the hardware layout of a particular Analog Paradigm
   Model-1 machine with software-definable XBAR in the smallest setup,
   containing SUM8, INT4, MUL4, DPT32 (and of course XBAR, HC).

   The number of Summers limits the number of negating-macro elements
   which can be built. The configurable parts show what can be done

      SUM8.1 => 1 x SUM2m
      SUM8.2 => 0.5 SUM2pm
      SUM8.3 => 0.5 SUM2pm
```

(continues on next page)

[80] http://analogparadigm.com/downloads/alpaca_15.pdf

(continued from previous page)

```
      SUM8.4 => 0.5 SUM2pm
      SUM8.5 => 0.5 SUM2pm
      SUM8.6 => 1 x INT2pm
      SUM8.7 => 1 x INT2pm, leaving 1 x INT2m
      SUM8.8 => 1 x MUL2pm, leaving 1 x MUL2m

   In total 8 computing elements. See configurable_parts for the formal
   definition.

configurable_parts:
   # naming follows the HyCon YML file, part "elements"
   INT0-0:
     type: INTpinv
     address:
       out: 0x0160 # integrator INT8.1
       inv: 0x0123 # summer SUM8.4 as inverter
   INT0-1:
     type: INTpinv
     address:
       out: 0x0161 # integrator INT8.2
       inv: 0x0126 # summer SUM8.7 as inverter
   INT0-2:
     type: INTp
     address:
       out: 0x0162 # integrator INT8.3
   MLT0-0: # HyCon calls it MLT8-0
     type: MULpinv
     address:
       out: 0x0100 # multiplier MUL4.1
       inv: 0x0127 # summer SUM8.8 as inverter
   MLT0-1: # HyCon calls it MLT8-1
     type: MULpinv
     address:
       out: 0x0100 # multiplier MUL4.2
   SUM0-0:
     type: SUMpinv
     address:
       out: 0x0120 # summer SUM8.1
       inv: 0x0124 # summer SUM8.5 as inverter (is below 1)
   SUM0-1:
     type: SUMpinv
     address:
       out: 0x0121 # summer SUM8.2
       inv: 0x0125 # summer SUM8.6 as inverter (is below 2)
   SUM0-2:
     type: SUMp
     address:
       out: 0x0122 # summer SUM8.3
   PlusOne:
     description: |
       Positive supply voltage/machine unit, also refered to as Vcc or Vdd,
       i.e. +10V in this machine.
     cannot_be_allocated: True
     type: ConstantVoltage
     input: [ +1 ]
   MinusOne:
```

(continues on next page)

(continued from previous page)

```
      description: |
        Negative supply voltage/machine unit, also refered to as Vss,
        i.e. -10V in this machine.
      cannot_be_allocated: True
      type: ConstantVoltage
      input: [ -1 ]
    None:
      description: See description of NoneType
      cannot_be_allocated: True
      type: NoneType


# We currently not make (yet) use of these levels anywhere, but they
# are later required for evaluating MUL's, for instance.
logic:
    boolean: [ True, False ]
    machine_units: [ +1, -1 ]
    voltage_range: [ +10, -10 ]
    voltage_unit: V
    DGND_voltage: 0 # won't this be different
    AGND_voltage: 0 # on the chip?

entities:

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪# # #
#
#             Elementary building blocks of Model-1 Computer
#
## # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪# # #

    DPT:
      type: elementary
      description: |
         A single digital potentiometer (part of a DPT24 or HC module).
      computes:
        vs: vs=v0*v
      input:
         - { name: v0, type: numeric, range: [0, 1] }
         - { name: v, type: analog }
      output:
         - { name: vs, type: analog }

    SUM:
      type: elementary
      description: |
         A single summer (part of a SUM8 module).
         The summer is a negating summer featuring
           * three inputs of weight a=10 (input prefix "d" for "decade") and
           * three inputs of weight a= 1 (input prefix "u" for "unit")
         We are lazy so we only describe two input lines each.
         We dont model the summing junction (SJ) yet.
      computes:
        out: out = -(10*d1 + 10*d2 + u1 + u2)
      input:
```

(continues on next page)

(continued from previous page)

```
153       - { name: d1, type: analog }
154       - { name: d2, type: analog }
155       - { name: u1, type: analog }
156       - { name: u2, type: analog }
157     output:
158       - name: out
159         type: analog
160         description: Output of negating summer
161     default_inputs: { d1: None, d2: None, u1: None, u2: None }
162
163   INT:
164     type: elementary
165     description: |
166        A single integrator (one of the INT4 module).
167
168        The integrator is an (implicite negating summing) intergrator
169        featuring
170          * three inputs of weight a=10 (input prefix "d" for "decade") and
171          * three inputs of weight a= 1 (input prefix "u" for "unit")
172        We are lazy so we only describe two input lines each.
173        For the moment, we dont model/make use of ModeIC and ModeOP.
174     computes:
175       out: out  =  integral(-10*d1(t) - 10*d2(t) - u1(t) - u2(t), t)
176     input:
177       - { name: d1, type: analog }
178       - { name: d2, type: analog }
179       - { name: u1, type: analog }
180       - { name: u2, type: analog }
181       - name: ic
182         type: analog
183         description: Initial condition
184     output:
185       - name: out
186         type: analog
187         description: Output of integral
188     default_inputs: { d1: None, d2: None, u1: None, u2: None, ic: None }
189
190
191   MUL:
192     type: elementary
193     description: |
194       A single multiplier (one of the MLT8 module).
195
196       Remember that multiplication is implemented normalized. For␣
↪instance,
197       if machine voltage is level=+10V (actually +-10V), the␣
↪multiplication
198       computes res = (a/level * b/level)*level in order to make sure
199       |res|<10V.
200
201       -> In order to decouple analog programming from the machine where
202          it runs, the normalization should be implemented in the
203          compilation step!
204     input:
205       - { name: a,   type: analog }
206       - { name: b,   type: analog }
```

(continues on next page)

(continued from previous page)

```
    output:
      - { name: out, type: analog }
    computes:
      out: out = a*b/voltage_level
    default_inputs: [ a: PlusOne, b: PlusOne ] # useful?

  CMPad:
    description: |
       The left part of a CMP4 comparator (analog2digital),
       i.e. a treshold based binarization.
    computes:
      res: res = a+b>0
    input:
      - { name:   a, type: analog }
      - { name:   b, type: analog }
    output:
      - { name: res, type: digital }

  CMPda:
    description: |
       The right part of a CMP4 comparator (digital2analog),
       i.e. a single electronic switch switching between two analog
       signals.
    computes:
      u: u = if(i, a, b)
    input:
      - { name: i, type: digital } # boolean
      - { name: a, type: analog  }
      - { name: b, type: analog  }
    output:
      - { name: u, type: analog  }


# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪# # # #
#
#         Macro computing elements plugged together by elementary ones
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪# # # #

  SUMp:
    type: macro
    description: |
       A single summer (SUM8 part) and two digital potentiometers.
       We only use weights a=10 from the summer.
       The flow chart is given by:

         (a0)-----+
                  |
                  v
         (a)->[DPTa]--(as)-->[10|         ]
                             [  | SUMo    ]---> out
         (b)->[DPTb]--(bs)-->[10|         ]
                  ^
                  |
```

(continues on next page)

(continued from previous page)

```
          (b0)----+

    computes:
      out: out = -(a0*a + b0*b)
    input:
      - name: a0
        type: numeric
        description: Potentiometer prefactor
        range: [0, 1]
      - name: a
        type: analog
      - name: b0
        type: numeric
        description: Potentiometer prefactor
        range: [0, 1]
      - name: b
        type: analog
    output:
      - name: out
        type: analog
        description: Output of negating summer
    default_inputs:
      a: None
      b: None
      a0: 0
      b0: 0
    internal_wires:
      - name: as
        type: analog
        description: Scaled a (output of DPTa)
      - name: bs
        type: analog
        description: Scaled b (output of DPTb)
    partlist:
      DPTa:
        type: DPT
        input: [ a0, a ]
        output: as
      DPTb:
        type: DPT
        input: [ b0, b ]
        output: bs
      SUMo:
        type: SUM
        input: { d1: as, d2: bs }
        output: out
    # These "is a"-relations should allow using elements instead of others
    is_a:
      SUM:
        description: Can mimic a simple summer when having both DPTs = 1.
        default_inputs: [a0: 1, b0: 1]
      DPT:
        description: Can mimic a simple potentiometer when not summing at␣
↪all.
        default_inputs: [a: None, b: None, b0: 0]

```

(continues on next page)

(continued from previous page)

```
  MULp:
    type: macro
    description: |
      A simple macro element made of a single multiplier (MLT8) and two
      digital potentiometers (on DPT24).

      The flow chart is similar to SUMp:

          (a0)----+
                  |
                  v
          (a)->[DPTa]--(as)-->[        ]
                               [  MULo ]---> out
          (b)->[DPTb]--(bs)-->[        ]
                  ^
                  |
          (b0)----+

    input:
      - { name: a0,  type: digital }
      - { name: a,   type: analog  }
      - { name: b0,  type: digital }
      - { name: b,   type: analog  }
    output:
      - { name: out, type: analog  }
    computes:
      out: out = a0*a*b0*b/voltage_level
    default_inputs:
      a: PlusOne
      b: PlusOne
      a0: 0
      b0: 0
    internal_wires:
      - name: as
        type: analog
        description: Scaled a (output of DPTa)
      - name: bs
        type: analog
        description: Scaled b (output of DPTb)
    partlist:
      DPTa:
        type: DPT
        input: [ a0, a ]
        output: as
      DPTb:
        type: DPT
        input: [ b0, b]
        output: bs
      MULo:
        type: MUL
        input: [as, bs]
        output: out
    is_a:
      MUL:
        description: Can mimic a simple multiplier when having both DPTs␣
↪= 1.
```

(continues on next page)

(continued from previous page)

```
        default_inputs: [a0: 1, b0: 1]
      # DPT: would be ironic since MLT also multiplies. Would rather compute MLT(DPT(x,PlusOne),PlusOne)


  INTp:
    type: macro
    description: |
      A macro component made of an integrator (one in the INT4 module)
      and three potentiometers (two shall be implemented on the DPT24,
      one on the HC, but that doesn't matter at this stage).

      The integrator is an (implicite negating summing) intergrator
      with two inputs. For the moment, we dont make use of ModeIC and
      ModeOP. All inputs use weight a=10.

      In the moment, the ICs are realized by one potentiometer and
      a comparator. The compiler creates the binary sign which steers
      the comparator and is outputted by the Hybrid Controller (HC).

      The summer is a (negating) summer with a single input.
      The summer uses weight a=1.
      Neither in integrator nor summer, we make use of the SJ.

      The flow chart looks like

       ic ----------------------------+
                                      |
       HC digital output,             |
       icSign -----+                  |
                   | switches         | controls
                   v                  v
        +1 -(p1)->[ SignGen ]-(pm)->[DPTic]   (HC DPT)
        -1 -(m1)->[  (CMPda) ]         |
                                     (aic)
        a0 ------+                     |
                 | controls            | sets IC
                 v                     v
        a ->[DPTa]---(as)--->[10|            ]
                               [    Integrator ]----> out
        b ->[DPTb]---(bs)--->[10|            ]
                 ^
                 |  controls
        b0 ------+
    computes:
      out: out = -integral( a0*a(t) + b0*b(t), t)
    input:
      - name: a0
        type: numeric
        description: Potentiometer prefactor
        range: [0, 1]
      - name: a
        type: analog
      - name: b0
        type: numeric
        description: Potentiometer prefactor
```

(continues on next page)

(continued from previous page)

```
        range: [0, 1]
      - name: b
        type: analog
      - name: icSign
        type: boolean
        description: Sign of initial condition value
      - name: ic
        type: numeric
        description: Value for initial condition
        range: [0, 1]
    output:
      - name: out
        type: analog
        description: Output of integral
    default_inputs:
      a: None
      b: None
      a0: 0
      b0: 0
      icSign: True
      ic: 0
    internal_wires:
      - name: as
        type: analog
        description: Scaled a (output of DPTa)
      - name: bs
        type: analog
        description: Scaled b (output of DPTb)
      - name: aic
        type: analog
        description: Analog Initial conditions (output of Comparator and multiplied by HC DPT)
      - name: pm
        type: analog
        description: Plus or Minus, depending on the sign bit
    partlist:
      DPTa:
        type: DPT
        input: [ a0, a ]
        output: as
      DPTb:
        type: DPT
        input: [ b0, b]
        output: bs
      SignGen:
        type: CMPda0
        input:
          i: icSign
          a: PlusOne
          b: MinusOne
        output: pm
      DPTic:
        type: DPT # but shall be implemented on HC for clarity
        input: [ ic, pm ]
        output: aic
      INTo:
```

(continues on next page)

(continued from previous page)

```
        type: INT
        input: { d1: as, d2: bs, ic: aic }
        output: out

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
#
#         Macro elements with attached inverter (negation)
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

  SUMpinv:
    type: macro
    is_a:
      SUMp:
        description: |
          Can mimic a single potentiometered summer. Just ignore the inv output.
    description: |
       A macro component made of two summers (SUM8 module) and two digital
       potentiometers (on DPT24). It is an easy extension of the SUMp
       module, the flow chart is just:

                +--------+
         a0  -->|        |        +--[ 1| SUMinv ]-> inv
         a   -->| SUMp   |        |
         b0  -->|        |--------+----------------> out
         b   -->|        |
                +--------+

       Remember, the summer SUMo in SUMp uses weights a=10, while the
       inverting summer SUMinv uses weight a=1.

    computes:
      out: out =   a0*a + b0*b
      inv: inv = -(a0*a + b0*b)
    input:
      - name: a0
        type: numeric
        description: Potentiometer prefactor
        range: [0, 1]
      - name: a
        type: analog
      - name: b0
        type: numeric
        description: Potentiometer prefactor
        range: [0, 1]
      - name: b
        type: analog
    output:
      - name: out
        type: analog
        description: Output of minus operation (actual summer output)
      - name: inv
        type: analog
```

(continues on next page)

(continued from previous page)

```
        description: Output of plus operation (negation of summer output)
    default_inputs:
      a: None
      b: None
      a0: 0
      b0: 0
    partlist:
      MainSummer:
        type: SUMp
        input: [ a0, a, b0, b ]
        output: out
      SUMinv:
        type: SUM
        input: { u1: out }
        output: inv

  MULpinv:
    type: macro
    description: |
      Extension of MUL2m with two outputs, where one is negated, thus
      requiring an internal summer.
      Flow chart is just:


              +-------+
       a0  -->|       |         +--[ 1| SUMinv ]-> inv
       a   -->| MULp  |         |
       b0  -->|       |---------+-----------------> out
       b   -->|       |
              +-------+
    input:
      - name: a0
        type: digital
      - name: a
        type: analog
      - name: b0
        type: digital
      - name: b
        type: analog
    output:
      - name: out
        type: analog
      - name: inv
        type: analog
    computes:
      out: out =  a0*a*b0*b/voltage_level
      inv: inv = -a0*a*b0*b/voltage_level
    default_inputs:
      a: PlusOne
      b: PlusOne
      a0: 0
      b0: 0
    partlist:
      Multiplier:
        type: MULp
        input: [ a0, a, b0, b ]
```

(continues on next page)

(continued from previous page)

```
        output: out
      SUMinv:
        type: SUM
        input: { u1: out }
        output: inv

  INTpinv:
    type: macro
    is_a:
      INTp:
        description: |
          Can mimic a single potentiometered integrator. Just ignore the inv output.
    description: |
       A macro component which can invert the output of INTp.
       The flow chart is just:

                +-------+
         a0  -->|       |        +--[ 1| SUMinv ]-> inv
         a   -->| INTp  |        |
         b0  -->|       |--------+----------------> out
         b   -->|       |
         icSign>|       |
         ic  -->|       |
                +-------+

    computes:
      out: out  =  integral( a0*a(t) + b0*b(t), t)
      inv: inv  = -integral( a0*a(t) + b0*b(t), t)
    input:
      - name: a0
        type: numeric
        description: Potentiometer prefactor
        range: [0, 1]
      - name: a
        type: analog
      - name: b0
        type: numeric
        description: Potentiometer prefactor
        range: [0, 1]
      - name: b
        type: analog
      - name: icSign
        type: boolean
        description: Sign of initial condition value
      - name: ic
        type: numeric
        description: Value for initial condition
        range: [0, 1]
    output:
      - name: inv
        type: analog
        description: Actual output of integrator
      - name: out
        type: analog
        description: Output of summer (negation of integrator)
```

(continues on next page)

(continued from previous page)

```
    default_inputs:
      a: None
      b: None
      a0: 0
      b0: 0
      icSign: True
      ic: 0
    partlist:
      MainINT:
        type: INTp
        input: [ a0, a, b0, b, icSign, ic ]
        output: out
      SUMinv:
        type: SUM
        input: { u1: out }
        output: inv

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪# # # #
#
#         Pseudo elements (Helpers)
#
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
↪# # # #

  ConstantVoltage:
    description: |
      This is a pseudo-element only used for allocate a logic +1 and
      logic -1 on the XBAR.
      Usage examples are given in the partlist with the instances
      PlusOne and MinusOne. These parts are subject to be used
      also in the actual programs (for the time being), in order to
      realize constant inputs for computatational parts. We need
      +1 and -1 because the potentiometers only implement range [0,1].
      The operation x-0.5 is then realized by the pseudo-code
      SUMMER(a=x,a0=1,b=MinusOne,b0=0.5), while x+0.7 is realized by
      SUMMER(a=x,a0=1,b=PlusOne,b0=0.7).
    computes:
      value: value = x
    input:  [{ name: x,     type: analog }]
    output: [{ name: value, type: analog }]

  NoneType:
    description: |
      This pseudo-element is only used to implement computational
      elements with unconnected inputs. For instance, the pseudo-code
      SUMMER(x)=-x is actually implemented as
      SUMMER(a=x,a0=1,b=None,b0=0)  (the value of b0 is not relevant!).
      The compiler will throw out None elements in favour of disabling
      the corresponding output line of the XBAR.

      The None is also used for slurping input from the DPTs, therefore
      they have an input.
    computes: nothing
    input:          [{ name: none, type: analog }]
    output:         [{ name: none, type: analog }]
```

(continues on next page)

(continued from previous page)

```
    default_inputs:  { none: 0 }


# Each of the wired hardware needs configuration by the HC
# (which also needs to be configured itself)
wired_parts:
  X1:
    type: XBAR
    address: 0x0040
    number_of_config_bytes: 10
    # Datasheet of chip AD8113:
    # https://www.analog.com/media/en/technical-documentation/data-sheets/AD8113.pdf
    topology: Single AD8113
    size: [ 16, 16 ]
    input_columns:
       - PlusOne           #  1
       - MinusOne          #  2
       - SUM0-2            #  3
       - SUM0-1: out       #  4
       - SUM0-1: inv       #  5
       - SUM0-0: out       #  6
       - SUM0-0: inv       #  7
       - MLT0-1            #  8
       - MLT0-0: out       #  9
       - MLT0-0: inv       # 10
       - INT0-2            # 11
       - INT0-1: out       # 12
       - INT0-1: inv       # 13
       - INT0-0: out       # 14
       - INT0-0: inv       # 15
       - None              # 16, empty!
    output_rows:
       - INT0-0: a         #  1
       - INT0-0: b         #  2
       - INT0-1: a         #  3
       - INT0-1: b         #  4
       - INT0-2: a         #  5
       - INT0-2: b         #  6
       - MLT0-0: a         #  7
       - MLT0-0: b         #  8
       - MLT0-1: a         #  9
       - MLT0-1: b         # 10
       - SUM0-0: a         # 11
       - SUM0-0: b         # 12
       - SUM0-1: a         # 13
       - SUM0-1: b         # 14
       - SUM0-2: a         # 15
       - SUM0-2: b         # 16

  D1:
    type: DPT24
    address: 0x0060
    size: 24
    enumeration:
       - INT0-0: a0     #  1
```

(continues on next page)

(continued from previous page)

```
      - INT0-0: b0     #  2
      - INT0-1: a0     #  3
      - INT0-1: b0     #  4
      - INT0-2: a0     #  5
      - INT0-2: b0     #  6
      - MLT0-0: a0     #  7
      - MLT0-0: b0     #  8
      - MLT0-1: a0     #  9
      - MLT0-1: b0     # 10
      - SUM0-0: a0     # 11
      - SUM0-0: b0     # 12
      - SUM0-1: a0     # 13
      - SUM0-1: b0     # 14
      - SUM0-2: a0     # 15
      - SUM0-2: b0     # 16

  Main:
    # A Hybrid Controller has 8 DPTs, 8 digital inputs and
    # 8 digital outputs. There can actually be only one HC per
    # machine.
    type: HC
    address: 0x0080
    number_of_digital_output_ports: 8
    number_of_digital_input_ports: 8
    dpt_resolution: 10
    dpt_enumeration:
      - INT0-0: ic
      - INT0-1: ic
      - INT0-2: ic
    digital_input: []
    digital_output:
      - INT0-0: icSign
      - INT0-1: icSign
      - INT0-2: icSign
```

4 PyHyCon

4.1 The Python Hybrid Controller interface

PyHyCon – a Python Hybrid Controller interface.

Note that the `IO::HyCon` Perl 5 module is the reference implementation that is maintained by the HyConAVR firmware author (Bernd). You can find the `IO::HyCon` Perl module as well as the mentioned Arduino firmware at https://github.com/anabrid/Model-1/tree/main/software/HC. The Perl code is also part of the CPAN at https://metacpan.org/pod/IO::HyCon.

While this implementation tries to be API-compatible with the reference implementation, it tries to be minimal/low-level and won't implement any client-side luxury (such as address mapping). It is the task of the user to implement something high-level ontop of this.

Especially, the following tasks are implemented by different modules which can but do not needed to be used:

- Connection managament: HyCon just assumes a file handle, but different connection types are proposed in the `connections` module.
- Autosetup: PyHyCon is plain python and has no dependency, for instance on YAML. There is `autosetup` which implements the "autosetup" functionality of Perl-HyCon.
- High level functionality is implemented on top of HyCon and not within. See for instance *`fpaa`* (page 81) for an abstraction which can generate HyCon instructions and is aware of the circuit design at the same time.

The hycon module also includes an interpreter for the HyCon serial stream "protocol". See `replay` for further details.

4.1.1 Logging and Debugging

There are several ways to inspect what HyCon is doing. One of the simplest is to activate logging on `INFO` level:

```
>>> import logging
>>> logging.basicConfig(level=logging.INFO)
>>> # proceed here as usual, i.e.: hc = HyCon(...)
```

hycon.HyCon.**ensure**(*var*, ***q*)

This is our assert function which is used widely over the code for dynamic parameter checking. `q` stands for *query*. The function will return silently if the query is fullfilled and raise a `ValueError` otherwise. Examples for success:

```
>>> ensure(42, eq=42)
>>> ensure("foo bar", re="fo+.*")
>>> ensure(17, inrange=(0,20))
>>> ensure("x", within="xyz")
>>> ensure("bla", length=3)
>>> ensure("blub", isa=str, length=4, re="b.*")
```

And in case of failure: >>> ensure(3, within=[1,2,9]) Traceback (most recent call last): ... ValueError: Got var=3, but it is none of [1, 2, 9].

class hycon.HyCon.**expect**(***q*)

`ensure` delayed and on stereoids: Can be initialized with a `query` (but with further options) and then called with a `HyConRequest`. Will check the *response* and also allows *return value mapping*, for instance with regexpes or by splitting. Example:

```
>>> R = HyConRequest("dummy")
>>> R.response = "1,2,3"
>>> E = expect(split=",", type=int)
>>> print(list(E(R)))
[1, 2, 3]
```

hycon.HyCon.**wont_implement**(*reason*)

Will not implement: Returns a function which raises `NotImplementedError(reason)` when called.

class hycon.HyCon.**HyConRequest**(*command*, *expected_response=None*)

A HyConRequest models a single *request* and *response* cycle. It stores the ASCII `command` emitted by the HyCon and can save a *expected response* future/promise (see `expect` class). A HyConRequest can only be made once. If you want to do it several times, you have to (deep) copy the instance.

write(*hycon*)

Run this request. Can only be executed once. Can be chained.

read(*hycon*, *expected_response=None*, *read_again=False*)

Read response from HyCon. If *read_again* is given, will read several times. Can be chained.

class hycon.HyCon.**HyCon**(*fh*, *unidirectional=False*)

Low-Level Hybrid Controller OOP interface, similar to the Perl Hybrid controller.

This is a minimalistic implementation which tries to implement all neccessary checking of input/output request/reply structure correctness, but won't do any *high level* support for applica-

tions. Users are assumed to write such code on themselves. The PyFPAA library is an example for a high level "frontend" against HyCon, which includes a circuit understanding, etc.

query(**args*, ***kwargs*)
Create a request, run it and check the reply

command(*, *help=None*, ***kwargs*)
Return a method which, when called, creates a request, runs it and checks the reply

ic()
Switch AC to IC-mode

op()
Switch AC to OP-mode

halt()
Switch AC to HALT-mode

disable_ovl_halt()
Disable HALT-on-overflow

enable_ovl_halt()
Enable HALT-on-overflow

disable_ext_halt()
Disable external HALT

enable_ext_halt()
Enable external HALT

repetitive_run()
Switch to RepOp

single_run()
One IC-OP-HALT-cycle

pot_set()
Activate POTSET-mode

single_run_sync()
Synchronous run (finishes after a single run finished). Return value is true if terminated by ext. halt condition

set_ic_time(*ictime*)
Sets IC (initial condition) time in MILLISECONDS.

set_op_time(*optime*)
Sets OP (operation mode) time in MILLISECONDS

get_data()
Supposed to be called when a read out group is defined and the machine is in (synchronous) OP mode.

read_element_by_address(*address*)
Read any machine element voltage. Expecting 16-bit element address as integer.

set_ro_group(*addresses*)
Defines a read out group, expects addresses to be an integer list of 16-bit element addresses.

read_ro_group()
Query for currently set read out group

read_digital()
Read digital inputs

digital_output(*port*, *state*)
Set digital output pins of the Hybrid Controller

set_xbar(*address*, *config*)
Exactly {self.XBAR_CONFIG_BYTES*2} HEX-nibbles are required to config data.

read_mpts(**kw*)
Not implemented because because it is just a high-level function which calls pot_set and iterates a list of potentiometer address/names.

set_pt(*address*, *number*, *value*)
Set a digital potentiometer by address/number.

read_dpts()
Asks the Hybridcontroller for reading out *all* DPTs in the machine (also DPT24 modules). Returns mapping of PT module to list of values in that module.

get_status()
Queries the HybridController about it's current status. Will return a dictionary.

get_op_time()
Asks about current OP time

reset()
Resets the HybridController (has no effect on python instance itself)

4.1.2 Connection managers

Connection or "backends" for the PyHyCon.

The `HyCon.HyCon` class requires a file handle to be passed. Usually, file APIs are cursed in many languages (also python), but you can get your way out with the following examples and also classes in this module.

Tested or "proven" connection interfaces are:

- *`tcpsocket`* (page 104): A small adapter for the `socket()` python builtin.
- *`human`* (page 104): A small dummy adapter which prints to the interactive user terminal session and expects commands from there (the naming is ironically pointing to the human acting as actual Hybrid controller hardware endpoint).

Somewhat experimental but known to work is especially for unidirectional access:

- `StringIO.StringIO`: Circumventing file access by reading from/to strings.
- `sys.stdout` for just dumping HyCon-generated instructions

Note: All functions in this module do some progress reporting if you enable python logging facilities. Do so with

```
>>> import logging
>>> logging.basicConfig(level=logging.INFO)
```

Usage Examples

The following examples are suitable to be run in an interactive python REPL to explore the APIs.

Using PyHyCon with a microcontroller "simulator"

```
>>> from hycon import HyCon
>>> ac = HyCon(human())
>>> ac.set_ic_time(1234)
<< Sending [C001234] to uC
[type reply of uC]>> T_IC=1234
HyConRequest(C001234, expect(eq: T_IC=1234), self.executed=True,␣
↪response=T_IC=1234, reply=T_IC=1234)
```

Using PyHyCon only for writing firmware commands

```
>>> import hycon, sys
>>> ac = hycon.HyCon(sys.stdout, unidirectional=True)
>>> ac.set_ic_time(234)
C000234HyConRequest(C000234, expect(eq: T_IC=234), self.executed=True,␣
↪response=n.a., reply=n.a.)
```

Such a unidirectional approach can be interesting when generating bitstreams, for larger integration tests, etc.

Using PyHyCon over TCP/IP

```
>>> sock = tcpsocket("localhost", 12345)
>>> ac = HyCon(sock)
>>> ac.reset()
>>> ac.digital_output(3, True)
>>> ac.set_op_time(123)
>>> ac.set_xbar(0x0040, "00000002108400007818")
```

This setup is particularly interesting when connecting network-transparently to actual hardware. The target TCP server is expected to route the contents to a serial port/USB UART without introducing buffering. Examples for this kind of stub servers are given at **networking-hc_**.

Using PyHyCon over Serial

```
>>> fh = serial("/dev/ttyUSB0", 115200)
>>> ac = HyCon(fh)
>>> ac.digital_output(3, True)
>>> # etc.
```

You are encouraged to use the *serial* (page 104) class, which uses PySerial[81] under the hood and does the clearing/resetting of the stream for you (something which is more cumbersome over serial then over TCP).

[81] https://pythonhosted.org/pyserial/

If you really want, you can also use PySerial directly:

```
>>> import Serial from serial
>>> fh = Serial("/dev/ttyUSB0", 115200)
>>> ac = HyCon(fh)
```

Note that this approach suffers from binary/string conversions, but you could probably wrap `open(fh)` in some text mode.

If you (also) do not like PySerial, you can connect to a char device on a unixoid operating system with vanilla python (this example is kind-of-untested):

```
>>> import os
>>> fd = os.open("/dev/ttyUSB0", os.O_RDWR | os.O_NOCTTY)
>>> fh = os.fdopen(self.fd, "wb+", buffering=0)
>>> ac = HyCon(fh)
```

In this case, you certainly want to set the connection parameters (baud rate, etc.) by `ioctl`, for instance in before on your linux terminal using a command like `stty -F /dev/ttyUSB0 115200`, or with `stty ospeed 115200` and `stty ispeed 115200` on Mac. Furthermore, when using this approach, consider writing a small wrapper which runs `fh.flush()` after writing.

`hycon.connections.`**`repeated_reset`**(*fh*)

This routine tries to *clear* output buffers of the hycon UART by sending repeated reset instructions and waiting until the reply "RESET" appears on the line. Doing this, it implements the HyCon protocol, but the `HyCon` code does not deal with connection issues, which is why this function is aprt of `connections`.

Calling this function on beginning the setup is recommended for direct serial connections.

You can also call to this method with the **:fun:`HyCon.HyCon.repeated_reset()`** shorthand.

This function returns `True` when the connection suceeded, else `False`.

class `hycon.connections.`**`human`**

Dummy IOWrapper for testing HyCon.py without the actual hardware

class `hycon.connections.`**`tcpsocket`**(*host*, *port*)

Wrapper for communicating with HyCon over TCP/IP. See also HyCon-over-TCP.README for further instructions

`write`(*sth*)

Expects sth to be a string

class `hycon.connections.`**`serial`**(*port*, *baudrate*, ***passed_options*)

Small wrapper for making the use of PySerial more handy (no need for extra import)

4.1.3 Autosetup features

The autosetup module of the hycon package is the python implementation of the similar named feature of the Perl HyCon library.

It is used to setup a hybrid controller based from a YAML file which includes a mapping from names to computing element and potentiometer addresses and a problem description containing information about timing, potentiometer values (coefficients) and a readout group of interest. It can also describe the configuration of an XBAR module.

The idea of this YAML file is to describe the analog circuit as complete as possible, to keep the steering hycon code in perl (or python, respectively) short. Furthermore, it brings some kind of highlevel description of the circuit, since many parts of the circuit are given names.

This idea is some intermediate idea to the pyFPAA code which I wrote. It can be seen as an alternative high-level frontend to pyHyCon. Remember, the idiom of pyHyCon is to provide only a lowest level API for interaction with the hardware hybrid controller.

About the history of this code: Bernd started to write his auto-setup code at 25-DEC-2019. I started to write my pyFPAA code at the same time. Roughly one year later, where most of the time was spent at other stuff, I now port parts of Bernds auto-setup code to python in order to be able to use the same YAML files.

class hycon.autosetup.**DotDict**

Small syntactic sugar: Dot notation to access to dictionary attributes, which is especially handy for deeply nested dicts. There are plenty of similar library for python around, but this implementation is only five lines (yes, five). The following usage example is longer then the implementation:

```
>>> a = { "b": 42, "non-identifier": 3, "foo": { "bar": { 3: 123 } }}
>>> a = DotDict(a)
>>> a.b
42
>>> a.foo.bar
{3: 123}
>>> a["non-identifier"]    # traditional __getitem__ access is still␣
↪possible
3
>>> a.foo.bar[3]           # especially hand for non-pure-ascii␣
↪identifiers
123
>>> DotDict(DotDict(a)).b # DotDict can be applied repeatedly without␣
↪loss of functionality
42
>>> c = DotDict()
>>> c.foo = "b"
>>> c                      # also works for setting, not only reading
{'foo': 'b'}
>>> c.bar = DotDict()
>>> c.bar.baz = "bla"      # Limitation for nested setting: create␣
↪nested DotDicts first.
```

class hycon.autosetup.**PotentiometerAddress**(*address*, *number*)

Stores a potentiometer address, which is a tuple of a (typically hex-given) bus address of the hardware element and an element-internal number. Example:

```
>>> a = PotentiometerAddress(0x200, 0x20)
>>> b = PotentiometerAddress.fromText("0x200/20")  # FIXME: Is number␣
↪really base 16?
>>> a == b
True
>>> a.address   # Don't forget that python standard numeric output is␣
↪in decimal
512
>>> a.toText()
'0x200/20'
```

classmethod fromText(*text*)

Parses something like 0x200/2 to (0x200, 2). Will also accept 0200/2 as hex.

hycon.autosetup.**autosetup**(*hycon*, *conf*, *reset=True*)

hycon is expected to be an instance of HyCon. conf is expected to be a dictionary.

If you want to load from a YAML file, use the yaml_load function.

TODO: XBAR support not yet implemented.

hycon.autosetup.**autoconnect**(*conf*)

Opens a file handle to the target position found in the YAML file. Follows the same rules as the perl routine, i.e. looks for *serial* or *tcp* key and connects according to the parameters.

Example serial port configuration:

```
serial:
  port: /dev/cu.usbserial-DN050L21
  bits: 8
  baud: 250000
  parity: none
  stopbits: 1
  poll_interval: 10
  poll_attempts: 20000
```

Note that we only take into account port and baud rate, since everything else looks standard and the pySerial port cannot deal with an integer stopbit `1` but expects something like `serial.STOPBITS_ONE`. As a note to the future, https://tools.ietf.org/html/rfc2217.html is supported by pySerial and should be adopted by the YAML definition.,

Example TCP port configuration:

```
tcp:
    addr: 192.168.31.190
    port: 12345
    connection_timeout: 2
    timeout: 0.1
    poll_interval: 10
    poll_attempts: 2000
    quick_start: False
```

Again, we only take into account the IP address and the TCP port, everything else is ignored for the time being.

class hycon.autosetup.**AutoConfHyCon**(*conf*)

Syntactic sugar to provide a "setup" method similar to the perl HyCon API.

TODO: Should also provide other methods for high level value read and set access.

`conf` can either be a string holding the YAML filename or a dictionary (holding the configuration content, i.e. parsed YAML file).

get_data_by_name()

Get readout group data handily labeled by name

set_pt_by_name(*name*, *value*)

Set a digital potentiometer by name

read_element_by_name(*name*)

Reads element by name

read_dpts_by_name()
Asks the Hybridcontroller for reading out *all* DPTs in the machine (also DPT24 modules). Returns single map of DPT name to value (as float).

read_ro_group_by_name()
Returns an OrderedDict of read-out group elements, with names

4.1.4 Protocol Replay features

A HyCon command stream interpreter.

Will spill out LISP-like commands which can be fed into the hycon again. This allows for replaying, which is helpful for a number of special scenarios such as:

- Offline-validating a HyCon instruction stream
- Man-in-the-middle inspecting an HyCon instruction stream
- Validating the correctness of high-level HyCon instructions (such as emitted by the PyFPAA or autosetup codes)

The code basically implements a character-by-character tokenizer/parser. It is built based on a simple mapping datastructure which assigns each one-letter command the respective PyHyCon method name. Furthermore, method arguments can be read and converted.

It would be nice to join HyCon.py and replay.py to a single file which translates between the serial protocol and the OOP API calls. The transformation is quite trivial, but now we have a lot of code doing nothing of bigger interest.

The ordering follows the AVR Ino code.

hycon.replay.**delayed**(*static_method*)
This is a decorator for a static method in a class. It will make the function body "delayed", i.e. when calling the function, a future/promise/delay/deferred element is returned. Some parameter bounding (closure) happens: The decorated arguments are evaluated early while the later execution expects only a single reader argument. Example to follow the logic:

```
>>> f = lambda a,b,c,d: print(a,b,c,d)
>>> g = delayed(f)
>>> h = g(1,2,3)
>>> h(4)
4 1 2 3
```

class hycon.replay.**consume**
Consume is an ugly namespace and not a class, actually. The basic idea of these (static!) functions is to be called delayedly with a function as it's argument which acts like the io.IOBase.read() function, i.e. advances an internal cursor (side effect) and returns *n* characters from the stream. Crude Example:

```
>>> tokenizer = [consume.exact("test"), consume.decimals(3), consume.
↪exact("foo"), consume.hex(2)]
>>> test = io.StringIO("test123fooAA")
>>> reader = test.read
>>> [ token(reader) for token in tokenizer ]
['test', 123, 'foo', 170]
```

number (*digits*, *base*, *multiply=1*)

Reads a number with #digit digits in some base. Can perform a multiplication afterwards.

```
>>> consume.number(8,16)(io.StringIO("deadbeef").read)
3735928559
>>> consume.number(2,10,multiply=2)(io.StringIO("42").read)
84
```

list (*split*, *end*, *digits*, *base*)

Reads a list of numbers. Limitations: * Always expects end token to come * All numbers must have same number of digits (and same base) * Cannot handle empty list or and won't accept end-of-file before end token.

Examples:

```
>>> consume.list(split=",",digits=1,base=10,end=".")(io.StringIO(
↪"1,5,2,3,9.").read)
[1, 5, 2, 3, 9]
>>> consume.list(split=":",digits=2,base=16,end=";")(io.StringIO(
↪"5a:88:ff:ff;").read)
[90, 136, 255, 255]
```

class hycon.replay.**HyConRequestReader**(*stream_or_string*, *mapping={'?': ('NOT_IMPLEMENTED', 'Prints help'), 'A': 'enable_ovl_halt', 'B': 'enable_ext_halt', 'C': ('set_ic_time', <function delayed.<locals>.decorated.<locals>.deferred>), 'D': ('digital_output', <function delayed.<locals>.decorated.<locals>.deferred>, True), 'E': 'single_run', 'F': 'single_run_sync', 'G': ('set_ro_group', <function delayed.<locals>.decorated.<locals>.deferred>), 'L': ('NOT_IMPLEMENTED', 'Locate a computing element'), 'P': ('set_pt', <function delayed.<locals>.decorated.<locals>.deferred>, <function delayed.<locals>.decorated.<locals>.deferred>, <function delayed.<locals>.decorated.<locals>.deferred>), 'R': 'read_digital', 'S': 'pot_set', 'X': ('set_xbar', <function delayed.<locals>.decorated.<locals>.deferred>, <function delayed.<locals>.decorated.<locals>.deferred>), 'a': 'disable_ovl_halt', 'b': 'disable_ext_halt', 'c': ('set_op_time', <function delayed.<locals>.decorated.<locals>.deferred>), 'd': ('digital_output', <function delayed.<locals>.decorated.<locals>.deferred>, False), 'e': 'repetitive_run', 'f': 'read_ro_group', 'g': ('read_element_by_address', <function delayed.<locals>.decorated.<locals>.deferred>), 'h': 'halt', 'i': 'ic', 'l': 'get_data', 'o': 'op', 'q': 'read_dpts', 's': 'get_status', 't': 'get_op_time', 'x': 'reset'}*)

Converts HyCon "configuration strings" to high level API calls. This can be seen as the inverse operation to calling the HyCon.

Instances of this class act as iterator. It will consume the incoming stream character by character (or the whole string, if given as a string).

Example:

```
>>> instructions = 'C000100c015000P0200000204P0300030000G0362;0363;
↪0220;0221;0222;0223.'
>>> commands = list(HyConRequestReader(instructions))
>>> for c in commands: print(c)
('set_ic_time', 100)
('set_op_time', 15000)
```

(continues on next page)

(continued from previous page)

```
('set_pt', 512, 0, 0.19941348973607037)
('set_pt', 768, 3, 0.0)
('set_ro_group', [866, 867, 544, 545, 546, 547])
>>> replayed = io.StringIO()
>>> hc = HyCon(replayed, unidirectional=True)
>>> replay(hc, commands)
>>> replayed.getvalue() == instructions
True
```

hycon.replay.**replay**(*hycon*, *commands*)

Given commands a list of tuples, this will mostly act like operator.methodcaller on them. If no arguments are given, the tuple can be omitted.

Basic example:

```
>>> class WannaBeHyCon:
...     def toot(self,x): print("too(%s)ooot" % x)
...     def bar(self): print("this is bar")
...     def buz(self,a,b,c): print(f"a*b = {a*b} but what is {c}")
>>> replay(WannaBeHyCon(), [ "bar", ("toot", "fuz"), ("buz", 1,2,3) ])
this is bar
too(fuz)ooot
a*b = 2 but what is 3
```

The command format is produced by the HyConRequestReader and thus can be fed into this replay function:

```
>>> replay(HyCon(sys.stdout, unidirectional=True), HyConRequestReader(
↪"xiohaARt"))
xiohaARt
```

This works for almost any useful instruction stream.

4.2 HyCon Serial (USB/RS-232) over TCP/IP

On the machine where the HyCon board is attached to, run something like `socat` or for instance the `tcp_serial_redirect.py` which is shipped with `PySerial` and can be obtained from https://raw.githubusercontent.com/pyserial/pyserial/master/examples/tcp_serial_redirect.py

Other Unix binaries which can be used are for instance `socat` https://bloggerbust.ca/post/let-socket-cat-be-thy-glue-over-serial/ or `netcat`.

Next, put a script like the following at that machine (now refered to as the server):

```
#!/bin/bash -x
source ~/.bash_profile
cd "$(dirname $0)"
/usr/local/bin/python3 tcp_serial_redirect.py -P 12345 /dev/cu.usbserial-
↪DN050L1O 115200
```

Then, from the client machine, run something like

```
ssh -L 12345:localhost:12345 ac /path/to/the/ac-bridge/run-bridge.sh
```

Where `run-bridge.sh` contains that script above. Now you can connect to TCP port `12345` at the client machine in order to speak directly with the HyCon. This works remarkably well, also TCP buffering or network latency is not a problem at all for me even with Bernds slow internet connection :-)

Note: Known bugs: When quitting the ssh session, the bridge is sometimes not killed properly. In this case, log into the server and kill the script directly, for instance with the following command:

```
ps aux | grep tcp_serial_redirect | awk '{print $2}' | xargs kill
```

4.3 Analog data aquisition

The *hycon.aquisition* (page 111) package collects various codes related to *data aquisition*. Currently, we have a number of pythonic analog2digital data auqisition routines:

- The Hybrid Controller uC itself offers methods such as *hycon.HyCon.HyCon.read_ro_group()* (page 101) and similar. The aquisition is high precision (16bit) but quite limited in time, given the small memory of the uC which is used for store the full aquisition before a transfer is possible (no streaming implemented).
- DSOs (digital storage oscilloscopes) can be used for high quality and industry standard data aquisition. A particular python example is provided in the *siglent_scpi.siglent* (page 113) code. These devices are typically also limited in aquisition time, which is naturally mapped to the display. Furthermore, these devices are always very limited in the number of analog channels, and devices with many channels are expensive.
- Custom Data loggers provide maximum flexibilty, given the large amount of channels implementable with cheap microcontrollers such as the Teensy[82]. However, these cheap uCs typically only have an average resolution of 10bit. A particular example is given at https://github.com/anabrid/TeensyLogger (see also https://the-analog-thing.org/wiki/Teensy).

When it comes to classical analog computing, one would like to optimize the aquisition for high resolution (16bit), many channels (as many as possible). What is not so important is the sampling rate, since the cutoff frequency of the discrete circuits is rather low (at the order of MHz).

4.3.1 Siglent SCPI

Pythonic API for Siglent Digital Oscilloscope in order to perform data aquisition. This is a python standalone code implementing the SCPI protocol (without further Python libraries) over sockets. The connection is assumed to be established over Ethernet (not USB, despite this should also work, in principle, as the siglent registers as UART device over USB). The code can work in blocking and non-blocking fashion (but is not fully async).

See the programming guide https://siglentna.com/wp-content/uploads/dlm_uploads/2021/01/SDS1000-SeriesSDS2000XSDS2000X-E_ProgrammingGuide_PG01-E02D.pdf for the reference. This code was tested against an Siglent SDS1104X-E with four 10bit channels.

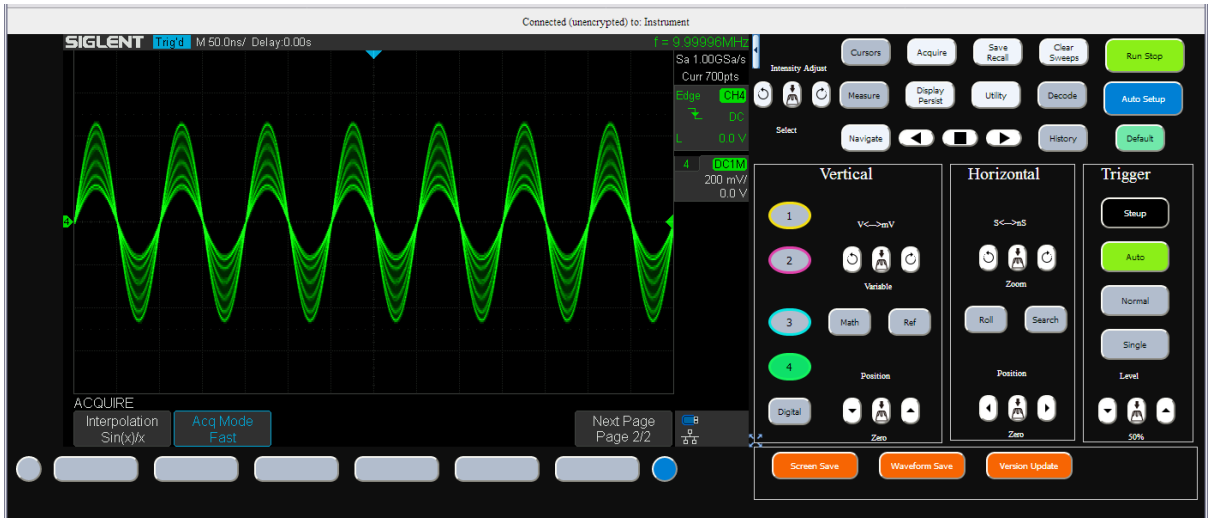
Here is a picture of such an oscilloscope. On the back, it has an RJ45 ethernet connector and a USB client socket. Whether over USB-UART or TCP, it allows to be programmed with the Standards Commands for Programmable Instruments (SCPI)[83] standard, which is basically a simple command-reply text protocol

[82] https://www.pjrc.com/teensy/

[83] https://en.wikipedia.org/wiki/Standard_Commands_for_Programmable_Instruments

similar to the one we implement with the *hycon.HyCon* (page 99).

Over ethernet, one can also access the oscilloscope screen via VNC[84]. The oscilloscope provides a web interface which includes an in-browser VNC client and models the interface controls. Both is shown side on side in the UI. The interface knobs are translated to SCPI commands via Javascript. The status indicators are polled. The overall UI is quite heavy, rather slow and generates, in particular, high load on the oscilloscope.



[84] https://en.wikipedia.org/wiki/Virtual_Network_Computing

Warning: Make sure you **never ever run the browser interface at the same time as a telnet session or this script**. This will break the oscilloscope randomly and requires you to cold start it. This is a reproducable bug in the Siglent firmware.

It is, however, fine to run a VNC connection to the oscilloscope while running this script.

This script is also executable as standalone and offers an API to contact the siglent and do a single readout, assuming the triggering took place independently.

`hycon.aquisition.siglent_scpi.`**`downsample`**(*data*, *factor*)

Downsample all numpy 1d arrays, since `query("MEMORY_SIZE 70K")` has no effect... TODO: MEMSIZE only works in STOP mode.

`hycon.aquisition.siglent_scpi.`**`write_npz`**(*data*, *fname=None*)

Write out NPZ files. See https://numpy.org/doc/stable/reference/generated/numpy.savez.html for this numpy specific binary file format. Expects data to be a python dictionary (ideally some `np.array`).

`class` `hycon.aquisition.siglent_scpi.`**`siglent`**(*host*, *port=5025*)

This is a classy API to the oscilloscope. A usage example intermixed with HyCon controlling may look like this:

```
import hycon.aquisition.siglent_scpi as sig

hc = AutoConfHyCon("something.yml")
print(hc.get_status())

siglent = sig.siglent("192.168.32.68")
print("Siglent Status: ", siglent.status())

hc.ic()
print("ICs freely running, preparing Siglent trigger...")
#input("Press enter to continue")
prepped = siglent.trigger_single()
siglent.log("Siglent Status: ", siglent.status())
if not prepped:
    raise ValueError("Could not properly prepare the Siglent.")
time.sleep(1)
hc.single_run_sync()
time.sleep(2) # 2 seconds "OP time".
print("Finished with single run sync. Waiting another second for␣
↪siglent.")
time.sleep(1)
hc.ic()
#input("HC stopped. Press enter to readout siglent.")
data = siglent.read_all()
sig.write_npz(data, f"output.npz")
```

Warning: While this code basically works, the refactoring at PyAnalog-introduction time was not tested. That is, this code is not yet fully tested.

`send`(*md*, *log_end='\n'*, *blocking=False*, *sleep_sec=0.5*)

Send SCPI command to siglent.

query(*sleep_sec=0.5*)
Query the siglent for something. This is basically a *send()* (page 113)/`recv()` cycle, according to the SCPI standard.

query_num(*pat*)
A shorthand which returns the number in some readout which looks like `a = 123`.

status()
Reads out the status bit (`INR?`) and translates it according to the manual, thus returning a string.

> **Warning:** Attention, Querying INR? is *NOT* a read-only operation but changes the oscilloscope status.

trigger_single()
Brings the Oscilloscope in single trig mode and checks the success of this operation (returns `True` in case of passed check).

read_wf(*channel*, *try_multiples=2*)
Read the actual waveform. This can be called after an aquisition took place and the oscilloscope trigger fired. In many times, this will fail and return no data. This results in a thrown `ValueError`. Otherwise, data recieving is looped (with lot's of logging since it takes some seconds). When some consistency is reached (all announced data is recieved), will return the *raw recieved channel data* as 1d list of numbers. That is, these are 10bit numbers.

This function will by default try several times to ask for data, in case the siglent did not properly answer.

read_all(*channels=None*)
Read the actual waveforms by querying the passed channel list. A channel is an integer between `0` and `4`, see `siglent.channels` for the valid channel list. If not provided, all channels are read.

Will do the conversion to voltages, that is, returns proper floats representing a voltage in unit V. Returns a dictionary mapping the channel name to the 1d data.

Appendix

A.1 About this documentation

This documentation is written in ReStructured text[85] (RST) and rendered using the Sphinx[86]. documentation system. Possible output formats are not only web pages (HTML), but also PDF, amongst others. If you have sphinx installed (`pip install sphinx`), you can just go to the `doc` directory and type `make html` or `make pdf` to generate the docs locally. If you have a mixed Python2/Python3 system, call `make html SPHINXBUILD="python3 -msphinx"` to ensure using Python3.

If you want to start editing/improving this documentation, you might want to read the ReStructuredText Primer[87]. A handy tool for hot reloading (regenerating) the documentation during editing is Sphinx-Reload[88]. After installation, just run `sphinx-reload doc/` from the root directory and point your browser to an address such as http://localhost:5500/.

This documentation is automatically updated/built at Git commit/push time by our Gitlab Continous Integration infrastructured and uploaded as static files to our Anabrid Dev Server[89] (formally Read The Docs[90]). you can find the docs at:

- https://anabrid.dev/pyanalog/dirhtml/ primary link
- https://anabrid.dev/pyanalog/latex/pyanalog.pdf for a +100 page printable/downloadable/searchable PDF version
- Various different versions for download at https://anabrid.dev/pyanalog/

[85] https://en.wikipedia.org/wiki/ReStructuredText
[86] https://www.sphinx-doc.org/
[87] https://docutils.sourceforge.io/docs/user/rst/quickstart.html
[88] https://github.com/prkumar/sphinx-reload
[89] https://www.anabrid.dev/
[90] https://readthedocs.org/

A.2 Indices and tables

- Python Function/Class index
- Python module index

Python Module Index

d

f

h

Index

F

G

H

I

L

M

N

O

P

U

V

W