glyph

Release 0.5.3

Jan 22, 2020

Contents

glyph is a python 3 library based on deap providing abstraction layers for symbolic regression problems.

It comes with batteries included:

- predefined primitive sets
- n-dimensional expression tree class
- symbolic and structural constants
- interfacing constant optimization to scipy.optimize
- easy integration with joblib or dask.distributed
- symbolic constraints
- boilerplate code for logging, checkpointing, break conditions and command line applications
- rich set of algorithms

glyph also includes a plug and play command line application **glyph-remote** which lets non-domain experts apply symbolic regression to their optimization tasks.

> **Warning:** While fully usable, glyph is still pre-1.0 software and has **no** backwards compatibility guarantees until the 1.0 release occurs!

CHAPTER 1

Content

1.1 Getting Started

1.1.1 Installation

Glyph is a **python 3.5+** only package.

You can install the latest stable version from PyPI with pip

```
pip install pyglyph
```

or get the bleeding edge

```
pip install git+git://github.com/ambrosys/glyph.git#egg=glyph
```

1.1.2 Examples

Examples can be found in the repo. To run them you need to:

- Clone the repo.
- `make init`
- `cd examples`
- Run any example, e.g. `python lorenz.py --help`

1.2 Concepts

Glyph has several abstraction layers. Not all of them are required to use.

1.2.1 Individual & genetic operators

This wraps around the backend, which is currently deap. In contrast to deap, the individual class has to be associated with a primitive set. This makes checkpointing and later evaluation of results easier.

This abstraction layer also allows for an interchangeable representation. We plan to support graphs and stacks in the future.

Genetric operators mutation and crossover operators.

Currently, we also rely on deaps sorting algorithms.

Creating an individual class is as simple as:

```
from glyph.gp.individual import AExpressionTree, numpy_primitive_set

pset = numpy_primitive_set(1)

class Individual(AExpressionTree):
    pset = pset
```

Here, we use the convinience function `numpy_primitive_set` to create a primitive set based on categeories.

1.2.2 Algorithm

This encapsulates selecting parents and breeding offspring.

Glyph comes with the following algorithms:

- AgeFitness Pareto Optimization
- SPEA2
- NSGA2
- and the "unique" counterparts of all of the above.

Algorithms need the genetic operators. The chose to implement them as classes. You can change the default parameters by simply overwriting the corresponding attribute. All algorithms only expose a single method `evolve(population)`. This assumes all individuals in the population have a valid fitness. `evolve(population)` will first select the parents and then produce offspring. Both, parents and offspring will be returned by the method. By doing so, so can re-evaluate the parent generation if desired (e.g. to account for different operating conditions of an experiment).

```
from functools import partial
import deap
from glyph import gp

mate = deap.gp.cxOnePoint
expr_mut = partial(deap.gp.genFull, min_=0, max_=2)
mutate = partial(deap.gp.mutUniform, expr=expr_mut, pset=Individual.pset)
algorithm = gp.NSGA2(mate, mutate)
```

1.2.3 AssessmentRunner

The AssesmentRunner is a callable which takes a list of Individuals and assigns a fitness to them. This can be as simple as:

```
def meassure(ind):
    g = lambda x: x**2 - 1.1
    points = np.linspace(-1, 1, 100, endpoint=True)
    y = g(points)
    f = gp.individual.numpy_phenotype(ind)
    yhat = f(points)
    if np.isscalar(yhat):
        yhat = np.ones_like(y) * yhat
    return nrmse(y, yhat), len(ind)

 def update_fitness(population, map=map):
     invalid = [p for p in population if not p.fitness.valid]
     fitnesses = map(meassure, invalid)
     for ind, fit in zip(invalid, fitnesses):
         ind.fitness.values = fit
     return population
```

`update_fitness` is taken directly from the deap library. You can interface your symbolic regression problem by providing a different `map` function. The recommenced solution is scoop. Why this does not work in most cases see *Parallel*. Which can be a bit cumbersome to write for more complex problems.

The glyph.assessment submodule has many out of the box solutions for boilerplate/utility code, constant optimization and integration multiprocessing/distributed frameworks.

The code above with constant optimization simply becomes:

```
class AssessmentRunner(AAssessmentRunner):
    def setup(self):
        self.points = np.linspace(-1, 1, 100, endpoint=True)
        self.g = lambda x: x**2 - 1.1
        self.y = self.g(self.points)

    def measure(self, ind):
        popt, error = const_opt_scalar(self.error, ind)
        ind.popt = popt
        return error, len(ind)

    def error(self, ind, *consts):
        f = numpy_phenotype(ind)
        yhat = f(self.points, *consts)
        return nrmse(self.y, yhat)
```

Algorithm and assessment runner already make up a program:

```
runner = AssessmentRunner()
pop = Individual.create_population(lambda_)
runner(pop)

for i in range(generations):
     pop = runner(algorithm(pop))
```

1.2.4 GPRunner

The GPRunner lets you conveniently steps cycle through the evolutionary algrithm whilst taken care for statistics and a hall of fame.

It's mostly syntatic sugar:

```
gp_runner = GPRunner(Individual, lambda: algorithm, AssessmentRunner())
gp_runner.init()
for i in range(generations):
    gp_runner.step()
```

1.2.5 Application

If you want a command line interface for all your hyper-parameters, checkpointing, ensuring random state handling on resume, as well as breaking conditions, the glyph.application submodule has you covered.

The module provides several facory classes which can dynamically expand an existing `argparse.ArgumentParser`. As a starting point, you can use the `default_console_app` to create an app. You will only need a primitive set and an assessment runner as explained above.

```
parser = argparse.ArgumentParser(program_description)

app, args = application.default_console_app(Individual, AssessmentRunner, parser)
app.run()
```

For more involved applications you can inherit from the Application class. (see `/../../glyph/cli/glyph_remote.py`).

We recommence having a look at the `/../../examples/control/minimal_example.py` as well as the `/../../examples/control/lorenz.py` example to see these concepts in action.

1.3 Glyph remote

glyph-remote is shipped together with the glyph package. After installation, the `glyph-remote` command is available at the command line.

1.3.1 Concept

With glyph-remote the separation between optimization method and optimization task is made easy. glyph-remote runs multi IO symbolic regression and sends candidate solution via ZeroMQ to an experiment controller for assessment. Every hyper-parameter used is assessable and fully configurable.

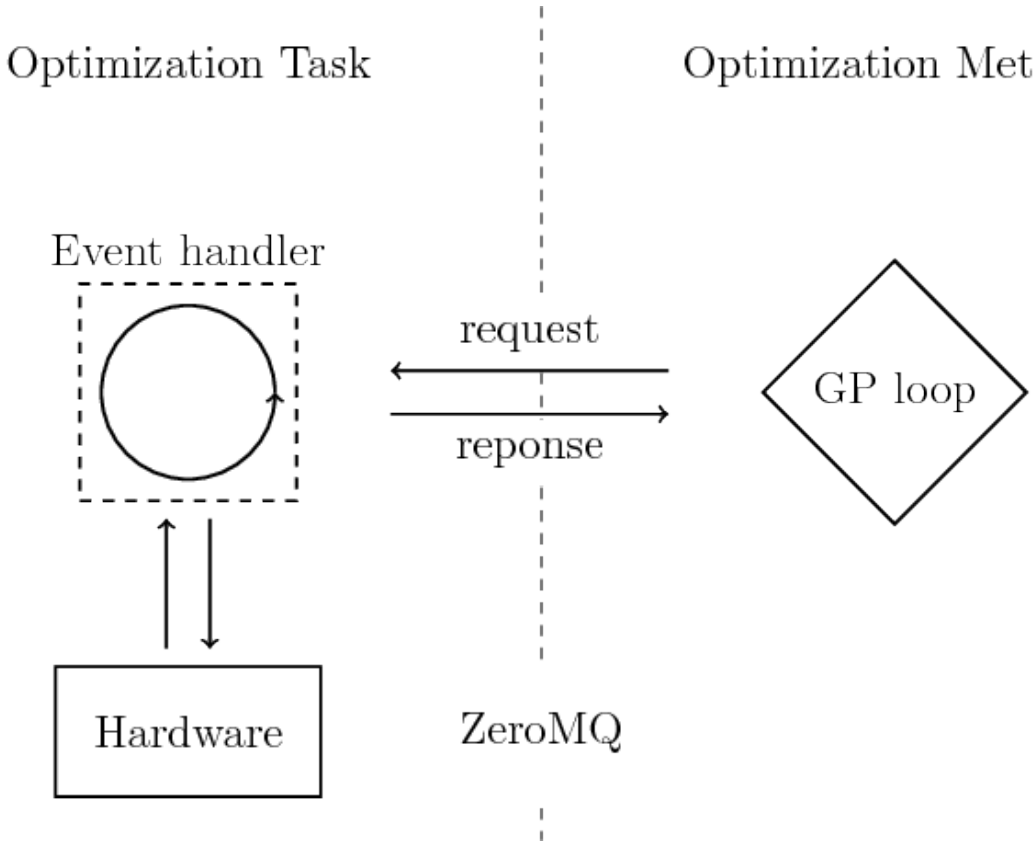
1.3.2 Overview

To the right the optimization method is represented. The GP program can be seen as a black box which is only accessible by the specified interface. To the left a single experiment plus an event handler is depicted. The latter glues optimization method and task together and needs to understand the communication protocol defined in

Currently we use client server sockets for glyph remote. **The user needs to implement the zmq.REP socket.**

1.3.3 Communication protocol

The communication is encoded in json. A message is a json object with two members:




```
{
    "action": "value",
    "payload": "value",
}
```

The possible action values are:

Action name	Payload	Expected return Value
CONFIG	–	config settings
EXPERIMENT	list of expressions	list of fitness value(s)
METADATA	–	any
SHUTDOWN	–	–

The config action is performed prior to the evolutionary loop. Entering the loop, for every discovered solution an *experiment* action will be requested. Since most experiments have an intermediate compiling step, expressions will come in chunks. You can configure optional caching for re-discovered solutions. The *shutdown* action will let the experiment program know that the gp loop is finished and you can safely stop the hardware.

Config

See Configuration section.

Experiment

The *experiment* request expects a fitness value for each expression:

```
{
    "fitness": ["value0", "value1", ...],
}
```

Shutdown

You can properly shut down the experiment hardware.

1.3.4 Configuration

For a full list of configuration options and their default values type `glyph-remote --help`.

All hyper-parameters and algorithms used have default values. You have three options to set parameters:

- use the command line interface
- read from file (using `--cfile myfile.yaml`)
- request from event handler (using `--remote`)

At cli, options are specified using `--key value`. The configuration file has to be written in yaml, i.e.

```
key: value
```

The event handler should send back a similar json array

```
{
  "key": "value",
}
```

It is mandatory to provide a information about the primitives you want to use. The value of the "primitives" key is again a json/yaml list specifying name: arity pairs. Arities greater one are functions, equal to one are variables and -1 is reserved for symbolic constants.

```
{
  "primitives":
  {
    "add": 2,
    "x": 0,
  },
}
```

1.3.5 GUI

Install

Glyph comes with an optional GUI to use the `glyph-remote` script with more convenience.

The GUI uses the package `wxPython`. The installation manual can be found here and Website.

Manual Gooey installtion

Since up-to-date (28.08.2018) the necessary changes to the used graphic library Gooey are not part of the master branch, it might be necessary to install Gooey by hand from this fork:

- `pip install -e "git+git@github.com:Magnati/Gooey.git#egg=gooey"`

Installation with pip installtion

To install glyph including the gui option use the following command:

To start the script with the gui just use the `--gui` parameter:

Usage

Within the GUI there is a tab for each group of parameters. If all parameters are set, click the start-button to start the experiment.

1.3.6 Pretesting & Constraints

In glyph-remote, genetic operations can be constrained. A genetic operation (i.e. every operation that create or modifies the genotype of an individual). If a constraint is violated, the genetic operation is rejected. If out of time, the last candidate is used.

Currently, two different types of constraints are implemented: - algebraic constraints using sympy - pretesting constraints

Algebraic constraints

Sympy is used to check whether expressions are:

- zero
- constant
- infinite

The three options can be individually activated.

Pretesting

You can invoke file-based pretesting with the `--constraints_pretest filename.py` flag. The flag `--constraints_pretest_function` lets you pass the function name which will be invoked to pretest individuals.

The function is expected to return a boolean, depending on the individual is rejected (False) or accepted (True).

An example file could look like this:

```
import time


def chi(ind):
    time.sleep(1)
    print(f"Hello World, this is {ind}")
    return True
```

1.4 Publications using glyph

If you use glyph please consider citing glyph:

```
@article{quade2019,
 author = {Quade, Markus and Gout, Julien and Abel, Markus},
 title = {{Glyph: {Symbolic} Regression Tools}},
 journal = {J Open Res Softw},
 year = {2019},
 month = jun,
 doi = {10.5334/jors.192},
 volume = {19},
 issue = {7(1)},
}
```

```
@misc{glyph,
 author = {Quade, Markus and Gout, Julien and Abel, Markus},
 title = {{glyph} - Symbolic Regression Tools},
 month = jan,
 year = {2018},
 doi = {10.5281/zenodo.1156654},
 url = {https://github.com/Ambrosys/glyph},
 note = {Version 0.3.5},
}
```

1.4.1 Experiments

1. El Sayed M, Y., Oswald, P., Sattler, S., Kumar, P., Radespiel, R., Behr, C., ... & Abel, M. (2018). Open- and closed-loop control investigations of unsteady Coanda actuation on a high-lift configuration. In 2018 Flow Control Conference (p. 3684).

1.4.2 Simulations

1. Gout, J., Quade, M., Shafi, K., Niven, R. K., & Abel, M. (2018). Synchronization control of oscillator networks using symbolic regression. Nonlinear Dynamics, 91(2), 1001-1021.

1.5 About glyph

glyph has been deleveloped by (alphabetical order):

- Markus Abel
- Julien Gout
- Markus Quade

at Ambrosys GmbH.

1.6 Funding Acknowledgements

The development of glyph was supported by the following:

- MQ was supported by a fellowship within the FITweltweit program of the German Academic Exchange Service (DAAD)
- German Science Foundation via SFB 880
- German Ministry for economy via ZIM projekt, Nr. KF2768302

1.7 Tutorials

1.7.1 Parallel

Pickling problems

Most parallelization frameworks rely on the built-in pickle module which has limited functionality regarding lambda expressions. Deap relies heavily on those functionalities and thus most parallelization frameworks do not work well with deap.

Dill can handle everything we need and can be monkey patched to replace pickle.

1.7.2 Labview Tutorial

Contributed by P. Oswald.

Install Python

1. Install Miniconda (Python 3.5 or higher).
2. Open a command window as administrator:
   - cd to Miniconda3 directory
   - run `conda update conda`
3. Install the numpy and scipy wheels using conda, or download them directly here and here. You can install them with `pip install path_to_wheel/wheel_file_name`.

Install Glyph

0. If you have git installed, run `pip install -e git+https://github.com/Ambrosys/glyph.git#egg=glyph`. Go to step 5.
1. Download the latest version from Github.
2. Unzip / move to somewhere useful
3. Upen a cmd window, navigate the the glyph-master folder
4. Run `pip install -e .` (don't forget the period at the end)
5. Test the installation by running `glyph-remote --help`.

Install ZeroMQ

1. Download ZeroMQ bindings for LabView from http://labview-zmq.sourceforge.net/
2. The download is a VI-Package (*.vip-file)
3. Double clicking the *.vip-file opens it in the VI Package Manager (further info http://www.ni.com/tutorial/12397/en/)
4. Use the VI Package Manager to install the package

Use ZeroMQ

1. After successful installation you can find examples on the usage of ZeroMQ either

a. through the VI Package Manager by double clicking on the entry "ZeroMQ Socket Library" and then on the button "Show Examples"

b. in your LabView installation folder in the subdirectory /examples/zeromq/examples/

c. online (e.g. the basic examples at http://labview-zmq.sourceforge.net/)

2. For communication with glyph-remote one has to implement a server that listens for requests from glyph and sends the apropriate responses
3. The ZeroMQ programming blocks can be accessed by right clicking on the block diagram and navigating to the section "Add-ons"
4. The block "Unflatten from JSON" can be used to convert the JSON encoded strings sent by glyph to LabView clusters

1.7.3 Matlab Tutorial

Contributed by B. Strom.

Note: This was performed on Windows 7 and using MATLAB R2016b (2016b or later is needed for `jasondecode()` and `jasonencode()` commands)

Install Python

1. Install Miniconda (Python 3.5 or higher).
2. Open a command window as administrator:

- cd to Miniconda3 directory
- run `conda update conda`

3. Install the numpy and scipy wheels using conda, or download them directly here and here. You can install them with `pip install path_to_wheel/wheel_file_name`.

Install Glyph

0. If you have git installed, run `pip install -e git+https://github.com/Ambrosys/glyph.git#egg=glyph`. Go to step 5.

1. Download the latest version from Github.
2. Unzip / move to somewhere useful
3. Upen a cmd window, navigate the the glyph-master folder
4. Run `pip install -e .` (don't forget the period at the end)
5. Test the installation by running `glyph-remote --help`.

Install jeroMQ (java implementation of zeroMQ)

This will be used for zeroMQ in MATLAB.

1. If you don't have it, install the Java developer kit.
2. Set the JAVA_HOME environment variable

a. Right click My Computer and select properties

b. On the Advanced tab, select Environment Variables, and then edit or create the system variable JAVA_HOME to point to where the JDK software is located, for example, `C:\Program Files\Java\jdk1.8.0_131`

3. Install Maven.

a. Add the bin directory of the created directory apache-maven-3.5.0 to the PATH environment variable (same steps as the setting the `JAVA_HOME` variable, but this is a user variable instead of a system variable)

b. Confirm installation with `mvn -v` in a command window

4. Download the latest stable release of jeroMQ.

a. Unpack the zip file

b. In a command window, navigate to the resulting jeroMQ folder

c. Run the command `mvn package`

d. This will take a while, but you should see "Build Success" when it is finished

e. This will have created a "target" directory in the jeroMQ folder. The Jar file we need is in here, something like `.../target/jeromq-0.4.1-SNAPSHOT.jar`

5. Add the path to this Jar file to MATLAB's static Java path

a. Run the command `prefdir` in MATLAB. Navigate to that folder and check for a file named `javaclasspath.txt`.

b. Open this file in a text editor or create anASCII text file named `javaclasspath.txt`.

c. On its own line, add the full path to the jar file, including the file name. You can move it or rename it first if you wish.

d. Restart MATLAB

6. To test that MATLAB can access jeroMQ, run `import org.zeromq.ZMQ` in at the MATLAB command prompt. If no error, it was successful.

Test a basic example

1.8 Development:

Know what you're looking for & just need API details? View our auto-generated API documentation:

1.8.1 glyph

glyph package

Subpackages

glyph.cli package

Submodules

glyph.cli.glyph_remote module

class Communicator(*ip*, *port*)
Bases: object

Holds the socket for 0mq communication.

Parameters

- **ip** – ip of the client
- **port** – port of the client

connect()

recv(*serializer=<module 'json' from '/home/docs/.pyenv/versions/3.6.8/lib/python3.6/json/__init__.py'>*)

send(*msg*, *serializer=<module 'json' from '/home/docs/.pyenv/versions/3.6.8/lib/python3.6/json/__init__.py'>*)

class EvalQueue(*com*, *result_queue*, *expect*)
Bases: queue.Queue

run(*chunk_size=100*)

class ExperimentProtocol
Bases: enum.EnumMeta

Communication Protocol with remote experiments.

CONFIG = 'CONFIG'

EXPERIMENT = 'EXPERIMENT'

METADATA = 'METADATA'

SHUTDOWN = 'SHUTDOWN'

class Individual(*content*)
Bases: *glyph.gp.individual.AExpressionTree*

Abstract base class for the genotype.

Derived classes need to specify a primitive set from which the expression tree can be build, as well as a phenotype method.

class NDTree(*trees*)
Bases: *glyph.gp.individual.ANDimTree*

base
alias of *Individual*

class RemoteApp(*config*, *gp_runner*, *checkpoint_file=None*, *callbacks=(<function make_checkpoint>, <function log>)*)

Bases: *glyph.application.Application*

An application based on *GPRunner*.

Controls execution of the runner and adds checkpointing and logging functionality; also defines a set of available command line options and their default values.

To create a full console application one can use the factory function default_console_app().

Parameters

- **config** (*dict or argparse.Namespace*) – Container holding all configs
- **gp_runner** – Instance of *GPRunner*
- **checkpoint_file** – Path to checkpoint_file
- **callbacks** –

checkpoint()

Checkpoint current state of evolution.

classmethod from_checkpoint(*file_name*, *com*)

Create application from checkpoint file.

run(*break_condition=None*)

For details see application.Application. Will checkpoint and close zmq connection on keyboard interruption.

class RemoteAssessmentRunner(*com*, *complexity_measure=None*, *multi_objective=False*, *method='Nelder-Mead'*, *options={'smart_options': {'use': False}}*, *caching=True*, *persistent_caching=None*, *simplify=False*, *chunk_size=30*, *send_symbolic=False*, *reevaluate=False*)

Bases: object

Contains assessment logic. Uses zmq connection to request evaluation.

evaluate_single(*individual*, **consts*, *meta=None*)

Evaluate a single individual.

measure(*individual*, *meta=None*)

Construct fitness for given individual.

predicate(*ind*)

Does this individual need to be evaluated?

recv

Backwards compatibility

send

Backwards compatibility

update_fitness(*population*, *meta=None*)

build_pset_gp(*primitives*, *structural_constants=False*, *cmin=-1*, *cmax=1*)

Build a primitive set used in remote evaluation.

Locally, all primitives correspond to the id() function.

const_opt_options_transform(*options*)

get_version_info()

handle_const_opt_config(*args*)

handle_gpconfig(*config*, *com*)
Will try to load config from file or from remote and update the cli/default config accordingly.

log_info(*args*)

main()

make_callback(*factories*, *args*)

make_remote_app(*callbacks=()*, *callback_factories=()*, *parser=None*)

send_meta_data(*app*)

update_namespace(*ns*, *up*)
Update the argparse.Namespace ns with a dictionairy up.

Module contents

glyph.gp package

Submodules

glyph.gp.algorithms module

class NSGA2(*mate_func*, *mutate_func*)
Bases: `glyph.gp.algorithms.MOGP`

Implementation of the NSGA-II algorithm as described in Essentials of Metaheuristics

class SPEA2(*mate_func*, *mutate_func*)
Bases: `glyph.gp.algorithms.MOGP`

Implementation of the SPEA2 algorithm as described in Essentials of Metaheuristics

class DeapEaSimple(*mate_func*, *mutate_func*)
Bases: `object`

Basically a copy of `deap.algorithms.eaSimple` algorithm.

evolve(*population*)

class AgeFitness(*mate_func*, *mutate_func*, *select*, *create_func*)
Bases: `glyph.gp.algorithms.MOGP`

AgeFitness algorithm as described in Schmidt & Lipson. DOI: 10.1007/978-1-4419-7747-2_8

evolve(*population*)

class UNSGA2(*mate_func*, *mutate_func*)
Bases: *glyph.gp.algorithms.NSGA2*

evlolve(*population*)

class USPEA2(*mate_func*, *mutate_func*)
Bases: *glyph.gp.algorithms.SPEA2*

evlolve(*population*)

class UDeapEaSimple(*mate_func*, *mutate_func*)
Bases: *glyph.gp.algorithms.DeapEaSimple*

evlolve(*population*)

class UAgeFitness(*mate_func*, *mutate_func*, *select*, *create_func*)
Bases: *glyph.gp.algorithms.AgeFitness*

evlolve(*population*)

glyph.gp.breeding module

mutuniform(*pset*, ***kwargs*)
Factory for mutuniform

mutnodereplacement(*pset*, ***kwargs*)
Factory for mutnodereplacement

mutshrink(*pset*, ***kwargs*)
Factory for mutshrink

mutshrink(*pset*, ***kwargs*)
Factory for mutshrink

cxonepoint(***kwargs*)
Factory for cxonepoint

cxonepointleafbiased(***kwargs*)
Factory for cxonepointleafbiased

glyph.gp.constraints module

class Constraint(*spaces*)
Bases: object

class NonFiniteExpression(*zero=True*, *infty=True*, *constant=False*)
Bases: *glyph.gp.constraints.Constraint*

Use sympy to check for finite expressions.

Parameters

- **zero** – flag to check for zero expressions
- **infty** – flag to check for infinite expressions
- **constant** – flag to check for constant expressions

class PreTest(*fn*, *fun='chi'*)
Bases: *glyph.gp.constraints.Constraint*

Apply pre-testing to check for constraint violation.

The python script needs to provide a callable fun(ind).

Parameters

- **fn** – filename of the python script.
- **fun** – name of the function in fn.

class PreTestService(*assessment_runner*)
Bases: *glyph.gp.constraints.Constraint*

com

make_str

constrain(*funcs*, *constraint*, *n_trials=30*, *timeout=60*)
Decorate a list of genetic operators with constraints.

Parameters

- **funcs** – list of operators (mate, mutate, create)
- **constraint** – instance of Nullspace

Returns constrained operators

reject_constrain_violation(*constraint*, *n_trials=30*, *timeout=60*)
Create constraints decorators based on rules.

Parameters

- **constraint** –
- **n_trials** – Number of tries. Give up afterwards (return input).

Returns list of constraint decorators

glyph.gp.individual module

Provide Individual class for gp.

class AExpressionTree(*content*)
Bases: deap.gp.PrimitiveTree

Abstract base class for the genotype.

Derived classes need to specify a primitive set from which the expression tree can be build, as well as a phenotype method.

const_opt

classmethod create(*gen_method=<function genHalfAndHalf>*, *min=1*, *max=4*)

classmethod create_population(*size*, *gen_method=<function genHalfAndHalf>*, *min=1*, *max=4*)
Create a list of individuals of class Individual.

classmethod from_string(*string*)
Try to convert a string expression into a PrimitiveTree given a PrimitiveSet *pset*. The primitive set needs to contain every primitive present in the expression.

Parameters

- **string** – String representation of a Python expression.
- **pset** – Primitive set from which primitives are selected.

Returns PrimitiveTree populated with the deserialized primitives.

hasher
alias of builtins.str

pset

resolve_sc()
Evaluate StructConst in individual top to bottom.

terminals
Return terminals that occur in the expression tree.

to_polish(*for_sympy=False*, *replace_struct=True*)
Symbolic representation of the expression tree.

class ANDimTree(*trees*)
Bases: `list`

A naive tree class representing a vector-valued expression.

Each dimension is encoded as a expression tree.

base

classmethod create_individual(*ndim*)

classmethod create_population(*size*, *ndim*)

classmethod from_string(*strs*)

height

pset

terminals
Return terminals that occur in the expression tree.

class Individual(*pset*, *name='MyIndividual'*, ***kwargs*)
Bases: `type`

Construct a new expression tree type.

Parameters

- **pset** – `deap.gp.PrimitiveSet`
- **name** – name of the expression tree class
- **kwargs** – additional attributes

Returns: expression tree class

static __new__(*mcs*, *pset*, *name='MyIndividual'*, ***kwargs*)
Construct a new expression tree type.

Parameters

- **pset** – `deap.gp.PrimitiveSet`
- **name** – name of the expression tree class
- **kwargs** – additional attributes

Returns: expression tree class

class Measure(*values=()*)
Bases: `deap.base.Fitness`

This is basically a wrapper around deap.base.Fitness.

It provides the following enhancements over the base class: - more adequate naming - copy constructable - no weight attribute

del_values()

get_values()

set_values(*values*)

values

weights = repeat(-1)

class NDIndividual(*base*, *name='MyNDIndividual'*, ***kwargs*)
Bases: type

Construct a new n-dimensional expression tree type.

Parameters

- **base** (Individual) – one dimensional base class
- **name** – name of the n-dimensional expression tree class
- ****kwargs** – addtional attributes

Returns: n-dimensional expression tree class

class StructConst(*func*, *arity=2*)
Bases: deap.gp.Primitive

Parameters **func** – evaluate left and right subtree and assign a constant.

as_terminal(**args*)

static get_len(*expr*, *tokens='('*, *')*

add_sc(*pset*, *func*)
Adds a structural constant to a given primitive set.

Parameters

- **func** – `callable(x, y) -> float` where x and y are the expressions of the left and right subtree
- **pset** (deap.gp.PrimitiveSet) – You may want to use *sympy_primitive_set* or *numpy_primitive_set* without symbolic constants.

child_trees(*ind*)
Yield all child tree which are used as arguments for the head node of ind.

convert_inverse_prim(*prim*, *args*)
Convert inverse prims according to: [Dd]iv(a,b) -> Mul[a, 1/b] [Ss]ub(a,b) -> Add[a, -b]

We achieve this by overwriting the corresponding format method of the sub and div prim.

nd_phenotype(*nd_tree*, *backend=<function sympy_phenotype>*)

Parameters

- **nd_tree** (ANDimTree) –
- **backend** – sympy_phenotype or numpy_phenotype

Returns lambda function

numpy_phenotype(*individual*)
Lambdify the individual

Parameters **individual** (*glyph.gp.individual.AExpressionTree*) –

Returns lambda function

Note

In constrast to sympy_phenotype the callable will have a variable number of keyword arguments depending on the number of symbolic constants in the individual.

Example

```
>>> pset = numpy_primitive_set(1)
>>> MyIndividual = Individual(pset=pset)
>>> ind = MyIndividual.from_string("Add(x_0, Symc)")
>>> f = numpy_phenotype(ind)
>>> f(1, 1)
2
```

numpy_primitive_set (*arity*, *categories=('algebraic', 'trigonometric', 'exponential', 'symc')*)
Create a primitive set based on numpys vectorized functions.

Parameters

- **arity** – Number of variables in the primitive set
- **categories** –

Returns deap.gp.PrimitiveSet

Note: All functions will be closed, that is non-defined values will be mapped to 1. 1/0 = 1!

---

pretty_print (*expr*, *constants*, *consts_values*, *count=0*)
Replace symbolic constants in the str representation of an individual by their numeric values.

This checks for

- c followed by ")" or ","
- c followed by infix operators
- c

sc_mmqout (*x*, *y*, *cmin=-1*, *cmax=1*)
SC is the minimum-maximum quotient of the number of nodes of both child-trees x and y mapped into the constant interval [cmin, cmax]

sc_qout (*x*, *y*)
SC is the quotient of the number of nodes of its left and right child-trees x and y

sympy_phenotype (*individual*)
Compile python function from individual.

Uses sympy's lambdify function. Terminals from the primitive set will be used as parameters to the constructed lambda function; primitives (like sympy.exp) will be converted into numpy expressions (eg. numpy.exp).

Parameters **individual** (*glyph.gp.individual.AExpressionTree*) –

Returns lambda function

sympy_primitive_set (*categories=('algebraic', 'trigonometric', 'exponential'), arguments=['y_0'], constants=[]*)
Create a primitive set with sympy primitves.

Parameters

- **arguments** – variables to use in primitive set
- **constants** – symbolic constants to use in primitive set
- **categories** –

  an optional list of function categories for the primitive set. The following are available 'algebraic', 'neg', 'trigonometric', 'exponential', 'exponential', 'logarithm', 'sqrt'.

return deap.gp.PrimitiveSet

Module contents

glyph.utils package

Submodules

glyph.utils.argparse module

Collection of helper functions for arparse.

non_negative_int(*string*)
Check whether string is an integer greater than -1.

np_infinity_int(*string*)

ntuple(*n*, *to_type=<class 'float'>*)
Check whether string is an n-tuple.

positive_int(*string*)
Check whether string is an integer greater than 0.

readable_file(*string*)
Check weather file is readable

readable_yaml_file(*string*)
Check weather file is a .yaml file and readable

unit_interval(*string*)
Check whether string is a float in the interval [0.0, 1.0].

glyph.utils.break_condition module

class SoftTimeOut(*ttl*)
Bases: object

Break condition based on a soft time out.

Start a new generation as long as there is some time left.

Parameters **ttl** – time to live in seconds

alive

now

break_condition(*target=0*, *error_index=0*, *ttl=0*, *max_iter=inf*)
Combined breaking condition based on time to live, minimum target and maximum number of iterations.

Parameters

- **target** – value of desired error metric
- **error_index** – index in fitness tuple
- **ttl** – time to live in seconds
- **max_iter** – maximum number of iterations

soft_max_iter (*app*, *max_iter=inf*)
Soft breaking condition. Will check after each generation weather maximum number of iterations is exceeded.

Parameters **max_iter** – maximum number of function evaluations

Returns bool(iter) > max_iter

soft_target (*app*, *target=0*, *error_index=0*)
Soft breaking condition. Will check after each generation minimum error is reached.

Parameters

- **target** – value of desired error metric
- **error_index** – index in fitness tuple

Returns bool(min_error) <= target

glyph.utils.logging module

load_config (*config_file*, *placeholders=None*, *level=20*)
Load logging configuration from .yaml file.

log_level (*verbosity*)
Convert numeric verbosity to logging log levels.

print_dict (*p_func*, *d*)
Pretty print a dictionary

Parameters **p_func** – printer to use (print or logging)

print_params (*p_func*, *gp_config*)
Pretty print a glyph app config

glyph.utils.numeric module

class SlowConversionTerminator (*method*, *step_size=10*, *min_stat=10*, *threshold=25*)
Bases: object

Decorate a minimize method used in scipy.optimize.minimize to cancel non promising constant optimizations.

The stopping criteria is based on the improvement rate :math:‘

rac{Delta f}[Delta fev}‘.

If the improvement rate is below the $q_{threshold}$ quantile for a given number of function evaluations, optimization is stopped. :params method: see scipy.optimize.minimize method :params step_size: number of function evaluations betweem iterations :params min_stat: minimum sample size before stopping :params threshold: quantile

cvrmse (*x*, *y*)
Coefficient of variation, with respect to x, of the rmse.

hill_climb (*fun*, *x0*, *args*, *precision=5*, *maxfev=100*, *directions=5*, *target=0*, *rng=<module 'numpy.random' from '/home/docs/checkouts/readthedocs.org/user_builds/glyph/envs/stable/lib/python3.6/site-packages/numpy/random/__init__.py'>*, ***kwargs*)
Stochastic hill climber for constant optimization. Try self.directions different solutions per iteration to select a new best individual.

Parameters

- **fun** – function to optimize
- **x0** – initial guess
- **args** – additional arguments to pass to fun
- **precision** – maximum precision of x0
- **maxfev** – maximum number of function calls before stopping
- **directions** – number of directions to explore before doing a hill climb step
- **target** – stop if fun(x) <= target
- **rng** – (seeded) random number generator

Returns scipy.optimize.OptimizeResult

nrmse(*x*, *y*)
Normalized, with respect to x, root mean square error.

rms(*y*)
Root mean square.

rmse(*x*, *y*)
Root mean square error.

silent_numpy(*func*)

strict_subtract(*x*, *y*)

Module contents

class Memoize(*fn*)
Bases: object

Memoize(fn) - an instance which acts like fn but memoizes its arguments

Will only work on functions with non-mutable arguments http://code.activestate.com/recipes/52201/

key_set(*itr*, *key=<built-in function hash>*)

partition(*pred*, *iterable*)
Use a predicate to partition entries into false entries and true entries.

```
>>> is_odd = lambda x: x % 2
>>> odd, even = partition(is_odd, range(10))
>>> list(odd)
[0, 2, 4, 6, 8]
```

random_state(*obj*, *rng=<module 'random' from '/home/docs/checkouts/readthedocs.org/user_builds/glyph/envs/stable/lib/python3.*
Do work inside this contextmanager with a random state defined by obj.

Looks for _prev_state to seed the rng. On exit, it will write the current state of the rng as _tmp_state to the obj.

Params obj Any object.

Params rng Instance of a random number generator.

Submodules

glyph.application module

Convenience classes and functions that allow you to quickly build gp apps.

class AFactory
Bases: object

static add_options(*parser*)
Add available parser options.

classmethod create(*config*, **args*, ***kwargs*)

classmethod get_from_mapping(*key*)

class AlgorithmFactory
Bases: *glyph.application.AFactory*

Factory class for gp algorithms.

static add_options(*parser*)
Add available parser options.

class Application(*config*, *gp_runner*, *checkpoint_file=None*, *callbacks=(<function make_checkpoint>, <function log>)*)
Bases: object

An application based on *GPRunner*.

Controls execution of the runner and adds checkpointing and logging functionality; also defines a set of available command line options and their default values.

To create a full console application one can use the factory function default_console_app().

Parameters

- **config** (*dict or argparse.Namespace*) – Container holding all configs
- **gp_runner** – Instance of *GPRunner*
- **checkpoint_file** – Path to checkpoint_file
- **callbacks** –

static add_options(*parser*)
Add available parser options.

assessment_runner

checkpoint()
Checkpoint current state of evolution.

classmethod from_checkpoint(*file_name*)
Create application from checkpoint file.

logbook

run(*break_condition=None*)
Run gp app.

Parameters **break_condition** (*callable(application)*) – is called after every evolutionary step.

Returns number of iterations executed during run.

workdir

class ConstraintsFactory
Bases: *glyph.application.AFactory*

static add_options(*parser*)
Add available parser options.

class CreateFactory
Bases: *glyph.application.AFactory*

Factory class for creation

add_options()
Add available parser options.

class GPRunner(*IndividualClass*, *algorithm_factory*, *assessment_runner*, *callbacks=(<function update_pareto_front>, <function update_logbook_record>)*)
Bases: object

Runner for gp problem sets.

Takes care of propper initialization, execution, and accounting of a gp run (i.e. population creation, random state, generation count, hall of fame, and logbook). The method init() has to be called once before stepping through the evolution process with method step(); init() and step() invoke the assessment runner.

Parameters

- **IndividualClass** – Class inherited from gp.AExpressionTree.
- **algorithm_factory** – callable() -> gp algorithm, as defined in gp.algorithms.
- **assessment_runner** – callable(population) -> None, updates fitness values of each invalid individual in population.

init(*pop_size*)
Initialize the gp run.

step()
Step through the evolution process.

class MateFactory
Bases: *glyph.application.AFactory*

Factory class for gp mating functions.

static add_options(*parser*)
Add available parser options.

class MutateFactory
Bases: *glyph.application.AFactory*

Factory class for gp mutation functions.

static add_options(*parser*)
Add available parser options.

class ParallelizationFactory
Bases: *glyph.application.AFactory*

Factory class for parallel execution schemes.

static add_options(*parser*)
Add available parser options.

class SelectFactory
Bases: *glyph.application.AFactory*

Factory class for selection

static add_options(*parser*)
Add available parser options.

create_stats(*n*)
Create deap.tools.MultiStatistics object for n fitness values.

create_tmp_dir(*prefix='run-'*)
Create directory with current time as signature.

default_console_app(*IndividualClass*, *AssessmentRunnerClass*, *parser=ArgumentParser(prog='sphinx-build'*, *usage=None*, *description=None*, *formatter_class=<class 'argparse.HelpFormatter'>*, *conflict_handler='error'*, *add_help=True)*, *callbacks=(<function make_checkpoint>*, *<function log>)*)
Factory function for a console application.

default_gprunner(*Individual*, *assessment_runner*, *callbacks=(<function update_pareto_front>*, *<function update_logbook_record>)*, ***kwargs*)
Create a default GPRunner instance.

For config options see *MateFactory*, *MutateFactory*, *AlgorithmFactory*.

get_mapping(*group*)

load(*file_name*)
Load data saved with safe().

log(*app*)

make_checkpoint(*app*)

safe(*file_name*, ***kwargs*)
Dump kwargs to file.

to_argparse_namespace(*d*)
Return argparse.Namespace object created from dictionary d.

update_logbook_record(*runner*)

update_pareto_front(*runner*)

glyph.assessment module

Some usefull classes/functions for the fitness assessment part in gp problems.

class AAssessmentRunner(*parallel_factory=<glyph.assessment.SingleProcessFactory object>*)
Bases: object

Abstract runner for the (parallel) assessment of individuals in a population.

Child classes have to at least override the measure() method, which might be executed in a different process or even on a remote machine depending on the parallelization scheme. Child classes may override the setup() method, which is executed once on object instantiation. Child classes may override the assign_fitness() method, which is executed in the main process. This can be usefull if you want to locally post-process the results of measure(), when collected from remote processes.

Parameters **parallel_factory** – callable() -> obj, obj has to implement some kind of (parallel) map() method.

__call__(*population*)
Update the fitness of each individual in population that has an invalid fitness.

Parameters **population** – a squence of individuals.

__getstate__()
Modify pickling behavior for the class.

All the attributes except 'parallel' can be pickled.

__setstate__(*state*)
Modify unpickling behavior for the class.

assign_fitness(*individual*, *fitness*)
Assign a fitness value (as returned by self.measure()) to idividual.

Default implementation.

measure(*individual*)
Return a fitness value for individual.

setup()
Default implementation.

class SingleProcessFactory
Bases: object

class map
Bases: object

map(func, *iterables) –> map object

Make an iterator that computes the function using arguments from each of the iterables. Stops when the shortest iterable is exhausted.

annotate(*func*, *annotations*)
Add annoations to func.

const_opt(*measure*, *individual*, *lsq=False*, *default_constants=<function default_constants>*, *f_kwargs=None*, ***kwargs*)
Apply constant optimization

Parameters

- **measure** – callable(individual, *f_args) -> scalar.
- **individual** – an individual tha is passed on to measure.
- **bounds** – bounds for the constant values (s. scipy.optimize.minimize).
- **method** – Type of solver. Should either be 'leastsq', or one of scipy.optimize.minimize's solvers.

Returns (popt, measure_opt), popt: the optimal values for the constants; measure_opt: the measure evaluated at popt.

const_opt_leastsq(*measure*, *individual*, *default_constants=<function default_constants>*, *f_kwargs=None*, ***kwargs*)

const_opt_scalar(**args*, ***kwargs*)

default_constants(*ind*)
Return a one for each different constant in the primitive set.

Parameters **ind** (*glyph.gp.individual.AExpressionTree*) –

Returns A value for each constant in the primitive set.

expressional_complexity (*ind*)
Sum of length of all subtrees of the individual.

max_fitness_on_timeout (*max_fitness*, *timeout*)
Decorate a function. Associate max_fitness with long running individuals.

Parameters

- **max_fitness** – fitness of aborted individual calls.
- **timeout** – time until timeout

Returns fitness or max_fitness

measure (**funcs*, *pre=<function identity>*, *post=<function identity>*)
Combine several measurement functions into one.

Optionaly do pre- and/or post-processing.

Parameters

- **funcs** – a sequence of measure functions as returned by measure() (eg. `callable(*a, **kw) -> tuple`), and/or single valued functions (eg. `callable(*a, **kw)` -> numerical value).
- **pre** – some pre-processing function that is to be apllied on input *once* before passing the result to each function in funcs.
- **post** – some post-processing function that is to be apllied on the tuple of measure values as returned by the combined funcs.

Returns callable(input) -> tuple of measure values, where input is usually a phenotype (eg. an expression tree).

replace_nan (*x*, *rep=inf*)
Replace occurences of np.nan in x.

Parameters

- **x** (*list, tuple, float, np.ndarray*) – Any data structure
- **rep** – value to replace np.nan with

Returns x without nan's

returns (*func*, *types*)
Check func's annotation dictionary for return type tuple.

tuple_wrap (*func*)
Wrap func's return value into a tuple if it is not one already.

glyph.observer module

class ProgressObserver
Bases: `object`

Animates the progress of the evolutionary optimization.

Note: Uses matplotlib's interactive mode.

__call__(*app*)

> **Note:** To be used as a callback in *glyph.application.Application*.
>
> **Parameters** **app** (glyph.application.Application) –

get_limits(*x*, *factor=1.1*)
Calculates the plot range given an array x.

Module contents

Python Module Index

g

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

P

R

S

T

U

V

W