

---

# glyph

*Release 0.5.3+13.g8d360fa*

Jan 22, 2020



---

## Contents

---

<b>1</b>	<b>Content</b>	<b>3</b>
	<b>Python Module Index</b>	<b>45</b>
	<b>Index</b>	<b>47</b>



**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.

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



## 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.

Genetic 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 convenience function `numpy_primitive_set` to create a primitive set based on categories.

### 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 measure(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(measure, 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 recommended 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 step cycle through the evolutionary algorithm whilst taken care for statistics and a hall of fame.

It's mostly syntactic 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 factory 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.

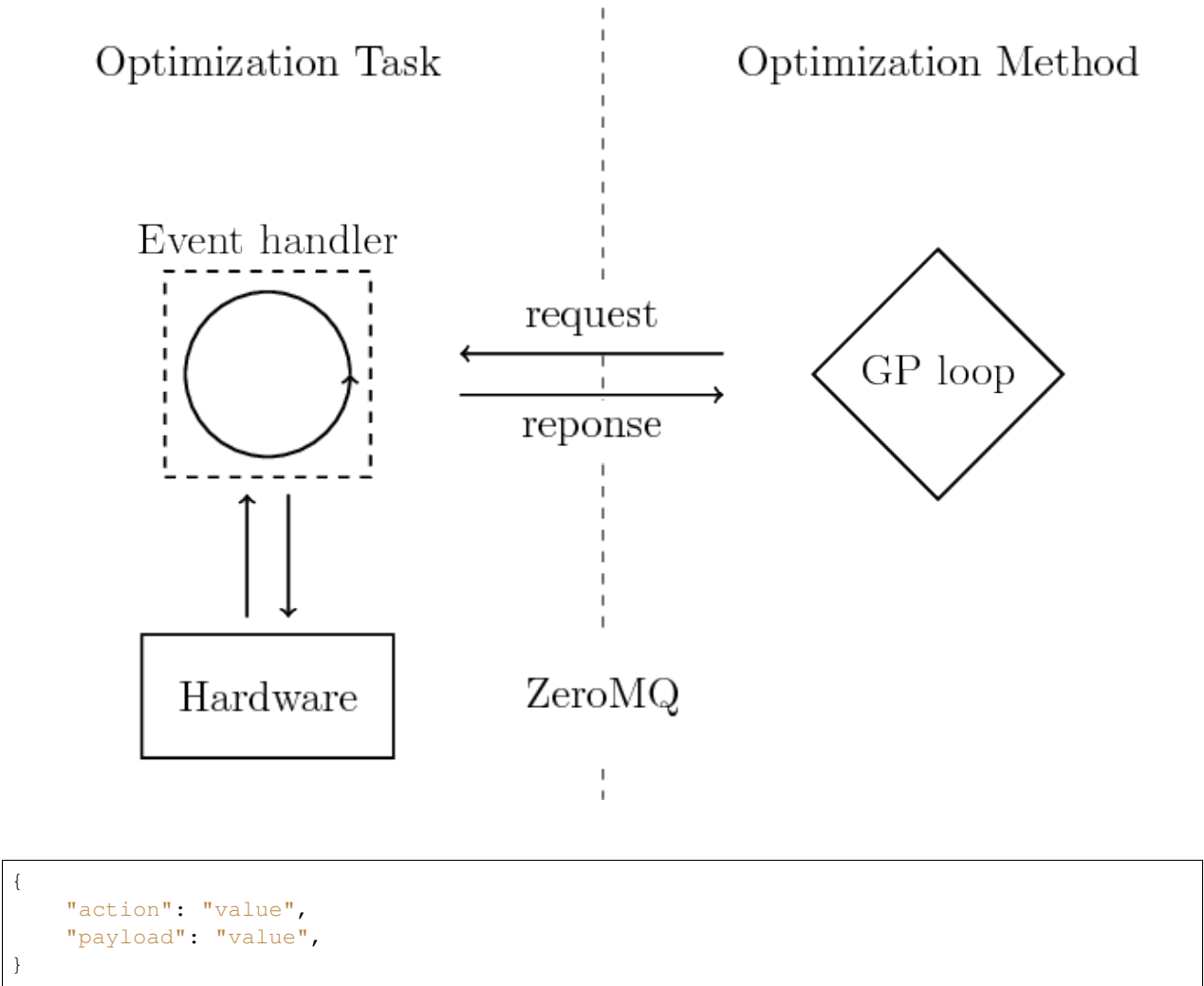
### 1.3.2 Overview

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:



The possible action values are:

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

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 Goocy installtion

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

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

### 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 developed 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 Example Gallery

#### Symbolic regression

This section demonstrates the glyph solving symbolic regression tasks.

---

**Note:** Click [here](#) to download the full example code

---

#### structural constants

Out:

```
/home/docs/checkouts/readthedocs.org/user_builds/glyph/envs/latest/lib/python3.6/site-
↳ packages/deap/tools/emo.py:138: ComplexWarning: Casting complex values to real_
↳ discards the imaginary part
    norm = nobj * float(crowd[-1][0][i] - crowd[0][0][i])
/home/docs/checkouts/readthedocs.org/user_builds/glyph/envs/latest/lib/python3.6/site-
↳ packages/deap/tools/emo.py:140: RuntimeWarning: invalid value encountered in double_
↳ scalars
    distances[cur[1]] += (next[0][i] - prev[0][i]) / norm
-0.7884573603642701 + I*pi ((0.028662213987980155-3.127158238984308j), 2.0)
-0.7884573603642701 + I*pi ((0.028662213987980155-3.127158238984308j), 2.0)
-0.7884573603642701 + I*pi ((0.028662213987980155-3.127158238984308j), 2.0)
-0.7884573603642701 + I*pi ((0.028662213987980155-3.127158238984308j), 2.0)
-0.7884573603642701 + I*pi ((0.028662213987980155-3.127158238984308j), 2.0)
-0.7884573603642701 + I*pi ((0.028662213987980155-3.127158238984308j), 2.0)
-0.5*log(2*x_0) - 0.5877866649021189 + I*pi ((0.005502350949910848-3.
↳ 0735536742862175j), 9.0)
-0.5*log(2*x_0) - 0.5877866649021189 + I*pi ((0.005502350949910848-3.
↳ 0735536742862175j), 9.0)
-0.5*log(2*x_0) - 0.5877866649021189 + I*pi ((0.005502350949910848-3.
↳ 0735536742862175j), 9.0)
-0.5*log(2*x_0) - 0.5877866649021189 + I*pi ((0.005502350949910848-3.
↳ 0735536742862175j), 9.0)
-0.5*log(2*x_0) - 0.5877866649021189 + I*pi ((0.005502350949910848-3.
↳ 0735536742862175j), 9.0)
-0.5*log(2*x_0) - 0.5877866649021189 + I*pi ((0.005502350949910848-3.
↳ 0735536742862175j), 9.0)
-0.5*log(2*x_0) - 0.5877866649021189 + I*pi ((0.005502350949910848-3.
↳ 0735536742862175j), 9.0)
-0.5*log(2*x_0) - 0.5877866649021189 + I*pi ((0.005502350949910848-3.
↳ 0735536742862175j), 9.0)
-0.5*log(2*x_0) - 0.5877866649021189 + I*pi ((0.005502350949910848-3.
↳ 0735536742862175j), 9.0)
```

(continues on next page)

(continued from previous page)

[illegible]

(continues on next page)



(continued from previous page)

```

-0.5*log(2*x_0) - 0.5877866649021189 + I*pi ((0.005502350949910848-3.
↪0735536742862175j), 9.0)
-0.5*log(2*x_0) - 0.5877866649021189 + I*pi ((0.005502350949910848-3.
↪0735536742862175j), 9.0)
-0.5*log(2*x_0) - 0.5877866649021189 + I*pi ((0.005502350949910848-3.
↪0735536742862175j), 9.0)
-0.5*log(2*x_0) - 0.5877866649021189 + I*pi ((0.005502350949910848-3.
↪0735536742862175j), 9.0)
-0.5*log(2*x_0) - 0.5877866649021189 + I*pi ((0.005502350949910848-3.
↪0735536742862175j), 9.0)
-0.5*log(2*x_0) - 0.5877866649021189 + I*pi ((0.005502350949910848-3.
↪0735536742862175j), 9.0)

```

```

from functools import partial

import deap.gp
import deap.tools
import numpy as np

from glyph import gp
from glyph.utils import Memoize
from glyph.utils.numeric import nrmse, silent_numpy

pset = gp.numpy_primitive_set(arity=1, categories=["algebraic", "trigonometric",
↪"exponential"])
pset = gp.individual.add_sc(pset, gp.individual.sc_mmqout)
Individual = gp.Individual(pset=pset)

@Memoize
@silent_numpy
def measure(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)
    try:
        yhat = f(points)
    except TypeError:
        yhat = np.infty
    if np.isscalar(yhat):
        yhat = np.ones_like(y) * yhat
    return nrmse(y, yhat), len(ind.resolve_sc())

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

```

(continues on next page)

(continued from previous page)

```

def main():
    pop_size = 400

    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.algorithms.AgeFitness(mate, mutate, deap.tools.selNSGA2,
    ↪Individual.create_population)

    pop = update_fitness(Individual.create_population(pop_size))

    for gen in range(50):
        pop = algorithm.evolve(pop)
        pop = update_fitness(pop)
        best = deap.tools.selBest(pop, 1)[0]
        print(gp.individual.simplify_this(best), best.fitness.values)

        if best.fitness.values[0] <= 1e-3:
            break

if __name__ == "__main__":
    main()

```

**Total running time of the script:** ( 0 minutes 14.235 seconds)

**Note:** Click [here](#) to download the full example code

## Minimal Example

Out:

```

cos(Symc*(x_0 - 1)) (0.07543569469969845, 6.0)
cos(Symc*(x_0 - 1)) (0.07543569469969845, 6.0)
cos(Symc*(x_0 - 1)) (0.07543569469969845, 6.0)
cos(Symc*(x_0 - 1)) (0.07543569469969845, 6.0)
cos(Symc*(x_0 - 1)) (0.07543569469969845, 6.0)
cos(Symc*(x_0 - 1)) (0.07543569469969845, 6.0)
cos(Symc*x_0**2) (0.06095387537460951, 12.0)
Symc*(x_0**2 + 2) (4.870737297739781e-09, 11.0)

```

```

from functools import partial

import deap.gp
import deap.tools
import numpy as np

```

(continues on next page)

(continued from previous page)

```

from glyph import gp
from glyph.assessment import const_opt
from glyph.utils import Memoize
from glyph.utils.numeric import nrmse, silent_numpy

pset = gp.numpy_primitive_set(arity=1, categories=["algebraic", "trigonometric",
↪ "exponential", "symc"])
Individual = gp.Individual(pset=pset)

@silent_numpy
def error(ind, *args):
    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, *args)

    if np.isscalar(yhat):
        yhat = np.ones_like(y) * yhat
    return nrmse(y, yhat)

@Memoize
def measure(ind):
    popt, err_opr = const_opt(error, ind)
    ind.popt = popt
    return err_opr, len(ind)

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

def main():
    pop_size = 100

    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.algorithms.AgeFitness(mate, mutate, deap.tools.selNSGA2, ↪
↪ Individual.create_population)

    pop = update_fitness(Individual.create_population(pop_size))

    for gen in range(20):
        pop = algorithm.evolve(pop)
        pop = update_fitness(pop)
        best = deap.tools.selBest(pop, 1)[0]
        print(gp.individual.simplify_this(best), best.fitness.values)

```

(continues on next page)

(continued from previous page)

```

        if best.fitness.values[0] <= 1e-3:
            break

if __name__ == "__main__":
    main()

```

**Total running time of the script:** ( 0 minutes 53.690 seconds)

**Note:** Click [here](#) to download the full example code

## Simplification by pruning

Out:

```

Symc + sin(Symc + x_0**2) (0.009182273966118449, 8.0)
Symc + sin(Symc*(x_0**2 - 2)) (0.003648423477499919, 12.0)
Symc + sin(Symc*(x_0**2 - 2)) (0.003648423477499919, 12.0)
Symc + sin(Symc*(x_0**2 - 2)) (0.003648423477499919, 12.0)
Symc + sin(Symc*(x_0**2 - 2)) (0.003648423477499919, 12.0)
Symc + sin(Symc*(x_0**2 - 2)) (0.003648423477499919, 12.0)
Symc + sin(Symc*(x_0**2 - 2)) (0.003648423477499919, 12.0)
Symc + sin(Symc*(x_0**2 - 2)) (0.003648423477499919, 12.0)
Symc + sin(Symc*(x_0**2 - 2)) (0.003648423477499919, 12.0)
Symc + sin(Symc*(x_0**2 - 2)) (0.003648423477499919, 12.0)
-Symc*cos(x_0) - Symc + cos(Symc*(x_0**2 + 1)) (5.714991547451496e-06, 15.0)

```

```

from functools import partial

import numpy as np
import deap.gp
import deap.tools

from glyph import gp
from glyph.assessment import const_opt
from glyph.utils import Memoize
from glyph.utils.numeric import silent_numpy, nrmse

pset = gp.numpy_primitive_set(arity=1, categories=["algebraic", "trigonometric",
↪ "exponential", "symc"])
Individual = gp.Individual(pset=pset)

@silent_numpy
def error(ind, *args):
    g = lambda x: x ** 2 - 1.1
    points = np.linspace(-1, 1, 100, endpoint=True)
    y = g(points)

```

(continues on next page)

(continued from previous page)

```

    f = gp.individual.numpy_phenotype(ind)
    yhat = f(points, *args)

    if np.isscalar(yhat):
        yhat = np.ones_like(y) * yhat
    return nrmse(y, yhat)

@Memoize
def measure(ind):
    pop_t, err_opr = const_opt(error, ind)
    ind.pop_t = pop_t
    return err_opr, len(ind)

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

def main():
    pop_size = 100

    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)
    simplify = gp.individual.simplify_constant

    algorithm = gp.algorithms.AgeFitness(mate, mutate, deap.tools.selNSGA2,
    ↪ Individual.create_population)

    pop = update_fitness(Individual.create_population(pop_size))

    for gen in range(20):
        pop = algorithm.evolve(pop)
        pop = [Individual(simplify(ind)) for ind in pop]

        pop = update_fitness(pop)
        best = deap.tools.selBest(pop, 1)[0]
        print(gp.individual.simplify_this(best), best.fitness.values)

        if best.fitness.values[0] <= 1e-3:
            break

if __name__ == "__main__":
    main()

```

**Total running time of the script:** ( 1 minutes 2.698 seconds)

**Note:** Click [here](#) to download the full example code

## MIMO

Out:

```
[ 'Sub(Symc, Sub(Add(Symc, x_0), Mul(x_0, Symc)))', 'Div(x_0, Symc)'] (0.
↪3377858084573795,)
[ 'Sub(Symc, Sub(Add(Symc, x_0), Mul(x_0, Symc)))', 'Div(x_0, Symc)'] (0.
↪3377858084573795,)
[ 'Sub(Symc, Sub(Add(Symc, x_0), Mul(x_0, Symc)))', 'Div(x_0, Symc)'] (0.
↪3377858084573795,)
[ 'Mul(x_0, x_0)', 'Mul(x_0, Symc)'] (0.0,)
```

```
from functools import partial, partialmethod

import numpy as np
from deap.tools import selBest, selNSGA2

from glyph.gp.algorithms import AgeFitness
from glyph.gp.breeding import cxonepoint, mutuniform, nd_crossover, nd_mutation
from glyph.gp.individual import Individual, NDIndividual, nd_phenotype, numpy_
↪phenotype, numpy_primitive_set
from glyph.utils.numeric import rmse

pset = numpy_primitive_set(1, categories=("algebraic", "symc"))
MyTree = Individual(pset=pset, name="MyTree")
MyNDTree = NDIndividual(base=MyTree, name="MyNDTree")
MyNDTree.create_population = partialmethod(MyNDTree.create_population, ndim=2)

def target(x):
    return np.array([f(x) for f in [lambda x: x ** 2, lambda x: x]])

x = np.linspace(-1, 1, 30)
y = target(x)

def evaluate_(individual, x, y):
    func = nd_phenotype(individual, backend=numpy_phenotype)
    yhat = func(x)
    for i in range(len(yhat)):
        if np.isscalar(yhat[i]):
            yhat[i] = np.ones_like(y[i]) * yhat[i]
    yhat = np.array(yhat)
    return (rmse(yhat, y),)

evaluate = partial(evaluate_, x=x, y=y)

def update_fitness(population):
    invalid = [p for p in population if not p.fitness.valid]
```

(continues on next page)

(continued from previous page)

```

fitnesses = map(evaluate, invalid)
for ind, fit in zip(invalid, fitnesses):
    ind.fitness.values = fit
return population

def main():
    pop_size = 100
    mutate = partial(nd_mutation, mutld=mutuniform(pset=pset))
    mate = partial(nd_crossover, cxld=cxonepoint())

    algorithm = AgeFitness(mate, mutate, selNSGA2, MyNDTree.create_population)

    pop = update_fitness(MyNDTree.create_population(pop_size))

    for gen in range(20):
        pop = algorithm.evolve(pop)
        pop = update_fitness(pop)
        best = selBest(pop, 1)[0]
        print(best, best.fitness.values)

        if best.fitness.values[0] <= 1e-3:
            break

if __name__ == "__main__":
    main()

```

**Total running time of the script:** ( 0 minutes 2.267 seconds)

**Note:** Click [here](#) to download the full example code

## Motifs

Out:

```

/home/docs/checkouts/readthedocs.org/user_builds/glyph/envs/latest/lib/python3.6/site-
↳ packages/scipy/optimize/optimize.py:697: RuntimeWarning: invalid value encountered_
↳ in double_scalars
    df = (f*((xk + d,) + args)) - f0) / d[k]
/home/docs/checkouts/readthedocs.org/user_builds/glyph/envs/latest/lib/python3.6/site-
↳ packages/scipy/optimize/optimize.py:697: RuntimeWarning: invalid value encountered_
↳ in double_scalars
    df = (f*((xk + d,) + args)) - f0) / d[k]
sin(Symc)*cos(x_0) (0.1736517123525045, 9.0)
sin(Symc)*cos(x_0) (0.1736517123525045, 9.0)
Symc*sin(Symc)*cos(x_0) (0.17365171235250448, 7.0)
Symc*sin(Symc)*cos(x_0) (0.17365171235250448, 7.0)
cos(exp(Symc) + sin(Symc*x_0)) (0.1563201842478601, 8.0)
cos(exp(Symc) + sin(Symc*x_0)) (0.1563201842478601, 8.0)
cos(exp(Symc) + sin(Symc*x_0)) (0.1563201842478601, 8.0)
cos(exp(Symc) + sin(Symc*x_0)) (0.1563201842478601, 8.0)
Symc - 0.94077090771611322*cos(Symc*x_0) (0.018082249356029582, 6.0)

```

(continues on next page)

(continued from previous page)

```
Symc - 0.94077090771611322*cos(Symc*x_0) (0.018082249356029582, 6.0)
{'cos(Add(exp(Symc), sin(Mul(Symc, x_0))))', 'Mul(Mul(cos(x_0), Symc), sin(Symc))',
↪'Add(Symc, Mul(Mul(cos(Mul(Symc, x_0)), -1.2308114705268705), sin(2.
↪271559786517541)))', 'Mul(Mul(cos(x_0), Div(Symc, Symc)), sin(Symc))'}
```

```
from functools import partial

import numpy as np
import deap.gp
import deap.tools

from glyph import gp
from glyph.assessment import const_opt
from glyph.utils import Memoize
from glyph.utils.numeric import silent_numpy, nrmse

pset = gp.numpy_primitive_set(arity=1, categories=["algebraic", "trigonometric",
↪"exponential", "symc"])
Individual = gp.Individual(pset=pset)

class ADF(deap.gp.Primitive):
    def __init__(self, name, arity, variable_names=None):
        self.name = name
        self.arity = arity
        self.args = [deap.gp.__type__] * arity
        self.ret = deap.gp.__type__
        self.variable_names = variable_names or ["x_{}".format(i) for i in_
↪range(arity)]
        self._format()

    def _format(self):
        self.fmt = self.name
        for i, v in enumerate(self.variable_names):
            self.fmt = self.fmt.replace(v, "{{{0}}}".format(i))

    def format(self, *args):
        return self.fmt.format(*args)

def pprint_individual(ind):
    name = str(ind)
    for c in ind.const_opt:
        name = name.replace("Symc", str(c), 1)
    return name

@silent_numpy
def error(ind, *args):
    g = lambda x: x ** 2 - 1.1
```

(continues on next page)



(continued from previous page)

```

points = np.linspace(-1, 1, 100, endpoint=True)
y = g(points)
f = gp.individual.numpy_phenotype(ind)
yhat = f(points, *args)

if np.isscalar(yhat):
    yhat = np.ones_like(y) * yhat
return nrmse(y, yhat)

@Memoize
def measure(ind):
    popt, err_opr = const_opt(error, ind)
    ind.popt = popt
    return err_opr, len(ind)

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

MOTIFS = set()

def add_motif(ind, pset):
    name = repr(ind)
    if name not in MOTIFS:
        motif = ADF(pprint_individual(ind), len(pset.arguments))
        pset._add(motif)
        pset.context[motif.name] = motif
        pset.prims_count += 1
        MOTIFS.add(name)

    return pset

def main():
    pop_size = 20

    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.algorithms.AgeFitness(mate, mutate, deap.tools.selNSGA2,
    ↪Individual.create_population)

    pop = update_fitness(Individual.create_population(pop_size))

    for gen in range(10):
        pop = algorithm.evolve(pop)
        pop = update_fitness(pop)
        best = deap.tools.selBest(pop, 1)[0]
        print(gp.individual.simplify_this(best), best.fitness.values)

```

(continues on next page)

(continued from previous page)

```

        Individual.pset = add_motif(best, Individual.pset)

        if best.fitness.values[0] <= 1e-3:
            break

    print(MOTIFS)

if __name__ == "__main__":
    main()

```

**Total running time of the script:** ( 0 minutes 20.092 seconds)

**Note:** Click [here](#) to download the full example code

## Linear Trafo Leaf

Out:

```

/home/docs/checkouts/readthedocs.org/user_builds/glyph/checkouts/latest/examples/
↳ symbolic_regression/linear_transformation_leaf.py:75: UserWarning: Maximum number
↳ of function evaluations has been exceeded.
  warnings.warn(res.message, UserWarning)
Mul(Div(w.dot(x) + b, w.dot(x) + b), Div(w.dot(x) + b, w.dot(x) + b)) [ 1.83606829  1.
↳ 92567857 -0.00433993  1.01260885  1.75845072 -1.8442725
  0.01366528  3.18841438] (1.5899852477247629e-06, 7.0)

```

```

import warnings
from functools import partial, wraps

import deap.gp
import deap.tools
import numpy as np
import scipy.optimize

from glyph import gp
from glyph.utils import Memoize
from glyph.utils.numeric import nrmse, silent_numpy

class Terminal(deap.gp.Terminal):
    def __init__(self):
        self.name = "w.dot(x) + b"
        self.ret = deap.gp.__type__

    def format(self, *args):
        return "w.dot(x) + b"

```

(continues on next page)

(continued from previous page)

```

pset = gp.numpy_primitive_set(arity=0, categories=["algebraic"])
pset.terminals[object].append(Terminal())

def phenotype(individual):
    params = ",".join(" w{i}, b{i}".format(i=i) for i in range(individual.n_args))
    args = "x," + params
    code = repr(individual)
    for i in range(individual.n_args):
        code = code.replace("w.dot(x) + b", "w{i}.dot(x) + b{i}".format(i=i), 1)
    expr = "lambda {}: {}".format(args, code)
    func = eval(expr, pset.context)
    return func

class Individual(gp.individual.AExpressionTree):
    """The gp representation (genotype) of the actuator for the control problem."""

    pset = pset

    @property
    def n_args(self):
        return sum([1 for t in self if isinstance(t, Terminal)])

    @property
    def arity(self):
        return 1

def const_opt(f, individual):
    arity = individual.arity

    @wraps(f)
    def closure(consts):
        new_consts = []
        for node in [
            consts[i : i + arity + 1] for i in range(0, len(consts), arity + 1)
        ]:
            new_consts.append(np.array(node[:-1]))
            new_consts.append(node[-1])
        return f(individual, *new_consts)

    p0 = np.ones((arity + 1) * individual.n_args)
    res = scipy.optimize.minimize(fun=closure, x0=p0, method="Nelder-Mead", tol=1e-3)
    popt = res.x if res.x.shape else np.array([res.x])
    measure_opt = res.fun
    if not res.success:
        warnings.warn(res.message, UserWarning)
    if measure_opt is None:
        measure_opt = closure(popt)
    return popt, measure_opt

@silent_numpy
def error(ind, *args):
    g = lambda x: x ** 2 - 1.1

```

(continues on next page)

(continued from previous page)

```

points = np.linspace(-1, 1, 100, endpoint=True).reshape(ind.arity, -1)
y = g(points)
f = phenotype(ind)
args[0].dot(points) + args[1]  # ??? without this there will be segmentation fault
yhat = f(points, *args).reshape(y.shape)

return nrmse(y, yhat)

@Memoize
def measure(ind):
    popt, err_opt = const_opt(error, ind)
    ind.popt = popt
    return err_opt, len(ind)

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

def main():
    pop_size = 20

    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.algorithms.AgeFitness(
        mate, mutate, deap.tools.selNSGA2, Individual.create_population
    )

    pop = update_fitness(Individual.create_population(pop_size))

    for gen in range(20):
        pop = algorithm.evolve(pop)
        pop = update_fitness(pop)
        best = deap.tools.selBest(pop, 1)[0]

        print(best, best.popt, best.fitness.values)

        if best.fitness.values[0] <= 1e-3:
            break

if __name__ == "__main__":
    main()

```

Total running time of the script: ( 0 minutes 31.270 seconds)

## 1.7.2 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.

## 1.7.3 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. Open 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 appropriate 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

## 1.7.4 Matlab Tutorial

Contributed by B. Strom.

---

**Note:** This was performed on Windows 7 and using MATLAB R2016b (2016b or later is needed for `jsondecode()` and `jsonencode()` 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. Open 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 an ASCII 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 Omq communication.

### Parameters

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

**connect** ()

**recv** (*serializer=<module 'json' from '/home/docs/.pyenv/versions/3.7.3/lib/python3.7/json/\_\_init\_\_.py'>*)

**send** (*msg, serializer=<module 'json' from '/home/docs/.pyenv/versions/3.7.3/lib/python3.7/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 dictionary `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

    evolve (population)

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

    evolve (population)

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

    evolve (population)

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

    evolve (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** – additional 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 primis 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 contrast 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, Symbc)")
>>> f = numpy_phenotype(ind)
>>> f(1, 1)
2
```

**numpy\_primitive\_set** (*arity, categories=('algebraic', 'trigonometric', 'exponential', 'symbc')*)

Create a primitive set based on numpy 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 primitives.

**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 argparse.

**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 whether file is readable

**readable\_yaml\_file** (*string*)

Check whether 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*, *tll=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
- **tll** – 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 whether 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:

$\frac{f(x) - f(x_{best})}{f(x)}$ .

If the improvement rate is below the *qthreshold* 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 between iterations :params min\_stat: minimum sample size before stopping :params threshold: quantile

**cvarmse** (*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/latest/lib/python3.7/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/latest/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 proper 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 *save()*.

**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 individual.

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 annotations 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 that 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.

Optionally 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 applied on input *once* before passing the result to each function in `funcs`.
- **post** – some post-processing function that is to be applied 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 occurrences 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`. Needs an interactive mpl backend.

---

**Parameters** **app** (`glyph.application.Application`) –

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

## Module contents





### g

- `glyph`, [43](#)
- `glyph.application`, [38](#)
- `glyph.assessment`, [41](#)
- `glyph.cli`, [30](#)
- `glyph.cli.glyph_remote`, [28](#)
- `glyph.gp`, [35](#)
- `glyph.gp.algorithms`, [30](#)
- `glyph.gp.breeding`, [30](#)
- `glyph.gp.constraints`, [31](#)
- `glyph.gp.individual`, [32](#)
- `glyph.observer`, [43](#)
- `glyph.utils`, [38](#)
- `glyph.utils.argparse`, [35](#)
- `glyph.utils.break_condition`, [36](#)
- `glyph.utils.logging`, [37](#)
- `glyph.utils.numeric`, [37](#)



## Symbols

`__call__()` (*AAssessmentRunner* method), 41  
`__call__()` (*ProgressObserver* method), 43  
`__getstate__()` (*AAssessmentRunner* method), 41  
`__new__()` (*Individual* static method), 33  
`__setstate__()` (*AAssessmentRunner* method), 41

## A

*AAssessmentRunner* (class in *glyph.assessment*), 41  
`add_options()` (*AFactory* static method), 38  
`add_options()` (*AlgorithmFactory* static method), 38  
`add_options()` (*Application* static method), 39  
`add_options()` (*ConstraintsFactory* static method), 39  
`add_options()` (*CreateFactory* method), 39  
`add_options()` (*MateFactory* static method), 40  
`add_options()` (*MutateFactory* static method), 40  
`add_options()` (*ParallelizationFactory* static method), 40  
`add_options()` (*SelectFactory* static method), 40  
`add_sc()` (in module *glyph.gp.individual*), 34  
*AExpressionTree* (class in *glyph.gp.individual*), 32  
*AFactory* (class in *glyph.application*), 38  
*AgeFitness* (class in *glyph.gp.algorithms*), 30  
*AlgorithmFactory* (class in *glyph.application*), 38  
`alive` (*SoftTimeOut* attribute), 36  
*ANDimTree* (class in *glyph.gp.individual*), 32  
`annotate()` (in module *glyph.assessment*), 42  
*Application* (class in *glyph.application*), 39  
`as_terminal()` (*StructConst* method), 34  
`assessment_runner` (*Application* attribute), 39  
`assign_fitness()` (*AAssessmentRunner* method), 41

## B

`base` (*ANDimTree* attribute), 32  
`base` (*NDTree* attribute), 28  
`break_condition()` (in module *glyph.utils.break\_condition*), 36

`build_pset_gp()` (in module *glyph.cli.glyph\_remote*), 29

## C

`checkpoint()` (*Application* method), 39  
`checkpoint()` (*RemoteApp* method), 29  
`child_trees()` (in module *glyph.gp.individual*), 34  
`com` (*PreTestService* attribute), 31  
*Communicator* (class in *glyph.cli.glyph\_remote*), 28  
`CONFIG` (*ExperimentProtocol* attribute), 28  
`connect()` (*Communicator* method), 28  
`const_opt` (*AExpressionTree* attribute), 32  
`const_opt()` (in module *glyph.assessment*), 42  
`const_opt_leastsq()` (in module *glyph.assessment*), 42  
`const_opt_options_transform()` (in module *glyph.cli.glyph\_remote*), 29  
`const_opt_scalar()` (in module *glyph.assessment*), 42  
`constrain()` (in module *glyph.gp.constraints*), 31  
*Constraint* (class in *glyph.gp.constraints*), 31  
*ConstraintsFactory* (class in *glyph.application*), 39  
`convert_inverse_prim()` (in module *glyph.gp.individual*), 34  
`create()` (*glyph.application.AFactory* class method), 38  
`create()` (*glyph.gp.individual.AExpressionTree* class method), 32  
`create_individual()` (*glyph.gp.individual.ANDimTree* class method), 32  
`create_population()` (*glyph.gp.individual.AExpressionTree* class method), 32  
`create_population()` (*glyph.gp.individual.ANDimTree* class method), 32  
`create_stats()` (in module *glyph.application*), 40

`create_tmp_dir()` (in module `glyph.application`), 40  
`CreateFactory` (class in `glyph.application`), 39  
`cvrmsc()` (in module `glyph.utils.numeric`), 37  
`cxonepoint()` (in module `glyph.gp.breeding`), 31  
`cxonepointleafbiased()` (in module `glyph.gp.breeding`), 31

## D

`DeapEaSimple` (class in `glyph.gp.algorithms`), 30  
`default_console_app()` (in module `glyph.application`), 40  
`default_constants()` (in module `glyph.assessment`), 42  
`default_gprunner()` (in module `glyph.application`), 40  
`del_values()` (Measure method), 33

## E

`EvalQueue` (class in `glyph.cli.glyph_remote`), 28  
`evaluate_single()` (RemoteAssessmentRunner method), 29  
`evolve()` (UAgeFitness method), 30  
`evolve()` (UDeapEaSimple method), 30  
`evolve()` (UNSGA2 method), 30  
`evolve()` (USPEA2 method), 30  
`evolve()` (AgeFitness method), 30  
`evolve()` (DeapEaSimple method), 30  
`EXPERIMENT` (ExperimentProtocol attribute), 28  
`ExperimentProtocol` (class in `glyph.cli.glyph_remote`), 28  
`expressional_complexity()` (in module `glyph.assessment`), 42

## F

`from_checkpoint()` (`glyph.application.Application` class method), 39  
`from_checkpoint()` (`glyph.cli.glyph_remote.RemoteApp` class method), 29  
`from_string()` (`glyph.gp.individual.AExpressionTree` class method), 32  
`from_string()` (`glyph.gp.individual.ANDimTree` class method), 32

## G

`get_from_mapping()` (`glyph.application.AFactory` class method), 38  
`get_len()` (StructConst static method), 34  
`get_limits()` (in module `glyph.observer`), 43  
`get_mapping()` (in module `glyph.application`), 41  
`get_values()` (Measure method), 33  
`get_version_info()` (in module `glyph.cli.glyph_remote`), 29

`glyph` (module), 43  
`glyph.application` (module), 38  
`glyph.assessment` (module), 41  
`glyph.cli` (module), 30  
`glyph.cli.glyph_remote` (module), 28  
`glyph.gp` (module), 35  
`glyph.gp.algorithms` (module), 30  
`glyph.gp.breeding` (module), 30  
`glyph.gp.constraints` (module), 31  
`glyph.gp.individual` (module), 32  
`glyph.observer` (module), 43  
`glyph.utils` (module), 38  
`glyph.utils.argparse` (module), 35  
`glyph.utils.break_condition` (module), 36  
`glyph.utils.logging` (module), 37  
`glyph.utils.numeric` (module), 37  
`GPRunner` (class in `glyph.application`), 39

## H

`handle_const_opt_config()` (in module `glyph.cli.glyph_remote`), 29  
`handle_gpconfig()` (in module `glyph.cli.glyph_remote`), 29  
`hasher` (AExpressionTree attribute), 32  
`height` (ANDimTree attribute), 32  
`hill_climb()` (in module `glyph.utils.numeric`), 37

## I

`Individual` (class in `glyph.cli.glyph_remote`), 28  
`Individual` (class in `glyph.gp.individual`), 33  
`init()` (GPRunner method), 40

## K

`key_set()` (in module `glyph.utils`), 38

## L

`load()` (in module `glyph.application`), 41  
`load_config()` (in module `glyph.utils.logging`), 37  
`log()` (in module `glyph.application`), 41  
`log_info()` (in module `glyph.cli.glyph_remote`), 29  
`log_level()` (in module `glyph.utils.logging`), 37  
`logbook` (Application attribute), 39

## M

`main()` (in module `glyph.cli.glyph_remote`), 29  
`make_callback()` (in module `glyph.cli.glyph_remote`), 29  
`make_checkpoint()` (in module `glyph.application`), 41  
`make_remote_app()` (in module `glyph.cli.glyph_remote`), 29  
`make_str` (PreTestService attribute), 31  
`MateFactory` (class in `glyph.application`), 40

max\_fitness\_on\_timeout() (in module *glyph.assessment*), 42  
 Measure (class in *glyph.gp.individual*), 33  
 measure() (AAssessmentRunner method), 41  
 measure() (in module *glyph.assessment*), 42  
 measure() (RemoteAssessmentRunner method), 29  
 Memoize (class in *glyph.utils*), 38  
 METADATA (ExperimentProtocol attribute), 28  
 MutateFactory (class in *glyph.application*), 40  
 mutnodereplacement() (in module *glyph.gp.breeding*), 30  
 mutshrink() (in module *glyph.gp.breeding*), 30, 31  
 mutuniform() (in module *glyph.gp.breeding*), 30

## N

nd\_phenotype() (in module *glyph.gp.individual*), 34  
 NDIndividual (class in *glyph.gp.individual*), 33  
 NDTREE (class in *glyph.cli.glyph\_remote*), 28  
 non\_negative\_int() (in module *glyph.utils.argparse*), 35  
 NonFiniteExpression (class in *glyph.gp.constraints*), 31  
 now (SoftTimeOut attribute), 36  
 np\_infinity\_int() (in module *glyph.utils.argparse*), 36  
 nrmse() (in module *glyph.utils.numeric*), 37  
 NSGA2 (class in *glyph.gp.algorithms*), 30  
 ntuple() (in module *glyph.utils.argparse*), 36  
 numpy\_phenotype() (in module *glyph.gp.individual*), 34  
 numpy\_primitive\_set() (in module *glyph.gp.individual*), 34

## P

ParallelizationFactory (class in *glyph.application*), 40  
 partition() (in module *glyph.utils*), 38  
 positive\_int() (in module *glyph.utils.argparse*), 36  
 predicate() (RemoteAssessmentRunner method), 29  
 PreTest (class in *glyph.gp.constraints*), 31  
 PreTestService (class in *glyph.gp.constraints*), 31  
 pretty\_print() (in module *glyph.gp.individual*), 35  
 print\_dict() (in module *glyph.utils.logging*), 37  
 print\_params() (in module *glyph.utils.logging*), 37  
 ProgressObserver (class in *glyph.observer*), 43  
 pset (AExpressionTree attribute), 32  
 pset (ANDimTree attribute), 33

## R

random\_state() (in module *glyph.utils*), 38  
 readable\_file() (in module *glyph.utils.argparse*), 36  
 readable\_yaml\_file() (in module *glyph.utils.argparse*), 36

recv (RemoteAssessmentRunner attribute), 29  
 recv() (Communicator method), 28  
 reject\_constrain\_violation() (in module *glyph.gp.constraints*), 31  
 RemoteApp (class in *glyph.cli.glyph\_remote*), 28  
 RemoteAssessmentRunner (class in *glyph.cli.glyph\_remote*), 29  
 replace\_nan() (in module *glyph.assessment*), 43  
 resolve\_sc() (AExpressionTree method), 32  
 returns() (in module *glyph.assessment*), 43  
 rms() (in module *glyph.utils.numeric*), 38  
 rmse() (in module *glyph.utils.numeric*), 38  
 run() (Application method), 39  
 run() (EvalQueue method), 28  
 run() (RemoteApp method), 29

## S

safe() (in module *glyph.application*), 41  
 sc\_mmqout() (in module *glyph.gp.individual*), 35  
 sc\_qout() (in module *glyph.gp.individual*), 35  
 SelectFactory (class in *glyph.application*), 40  
 send (RemoteAssessmentRunner attribute), 29  
 send() (Communicator method), 28  
 send\_meta\_data() (in module *glyph.cli.glyph\_remote*), 29  
 set\_values() (Measure method), 33  
 setup() (AAssessmentRunner method), 41  
 SHUTDOWN (ExperimentProtocol attribute), 28  
 silent\_numpy() (in module *glyph.utils.numeric*), 38  
 SingleProcessFactory (class in *glyph.assessment*), 42  
 SingleProcessFactory.map (class in *glyph.assessment*), 42  
 SlowConversionTerminator (class in *glyph.utils.numeric*), 37  
 soft\_max\_iter() (in module *glyph.utils.break\_condition*), 36  
 soft\_target() (in module *glyph.utils.break\_condition*), 36  
 SoftTimeOut (class in *glyph.utils.break\_condition*), 36  
 SPEA2 (class in *glyph.gp.algorithms*), 30  
 step() (GPRunner method), 40  
 strict\_subtract() (in module *glyph.utils.numeric*), 38  
 StructConst (class in *glyph.gp.individual*), 33  
 sympy\_phenotype() (in module *glyph.gp.individual*), 35  
 sympy\_primitive\_set() (in module *glyph.gp.individual*), 35

## T

terminals (AExpressionTree attribute), 32  
 terminals (ANDimTree attribute), 33

`to_argparse_namespace()` (in module `glyph.application`), 41  
`to_polish()` (*AExpressionTree* method), 32  
`tuple_wrap()` (in module `glyph.assessment`), 43

## U

*UAgeFitness* (class in `glyph.gp.algorithms`), 30  
*UDeapEaSimple* (class in `glyph.gp.algorithms`), 30  
`unit_interval()` (in module `glyph.utils.argparse`), 36  
*UNSGA2* (class in `glyph.gp.algorithms`), 30  
`update_fitness()` (*RemoteAssessmentRunner* method), 29  
`update_logbook_record()` (in module `glyph.application`), 41  
`update_namespace()` (in module `glyph.cli.glyph_remote`), 29  
`update_pareto_front()` (in module `glyph.application`), 41  
*USPEA2* (class in `glyph.gp.algorithms`), 30

## V

`values` (*Measure* attribute), 33

## W

`weights` (*Measure* attribute), 33  
`workdir` (*Application* attribute), 39