The μlab book

Release 1.4.0

Zoltán Vörös

with contributions by

Roberto Colistete Jr.

Jeff Epler

Taku Fukada

Diego Elio Pettenò

Scott Shawcroft

November 23, 2020

CHAPTER 1

Introduction

1.1 Enter ulab

`ulab` is a `numpy`-like module for `micropython` and its derivatives, meant to simplify and speed up common mathematical operations on arrays. `ulab` implements a small subset of `numpy`. The functions were chosen such that they might be useful in the context of a microcontroller. However, the project is a living one, and suggestions for new functions are always welcome.

This document discusses how you can use the library, starting from building your own firmware, through questions like what affects the firmware size, what are the trade-offs, and what are the most important differences to `numpy`. The document is organised as follows:

The chapter after this one helps you with firmware customisation.

The third chapter gives a very concise summary of the `ulab` functions and array methods. This chapter can be used as a quick reference.

The chapters after that are an in-depth review of most functions. Here you can find usage examples, benchmarks, as well as a thorough discussion of such concepts as broadcasting, and views versus copies.

The final chapter of this book can be regarded as the programming manual. The inner working of `ulab` is dissected here, and you will also find hints as to how to implement your own `numpy`-compatible functions.

1.2 Purpose

Of course, the first question that one has to answer is, why on Earth one would need a fast math library on a microcontroller. After all, it is not expected that heavy number crunching is going to take place on bare metal. It is not meant to. On a PC, the main reason for writing fast code is the sheer amount of data that one wants to process. On a microcontroller, the data volume is probably small, but it might lead to catastrophic system failure, if these data are not processed in time, because the microcontroller is supposed to interact with the outside world in a timely fashion. In fact, this latter objective was the initiator of this project: I needed the Fourier transform of a signal coming from the ADC of the `pyboard`, and all available options were simply too slow.

In addition to speed, another issue that one has to keep in mind when working with embedded systems is the amount of available RAM: I believe, everything here could be implemented in pure `python` with relatively little effort (in fact, there are a couple of `python`-only implementations of `numpy` functions out there), but the price we would have to pay for that is not only speed, but RAM, too. `python` code, if is not frozen, and compiled into the firmware, has to be compiled at runtime, which is not exactly a cheap process. On top of that, if numbers are stored in a list or tuple, which would be the high-level container, then they occupy 8 bytes, no matter, whether they are all smaller than 100, or larger than one hundred million. This is obviously a waste of resources in an environment, where resources are scarce.

Finally, there is a reason for using `micropython` in the first place. Namely, that a microcontroller can be programmed in a very elegant, and *pythonic* way. But if it is so, why should we not extend this idea to other tasks and concepts that might come up in this context? If there was no other reason than this *elegance*, I would find that convincing enough.

Based on the above-mentioned considerations, all functions in `ulab` are implemented in a way that

1. conforms to `numpy` as much as possible
2. is so frugal with RAM as possible,
3. and yet, fast. Much faster than pure python. Think of speed-ups of 30-50!

The main points of `ulab` are

- compact, iterable and slicable containers of numerical data in one to four dimensions. These containers support all the relevant unary and binary operators (e.g., `len`, ==, +, *, etc.)
- vectorised computations on `micropython` iterables and numerical arrays (in `numpy`-speak, universal functions)
- computing statistical properties (mean, standard deviation etc.) on arrays
- basic linear algebra routines (matrix inversion, multiplication, reshaping, transposition, determinant, and eigenvalues, Cholesky decomposition and so on)
- polynomial fits to numerical data, and evaluation of polynomials
- fast Fourier transforms
- filtering of data (convolution and second-order filters)
- function minimasation, fitting, and numerical approximation routines

`ulab` implements close to a hundred functions and array methods. At the time of writing this manual (for version 1.0.0), the library adds approximately 100 kB of extra compiled code to the `micropython` (pyboard.v.11) firmware. However, if you are tight with flash space, you can easily shave tens of kB off the firmware. In fact, if only a small sub-set of functions are needed, you can get away with less than 10 kB of flash space. See the section on *customising ulab*.

1.3 Resources and legal matters

The source code of the module can be found under https://github.com/v923z/micropython-ulab/tree/master/code. while the source of this user manual is under https://github.com/v923z/micropython-ulab/tree/master/docs.

The MIT licence applies to all material.

1.4 Friendly request

If you use `ulab`, and bump into a bug, or think that a particular function is missing, or its behaviour does not conform to `numpy`, please, raise a *ulab issue* on github, so that the community can profit from your experiences.

Even better, if you find the project useful, and think that it could be made better, faster, tighter, and shinier, please, consider contributing, and issue a pull request with the implementation of your improvements and new features. `ulab` can only become successful, if it offers what the community needs.

These last comments apply to the documentation, too. If, in your opinion, the documentation is obscure, misleading, or not detailed enough, please, let us know, so that *we* can fix it.

1.5 Differences between micropython-ulab and circuitpython-ulab

`ulab` has originally been developed for `micropython`, but has since been integrated into a number of its flavours. Most of these flavours are simply forks of `micropython` itself, with some additional functionality. One of the notable exceptions is `circuitpython`, which has slightly diverged at the core level, and this has some minor consequences. Some of these concern the C implementation details only, which all have been sorted out with the generous and enthusiastic support of Jeff Epler from Adafruit Industries.

There are, however, a couple of instances, where the usage in the two environments is different at the python level. These are how the packages can be imported, and how the class properties can be accessed. We will point out the differences and possible workarounds at the relevant places in this document.

CHAPTER 2

Customising ulab

As mentioned above, ulab has considerably grown since its conception, which also means that it might no longer fit on the microcontroller of your choice. There are, however, a couple of ways of customising the firmware, and thereby reducing its size.

All options are listed in a single header file, ulab.h, which contains pre-processor flags for each feature that can be fine-tuned. The first couple of lines of the file look like this

```
#ifndef __ULAB__
#define __ULAB__

// The pre-processor constants in this file determine how ulab behaves:
//
// - how many dimensions ulab can handle
// - which functions are included in the compiled firmware
// - whether the python syntax is numpy-like, or modular
// - whether arrays can be sliced and iterated over
// - which binary/unary operators are supported
//
// A considerable amount of flash space can be saved by removing (setting
// the corresponding constants to 0) the unnecessary functions and features.

// Setting this variable to 1 produces numpy-compatible firmware,
// i.e., functions can be called at the top level,
// without having to import the sub-modules (linalg and fft are exceptions,
// since those must be imported even in numpy)
#define ULAB_NUMPY_COMPATIBILITY        (1)

// The maximum number of dimensions the firmware should be able to support
// Possible values lie between 1, and 4, inclusive
#define ULAB_MAX_DIMS                   2

// By setting this constant to 1, iteration over array dimensions will be implemented
// as a function (ndarray_rewind_array), instead of writing out the loops in macros
// This reduces firmware size at the expense of speed
```

(continues on next page)

(continued from previous page)

```
#define ULAB_HAS_FUNCTION_ITERATOR      (0)

// If NDARRAY_IS_ITERABLE is 1, the ndarray object defines its own iterator function
// This option saves approx. 250 bytes of flash space
#define NDARRAY_IS_ITERABLE             (1)

// Slicing can be switched off by setting this variable to 0
#define NDARRAY_IS_SLICEABLE            (1)

// The default threshold for pretty printing. These variables can be overwritten
// at run-time via the set_printoptions() function
#define ULAB_HAS_PRINTOPTIONS           (1)
#define NDARRAY_PRINT_THRESHOLD         10
#define NDARRAY_PRINT_EDGEITEMS         3

// determines, whether pi, and e are defined in ulab itself
#define ULAB_HAS_MATH_CONSTANTS         (1)

// determines, whether the ndinfo function is available
#define ULAB_HAS_NDINFO                 (1)

// the ndarray binary operators
#define NDARRAY_HAS_BINARY_OPS              (1)
#define NDARRAY_HAS_BINARY_OP_ADD           (1)
#define NDARRAY_HAS_BINARY_OP_EQUAL         (1)
#define NDARRAY_HAS_BINARY_OP_LESS          (1)
#define NDARRAY_HAS_BINARY_OP_LESS_EQUAL    (1)
#define NDARRAY_HAS_BINARY_OP_MORE          (1)
#define NDARRAY_HAS_BINARY_OP_MORE_EQUAL    (1)
...
```

The meaning of flags with names `_HAS_` should obvious, so we will just explain the other options.

To see how much you can gain by un-setting the functions that you do not need, here are some pointers. In four dimensions, including all functions adds around 120 kB to the `micropython` firmware. On the other hand, if you are interested in Fourier transforms only, and strip everything else, you get away with less than 5 kB extra.

2.1 Compatibility with numpy

2.1.1 Working with sub-modules

The functions implemented in `ulab` are organised in sub-modules at the C level. This modularity is eleveted to python, if

```
#define ULAB_NUMPY_COMPATIBILITY        (0)
```

meaning that if you want to access a particular function, you would have to import the corresponding sub-module first.

```
import ulab
from ulab import poly

x = ulab.array([4, 5, 6])
p = ulab.array([1, 2, 3])
poly.polyval(p, x)
```

The idea of such grouping of functions and methods at the python level is to provide a means for granularity. At first, having to import everything in this way might appear to be overly complicated, but there is a very good reason behind all this: you can find out at the time of importing, whether a function or sub-module is part of your `ulab` firmware, or not. The alternative, namely, that you do not have to import anything beyond `ulab`, could prove catastrophic: you would learn only at run time (at the moment of calling the function in your code) that a particular function is not in the firmware, and that is most probably too late.

2.1.2 Generating numpy-compatible firmware

`circuitpython` follows the approach above, setting the `ULAB_NUMPY_COMPATIBILITY` flag to 0. On the other hand, if you want to generate truly `numpy`-compatible firmware, you can set

```
#define ULAB_NUMPY_COMPATIBILITY        (1)
```

If `ULAB_NUMPY_COMPATIBILITY` equals 1, functions will be bound at the top level, meaning that the example above now would look like

```
import ulab as numpy

x = numpy.array([4, 5, 6])
p = numpy.array([1, 2, 3])
numpy.polyval(p, x)
```

There are two exceptions to this rule, namely `fft`, and `linalg`, which are sub-modules even in `numpy`, thus you have to write them out as

```
import ulab
from ulab import linalg

A = ulab.array([1, 2, 3, 4]).reshape()
linalg.trace(A)
```

We should also note that the `numpy`-compatible firmware is a couple of hundred bytes smaller than the one with sub-modules, because defining the sub-modules requires some space.

2.2 The impact of dimensionality

2.2.1 Reducing the number of dimensions

`ulab` supports tensors of rank four, but this is expensive in terms of flash: with all available functions and options, the library adds around 100 kB to the flash. However, if such high dimensions are not required, significant reductions in size can be gotten by changing the value of

```
#define ULAB_MAX_DIMS                   2
```

Two dimensions cost a bit more than half of four, while you can get away with around 20 kB of flash in one dimension, because all those functions that don't make sense (e.g., matrix inversion, eigenvalues etc.) are automatically stripped from the firmware.

2.2.2 Using the function iterator

In higher dimensions, the firmware size increases, because each dimension (axis) adds another level of nested loops. An example of this is the macro of the binary operator in three dimensions

```
#define BINARY_LOOP(results, type_out, type_left, type_right, larray, lstrides,␣
↪rarray, rstrides, OPERATOR)
    type_out *array = (type_out *)results->array;
    size_t j = 0;
    do {
        size_t k = 0;
        do {
            size_t l = 0;
            do {
                *array++ = *((type_left *)(larray)) OPERATOR *((type_right␣
↪*)(rarray));
                (larray) += (lstrides)[ULAB_MAX_DIMS - 1];
                (rarray) += (rstrides)[ULAB_MAX_DIMS - 1];
                l++;
            } while(l < (results)->shape[ULAB_MAX_DIMS - 1]);
            (larray) -= (lstrides)[ULAB_MAX_DIMS - 1] * (results)->shape[ULAB_MAX_
↪DIMS-1];
            (larray) += (lstrides)[ULAB_MAX_DIMS - 2];
            (rarray) -= (rstrides)[ULAB_MAX_DIMS - 1] * (results)->shape[ULAB_MAX_
↪DIMS-1];
            (rarray) += (rstrides)[ULAB_MAX_DIMS - 2];
            k++;
        } while(k < (results)->shape[ULAB_MAX_DIMS - 2]);
        (larray) -= (lstrides)[ULAB_MAX_DIMS - 2] * results->shape[ULAB_MAX_DIMS-2];
        (larray) += (lstrides)[ULAB_MAX_DIMS - 3];
        (rarray) -= (rstrides)[ULAB_MAX_DIMS - 2] * results->shape[ULAB_MAX_DIMS-2];
        (rarray) += (rstrides)[ULAB_MAX_DIMS - 3];
        j++;
    } while(j < (results)->shape[ULAB_MAX_DIMS - 3]);
```

In order to reduce firmware size, it *might* make sense in higher dimensions to make use of the function iterator by setting the

```
#define ULAB_HAS_FUNCTION_ITERATOR      (1)
```

constant to 1. This allows the compiler to call the `ndarray_rewind_array` function, so that it doesn't have to unwrap the loops for `k`, and `j`. Instead of the macro above, we now have

```
#define BINARY_LOOP(results, type_out, type_left, type_right, larray, lstrides,␣
↪rarray, rstrides, OPERATOR)
    type_out *array = (type_out *)(results)->array;
    size_t *lcoords = ndarray_new_coords((results)->ndim);
    size_t *rcoords = ndarray_new_coords((results)->ndim);
    for(size_t i=0; i < (results)->len/(results)->shape[ULAB_MAX_DIMS -1]; i++) {
        size_t l = 0;
        do {
            *array++ = *((type_left *)(larray)) OPERATOR *((type_right *)(rarray));
            (larray) += (lstrides)[ULAB_MAX_DIMS - 1];
            (rarray) += (rstrides)[ULAB_MAX_DIMS - 1];
            l++;
        } while(l < (results)->shape[ULAB_MAX_DIMS - 1]);
        ndarray_rewind_array((results)->ndim, larray, (results)->shape, lstrides,␣
↪lcoords);
```

(continues on next page)

(continued from previous page)

```
        ndarray_rewind_array((results)->ndim, rarray, (results)->shape, rstrides,␣
↪rcoords);
    } while(0)
```

Since the `ndarray_rewind_array` function is implemented only once, a lot of space can be saved. Obviously, function calls cost time, thus such trade-offs must be evaluated for each application. The gain also depends on which functions and features you include. Operators and functions that involve two arrays are expensive, because at the C level, the number of cases that must be handled scales with the squares of the number of data types. As an example, the innocent-looking expression

```
import ulab as np

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

c = a + b
```

requires 25 loops in C, because the `dtypes` of both `a`, and `b` can assume 5 different values, and the addition has to be resolved for all possible cases. Hint: each binary operator costs between 3 and 4 kB in two dimensions.

2.3 The ulab version string

As is customary with `python` packages, information on the package version can be found be querying the `__version__` string.

```
# code to be run in micropython

import ulab as np

print('you are running ulab version', np.__version__)
```

```
you are running ulab version 0.99.0-2D-numpy
```

The first three numbers indicate the major, minor, and sub-minor versions of `ulab` (defined by the `ULAB_VERSION` constant in ulab.c). We usually change the minor version, whenever a new function is added to the code, and the sub-minor version will be incremented, if a bug fix is implemented.

`2D` tells us that the particular firmware supports tensors of rank 2 (defined by `ULAB_MAX_DIMS` in ulab.h), and the string `numpy` means that the firmware is `numpy`-compatible in the sense explained above. Otherwise, you would find `cpy`, i.e., firmware that conforms to `circuitpython`'s conventions.

If you find a bug, please, include the version string in your report!

2.4 Finding out what your firmware supports

`ulab` implements a number of array operators and functions, but this doesn't mean that all of these functions and methods are actually compiled into the firmware. You can fine-tune your firmware by setting/unsetting any of the `_HAS_` constants in ulab.h.

2.4.1 Functions included in the firmware

The version string will not tell you everything about your firmware, because the supported functions and sub-modules can still arbitrarily be included or excluded. One way of finding out what is compiled into the firmware is calling `dir` with `ulab` as its argument.

```
# code to be run in micropython

import ulab as np

print('class-level functions: \n', dir(np))

# since fft and linalg are sub-modules, print them separately
print('\nfunctions included in the fft module: \n', dir(np.fft))
print('\nfunctions included in the linalg module: \n', dir(np.linalg))
```

```
class-level functions:
 ['__class__', '__name__', 'bool', 'sort', 'sum', '__version__', 'acos', 'acosh',
↪'arange', 'arctan2', 'argmax', 'argmin', 'argsort', 'around', 'array', 'asin',
↪'asinh', 'atan', 'atanh', 'bisect', 'ceil', 'clip', 'concatenate', 'convolve', 'cos
↪', 'cosh', 'cross', 'degrees', 'diff', 'e', 'equal', 'erf', 'erfc', 'exp', 'expm1',
↪'eye', 'fft', 'flip', 'float', 'floor', 'fmin', 'full', 'gamma', 'get_printoptions',
↪ 'int16', 'int8', 'interp', 'lgamma', 'linalg', 'linspace', 'log', 'log10', 'log2',
↪'logspace', 'max', 'maximum', 'mean', 'min', 'minimum', 'ndinfo', 'newton', 'not_
↪equal', 'ones', 'pi', 'polyfit', 'polyval', 'radians', 'roll', 'set_printoptions',
↪'sin', 'sinh', 'sosfilt', 'sqrt', 'std', 'tan', 'tanh', 'trapz', 'uint16', 'uint8',
↪'user', 'vectorize', 'zeros']

functions included in the fft module:
 ['__class__', '__name__', 'fft', 'ifft', 'spectrogram']

functions included in the linalg module:
 ['__class__', '__name__', 'cholesky', 'det', 'dot', 'eig', 'inv', 'norm', 'size',
↪'trace']
```

2.4.2 Methods included in the firmware

The `dir` function applied to the module or its sub-modules gives information on what the module and sub-modules include, but is not enough to find out which methods the `ndarray` supports. We can list the methods by calling `dir` with the `array` object itself:

```
# code to be run in micropython

import ulab as np

print(dir(np.array))
```

```
['__class__', '__name__', 'copy', '__bases__', '__dict__', 'flatten', 'itemsize',
↪'reshape', 'shape', 'size', 'strides', 'tobytes', 'transpose']
```

2.4.3 Operators included in the firmware

A list of operators cannot be generated as shown above. If you need to find out, whether, e.g., the `**` operator is supported by the firmware, you have to `try` it:

```
# code to be run in micropython

import ulab as np

a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

try:
    print(a ** b)
except Exception as e:
    print('operator is not supported: ', e)
```

```
operator is not supported:  unsupported types for __pow__: 'ndarray', 'ndarray'
```

CHAPTER 3

`ulab` – Manipulate numeric data similar to numpy

ulab is a numpy-like module for micropython, meant to simplify and speed up common mathematical operations on arrays. The primary goal was to implement a small subset of numpy that might be useful in the context of a microcontroller. This means low-level data processing of linear (array) and two-dimensional (matrix) data.

ulab is adapted from micropython-ulab, and the original project's documentation can be found at https://micropython-ulab.readthedocs.io/en/latest/

ulab is modeled after numpy, and aims to be a compatible subset where possible. Numpy's documentation can be found at https://docs.scipy.org/doc/numpy/index.html

3.1 `ulab.approx` – Numerical approximation methods

`ulab.approx.`**`bisect`** (*fun: Callable[[float], float], a: float, b: float, *, xtol: float = 2.4e-07, maxiter: int = 100*) → float

Parameters

- **`f`** (*`callable`*) – The function to bisect
- **`a`** (*`float`*) – The left side of the interval
- **`b`** (*`float`*) – The right side of the interval
- **`xtol`** (*`float`*) – The tolerance value
- **`maxiter`** (*`float`*) – The maximum number of iterations to perform

Find a solution (zero) of the function `f(x)` on the interval (`a`..``b``) using the bisection method. The result is accurate to within `xtol` unless more than `maxiter` steps are required.

`ulab.approx.`**`fmin`** (*fun: Callable[[float], float], x0: float, *, xatol: float = 2.4e-07, fatol: float = 2.4e-07, maxiter: int = 200*) → float

Parameters

- **`f`** (*`callable`*) – The function to bisect

- **x0** (*float*) – The initial x value
- **xatol** (*float*) – The absolute tolerance value
- **fatol** (*float*) – The relative tolerance value

Find a minimum of the function `f(x)` using the downhill simplex method. The located `x` is within `fxtol` of the actual minimum, and `f(x)` is within `fatol` of the actual minimum unless more than `maxiter` steps are requried.

`ulab.approx.`**`interp`**(*x: ulab.array*, *xp: ulab.array*, *fp: ulab.array*, ***, *left: Optional[float] = None*, *right: Optional[float] = None*) → ulab.array

Parameters

- **x** (`ulab.array`) – The x-coordinates at which to evaluate the interpolated values.
- **xp** (`ulab.array`) – The x-coordinates of the data points, must be increasing
- **fp** (`ulab.array`) – The y-coordinates of the data points, same length as xp
- **left** – Value to return for `x < xp[0]`, default is `fp[0]`.
- **right** – Value to return for `x > xp[-1]`, default is `fp[-1]`.

Returns the one-dimensional piecewise linear interpolant to a function with given discrete data points (xp, fp), evaluated at x.

`ulab.approx.`**`newton`**(*fun: Callable[[float], float]*, *x0: float*, ***, *xtol: float = 2.4e-07*, *rtol: float = 0.0*, *maxiter: int = 50*) → float

Parameters

- **f** (*callable*) – The function to bisect
- **x0** (*float*) – The initial x value
- **xtol** (*float*) – The absolute tolerance value
- **rtol** (*float*) – The relative tolerance value
- **maxiter** (*float*) – The maximum number of iterations to perform

Find a solution (zero) of the function `f(x)` using Newton's Method. The result is accurate to within `xtol * rtol * |f(x)|` unless more than `maxiter` steps are requried.

`ulab.approx.`**`trapz`**(*y: ulab.array*, *x: Optional[ulab.array] = None*, *dx: float = 1.0*) → float

Parameters

- **ulab.array y** (*1D*) – the values of the dependent variable
- **ulab.array x** (*1D*) – optional, the coordinates of the independent variable. Defaults to uniformly spaced values.
- **dx** (*float*) – the spacing between sample points, if x=None

Returns the integral of y(x) using the trapezoidal rule.

3.2 `ulab.compare` – Comparison functions

`ulab.compare.`**`clip`**(*x1: Union[ulab.array, float]*, *x2: Union[ulab.array, float]*, *x3: Union[ulab.array, float]*) → ulab.array

Constrain the values from `x1` to be between `x2` and `x3`. `x2` is assumed to be less than or equal to `x3`.

Arguments may be ulab arrays or numbers. All array arguments must be the same size. If the inputs are all scalars, a single scalar is returned.

Shorthand for `ulab.maximum(x2, ulab.minimum(x1, x3))`

`ulab.compare.`**`equal`**(*x1: Union[ulab.array, float], x2: Union[ulab.array, float]*) → List[bool]
Return an array of bool which is true where x1[i] == x2[i] and false elsewhere

`ulab.compare.`**`not_equal`**(*x1: Union[ulab.array, float], x2: Union[ulab.array, float]*) → List[bool]
Return an array of bool which is false where x1[i] == x2[i] and true elsewhere

`ulab.compare.`**`maximum`**(*x1: Union[ulab.array, float], x2: Union[ulab.array, float]*) → ulab.array
Compute the element by element maximum of the arguments.

Arguments may be ulab arrays or numbers. All array arguments must be the same size. If the inputs are both scalars, a number is returned

`ulab.compare.`**`minimum`**(*x1: Union[ulab.array, float], x2: Union[ulab.array, float]*) → ulab.array
Compute the element by element minimum of the arguments.

Arguments may be ulab arrays or numbers. All array arguments must be the same size. If the inputs are both scalars, a number is returned

3.3 `ulab.fft` – Frequency-domain functions

`ulab.fft.`**`fft`**(*r: ulab.array*, *c: Optional[ulab.array] = None*) → Tuple[ulab.array, ulab.array]

Parameters

- **r** (`ulab.array`) – A 1-dimension array of values whose size is a power of 2
- **c** (`ulab.array`) – An optional 1-dimension array of values whose size is a power of 2, giving the complex part of the value

Return tuple (r, c) The real and complex parts of the FFT

Perform a Fast Fourier Transform from the time domain into the frequency domain

See also ~ulab.extras.spectrum, which computes the magnitude of the fft, rather than separately returning its real and imaginary parts.

`ulab.fft.`**`ifft`**(*r: ulab.array*, *c: Optional[ulab.array] = None*) → Tuple[ulab.array, ulab.array]

Parameters

- **r** (`ulab.array`) – A 1-dimension array of values whose size is a power of 2
- **c** (`ulab.array`) – An optional 1-dimension array of values whose size is a power of 2, giving the complex part of the value

Return tuple (r, c) The real and complex parts of the inverse FFT

Perform an Inverse Fast Fourier Transform from the frequeny domain into the time domain

`ulab.fft.`**`spectrogram`**(*r: ulab.array*) → ulab.array

Parameters **r** (`ulab.array`) – A 1-dimension array of values whose size is a power of 2

Computes the spectrum of the input signal. This is the absolute value of the (complex-valued) fft of the signal. This function is similar to scipy's `scipy.signal.spectrogram`.

3.4 `ulab.filter` – Filtering functions

`ulab.filter.`**`convolve`**`(`*a: ulab.array*, *v: ulab.array*`)` → ulab.array

Parameters

- **`a`** (`ulab.array`) –
- **`v`** (`ulab.array`) –

Returns the discrete, linear convolution of two one-dimensional sequences. The result is always an array of float. Only the `full` mode is supported, and the `mode` named parameter of numpy is not accepted. Note that all other modes can be had by slicing a `full` result.

Convolution filters can implement high pass, low pass, band pass, etc., filtering operations. Convolution filters are typically constructed ahead of time. This can be done using desktop python with scipy, or on web pages such as https://fiiir.com/

Convolution is most time-efficient when both inputs are of float type.

`ulab.filter.`**`sosfilt`**`(`*sos: _ArrayLike*, *x: _ArrayLike*`)` → ulab.array

`ulab.filter.`**`sosfilt`**`(`*sos: _ArrayLike*, *x: _ArrayLike*, ***, *zi: ulab.array*`)` → Tuple[ulab.array, ulab.array]

Parameters

- **`sos`** (`ulab.array`) – Array of second-order filter coefficients, must have shape (n_sections, 6). Each row corresponds to a second-order section, with the first three columns providing the numerator coefficients and the last three providing the denominator coefficients.
- **`x`** (`ulab.array`) – The data to be filtered
- **`zi`** (`ulab.array`) – Optional initial conditions for the filter

Returns If `zi` is not specified, the filter result alone is returned. If `zi` is specified, the return value is a 2-tuple of the filter result and the final filter conditions.

Filter data along one dimension using cascaded second-order sections.

Filter a data sequence, x, using a digital IIR filter defined by sos.

The filter function is implemented as a series of second-order filters with direct-form II transposed structure. It is designed to minimize numerical precision errors for high-order filters.

Filter coefficients can be generated by using scipy's filter generators such as `signal.ellip(..., output='sos')`.

3.5 `ulab.linalg` – Linear algebra functions

`ulab.linalg.`**`cholesky`**`(`*A: ulab.array*`)` → ulab.array

Parameters **`A`** (`array`) – a positive definite, symmetric square matrix

Return ~ulab.array L a square root matrix in the lower triangular form

Raises **`ValueError`** – If the input does not fulfill the necessary conditions

The returned matrix satisfies the equation m=LL*

`ulab.linalg.`**`det`**`(`*m: ulab.array*`)` → float

Param m, a square matrix

Return float The determinant of the matrix

Computes the eigenvalues and eigenvectors of a square matrix

`ulab.linalg.`**`dot`**(*m1: ulab.array*, *m2: ulab.array*) → Union[ulab.array, float]

Parameters

- **`m1`** (`array`) – a matrix, or a vector
- **`m2`** (`array`) – a matrix, or a vector

Computes the product of two matrices, or two vectors. In the letter case, the inner product is returned.

`ulab.linalg.`**`eig`**(*m: ulab.array*) → Tuple[ulab.array, ulab.array]

Parameters **`m`** – a square matrix

Return tuple (eigenvectors, eigenvalues)

Computes the eigenvalues and eigenvectors of a square matrix

`ulab.linalg.`**`inv`**(*m: ulab.array*) → ulab.array

Parameters **`m`** (`array`) – a square matrix

Returns The inverse of the matrix, if it exists

Raises **`ValueError`** – if the matrix is not invertible

Computes the inverse of a square matrix

`ulab.linalg.`**`norm`**(*x: ulab.array*) → float

Parameters **`x`** (`array`) – a vector or a matrix

Computes the 2-norm of a vector or a matrix, i.e., `sqrt(sum(x*x))`, however, without the RAM overhead.

`ulab.linalg.`**`trace`**(*m: ulab.array*) → float

Parameters **`m`** – a square matrix

Compute the trace of the matrix, the sum of its diagonal elements.

3.6 `ulab.numerical` – Numerical and Statistical functions

Most of these functions take an "axis" argument, which indicates whether to operate over the flattened array (None), or a particular axis (integer).

`ulab.numerical.`**`argmax`**(*array: _ArrayLike*, **, *axis: Optional[int] = None*) → int
Return the index of the maximum element of the 1D array

`ulab.numerical.`**`argmin`**(*array: _ArrayLike*, **, *axis: Optional[int] = None*) → int
Return the index of the minimum element of the 1D array

`ulab.numerical.`**`argsort`**(*array: ulab.array*, **, *axis: int = -1*) → ulab.array
Returns an array which gives indices into the input array from least to greatest.

`ulab.numerical.`**`cross`**(*a: ulab.array*, *b: ulab.array*) → ulab.array
Return the cross product of two vectors of length 3

`ulab.numerical.`**`diff`**(*array: ulab.array*, **, *n: int = 1*, *axis: int = -1*) → ulab.array
Return the numerical derivative of successive elements of the array, as an array. axis=None is not supported.

`ulab.numerical.`**`flip`**(*array: ulab.array*, *, *axis: Optional[int] = None*) → ulab.array
Returns a new array that reverses the order of the elements along the given axis, or along all axes if axis is None.

`ulab.numerical.`**`max`**(*array: _ArrayLike*, *, *axis: Optional[int] = None*) → float
Return the maximum element of the 1D array

`ulab.numerical.`**`mean`**(*array: _ArrayLike*, *, *axis: Optional[int] = None*) → float
Return the mean element of the 1D array, as a number if axis is None, otherwise as an array.

`ulab.numerical.`**`median`**(*array: ulab.array*, *, *axis: int = -1*) → ulab.array
Find the median value in an array along the given axis, or along all axes if axis is None.

`ulab.numerical.`**`min`**(*array: _ArrayLike*, *, *axis: Optional[int] = None*) → float
Return the minimum element of the 1D array

`ulab.numerical.`**`roll`**(*array: ulab.array*, *distance: int*, *, *axis: Optional[int] = None*) → None
Shift the content of a vector by the positions given as the second argument. If the `axis` keyword is supplied, the shift is applied to the given axis. The array is modified in place.

`ulab.numerical.`**`sort`**(*array: ulab.array*, *, *axis: int = -1*) → ulab.array
Sort the array along the given axis, or along all axes if axis is None. The array is modified in place.

`ulab.numerical.`**`std`**(*array: _ArrayLike*, *, *axis: Optional[int] = None*, *ddof: int = 0*) → float
Return the standard deviation of the array, as a number if axis is None, otherwise as an array.

`ulab.numerical.`**`sum`**(*array: _ArrayLike*, *, *axis: Optional[int] = None*) → Union[float, int, ulab.array]
Return the sum of the array, as a number if axis is None, otherwise as an array.

3.7 `ulab.poly` – Polynomial functions

`ulab.poly.`**`polyfit`**(*y: _ArrayLike*, *degree: int*) → ulab.array

`ulab.poly.`**`polyfit`**(*x: _ArrayLike*, *y: _ArrayLike*, *degree: int*) → ulab.array
Return a polynomial of given degree that approximates the function f(x)=y. If x is not supplied, it is the range(len(y)).

`ulab.poly.`**`polyval`**(*p: _ArrayLike*, *x: _ArrayLike*) → ulab.array
Evaluate the polynomial p at the points x. x must be an array.

3.8 `ulab.user` – This module should hold arbitrary user-defined functions.

3.9 `ulab.vector` – Element-by-element functions

These functions can operate on numbers, 1-D iterables, 1-D arrays, or 2-D arrays by applying the function to every element in the array. This is typically much more efficient than expressing the same operation as a Python loop.

`ulab.vector.`**`acos`**(*a: _ArrayLike*) → ulab.array
Computes the inverse cosine function

`ulab.vector.`**`acosh`**(*a: _ArrayLike*) → ulab.array
Computes the inverse hyperbolic cosine function

`ulab.vector.`**`asin`**(*a: _ArrayLike*) → ulab.array
Computes the inverse sine function

`ulab.vector.`**`asinh`**(*a: _ArrayLike*) → ulab.array
: Computes the inverse hyperbolic sine function

`ulab.vector.`**`around`**(*a: _ArrayLike*, ***, *decimals: int = 0*) → ulab.array
: Returns a new float array in which each element is rounded to `decimals` places.

`ulab.vector.`**`atan`**(*a: _ArrayLike*) → ulab.array
: Computes the inverse tangent function; the return values are in the range [-pi/2,pi/2].

`ulab.vector.`**`arctan2`**(*ya: _ArrayLike*, *xa: _ArrayLike*) → ulab.array
: Computes the inverse tangent function of y/x; the return values are in the range [-pi, pi].

`ulab.vector.`**`atanh`**(*a: _ArrayLike*) → ulab.array
: Computes the inverse hyperbolic tangent function

`ulab.vector.`**`ceil`**(*a: _ArrayLike*) → ulab.array
: Rounds numbers up to the next whole number

`ulab.vector.`**`cos`**(*a: _ArrayLike*) → ulab.array
: Computes the cosine function

`ulab.vector.`**`cosh`**(*a: _ArrayLike*) → ulab.array
: Computes the hyperbolic cosine function

`ulab.vector.`**`degrees`**(*a: _ArrayLike*) → ulab.array
: Converts angles from radians to degrees

`ulab.vector.`**`erf`**(*a: _ArrayLike*) → ulab.array
: Computes the error function, which has applications in statistics

`ulab.vector.`**`erfc`**(*a: _ArrayLike*) → ulab.array
: Computes the complementary error function, which has applications in statistics

`ulab.vector.`**`exp`**(*a: _ArrayLike*) → ulab.array
: Computes the exponent function.

`ulab.vector.`**`expm1`**(*a: _ArrayLike*) → ulab.array
: Computes e^x-1. In certain applications, using this function preserves numeric accuracy better than the *exp* function.

`ulab.vector.`**`floor`**(*a: _ArrayLike*) → ulab.array
: Rounds numbers up to the next whole number

`ulab.vector.`**`gamma`**(*a: _ArrayLike*) → ulab.array
: Computes the gamma function

`ulab.vector.`**`lgamma`**(*a: _ArrayLike*) → ulab.array
: Computes the natural log of the gamma function

`ulab.vector.`**`log`**(*a: _ArrayLike*) → ulab.array
: Computes the natural log

`ulab.vector.`**`log10`**(*a: _ArrayLike*) → ulab.array
: Computes the log base 10

`ulab.vector.`**`log2`**(*a: _ArrayLike*) → ulab.array
: Computes the log base 2

`ulab.vector.`**`radians`**(*a: _ArrayLike*) → ulab.array
: Converts angles from degrees to radians

`ulab.vector.`**`sin`**(*a: _ArrayLike*) → ulab.array
: Computes the sine function

`ulab.vector.`**`sinh`**(*a: _ArrayLike*) → ulab.array
    Computes the hyperbolic sine

`ulab.vector.`**`sqrt`**(*a: _ArrayLike*) → ulab.array
    Computes the square root

`ulab.vector.`**`tan`**(*a: _ArrayLike*) → ulab.array
    Computes the tangent

`ulab.vector.`**`tanh`**(*a: _ArrayLike*) → ulab.array
    Computes the hyperbolic tangent

`ulab.vector.`**`vectorize`**(*f: Union[Callable[[int], float], Callable[[float], float]], *, otypes: Optional[_DType] = None*) → Callable[[_ArrayLike], ulab.array]

Parameters

- **`f`** (*`callable`*) – The function to wrap
- **`otypes`** – List of array types that may be returned by the function. None is interpreted to mean the return value is float.

Wrap a Python function `f` so that it can be applied to arrays. The callable must return only values of the types specified by `otypes`, or the result is undefined.

`ulab.`**`_DType`**
    ulab.int8, *ulab.uint8*, *ulab.int16*, *ulab.uint16*, *ulab.float* or *ulab.bool*

`ulab.`**`_float`**
    Type alias of the bulitin float

`ulab.`**`_bool`**
    Type alias of the bulitin bool

`ulab.`**`_Index`**

`class` `ulab.`**`array`**(*values: Union[array, Iterable[Union[_float, _bool, Iterable[Any]]]], *, dtype: _DType = ulab.float*)
    1- and 2- dimensional array

Parameters

- **`values`** (*`sequence`*) – Sequence giving the initial content of the array.
- **`dtype`** (`_DType`) – The type of array values, *ulab.int8*, *ulab.uint8*, *ulab.int16*, *ulab.uint16*, *ulab.float* or *ulab.bool*

The `values` sequence can either be another ~ulab.array, sequence of numbers (in which case a 1-dimensional array is created), or a sequence where each subsequence has the same length (in which case a 2-dimensional array is created).

Passing a *ulab.array* and a different dtype can be used to convert an array from one dtype to another.

In many cases, it is more convenient to create an array from a function like *zeros* or *linspace*.

ulab.array implements the buffer protocol, so it can be used in many places an *array.array* can be used.

`shape :Tuple[int, ...]`
    The size of the array, a tuple of length 1 or 2

`size :int`
    The number of elements in the array

`itemsize :int`
    The size of a single item in the array

`strides :Tuple[int, ...]`
Tuple of bytes to step in each dimension, a tuple of length 1 or 2

`copy` (*self*)
Return a copy of the array

`flatten` (*self*, ***, *order: str = 'C'*)

Parameters **`order`** – Whether to flatten by rows ('C') or columns ('F')

Returns a new *ulab.array* object which is always 1 dimensional. If order is 'C' (the default", then the data is ordered in rows; If it is 'F', then the data is ordered in columns. "C" and "F" refer to the typical storage organization of the C and Fortran languages.

`reshape` (*self, shape: Tuple[int, ...]*)
Returns an array containing the same data with a new shape.

`sort` (*self*, ***, *axis: Optional[int] = 1*)

Parameters **`axis`** – Whether to sort elements within rows (0), columns (1), or elements (None)

`tobytes` (*self*)
Return the raw data bytes in the array

`transpose` (*self*)
Swap the rows and columns of a 2-dimensional array

`__add__` (*self, other: Union[array, _float]*)
Adds corresponding elements of the two arrays, or adds a number to all elements of the array. If both arguments are arrays, their sizes must match.

`__radd__` (*self*, *other: _float*)

`__sub__` (*self, other: Union[array, _float]*)
Subtracts corresponding elements of the two arrays, or subtracts a number from all elements of the array. If both arguments are arrays, their sizes must match.

`__rsub__` (*self*, *other: _float*)

`__mul__` (*self, other: Union[array, _float]*)
Multiplies corresponding elements of the two arrays, or multiplies all elements of the array by a number. If both arguments are arrays, their sizes must match.

`__rmul__` (*self*, *other: _float*)

`__div__` (*self, other: Union[array, _float]*)
Multiplies corresponding elements of the two arrays, or divides all elements of the array by a number. If both arguments are arrays, their sizes must match.

`__rdiv__` (*self*, *other: _float*)

`__pow__` (*self, other: Union[array, _float]*)
Computes the power (x**y) of corresponding elements of the the two arrays, or one number and one array. If both arguments are arrays, their sizes must match.

`__rpow__` (*self*, *other: _float*)

`__inv__` (*self*)

`__neg__` (*self*)

`__pos__` (*self*)

`__abs__` (*self*)

`__len__` (*self*)

`__lt__`(*self, other: Union[array, _float]*)
Return self<value.

`__le__`(*self, other: Union[array, _float]*)
Return self<=value.

`__gt__`(*self, other: Union[array, _float]*)
Return self>value.

`__ge__`(*self, other: Union[array, _float]*)
Return self>=value.

`__iter__`(*self*)

`__getitem__`(*self, index: _Index*)
Retrieve an element of the array.

`__setitem__`(*self, index: _Index, value: Union[array, _float]*)
Set an element of the array.

`ulab.`**`_ArrayLike`**
ulab.array, `List[float]`, `Tuple[float]` or *range*

`ulab.`**`int8 :_DType`**
Type code for signed integers in the range -128 .. 127 inclusive, like the 'b' typecode of *array.array*

`ulab.`**`int16 :_DType`**
Type code for signed integers in the range -32768 .. 32767 inclusive, like the 'h' typecode of *array.array*

`ulab.`**`float :_DType`**
Type code for floating point values, like the 'f' typecode of *array.array*

`ulab.`**`uint8 :_DType`**
Type code for unsigned integers in the range 0 .. 255 inclusive, like the 'H' typecode of *array.array*

`ulab.`**`uint16 :_DType`**
Type code for unsigned integers in the range 0 .. 65535 inclusive, like the 'h' typecode of *array.array*

`ulab.`**`bool :_DType`**
Type code for boolean values

`ulab.`**`get_printoptions`**() → Dict[str, int]
Get printing options

`ulab.`**`set_printoptions`**(*threshold: Optional[int] = None, edgeitems: Optional[int] = None*) → None
Set printing options

`ulab.`**`ndinfo`**(*array: ulab.array*) → None

`ulab.`**`arange`**(*stop: _float, step: _float = 1, *, dtype: _DType = ulab.float*) → ulab.array

`ulab.`**`arange`**(*start: _float, stop: _float, step: _float = 1, *, dtype: _DType = ulab.float*) → ulab.array
Return a new 1-D array with elements ranging from `start` to `stop`, with step size `step`.

`ulab.`**`concatenate`**(*arrays: Tuple[ulab.array], *, axis: int = 0*) → ulab.array
Join a sequence of arrays along an existing axis.

`ulab.`**`diag`**(*a: ulab.array, *, k: int = 0*) → ulab.array
Return specified diagonals.

`ulab.`**`eye`**(*size: int, *, M: Optional[int] = None, k: int = 0, dtype: _DType = ulab.float*) → ulab.array
Return a new square array of size, with the diagonal elements set to 1 and the other elements set to 0.

`ulab.`**`full`**(*shape: Union[int, Tuple[int, ...]], fill_value: Union[_float, _bool], *, dtype: _DType = ulab.float*) → ulab.array

Return a new array of the given shape with all elements set to 0.

`ulab.`**`linspace`**(*start: _float, stop: _float, *, dtype: _DType = ulab.float, num: int = 50, endpoint: _bool = True, retstep: _bool = False*) → ulab.array

Return a new 1-D array with `num` elements ranging from `start` to `stop` linearly.

`ulab.`**`logspace`**(*start: _float, stop: _float, *, dtype: _DType = ulab.float, num: int = 50, endpoint: _bool = True, base: _float = 10.0*) → ulab.array

Return a new 1-D array with `num` evenly spaced elements on a log scale. The sequence starts at `base ** start`, and ends with `base ** stop`.

`ulab.`**`ones`**(*shape: Union[int, Tuple[int, ...]], *, dtype: _DType = ulab.float*) → ulab.array

Return a new array of the given shape with all elements set to 1.

`ulab.`**`zeros`**(*shape: Union[int, Tuple[int, ...]], *, dtype: _DType = ulab.float*) → ulab.array

Return a new array of the given shape with all elements set to 0.

CHAPTER 4

ndarray, the basic container

The `ndarray` is the underlying container of numerical data. It can be thought of as micropython's own `array` object, but has a great number of extra features starting with how it can be initialised, which operations can be done on it, and which functions can accept it as an argument. One important property of an `ndarray` is that it is also a proper `micropython` iterable.

The `ndarray` consists of a short header, and a pointer that holds the data. The pointer always points to a contiguous segment in memory (`numpy` is more flexible in this regard), and the header tells the interpreter, how the data from this segment is to be read out, and what the bytes mean. Some operations, e.g., `reshape`, are fast, because they do not operate on the data, they work on the header, and therefore, only a couple of bytes are manipulated, even if there are a million data entries. A more detailed exposition of how operators are implemented can be found in the section titled *Programming ulab*.

Since the `ndarray` is a binary container, it is also compact, meaning that it takes only a couple of bytes of extra RAM in addition to what is required for storing the numbers themselves. `ndarrays` are also type-aware, i.e., one can save RAM by specifying a data type, and using the smallest reasonable one. Five such types are defined, namely `uint8`, `int8`, which occupy a single byte of memory per datum, `uint16`, and `int16`, which occupy two bytes per datum, and `float`, which occupies four or eight bytes per datum. The precision/size of the `float` type depends on the definition of `mp_float_t`. Some platforms, e.g., the PYBD, implement `doubles`, but some, e.g., the pyboard.v.11, don't. You can find out, what type of float your particular platform implements by looking at the output of the *.itemsize* class property.

In addition to the five above-mentioned numerical types, it is also possible to define Boolean arrays, which can be used in the indexing of data. However, Boolean arrays are really nothing but arrays of type `uint8` with an extra flag.

On the following pages, we will see how one can work with `ndarrays`. Those familiar with `numpy` should find that the nomenclature and naming conventions of `numpy` are adhered to as closely as possible. We will point out the few differences, where necessary.

For the sake of comparison, in addition to the `ulab` code snippets, sometimes the equivalent `numpy` code is also presented. You can find out, where the snippet is supposed to run by looking at its first line, the header of the code block.

4.1 The ndinfo function

A concise summary of a couple of the properties of an `ndarray` can be printed out by calling the `ndinfo` function. In addition to finding out what the *shape* and *strides* of the array array, we also get the `itemsize`, as well as the type. An interesting piece of information is the *data pointer*, which tells us, what the address of the data segment of the `ndarray` is. We will see the significance of this in the section *Slicing and indexing*.

Note that this function simply prints some information, but does not return anything. If you need to get a handle of the data contained in the printout, you should call the dedicated `shape`, `strides`, or `itemsize` functions directly.

```
# code to be run in micropython

import ulab as np

a = np.array(range(5), dtype=np.float)
b = np.array(range(25), dtype=np.uint8).reshape((5, 5))
np.ndinfo(a)
print('\n')
np.ndinfo(b)
```

```
class: ndarray
shape: (5,)
strides: (8,)
itemsize: 8
data pointer: 0x7f2bafabd220
type: float


class: ndarray
shape: (5, 5)
strides: (5, 1)
itemsize: 1
data pointer: 0x7f2bafabd3a0
type: uint8
```

4.2 Initialising an array

A new array can be created by passing either a standard micropython iterable, or another `ndarray` into the constructor.

4.2.1 Initialising by passing iterables

If the iterable is one-dimensional, i.e., one whose elements are numbers, then a row vector will be created and returned. If the iterable is two-dimensional, i.e., one whose elements are again iterables, a matrix will be created. If the lengths of the iterables are not consistent, a `ValueError` will be raised. Iterables of different types can be mixed in the initialisation function.

If the `dtype` keyword with the possible `uint8/int8/uint16/int16/float` values is supplied, the new `ndarray` will have that type, otherwise, it assumes `float` as default.

```
# code to be run in micropython

import ulab as np
```

(continues on next page)

(continued from previous page)

```
a = [1, 2, 3, 4, 5, 6, 7, 8]
b = np.array(a)

print("a:\t", a)
print("b:\t", b)

# a two-dimensional array with mixed-type initialisers
c = np.array([range(5), range(20, 25, 1), [44, 55, 66, 77, 88]], dtype=np.uint8)
print("\nc:\t", c)

# and now we throw an exception
d = np.array([range(5), range(10), [44, 55, 66, 77, 88]], dtype=np.uint8)
print("\nd:\t", d)
```

```
a:   [1, 2, 3, 4, 5, 6, 7, 8]
b:   array([1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0], dtype=float)

c:   array([[0, 1, 2, 3, 4],
    [20, 21, 22, 23, 24],
    [44, 55, 66, 77, 88]], dtype=uint8)

Traceback (most recent call last):
  File "/dev/shm/micropython.py", line 15, in <module>
ValueError: iterables are not of the same length
```

4.2.2 Initialising by passing arrays

An ndarray can be initialised by supplying another array. This statement is almost trivial, since ndarrays are iterables themselves, though it should be pointed out that initialising through arrays is a bit faster. This statement is especially true, if the dtypes of the source and output arrays are the same, because then the contents can simply be copied without further ado. While type conversion is also possible, it will always be slower than straight copying.

```
# code to be run in micropython

import ulab as np

a = [1, 2, 3, 4, 5, 6, 7, 8]
b = np.array(a)
c = np.array(b)
d = np.array(b, dtype=np.uint8)

print("a:\t", a)
print("\nb:\t", b)
print("\nc:\t", c)
print("\nd:\t", d)
```

```
a:   [1, 2, 3, 4, 5, 6, 7, 8]

b:   array([1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0], dtype=float)

c:   array([1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0], dtype=float)

d:   array([1, 2, 3, 4, 5, 6, 7, 8], dtype=uint8)
```

Note that the default type of the `ndarray` is `float`. Hence, if the array is initialised from another array, type conversion will always take place, except, when the output type is specifically supplied. I.e.,

```
# code to be run in micropython

import ulab as np

a = np.array(range(5), dtype=np.uint8)
b = np.array(a)
print("a:\t", a)
print("\nb:\t", b)
```

```
a:   array([0, 1, 2, 3, 4], dtype=uint8)

b:   array([0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
```

will iterate over the elements in `a`, since in the assignment `b = np.array(a)` no output type was given, therefore, `float` was assumed. On the other hand,

```
# code to be run in micropython

import ulab as np

a = np.array(range(5), dtype=np.uint8)
b = np.array(a, dtype=np.uint8)
print("a:\t", a)
print("\nb:\t", b)
```

```
a:   array([0, 1, 2, 3, 4], dtype=uint8)

b:   array([0, 1, 2, 3, 4], dtype=uint8)
```

will simply copy the content of `a` into `b` without any iteration, and will, therefore, be faster. Keep this in mind, whenever the output type, or performance is important.

4.3 Array initialisation functions

There are seven functions that can be used for initialising an array. These are bound to `ulab` itself at the top level, i.e., no module has to be imported for the function invocations.

4.3.1 arange

`numpy`: https://numpy.org/doc/stable/reference/generated/numpy.arange.html

The function returns a one-dimensional array with evenly spaced values. Takes 3 positional arguments (two are optional), and the `dtype` keyword argument.

```
# code to be run in micropython

import ulab

print(ulab.arange(10))
print(ulab.arange(2, 10))
```

(continues on next page)

(continued from previous page)

```
print(ulab.arange(2, 10, 3))
print(ulab.arange(2, 10, 3, dtype=ulab.float))
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=int16)
array([2, 3, 4, 5, 6, 7, 8, 9], dtype=int16)
array([2, 5, 8], dtype=int16)
array([2.0, 5.0, 8.0], dtype=float)
```

4.3.2 concatenate

`numpy`: https://numpy.org/doc/stable/reference/generated/numpy.concatenate.html

The function joins a sequence of arrays, if they are compatible in shape, if all shapes except the one along the joining axis are equal.

```
# code to be run in micropython

import ulab as np

a = np.array(range(25), dtype=np.uint8).reshape((5, 5))
b = np.array(range(15), dtype=np.uint8).reshape((3, 5))

c = np.concatenate((a, b), axis=0)
print(c)
```

```
array([[0, 1, 2, 3, 4],
    [5, 6, 7, 8, 9],
    [10, 11, 12, 13, 14],
    [15, 16, 17, 18, 19],
    [20, 21, 22, 23, 24],
    [0, 1, 2, 3, 4],
    [5, 6, 7, 8, 9],
    [10, 11, 12, 13, 14]], dtype=uint8)
```

WARNING: `numpy` accepts arbitrary `dtypes` in the sequence of arrays, in `ulab` the `dtypes` must be identical. If you want to concatenate different types, you have to convert all arrays to the same type first. Here `b` is of `float` type, so it cannot directly be concatenated to `a`. However, if we cast the `dtype` of `b`, the concatenation works:

```
# code to be run in micropython

import ulab as np

a = np.array(range(25), dtype=np.uint8).reshape((5, 5))
b = np.array(range(15), dtype=np.float).reshape((5, 3))
d = np.array(b+1, dtype=np.uint8)
print('a: ', a)
print('='*20 + '\nd: ', d)
c = np.concatenate((d, a), axis=1)
print('='*20 + '\nc: ', c)
```

```
a:  array([[0, 1, 2, 3, 4],
    [5, 6, 7, 8, 9],
    [10, 11, 12, 13, 14],
    [15, 16, 17, 18, 19],
```

(continues on next page)

(continued from previous page)

```
    [20, 21, 22, 23, 24]], dtype=uint8)
====================
d:  array([[1, 2, 3],
    [4, 5, 6],
    [7, 8, 9],
    [10, 11, 12],
    [13, 14, 15]], dtype=uint8)
====================
c:  array([[1, 2, 3, 0, 1, 2, 3, 4],
    [4, 5, 6, 5, 6, 7, 8, 9],
    [7, 8, 9, 10, 11, 12, 13, 14],
    [10, 11, 12, 15, 16, 17, 18, 19],
    [13, 14, 15, 20, 21, 22, 23, 24]], dtype=uint8)
```

4.3.3 eye

`numpy`: https://docs.scipy.org/doc/numpy/reference/generated/numpy.eye.html

Another special array method is the `eye` function, whose call signature is

```
eye(N, M, k=0, dtype=float)
```

where `N` (`M`) specify the dimensions of the matrix (if only `N` is supplied, then we get a square matrix, otherwise one with `M` rows, and `N` columns), and `k` is the shift of the ones (the main diagonal corresponds to `k=0`). Here are a couple of examples.

With a single argument

```
# code to be run in micropython

import ulab as np

print(np.eye(5))
```

```
array([[1.0, 0.0, 0.0, 0.0, 0.0],
       [0.0, 1.0, 0.0, 0.0, 0.0],
       [0.0, 0.0, 1.0, 0.0, 0.0],
       [0.0, 0.0, 0.0, 1.0, 0.0],
       [0.0, 0.0, 0.0, 0.0, 1.0]], dtype=float)
```

Specifying the dimensions of the matrix

```
# code to be run in micropython

import ulab as np

print(np.eye(4, M=6, k=-1, dtype=np.int16))
```

```
array([[0, 0, 0, 0],
       [1, 0, 0, 0],
       [0, 1, 0, 0],
```

(continues on next page)

(continued from previous page)

```
       [0, 0, 1, 0],
       [0, 0, 0, 1],
       [0, 0, 0, 0]], dtype=int16)
```

```
# code to be run in micropython

import ulab as np

print(np.eye(4, M=6, dtype=np.int8))
```

```
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0],
       [0, 0, 0, 1],
       [0, 0, 0, 0],
       [0, 0, 0, 0]], dtype=int8)
```

4.3.4 full

`numpy`: https://docs.scipy.org/doc/numpy/reference/generated/numpy.full.html

The function returns an array of arbitrary dimension, whose elements are all equal to the second positional argument. The first argument is a tuple describing the shape of the tensor. The `dtype` keyword argument with a default value of `float` can also be supplied.

```
# code to be run in micropython

import ulab as np

# create an array with the default type
print(np.full((2, 4), 3))

print('\n' + '='*20 + '\n')
# the array type is uint8 now
print(np.full((2, 4), 3, dtype=np.uint8))
```

```
array([[3.0, 3.0, 3.0, 3.0],
    [3.0, 3.0, 3.0, 3.0]], dtype=float)

====================

array([[3, 3, 3, 3],
    [3, 3, 3, 3]], dtype=uint8)
```

4.3.5 linspace

`numpy`: https://docs.scipy.org/doc/numpy/reference/generated/numpy.linspace.html

This function returns an array, whose elements are uniformly spaced between the `start`, and `stop` points. The number of intervals is determined by the `num` keyword argument, whose default value is 50. With the `endpoint` keyword argument (defaults to `True`) one can include `stop` in the sequence. In addition, the `dtype` keyword can be supplied to force type conversion of the output. The default is `float`. Note that, when `dtype` is of integer type, the sequence is not necessarily evenly spaced. This is not an error, rather a consequence of rounding. (This is also the `numpy` behaviour.)

```
# code to be run in micropython

import ulab as np

# generate a sequence with defaults
print('default sequence:\t', np.linspace(0, 10))

# num=5
print('num=5:\t\t\t', np.linspace(0, 10, num=5))

# num=5, endpoint=False
print('num=5:\t\t\t', np.linspace(0, 10, num=5, endpoint=False))

# num=5, endpoint=False, dtype=uint8
print('num=5:\t\t\t', np.linspace(0, 5, num=7, endpoint=False, dtype=np.uint8))
```

```
default sequence:    array([0.0, 0.2040816396474838, 0.4081632792949677, ..., 9.
↪591833114624023, 9.795914649963379, 9.999996185302734], dtype=float)
num=5:                         array([0.0, 2.5, 5.0, 7.5, 10.0], dtype=float)
num=5:                         array([0.0, 2.0, 4.0, 6.0, 8.0], dtype=float)
num=5:                         array([0, 0, 1, 2, 2, 3, 4], dtype=uint8)
```

4.3.6 logspace

`linspace`' equivalent for logarithmically spaced data is `logspace`. This function produces a sequence of numbers, in which the quotient of consecutive numbers is constant. This is a geometric sequence.

`numpy`: https://docs.scipy.org/doc/numpy/reference/generated/numpy.logspace.html

This function returns an array, whose elements are uniformly spaced between the `start`, and `stop` points. The number of intervals is determined by the `num` keyword argument, whose default value is 50. With the `endpoint` keyword argument (defaults to `True`) one can include `stop` in the sequence. In addition, the `dtype` keyword can be supplied to force type conversion of the output. The default is `float`. Note that, exactly as in `linspace`, when `dtype` is of integer type, the sequence is not necessarily evenly spaced in log space.

In addition to the keyword arguments found in `linspace`, `logspace` also accepts the `base` argument. The default value is 10.

```
# code to be run in micropython

import ulab as np

# generate a sequence with defaults
print('default sequence:\t', np.logspace(0, 3))

# num=5
print('num=5:\t\t\t', np.logspace(1, 10, num=5))

# num=5, endpoint=False
print('num=5:\t\t\t', np.logspace(1, 10, num=5, endpoint=False))

# num=5, endpoint=False
print('num=5:\t\t\t', np.logspace(1, 10, num=5, endpoint=False, base=2))
```

```
default sequence:    array([1.0, 1.151395399326447, 1.325711365590109, ..., 754.
↪3120063354646, 868.5113737513561, 1000.000000000004], dtype=float)
num=5:                       array([10.0, 1778.279410038923, 316227.766016838,␣
↪56234132.5190349, 10000000000.0], dtype=float)
num=5:                       array([10.0, 630.9573444801933, 39810.71705534974,␣
↪2511886.431509581, 158489319.2461114], dtype=float)
num=5:                       array([2.0, 6.964404506368993, 24.25146506416637, 84.
↪44850628946524, 294.066778879241], dtype=float)
```

4.3.7 ones, zeros

numpy: https://docs.scipy.org/doc/numpy/reference/generated/numpy.zeros.html

numpy: https://docs.scipy.org/doc/numpy/reference/generated/numpy.ones.html

A couple of special arrays and matrices can easily be initialised by calling one of the `ones`, or `zeros` functions. `ones` and `zeros` follow the same pattern, and have the call signature

```
ones(shape, dtype=float)
zeros(shape, dtype=float)
```

where shape is either an integer, or a 2-tuple.

```
# code to be run in micropython

import ulab as np

print(np.ones(6, dtype=np.uint8))
print(np.zeros((6, 4)))
```

```
array([1, 1, 1, 1, 1, 1], dtype=uint8)
array([[0.0, 0.0, 0.0, 0.0],
     [0.0, 0.0, 0.0, 0.0],
     [0.0, 0.0, 0.0, 0.0],
     [0.0, 0.0, 0.0, 0.0],
     [0.0, 0.0, 0.0, 0.0],
     [0.0, 0.0, 0.0, 0.0]], dtype=float)
```

4.4 Customising array printouts

`ndarrays` are pretty-printed, i.e., if the number of entries along the last axis is larger than 10 (default value), then only the first and last three entries will be printed. Also note that, as opposed to `numpy`, the printout always contains the `dtype`.

```
# code to be run in micropython

import ulab as np

a = np.array(range(200))
print("a:\t", a)
```

```
a:   array([0.0, 1.0, 2.0, ..., 197.0, 198.0, 199.0], dtype=float)
```

4.4.1 set_printoptions

The default values can be overwritten by means of the `set_printoptions` function numpy.set_printoptions, which accepts two keywords arguments, the `threshold`, and the `edgeitems`. The first of these arguments determines the length of the longest array that will be printed in full, while the second is the number of items that will be printed on the left and right hand side of the ellipsis, if the array is longer than `threshold`.

```
# code to be run in micropython

import ulab as np

a = np.array(range(20))
print("a printed with defaults:\t", a)

np.set_printoptions(threshold=200)
print("\na printed in full:\t\t", a)

np.set_printoptions(threshold=10, edgeitems=2)
print("\na truncated with 2 edgeitems:\t", a)
```

```
a printed with defaults:     array([0.0, 1.0, 2.0, ..., 17.0, 18.0, 19.0],␣
↪dtype=float)

a printed in full:           array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0,␣
↪10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0], dtype=float)

a truncated with 2 edgeitems:        array([0.0, 1.0, ..., 18.0, 19.0], dtype=float)
```

4.4.2 get_printoptions

The set value of the `threshold` and `edgeitems` can be retrieved by calling the `get_printoptions` function with no arguments. The function returns a *dictionary* with two keys.

```
# code to be run in micropython

import ulab as np

np.set_printoptions(threshold=100, edgeitems=20)
print(np.get_printoptions())
```

```
{'threshold': 100, 'edgeitems': 20}
```

4.5 Methods and properties of ndarrays

Arrays have several *properties* that can queried, and some methods that can be called. With the exception of the flatten and transpose operators, properties return an object that describe some feature of the array, while the methods return a new array-like object.

4.5.1 .copy

The `.copy` method creates a new *deep copy* of an array, i.e., the entries of the source array are *copied* into the target array.

```
# code to be run in micropython

import ulab as np

a = np.array([1, 2, 3, 4], dtype=np.int8)
b = a.copy()
print('a: ', a)
print('='*20)
print('b: ', b)
```

```
a:  array([1, 2, 3, 4], dtype=int8)
====================
b:  array([1, 2, 3, 4], dtype=int8)
```

4.5.2 .dtype

`numpy`: https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.dtype.htm

The `.dtype` property returns the `dtype` of an array. This can then be used for initialising another array with the matching type. `ulab` implements two versions of `dtype`; one that is `numpy`-like, i.e., one, which returns a `dtype` object, and one that is significantly cheaper in terms of flash space, but does not define a the `dtype` object, and returns a single character (number) instead.

WARNING: in `circuitpython`:

```
# code to be run in micropython

import ulab as np

a = np.array([1, 2, 3, 4], dtype=np.int8)
b = np.array([5, 6, 7], dtype=a.dtype)
print('a: ', a)
print('dtype of a: ', a.dtype)
print('\nb: ', b)
```

```
a:  array([1, 2, 3, 4], dtype=int8)
dtype of a:  dtype('int8')

b:  array([5, 6, 7], dtype=int8)
```

WARNING: in `micropython`:

```
# code to be run in micropython

import ulab as np

a = np.array([1, 2, 3, 4], dtype=np.int8)
b = np.array([5, 6, 7], dtype=a.dtype())
print('a: ', a)
print('dtype of a: ', a.dtype())
print('\nb: ', b)
```

```
a:  array([1, 2, 3, 4], dtype=int8)
dtype of a:  dtype('int8')
```

(continues on next page)

(continued from previous page)

```
b:  array([5, 6, 7], dtype=int8)
```

If the `ulab.h` header file sets the pre-processor constant `ULAB_HAS_DTYPE_OBJECT` to 0, then the output of the previous snippet will be

```
# code to be run in micropython

import ulab as np

a = np.array([1, 2, 3, 4], dtype=np.int8)
b = np.array([5, 6, 7], dtype=a.dtype())
print('a: ', a)
print('dtype of a: ', a.dtype())
print('\nb: ', b)
```

```
a:  array([1, 2, 3, 4], dtype=int8)
dtype of a:  98

b:  array([5, 6, 7], dtype=int8)
```

Here 98 is nothing but the ASCII value of the character `b`, which is the type code for signed 8-bit integers.

4.5.3 .flatten

`numpy`: https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.flatten.htm

`.flatten` returns the flattened array. The array can be flattened in `C` style (i.e., moving along the last axis in the tensor), or in `fortran` style (i.e., moving along the first axis in the tensor).

```
# code to be run in micropython

import ulab as np

a = np.array([1, 2, 3, 4], dtype=np.int8)
print("a: \t\t", a)
print("a flattened: \t", a.flatten())

b = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int8)
print("\nb:", b)

print("b flattened (C): \t", b.flatten())
print("b flattened (F): \t", b.flatten(order='F'))
```

```
a:            array([1, 2, 3, 4], dtype=int8)
a flattened:          array([1, 2, 3, 4], dtype=int8)

b: array([[1, 2, 3],
    [4, 5, 6]], dtype=int8)
b flattened (C):     array([1, 2, 3, 4, 5, 6], dtype=int8)
b flattened (F):     array([1, 4, 2, 5, 3, 6], dtype=int8)
```

4.5.4 .itemsize

numpy: https://numpy.org/doc/stable/reference/generated/numpy.ndarray.itemsize.html

The .itemsize method (property) returns an integer with the size of elements in the array.

WARNING: In circuitpython:

```
# code to be run in micropython

import ulab as np

a = np.array([1, 2, 3], dtype=np.int8)
print("a:\n", a)
print("itemsize of a:", a.itemsize)

b= np.array([[1, 2], [3, 4]], dtype=np.float)
print("\nb:\n", b)
print("itemsize of b:", b.itemsize)
```

```
a:
 array([1, 2, 3], dtype=int8)
itemsize of a: 1

b:
 array([[1.0, 2.0],
     [3.0, 4.0]], dtype=float)
itemsize of b: 8
```

WARNING: In micropython:

```
# code to be run in micropython

import ulab as np

a = np.array([1, 2, 3], dtype=np.int8)
print("a:\n", a)
print("itemsize of a:", a.itemsize)

b= np.array([[1, 2], [3, 4]], dtype=np.float)
print("\nb:\n", b)
print("itemsize of b:", b.itemsize())
```

```
a:
 array([1, 2, 3], dtype=int8)
itemsize of a: 1

b:
 array([[1.0, 2.0],
     [3.0, 4.0]], dtype=float)
itemsize of b: 8
```

4.5.5 .reshape

numpy: https://docs.scipy.org/doc/numpy/reference/generated/numpy.reshape.html reshape re-writes the shape properties of an ndarray, but the array will not be modified in any other way. The function takes a single 2-tuple with two integers as its argument. The 2-tuple should specify the desired number of rows and columns. If the new shape is not consistent with the old, a ValueError exception will be raised.

```
# code to be run in micropython

import ulab as np

a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]],␣
↪dtype=np.uint8)
print('a (4 by 4):', a)
print('a (2 by 8):', a.reshape((2, 8)))
print('a (1 by 16):', a.reshape((1, 16)))
```

```
a (4 by 4): array([[1, 2, 3, 4],
     [5, 6, 7, 8],
     [9, 10, 11, 12],
     [13, 14, 15, 16]], dtype=uint8)
a (2 by 8): array([[1, 2, 3, 4, 5, 6, 7, 8],
     [9, 10, 11, 12, 13, 14, 15, 16]], dtype=uint8)
a (1 by 16): array([1, 2, 3, ..., 14, 15, 16], dtype=uint8)
```

4.5.6 .shape

numpy: https://numpy.org/doc/stable/reference/generated/numpy.ndarray.shape.html

The .shape method (property) returns a tuple with the length of the array in along each dimension.

WARNING: In circuitpython, you can call the method as a property, i.e.,

```
# code to be run in micropython

import ulab as np

a = np.array([1, 2, 3, 4], dtype=np.int8)
print("a:\n", a)
print("shape of a:", a.shape)

b= np.array([[1, 2], [3, 4]], dtype=np.int8)
print("\nb:\n", b)
print("shape of b:", b.shape)
```

```
a:
 array([1, 2, 3, 4], dtype=int8)
shape of a: (1, 4)

b:
 array([[1, 2],
     [3, 4]], dtype=int8)
shape of b: (2, 2)
```

WARNING: On the other hand, since properties are not implemented in micropython, there you would call the method as a function, i.e.,

```
# code to be run in micropython
```

(continues on next page)

(continued from previous page)

```
import ulab as np

a = np.array([1, 2, 3, 4], dtype=np.int8)
print("a:\n", a)
print("shape of a:", a.shape)

b= np.array([[1, 2], [3, 4]], dtype=np.int8)
print("\nb:\n", b)
print("shape of b:", b.shape())
```

```
a:
 array([1, 2, 3, 4], dtype=int8)
shape of a: (1, 4)

b:
 array([[1, 2],
     [3, 4]], dtype=int8)
shape of b: (2, 2)
```

4.5.7 .size

numpy: https://numpy.org/doc/stable/reference/generated/numpy.ndarray.size.html

The `.size` method (property) returns an integer with the number of elements in the array.

WARNING: In `circuitpython`, the `numpy` nomenclature applies, i.e.,

```
# code to be run in micropython

import ulab as np

a = np.array([1, 2, 3], dtype=np.int8)
print("a:\n", a)
print("size of a:", a.size)

b= np.array([[1, 2], [3, 4]], dtype=np.int8)
print("\nb:\n", b)
print("size of b:", b.size)
```

```
a:
 array([1, 2, 3], dtype=int8)
size of a: 3

b:
 array([[1, 2],
     [3, 4]], dtype=int8)
size of b: 4
```

WARNING: In `micropython`, `size` is a method, i.e.,

```
# code to be run in micropython

import ulab as np

a = np.array([1, 2, 3], dtype=np.int8)
```

(continues on next page)

(continued from previous page)

```
print("a:\n", a)
print("size of a:", a.size)

b= np.array([[1, 2], [3, 4]], dtype=np.int8)
print("\nb:\n", b)
print("size of b:", b.size())
```

```
a:
 array([1, 2, 3], dtype=int8)
size of a: 3

b:
 array([[1, 2],
     [3, 4]], dtype=int8)
size of b: 4
```

4.5.8 .tobytes

numpy: https://numpy.org/doc/stable/reference/generated/numpy.ndarray.tobytes.html

The .tobytes method can be used for acquiring a handle of the underlying data pointer of an array, and it returns a new bytearray that can be fed into any method that can accep a bytearray, e.g., ADC data can be buffered into this bytearray, or the bytearray can be fed into a DAC. Since the bytearray is really nothing but the bare data container of the array, any manipulation on the bytearray automatically modifies the array itself.

Note that the method raises a ValueError exception, if the array is not dense (i.e., it has already been sliced).

```
# code to be run in micropython

import ulab as np

a = np.array(range(8), dtype=np.uint8)
print('a: ', a)
b = a.tobytes()
print('b: ', b)

# modify b
b[0] = 13

print('='*20)
print('b: ', b)
print('a: ', a)
```

```
a:  array([0, 1, 2, 3, 4, 5, 6, 7], dtype=uint8)
b:  bytearray(b'x00x01x02x03x04x05x06x07')
====================
b:  bytearray(b'rx01x02x03x04x05x06x07')
a:  array([13, 1, 2, 3, 4, 5, 6, 7], dtype=uint8)
```

4.5.9 .transpose

numpy: https://docs.scipy.org/doc/numpy/reference/generated/numpy.transpose.html

Returns the transposed array. Only defined, if the number of maximum dimensions is larger than 1.

```
# code to be run in micropython

import ulab as np

a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]], dtype=np.uint8)
print('a:\n', a)
print('shape of a:', a.shape())
a.transpose()
print('\ntranspose of a:\n', a)
print('shape of a:', a.shape())
```

```
a:
 array([[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9],
     [10, 11, 12]], dtype=uint8)
shape of a: (4, 3)

transpose of a:
 array([[1, 4, 7, 10],
     [2, 5, 8, 11],
     [3, 6, 9, 12]], dtype=uint8)
shape of a: (3, 4)
```

4.5.10 .sort

numpy: https://docs.scipy.org/doc/numpy/reference/generated/numpy.sort.html

In-place sorting of an `ndarray`. For a more detailed exposition, see *sort*.

```
# code to be run in micropython

import ulab as np

a = np.array([[1, 12, 3, 0], [5, 3, 4, 1], [9, 11, 1, 8], [7, 10, 0, 1]], dtype=np.
↪uint8)
print('\na:\n', a)
a.sort(axis=0)
print('\na sorted along vertical axis:\n', a)

a = np.array([[1, 12, 3, 0], [5, 3, 4, 1], [9, 11, 1, 8], [7, 10, 0, 1]], dtype=np.
↪uint8)
a.sort(a, axis=1)
print('\na sorted along horizontal axis:\n', a)

a = np.array([[1, 12, 3, 0], [5, 3, 4, 1], [9, 11, 1, 8], [7, 10, 0, 1]], dtype=np.
↪uint8)
a.sort(a, axis=None)
print('\nflattened a sorted:\n', a)
```

```
a:
 array([[1, 12, 3, 0],
     [5, 3, 4, 1],
     [9, 11, 1, 8],
     [7, 10, 0, 1]], dtype=uint8)
```

(continues on next page)

(continued from previous page)

```

a sorted along vertical axis:
 array([[1, 3, 0, 0],
     [5, 10, 1, 1],
     [7, 11, 3, 1],
     [9, 12, 4, 8]], dtype=uint8)

a sorted along horizontal axis:
 array([[0, 1, 3, 12],
     [1, 3, 4, 5],
     [1, 8, 9, 11],
     [0, 1, 7, 10]], dtype=uint8)

flattened a sorted:
 array([0, 0, 1, ..., 10, 11, 12], dtype=uint8)
```

4.6 Unary operators

With the exception of `len`, which returns a single number, all unary operators manipulate the underlying data element-wise.

4.6.1 len

This operator takes a single argument, the array, and returns either the length of the first axis.

```
# code to be run in micropython

import ulab as np

a = np.array([1, 2, 3, 4, 5], dtype=np.uint8)
b = np.array([range(5), range(5), range(5), range(5)], dtype=np.uint8)

print("a:\t", a)
print("length of a: ", len(a))
print("shape of a: ", a.shape())
print("\nb:\t", b)
print("length of b: ", len(b))
print("shape of b: ", b.shape())
```

```
a:   array([1, 2, 3, 4, 5], dtype=uint8)
length of a:  5
shape of a:  (1, 5)

b:   array([[0, 1, 2, 3, 4],
     [0, 1, 2, 3, 4],
     [0, 1, 2, 3, 4],
     [0, 1, 2, 3, 4]], dtype=uint8)
length of b:  4
shape of b:  (4, 5)
```

The number returned by `len` is also the length of the iterations, when the array supplies the elements for an iteration (see later).

4.6.2 invert

The function is defined for integer data types (`uint8`, `int8`, `uint16`, and `int16`) only, takes a single argument, and returns the element-by-element, bit-wise inverse of the array. If a `float` is supplied, the function raises a `ValueError` exception.

With signed integers (`int8`, and `int16`), the results might be unexpected, as in the example below:

```
# code to be run in micropython

import ulab as np

a = np.array([0, -1, -100], dtype=np.int8)
print("a:\t\t", a)
print("inverse of a:\t", ~a)

a = np.array([0, 1, 254, 255], dtype=np.uint8)
print("\na:\t\t", a)
print("inverse of a:\t", ~a)
```

```
a:           array([0, -1, -100], dtype=int8)
inverse of a:        array([-1, 0, 99], dtype=int8)

a:           array([0, 1, 254, 255], dtype=uint8)
inverse of a:        array([255, 254, 1, 0], dtype=uint8)
```

4.6.3 abs

This function takes a single argument, and returns the element-by-element absolute value of the array. When the data type is unsigned (`uint8`, or `uint16`), a copy of the array will be returned immediately, and no calculation takes place.

```
# code to be run in micropython

import ulab as np

a = np.array([0, -1, -100], dtype=np.int8)
print("a:\t\t\t ", a)
print("absolute value of a:\t ", abs(a))
```

```
a:                      array([0, -1, -100], dtype=int8)
absolute value of a:            array([0, 1, 100], dtype=int8)
```

4.6.4 neg

This operator takes a single argument, and changes the sign of each element in the array. Unsigned values are wrapped.

```
# code to be run in micropython

import ulab as np

a = np.array([10, -1, 1], dtype=np.int8)
print("a:\t\t", a)
```

(continues on next page)

(continued from previous page)

```
print("negative of a:\t", -a)

b = np.array([0, 100, 200], dtype=np.uint8)
print("\nb:\t\t", b)
print("negative of b:\t", -b)
```

```
a:		array([10, -1, 1], dtype=int8)
negative of a:		array([-10, 1, -1], dtype=int8)

b:		array([0, 100, 200], dtype=uint8)
negative of b:		array([0, 156, 56], dtype=uint8)
```

4.6.5 pos

This function takes a single argument, and simply returns a copy of the array.

```
# code to be run in micropython

import ulab as np

a = np.array([10, -1, 1], dtype=np.int8)
print("a:\t\t", a)
print("positive of a:\t", +a)
```

```
a:		array([10, -1, 1], dtype=int8)
positive of a:		array([10, -1, 1], dtype=int8)
```

4.7 Binary operators

`ulab` implements the `+`, `-`, `*`, `/`, `**`, `<`, `>`, `<=`, `>=`, `==`, `!=`, `+=`, `-=`, `*=`, `/=`, `**=` binary operators that work element-wise. Broadcasting is available, meaning that the two operands do not even have to have the same shape. If the lengths along the respective axes are equal, or one of them is 1, or the axis is missing, the element-wise operation can still be carried out. A thorough explanation of broadcasting can be found under https://numpy.org/doc/stable/user/basics.broadcasting.html.

WARNING: note that relational operators (`<`, `>`, `<=`, `>=`, `==`, `!=`) should have the `ndarray` on their left hand side, when compared to scalars. This means that the following works

```
# code to be run in micropython

import ulab
a = ulab.array([1, 2, 3])
print(a > 2)
```

```
array([False, False, True], dtype=bool)
```

while the equivalent statement, `2 < a`, will raise a `TypeError` exception:

```
# code to be run in micropython

import ulab
```

(continues on next page)

(continued from previous page)

```
a = ulab.array([1, 2, 3])
print(2 < a)
```

```
Traceback (most recent call last):
  File "/dev/shm/micropython.py", line 4, in <module>
TypeError: unsupported types for __lt__: 'int', 'ndarray'
```

WARNING: `circuitpython` users should use the `equal`, and `not_equal` operators instead of `==`, and `!=`. See the section on *array comparison* for details.

4.7.1 Upcasting

Binary operations require special attention, because two arrays with different typecodes can be the operands of an operation, in which case it is not trivial, what the typecode of the result is. This decision on the result's typecode is called upcasting. Since the number of typecodes in `ulab` is significantly smaller than in `numpy`, we have to define new upcasting rules. Where possible, I followed `numpy`'s conventions.

`ulab` observes the following upcasting rules:

1. Operations on two `ndarray`s of the same `dtype` preserve their `dtype`, even when the results overflow.
2. if either of the operands is a float, the result is automatically a float
3. When one of the operands is a scalar, it will internally be turned into a single-element `ndarray` with the *smallest* possible `dtype`. Thus, e.g., if the scalar is 123, it will be converted into an array of `dtype` `uint8`, while -1000 will be converted into `int16`. An `mp_obj_float`, will always be promoted to `dtype` `float`. Other micropython types (e.g., lists, tuples, etc.) raise a `TypeError` exception.
4.

left hand side	right hand side	ulab result	numpy result
`uint8`	`int8`	`int16`	`int16`
`uint8`	`int16`	`int16`	`int16`
`uint8`	`uint16`	`uint16`	`uint16`
`int8`	`int16`	`int16`	`int16`
`int8`	`uint16`	`uint16`	`int32`
`uint16`	`int16`	`float`	`int32`

Note that the last two operations are promoted to `int32` in `numpy`.

WARNING: Due to the lower number of available data types, the upcasting rules of `ulab` are slightly different to those of `numpy`. Watch out for this, when porting code!

Upcasting can be seen in action in the following snippet:

```
# code to be run in micropython

import ulab as np

a = np.array([1, 2, 3, 4], dtype=np.uint8)
b = np.array([1, 2, 3, 4], dtype=np.int8)
print("a:\t", a)
print("b:\t", b)
print("a+b:\t", a+b)
```

(continues on next page)

(continued from previous page)

```
c = np.array([1, 2, 3, 4], dtype=np.float)
print("\na:\t", a)
print("c:\t", c)
print("a*c:\t", a*c)
```

```
a:   array([1, 2, 3, 4], dtype=uint8)
b:   array([1, 2, 3, 4], dtype=int8)
a+b:         array([2, 4, 6, 8], dtype=int16)

a:   array([1, 2, 3, 4], dtype=uint8)
c:   array([1.0, 2.0, 3.0, 4.0], dtype=float)
a*c:         array([1.0, 4.0, 9.0, 16.0], dtype=float)
```

4.7.2 Benchmarks

The following snippet compares the performance of binary operations to a possible implementation in python. For the time measurement, we will take the following snippet from the micropython manual:

```
# code to be run in micropython

import utime

def timeit(f, *args, **kwargs):
    func_name = str(f).split(' ')[1]
    def new_func(*args, **kwargs):
        t = utime.ticks_us()
        result = f(*args, **kwargs)
        print('execution time: ', utime.ticks_diff(utime.ticks_us(), t), ' us')
        return result
    return new_func
```

```
# code to be run in micropython

import ulab as np

@timeit
def py_add(a, b):
    return [a[i]+b[i] for i in range(1000)]

@timeit
def py_multiply(a, b):
    return [a[i]*b[i] for i in range(1000)]

@timeit
def ulab_add(a, b):
    return a + b

@timeit
def ulab_multiply(a, b):
    return a * b

a = [0.0]*1000
b = range(1000)
```

(continues on next page)

(continued from previous page)

```
print('python add:')
py_add(a, b)

print('\npython multiply:')
py_multiply(a, b)

a = np.linspace(0, 10, num=1000)
b = np.ones(1000)

print('\nulab add:')
ulab_add(a, b)

print('\nulab multiply:')
ulab_multiply(a, b)
```

```
python add:
execution time:  10051  us

python multiply:
execution time:  14175  us

ulab add:
execution time:  222  us

ulab multiply:
execution time:  213  us
```

The python implementation above is not perfect, and certainly, there is much room for improvement. However, the factor of 50 difference in execution time is very spectacular. This is nothing but a consequence of the fact that the `ulab` functions run `C` code, with very little python overhead. The factor of 50 appears to be quite universal: the FFT routine obeys similar scaling (see *Speed of FFTs*), and this number came up with font rendering, too: fast font rendering on graphical displays.

4.8 Comparison operators

The smaller than, greater than, smaller or equal, and greater or equal operators return a vector of Booleans indicating the positions (`True`), where the condition is satisfied.

```
# code to be run in micropython

import ulab as np

a = np.array([1, 2, 3, 4, 5, 6, 7, 8], dtype=np.uint8)
print(a < 5)
```

```
array([True, True, True, True, False, False, False, False], dtype=bool)
```

WARNING: at the moment, due to `micropython`'s implementation details, the `ndarray` must be on the left hand side of the relational operators.

That is, while `a < 5` and `5 > a` have the same meaning, the following code will not work:

```
# code to be run in micropython

import ulab as np

a = np.array([1, 2, 3, 4, 5, 6, 7, 8], dtype=np.uint8)
print(5 > a)
```

```
Traceback (most recent call last):
  File "/dev/shm/micropython.py", line 5, in <module>
TypeError: unsupported types for __gt__: 'int', 'ndarray'
```

4.9 Iterating over arrays

`ndarrays` are iterable, which means that their elements can also be accessed as can the elements of a list, tuple, etc. If the array is one-dimensional, the iterator returns scalars, otherwise a new reduced-dimensional *view* is created and returned.

```
# code to be run in micropython

import ulab as np

a = np.array([1, 2, 3, 4, 5], dtype=np.uint8)
b = np.array([range(5), range(10, 15, 1), range(20, 25, 1), range(30, 35, 1)],␣
↪dtype=np.uint8)

print("a:\t", a)

for i, _a in enumerate(a):
    print("element %d in a:"%i, _a)

print("\nb:\t", b)

for i, _b in enumerate(b):
    print("element %d in b:"%i, _b)
```

```
a:   array([1, 2, 3, 4, 5], dtype=uint8)
element 0 in a: 1
element 1 in a: 2
element 2 in a: 3
element 3 in a: 4
element 4 in a: 5

b:   array([[0, 1, 2, 3, 4],
    [10, 11, 12, 13, 14],
    [20, 21, 22, 23, 24],
    [30, 31, 32, 33, 34]], dtype=uint8)
element 0 in b: array([0, 1, 2, 3, 4], dtype=uint8)
element 1 in b: array([10, 11, 12, 13, 14], dtype=uint8)
element 2 in b: array([20, 21, 22, 23, 24], dtype=uint8)
element 3 in b: array([30, 31, 32, 33, 34], dtype=uint8)
```

4.10 Slicing and indexing

4.10.1 Views vs. copies

`numpy` has a very important concept called *views*, which is a powerful extension of `python`'s own notion of slicing. Slices are special python objects of the form

```
slice = start:end:stop
```

where `start`, `end`, and `stop` are (not necessarily non-negative) integers. Not all of these three numbers must be specified in an index, in fact, all three of them can be missing. The interpreter takes care of filling in the missing values. (Note that slices cannot be defined in this way, only there, where an index is expected.) For a good explanation on how slices work in python, you can read the stackoverflow question https://stackoverflow.com/questions/509211/understanding-slice-notation.

In order to see what slicing does, let us take the string `a = '012345679'`! We can extract every second character by creating the slice `::2`, which is equivalent to `0:len(a):2`, i.e., increments the character pointer by 2 starting from 0, and traversing the string up to the very end.

```
# code to be run in CPython

string = '0123456789'
string[::2]
```

```
'02468'
```

Now, we can do the same with numerical arrays.

```
# code to be run in micropython

import ulab as np

a = np.array(range(10), dtype=np.uint8)
print('a:\t', a)

print('a[::2]:\t', a[::2])
```

```
a:   array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)
a[::2]:      array([0, 2, 4, 6, 8], dtype=uint8)
```

This looks similar to `string` above, but there is a very important difference that is not so obvious. Namely, `string[::2]` produces a partial copy of `string`, while `a[::2]` only produces a *view* of `a`. What this means is that `a`, and `a[::2]` share their data, and the only difference between the two is, how the data are read out. In other words, internally, `a[::2]` has the same data pointer as `a`. We can easily convince ourselves that this is indeed the case by calling the *ndinfo* function: the *data pointer* entry is the same in the two printouts.

```
# code to be run in micropython

import ulab as np

a = np.array(range(10), dtype=np.uint8)
print('a: ', a, '\n')
np.ndinfo(a)
print('\n' + '='*20)
print('a[::2]: ', a[::2], '\n')
np.ndinfo(a[::2])
```

```
a:  array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)

class: ndarray
shape: (10,)
strides: (1,)
itemsize: 1
data pointer: 0x7ff6c6193220
type: uint8

====================
a[::2]:  array([0, 2, 4, 6, 8], dtype=uint8)

class: ndarray
shape: (5,)
strides: (2,)
itemsize: 1
data pointer: 0x7ff6c6193220
type: uint8
```

If you are still a bit confused about the meaning of *views*, the section *Slicing and assigning to slices* should clarify the issue.

4.10.2 Indexing

The simplest form of indexing is specifying a single integer between the square brackets as in

```
# code to be run in micropython

import ulab as np

a = np.array(range(10), dtype=np.uint8)
print("a: ", a)
print("the first, and last element of a:\n", a[0], a[-1])
print("the second, and last but one element of a:\n", a[1], a[-2])
```

```
a:  array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)
the first, and last element of a:
 0 9
the second, and last but one element of a:
 1 8
```

Indexing can be applied to higher-dimensional tensors, too. When the length of the indexing sequences is smaller than the number of dimensions, a new *view* is returned, otherwise, we get a single number.

```
# code to be run in micropython

import ulab as np

a = np.array(range(9), dtype=np.uint8).reshape((3, 3))
print("a:\n", a)
print("a[0]:\n", a[0])
print("a[1,1]: ", a[1,1])
```

```
a:
 array([[0, 1, 2],
```

(continues on next page)

(continued from previous page)

```
    [3, 4, 5],
    [6, 7, 8]], dtype=uint8)
a[0]:
 array([[0, 1, 2]], dtype=uint8)
a[1,1]:  4
```

Indices can also be a list of Booleans. By using a Boolean list, we can select those elements of an array that satisfy a specific condition. At the moment, such indexing is defined for row vectors only; when the rank of the tensor is higher than 1, the function raises a `NotImplementedError` exception, though this will be rectified in a future version of `ulab`.

```
# code to be run in micropython

import ulab as np

a = np.array(range(9), dtype=np.float)
print("a:\t", a)
print("a < 5:\t", a[a < 5])
```

```
a:   array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0], dtype=float)
a < 5:       array([0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
```

Indexing with Boolean arrays can take more complicated expressions. This is a very concise way of comparing two vectors, e.g.:

```
# code to be run in micropython

import ulab as np

a = np.array(range(9), dtype=np.uint8)
b = np.array([4, 4, 4, 3, 3, 3, 13, 13, 13], dtype=np.uint8)
print("a:\t", a)
print("\na**2:\t", a*a)
print("\nb:\t", b)
print("\n100*sin(b):\t", np.sin(b)*100.0)
print("\na[a*a > np.sin(b)*100.0]:\t", a[a*a > np.sin(b)*100.0])
```

```
a:   array([0, 1, 2, 3, 4, 5, 6, 7, 8], dtype=uint8)

a**2:        array([0, 1, 4, 9, 16, 25, 36, 49, 64], dtype=uint16)

b:   array([4, 4, 4, 3, 3, 3, 13, 13, 13], dtype=uint8)

100*sin(b):  array([-75.68024953079282, -75.68024953079282, -75.68024953079282, 14.
↪11200080598672, 14.11200080598672, 14.11200080598672, 42.01670368266409, 42.
↪01670368266409, 42.01670368266409], dtype=float)

a[a*a > np.sin(b)*100.0]:    array([0, 1, 2, 4, 5, 7, 8], dtype=uint8)
```

Boolean indices can also be used in assignments, if the array is one-dimensional. The following example replaces the data in an array, wherever some condition is fulfilled.

```
# code to be run in micropython

import ulab as np
```

(continues on next page)

(continued from previous page)

```
a = np.array(range(9), dtype=np.uint8)
b = np.array(range(9)) + 12

print(a[b < 15])

a[b < 15] = 123
print(a)
```

```
array([0, 1, 2], dtype=uint8)
array([123, 123, 123, 3, 4, 5, 6, 7, 8], dtype=uint8)
```

On the right hand side of the assignment we can even have another array.

```
# code to be run in micropython

import ulab as np

a = np.array(range(9), dtype=np.uint8)
b = np.array(range(9)) + 12

print(a[b < 15], b[b < 15])

a[b < 15] = b[b < 15]
print(a)
```

```
array([0, 1, 2], dtype=uint8) array([12.0, 13.0, 14.0], dtype=float)
array([12, 13, 14, 3, 4, 5, 6, 7, 8], dtype=uint8)
```

4.10.3 Slicing and assigning to slices

You can also generate sub-arrays by specifying slices as the index of an array. Slices are special python objects of the form

```
# code to be run in micropython

import ulab as np

a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype=np.uint8)
print('a:\n', a)

# the first row
print('\na[0]:\n', a[0])

# the first two elements of the first row
print('\na[0,:2]:\n', a[0,:2])

# the zeroth element in each row (also known as the zeroth column)
print('\na[:,0]:\n', a[:,0])

# the last row
print('\na[-1]:\n', a[-1])
```

(continues on next page)

(continued from previous page)

```
# the last two rows backwards
print('\na[-1:-3:-1]:\n', a[-1:-3:-1])
```

```
a:
 array([[1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]], dtype=uint8)

a[0]:
 array([[1, 2, 3]], dtype=uint8)

a[0,:2]:
 array([[1, 2]], dtype=uint8)

a[:,0]:
 array([[1],
    [4],
    [7]], dtype=uint8)

a[-1]:
 array([[7, 8, 9]], dtype=uint8)

a[-1:-3:-1]:
 array([[7, 8, 9],
    [4, 5, 6]], dtype=uint8)
```

Assignment to slices can be done for the whole slice, per row, and per column. A couple of examples should make these statements clearer:

```
# code to be run in micropython

import ulab as np

a = np.zeros((3, 3), dtype=np.uint8)
print('a:\n', a)

# assigning to the whole row
a[0] = 1
print('\na[0] = 1\n', a)

a = np.zeros((3, 3), dtype=np.uint8)

# assigning to a column
a[:,2] = 3.0
print('\na[:,0]:\n', a)
```

```
a:
 array([[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]], dtype=uint8)

a[0] = 1
 array([[1, 1, 1],
    [0, 0, 0],
    [0, 0, 0]], dtype=uint8)
```

(continues on next page)

(continued from previous page)

```
a[:,0]:
 array([[0, 0, 3],
    [0, 0, 3],
    [0, 0, 3]], dtype=uint8)
```

Now, you should notice that we re-set the array `a` after the first assignment. Do you care to see what happens, if we do not do that? Well, here are the results:

```
# code to be run in micropython

import ulab as np

a = np.zeros((3, 3), dtype=np.uint8)
b = a[:,:]
# assign 1 to the first row
b[0] = 1

# assigning to the last column
b[:,2] = 3
print('a: ', a)
```

```
a:  array([[1, 1, 3],
    [0, 0, 3],
    [0, 0, 3]], dtype=uint8)
```

Note that both assignments involved `b`, and not `a`, yet, when we print out `a`, its entries are updated. This proves our earlier statement about the behaviour of *views*: in the statement `b = a[:,:]` we simply created a *view* of `a`, and not a *deep* copy of it, meaning that whenever we modify `b`, we actually modify `a`, because the underlying data container of `a` and `b` are shared between the two object. Having a single data container for two seemingly different objects provides an extremely powerful way of manipulating sub-sets of numerical data.

If you want to work on a *copy* of your data, you can use the `.copy` method of the `ndarray`. The following snippet should drive the point home:

```
# code to be run in micropython

import ulab as np

a = np.zeros((3, 3), dtype=np.uint8)
b = a.copy()

# get the address of the underlying data pointer

np.ndinfo(a)
print()
np.ndinfo(b)

# assign 1 to the first row of b, and do not touch a
b[0] = 1

print()
print('a: ', a)
print('='*20)
print('b: ', b)
```

```
class: ndarray
shape: (3, 3)
strides: (3, 1)
itemsize: 1
data pointer: 0x7ff737ea3220
type: uint8

class: ndarray
shape: (3, 3)
strides: (3, 1)
itemsize: 1
data pointer: 0x7ff737ea3340
type: uint8

a:  array([[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]], dtype=uint8)
====================
b:  array([[1, 1, 1],
    [0, 0, 0],
    [0, 0, 0]], dtype=uint8)
```

The `.copy` method can also be applied to views: below, `a[0]` is a *view* of `a`, out of which we create a *deep copy* called `b`. This is a row vector now. We can then do whatever we want to with `b`, and that leaves `a` unchanged.

```
# code to be run in micropython

import ulab as np

a = np.zeros((3, 3), dtype=np.uint8)
b = a[0].copy()
print('b: ', b)
print('='*20)
# assign 1 to the first entry of b, and do not touch a
b[0] = 1
print('a: ', a)
print('='*20)
print('b: ', b)
```

```
b:  array([0, 0, 0], dtype=uint8)
====================
a:  array([[0, 0, 0],
    [0, 0, 0],
    [0, 0, 0]], dtype=uint8)
====================
b:  array([1, 0, 0], dtype=uint8)
```

The fact that the underlying data of a view is the same as that of the original array has another important consequence, namely, that the creation of a view is cheap. Both in terms of RAM, and execution time. A view is really nothing but a short header with a data array that already exists, and is filled up. Hence, creating the view requires only the creation of its header. This operation is fast, and uses virtually no RAM.

```
# code to be run in CPython
```

CHAPTER 5

Interpolation, root finding, and function minimisation

The `approx` sub-module defines functions for interpolating numerical data, and finding the roots and the minimum of arbitrary functions defined in `python`. Note that routines that work with user-defined functions still have to call the underlying `python` code, and therefore, gains in speed are not as significant as with other vectorised operations. As a rule of thumb, a factor of two can be expected, when compared to an optimised python implementation.

5.1 interp

`numpy`: https://docs.scipy.org/doc/numpy/numpy.interp

The `interp` function returns the linearly interpolated values of a one-dimensional numerical array. It requires three positional arguments,`x`, at which the interpolated values are evaluated, `xp`, the array of the independent variables of the data, and `fp`, the array of the dependent values of the data. `xp` must be a monotonically increasing sequence of numbers.

Two keyword arguments, `left`, and `right` can also be supplied; these determine the return values, if `x < xp[0]`, and `x > xp[-1]`, respectively. If these arguments are not supplied, `left`, and `right` default to `fp[0]`, and `fp[-1]`, respectively.

```
# code to be run in micropython

import ulab
from ulab import approx

x = ulab.array([1, 2, 3, 4, 5])
xp = ulab.array([1, 2, 3, 4])
fp = ulab.array([1, 2, 3, 5])
x = x - 0.2
print(x)
print(approx.interp(x, xp, fp))
print(approx.interp(x, xp, fp, left=0.0))
print(approx.interp(x, xp, fp, right=10.0))
```

```
array([0.8, 1.8, 2.8, 3.8, 4.8], dtype=float)
array([1.0, 1.8, 2.8, 4.6, 5.0], dtype=float)
array([0.0, 1.8, 2.8, 4.6, 5.0], dtype=float)
array([1.0, 1.8, 2.8, 4.6, 10.0], dtype=float)
```

5.2 newton

`scipy`:https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.newton.html

`newton` finds a zero of a real, user-defined function using the Newton-Raphson (or secant or Halley's) method. The routine requires two positional arguments, the function, and the initial value. Three keyword arguments can be supplied to control the iteration. These are the absolute and relative tolerances `tol`, and `rtol`, respectively, and the number of iterations before stopping, `maxiter`. The function retuns a single scalar, the position of the root.

```
# code to be run in micropython

import ulab
from ulab import approx

def f(x):
    return x*x*x - 2.0

print(approx.newton(f, 3., tol=0.001, rtol=0.01))
```

```
1.260135727246117
```

5.3 bisect

`scipy`: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.bisect.html

`bisect` finds the root of a function of one variable using a simple bisection routine. It takes three positional arguments, the function itself, and two starting points. The function must have opposite signs at the starting points. Returned is the position of the root.

Two keyword arguments, `xtol`, and `maxiter` can be supplied to control the accuracy, and the number of bisections, respectively.

```
# code to be run in micropython

import ulab
from ulab import approx

def f(x):
    return x*x - 1

print(approx.bisect(f, 0, 4))

print('only 8 bisections: ',  approx.bisect(f, 0, 4, maxiter=8))

print('with 0.1 accuracy: ',  approx.bisect(f, 0, 4, xtol=0.1))
```

```
0.9999997615814209
only 8 bisections:  0.984375
with 0.1 accuracy:  0.9375
```

5.3.1 Performance

Since the `bisect` routine calls user-defined `python` functions, the speed gain is only about a factor of two, if compared to a purely `python` implementation.

```python
# code to be run in micropython

import ulab
from ulab import approx

def f(x):
    return (x-1)*(x-1) - 2.0

def bisect(f, a, b, xtol=2.4e-7, maxiter=100):
    if f(a) * f(b) > 0:
        raise ValueError

    rtb = a if f(a) < 0.0 else b
    dx = b - a if f(a) < 0.0 else a - b
    for i in range(maxiter):
        dx *= 0.5
        x_mid = rtb + dx
        mid_value = f(x_mid)
        if mid_value < 0:
            rtb = x_mid
        if abs(dx) < xtol:
            break

    return rtb

@timeit
def bisect_approx(f, a, b):
    return approx.bisect(f, a, b)

@timeit
def bisect_timed(f, a, b):
    return bisect(f, a, b)

print('bisect running in python')
bisect_timed(f, 3, 2)

print('bisect running in C')
bisect_approx(f, 3, 2)
```

```
bisect running in python
execution time:  1270  us
bisect running in C
execution time:  642  us
```

5.4 fmin

scipy: https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.fmin.html

The fmin function finds the position of the minimum of a user-defined function by using the downhill simplex method. Requires two positional arguments, the function, and the initial value. Three keyword arguments, xatol, fatol, and maxiter stipulate conditions for stopping.

```
# code to be run in micropython
import ulab
from ulab import approx

def f(x):
    return (x-1)**2 - 1

print(approx.fmin(f, 3.0))
print(approx.fmin(f, 3.0, xatol=0.1))
```

```
0.9996093749999952
1.199999999999996
```

5.5 trapz

numpy: https://numpy.org/doc/stable/reference/generated/numpy.trapz.html

The function takes one or two one-dimensional ndarrays, and integrates the dependent values (y) using the trapezoidal rule. If the independent variable (x) is given, that is taken as the sample points corresponding to y.

```
# code to be run in micropython

import ulab
from ulab import approx

x = ulab.linspace(0, 9, num=10)
y = x*x

print('x: ',  x)
print('y: ',  y)
print('============================')
print('integral of y: ', approx.trapz(y))
print('integral of y at x: ', approx.trapz(y, x=x))
```

```
x:  array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0], dtype=float)
y:  array([0.0, 1.0, 4.0, 9.0, 16.0, 25.0, 36.0, 49.0, 64.0, 81.0], dtype=float)
============================
integral of y:  244.5
integral of y at x:  244.5
```

CHAPTER 6

Comparison of arrays

Functions in the `compare` module can be called by importing the sub-module first.

6.1 equal, not_equal

`numpy`: https://numpy.org/doc/stable/reference/generated/numpy.equal.html

`numpy`: https://numpy.org/doc/stable/reference/generated/numpy.not_equal.html

In `micropython`, equality of arrays or scalars can be established by utilising the ==, !=, <, >, <=, or => binary operators. In `circuitpython`, == and != will produce unexpected results. In order to avoid this discrepancy, and to maintain compatibility with `numpy`, `ulab` implements the `equal` and `not_equal` operators that return the same results, irrespective of the `python` implementation.

These two functions take two `ndarrays`, or scalars as their arguments. No keyword arguments are implemented.

```
# code to be run in micropython

import ulab as np

a = np.array(range(9))
b = np.zeros(9)

print('a: ', a)
print('b: ', b)
print('\na == b: ', np.compare.equal(a, b))
print('a != b: ', np.compare.not_equal(a, b))

# comparison with scalars
print('a == 2: ', np.compare.equal(a, 2))
```

```
a:  array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0], dtype=float)
b:  array([0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], dtype=float)
```

(continues on next page)

(continued from previous page)

```
a == b:  [True, False, False, False, False, False, False, False, False]
a != b:  [False, True, True, True, True, True, True, True, True]
a == 2:  [False, False, True, False, False, False, False, False, False]
```

6.2 minimum

numpy: https://docs.scipy.org/doc/numpy/reference/generated/numpy.minimum.html

Returns the minimum of two arrays, or two scalars, or an array, and a scalar. Partial broadcasting is implemented. If the arrays are of different dtype, the output is upcast as in *Binary operators*. If both inputs are scalars, a scalar is returned. Only positional arguments are implemented.

6.3 maximum

numpy: https://docs.scipy.org/doc/numpy/reference/generated/numpy.maximum.html

Returns the maximum of two arrays, or two scalars, or an array, and a scalar. Partial broadcasting is implemented. If the arrays are of different dtype, the output is upcast as in *Binary operators*. If both inputs are scalars, a scalar is returned. Only positional arguments are implemented.

```
# code to be run in micropython

import ulab

a = ulab.array([1, 2, 3, 4, 5], dtype=ulab.uint8)
b = ulab.array([5, 4, 3, 2, 1], dtype=ulab.float)
print('minimum of a, and b:')
print(ulab.compare.minimum(a, b))

print('\nmaximum of a, and b:')
print(ulab.compare.maximum(a, b))

print('\nmaximum of 1, and 5.5:')
print(ulab.compare.maximum(1, 5.5))
```

```
minimum of a, and b:
array([1.0, 2.0, 3.0, 2.0, 1.0], dtype=float)

maximum of a, and b:
array([5.0, 4.0, 3.0, 4.0, 5.0], dtype=float)

maximum of 1, and 5.5:
5.5
```

6.4 clip

numpy: https://docs.scipy.org/doc/numpy/reference/generated/numpy.clip.html

Clips an array, i.e., values that are outside of an interval are clipped to the interval edges. The function is equivalent to `maximum(a_min, minimum(a, a_max))`. or two scalars, hence partial broadcasting takes place exactly as in *minimum*. If the arrays are of different `dtype`, the output is upcast as in *Binary operators*.

```
# code to be run in micropython

import ulab

a = ulab.array(range(9), dtype=ulab.uint8)
print('a:\t\t', a)
print('clipped:\t', ulab.compare.clip(a, 3, 7))

b = 3 * ulab.ones(len(a), dtype=ulab.float)
print('\na:\t\t', a)
print('b:\t\t', b)
print('clipped:\t', ulab.compare.clip(a, b, 7))
```

```
a:              array([0, 1, 2, 3, 4, 5, 6, 7, 8], dtype=uint8)
clipped:        array([3, 3, 3, 3, 4, 5, 6, 7, 7], dtype=uint8)

a:              array([0, 1, 2, 3, 4, 5, 6, 7, 8], dtype=uint8)
b:              array([3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0, 3.0], dtype=float)
clipped:        array([3.0, 3.0, 3.0, 3.0, 4.0, 5.0, 6.0, 7.0, 7.0], dtype=float)
```

CHAPTER 7

Fourier transforms

Functions related to Fourier transforms can be called by importing the `fft` sub-module first.

`numpy`: https://docs.scipy.org/doc/numpy/reference/generated/numpy.fft.ifft.html

7.1 fft

Since `ulab`'s `ndarray` does not support complex numbers, the invocation of the Fourier transform differs from that in `numpy`. In `numpy`, you can simply pass an array or iterable to the function, and it will be treated as a complex array:

```
# code to be run in CPython

fft.fft([1, 2, 3, 4, 1, 2, 3, 4])
```

```
array([20.+0.j,  0.+0.j, -4.+4.j,  0.+0.j, -4.+0.j,  0.+0.j, -4.-4.j,
        0.+0.j])
```

WARNING: The array that is returned is also complex, i.e., the real and imaginary components are cast together. In `ulab`, the real and imaginary parts are treated separately: you have to pass two `ndarrays` to the function, although, the second argument is optional, in which case the imaginary part is assumed to be zero.

WARNING: The function, as opposed to `numpy`, returns a 2-tuple, whose elements are two `ndarrays`, holding the real and imaginary parts of the transform separately.

```
# code to be run in micropython

import ulab as np
from ulab import vector
from ulab import fft

x = np.linspace(0, 10, num=1024)
y = vector.sin(x)
```

(continues on next page)

(continued from previous page)

```
z = np.zeros(len(x))

a, b = fft.fft(x)
print('real part:\t', a)
print('\nimaginary part:\t', b)

c, d = fft.fft(x, z)
print('\nreal part:\t', c)
print('\nimaginary part:\t', d)
```

```
real part:   array([5119.996, -5.004663, -5.004798, ..., -5.005482, -5.005643, -5.
↪006577], dtype=float)

imaginary part:      array([0.0, 1631.333, 815.659, ..., -543.764, -815.6588, -1631.
↪333], dtype=float)

real part:   array([5119.996, -5.004663, -5.004798, ..., -5.005482, -5.005643, -5.
↪006577], dtype=float)

imaginary part:      array([0.0, 1631.333, 815.659, ..., -543.764, -815.6588, -1631.
↪333], dtype=float)
```

7.2 ifft

The above-mentioned rules apply to the inverse Fourier transform. The inverse is also normalised by `N`, the number of elements, as is customary in `numpy`. With the normalisation, we can ascertain that the inverse of the transform is equal to the original array.

```
# code to be run in micropython

import ulab as np
from ulab import vector
from ulab import fft

x = np.linspace(0, 10, num=1024)
y = vector.sin(x)

a, b = fft.fft(y)

print('original vector:\t', y)

y, z = fft.ifft(a, b)
# the real part should be equal to y
print('\nreal part of inverse:\t', y)
# the imaginary part should be equal to zero
print('\nimaginary part of inverse:\t', z)
```

```
original vector:     array([0.0, 0.009775016, 0.0195491, ..., -0.5275068, -0.5357859,␣
↪-0.5440139], dtype=float)

real part of inverse:        array([-2.980232e-08, 0.0097754, 0.0195494, ..., -0.
↪5275064, -0.5357857, -0.5440133], dtype=float)
```

(continues on next page)

(continued from previous page)

```
imaginary part of inverse:   array([-2.980232e-08, -1.451171e-07, 3.693752e-08, ...,␣
↪6.44871e-08, 9.34986e-08, 2.18336e-07], dtype=float)
```

Note that unlike in numpy, the length of the array on which the Fourier transform is carried out must be a power of 2. If this is not the case, the function raises a ValueError exception.

7.3 spectrogram

In addition to the Fourier transform and its inverse, ulab also sports a function called spectrogram, which returns the absolute value of the Fourier transform. This could be used to find the dominant spectral component in a time series. The arguments are treated in the same way as in fft, and ifft.

```
# code to be run in micropython

import ulab as np
from ulab import vector
from ulab import fft

x = np.linspace(0, 10, num=1024)
y = vector.sin(x)

a = fft.spectrogram(y)

print('original vector:\t', y)
print('\nspectrum:\t', a)
```

```
original vector:     array([0.0, 0.009775015390171337, 0.01954909674625918, ..., -0.
↪5275140569487312, -0.5357931822978732, -0.5440211108893639], dtype=float)

spectrum:    array([187.8635087634579, 315.3112063607119, 347.8814873399374, ..., 84.
↪45888934298905, 347.8814873399374, 315.3112063607118], dtype=float)
```

As such, spectrogram is really just a shorthand for np.sqrt(a*a + b*b):

```
# code to be run in micropython

import ulab as np
from ulab import fft
from ulab import vector

x = np.linspace(0, 10, num=1024)
y = vector.sin(x)

a, b = fft.fft(y)

print('\nspectrum calculated the hard way:\t', vector.sqrt(a*a + b*b))

a = fft.spectrogram(y)

print('\nspectrum calculated the lazy way:\t', a)
```

```
spectrum calculated the hard way:    array([187.8641, 315.3125, 347.8804, ..., 84.
↪4587, 347.8803, 315.3124], dtype=float)
```

(continues on next page)

(continued from previous page)

```
spectrum calculated the lazy way:    array([187.8641, 315.3125, 347.8804, ..., 84.
↪4587, 347.8803, 315.3124], dtype=float)
```

7.4 Computation and storage costs

7.4.1 RAM

The FFT routine of `ulab` calculates the transform in place. This means that beyond reserving space for the two `ndarrays` that will be returned (the computation uses these two as intermediate storage space), only a handful of temporary variables, all floats or 32-bit integers, are required.

7.4.2 Speed of FFTs

A comment on the speed: a 1024-point transform implemented in python would cost around 90 ms, and 13 ms in assembly, if the code runs on the pyboard, v.1.1. You can gain a factor of four by moving to the D series https://github.com/peterhinch/micropython-fourier/blob/master/README.md#8-performance.

```
# code to be run in micropython

import ulab as np
from ulab import vector
from ulab import fft

x = np.linspace(0, 10, num=1024)
y = vector.sin(x)

@timeit
def np_fft(y):
    return fft.fft(y)

a, b = np_fft(y)
```

```
execution time:  1985  us
```

The C implementation runs in less than 2 ms on the pyboard (we have just measured that), and has been reported to run in under 0.8 ms on the D series board. That is an improvement of at least a factor of four.

CHAPTER 8

Filter routines

Functions in the `filter` module can be called by importing the sub-module first.

8.1 convolve

`numpy`: https://docs.scipy.org/doc/numpy/reference/generated/numpy.convolve.html

Returns the discrete, linear convolution of two one-dimensional sequences.

Only the `full` mode is supported, and the `mode` named parameter is not accepted. Note that all other modes can be had by slicing a `full` result.

```
# code to be run in micropython

import ulab as np
from ulab import filter

x = np.array((1,2,3))
y = np.array((1,10,100,1000))

print(filter.convolve(x, y))
```

```
array([1.0, 12.0, 123.0, 1230.0, 2300.0, 3000.0], dtype=float)
```

8.2 sosfilt

`scipy`: https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.sosfilt.html

Filter data along one dimension using cascaded second-order sections.

The function takes two positional arguments, `sos`, the filter segments of length 6, and the one-dimensional, uniformly sample data set to be filtered. Returns the filtered data, or the filtered data and the final filter delays, if the `zi` keyword arguments is supplied. The keyword argument be a float `ndarray` of shape `(n_sections, 2)`. If `zi` is not passed to the function, the initial values are assumed to be 0.

```
# code to be run in micropython

import ulab
from ulab import filter as filter

x = ulab.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
sos = [[1, 2, 3, 1, 5, 6], [1, 2, 3, 1, 5, 6]]
y = filter.sosfilt(sos, x)
print('y: ', y)
```

```
y:  array([0.0, 1.0, -4.0, 24.0, -104.0, 440.0, -1728.0, 6532.000000000001, -23848.0,␣
↪84864.0], dtype=float)
```

```
# code to be run in micropython

import ulab
from ulab import filter as filter

x = ulab.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
sos = [[1, 2, 3, 1, 5, 6], [1, 2, 3, 1, 5, 6]]
# initial conditions of the filter
zi = ulab.array([[1, 2], [3, 4]])

y, zf = filter.sosfilt(sos, x, zi=zi)
print('y: ', y)
print('\n' + '='*40 + '\nzf: ', zf)
```

```
y:  array([4.0, -16.0, 63.00000000000001, -227.0, 802.9999999999999, -2751.0, 9271.
↪000000000001, -30775.0, 101067.0, -328991.0000000001], dtype=float)

========================================
zf:  array([[37242.0, 74835.0],
     [1026187.0, 1936542.0]], dtype=float)
```

CHAPTER 9

Linalg

Functions in the `linalg` module can be called by importing the sub-module first.

9.1 inv

`numpy`: https://docs.scipy.org/doc/numpy-1.17.0/reference/generated/numpy.linalg.inv.html

A square matrix, provided that it is not singular, can be inverted by calling the `inv` function that takes a single argument. The inversion is based on successive elimination of elements in the lower left triangle, and raises a `ValueError` exception, if the matrix turns out to be singular (i.e., one of the diagonal entries is zero).

```
# code to be run in micropython

import ulab as np
from ulab import linalg

m = np.array([[1, 2, 3, 4], [4, 5, 6, 4], [7, 8.6, 9, 4], [3, 4, 5, 6]])

print(linalg.inv(m))
```

```
array([[-2.166666, 1.499999, -0.8333326, 1.0],
      [1.666666, -3.333331, 1.666666, -4.768516e-08],
      [0.1666672, 2.166666, -0.8333327, -1.0],
      [-0.1666666, -0.3333334, 4.96705e-08, 0.5]], dtype=float)
```

9.1.1 Computation expenses

Note that the cost of inverting a matrix is approximately twice as many floats (RAM), as the number of entries in the original matrix, and approximately as many operations, as the number of entries. Here are a couple of numbers:

```
# code to be run in micropython

import ulab as np
from ulab import linalg

@timeit
def invert_matrix(m):
    return linalg.inv(m)

m = np.array([[1, 2,], [4, 5]])
print('2 by 2 matrix:')
invert_matrix(m)

m = np.array([[1, 2, 3, 4], [4, 5, 6, 4], [7, 8.6, 9, 4], [3, 4, 5, 6]])
print('\n4 by 4 matrix:')
invert_matrix(m)

m = np.array([[1, 2, 3, 4, 5, 6, 7, 8], [0, 5, 6, 4, 5, 6, 4, 5],
              [0, 0, 9, 7, 8, 9, 7, 8], [0, 0, 0, 10, 11, 12, 11, 12],
             [0, 0, 0, 0, 4, 6, 7, 8], [0, 0, 0, 0, 0, 5, 6, 7],
             [0, 0, 0, 0, 0, 0, 7, 6], [0, 0, 0, 0, 0, 0, 0, 2]])
print('\n8 by 8 matrix:')
invert_matrix(m)
```

```
2 by 2 matrix:
execution time:  65  us

4 by 4 matrix:
execution time:  105  us

8 by 8 matrix:
execution time:  299  us
```

The above-mentioned scaling is not obeyed strictly. The reason for the discrepancy is that the function call is still the same for all three cases: the input must be inspected, the output array must be created, and so on.

9.2 dot

`numpy`: https://docs.scipy.org/doc/numpy/reference/generated/numpy.dot.html

WARNING: numpy applies upcasting rules for the multiplication of matrices, while `ulab` simply returns a float matrix.

Once you can invert a matrix, you might want to know, whether the inversion is correct. You can simply take the original matrix and its inverse, and multiply them by calling the `dot` function, which takes the two matrices as its arguments. If the matrix dimensions do not match, the function raises a `ValueError`. The result of the multiplication is expected to be the unit matrix, which is demonstrated below.

```
# code to be run in micropython

import ulab as np
from ulab import linalg

m = np.array([[1, 2, 3], [4, 5, 6], [7, 10, 9]], dtype=np.uint8)
n = linalg.inv(m)
```

(continues on next page)

(continued from previous page)

```
print("m:\n", m)
print("\nm^-1:\n", n)
# this should be the unit matrix
print("\nm*m^-1:\n", linalg.dot(m, n))
```

```
m:
 array([[1, 2, 3],
     [4, 5, 6],
     [7, 10, 9]], dtype=uint8)

m^-1:
 array([[-1.25, 1.0, -0.25],
     [0.5, -1.0, 0.5],
     [0.4166667, 0.3333334, -0.25]], dtype=float)

m*m^-1:
 array([[1.0, 2.384186e-07, -1.490116e-07],
     [-2.980232e-07, 1.000001, -4.172325e-07],
     [-3.278255e-07, 1.311302e-06, 0.9999992]], dtype=float)
```

Note that for matrix multiplication you don't necessarily need square matrices, it is enough, if their dimensions are compatible (i.e., the the left-hand-side matrix has as many columns, as does the right-hand-side matrix rows):

```
# code to be run in micropython

import ulab as np
from ulab import linalg

m = np.array([[1, 2, 3, 4], [5, 6, 7, 8]], dtype=np.uint8)
n = np.array([[1, 2], [3, 4], [5, 6], [7, 8]], dtype=np.uint8)
print(m)
print(n)
print(linalg.dot(m, n))
```

```
array([[1, 2, 3, 4],
     [5, 6, 7, 8]], dtype=uint8)
array([[1, 2],
     [3, 4],
     [5, 6],
     [7, 8]], dtype=uint8)
array([[7.0, 10.0],
     [23.0, 34.0]], dtype=float)
```

9.3 det

numpy: https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.det.html

The det function takes a square matrix as its single argument, and calculates the determinant. The calculation is based on successive elimination of the matrix elements, and the return value is a float, even if the input array was of integer type.

```
# code to be run in micropython

```

(continues on next page)

(continued from previous page)

```
import ulab as np
from ulab import linalg

a = np.array([[1, 2], [3, 4]], dtype=np.uint8)
print(linalg.det(a))
```

```
-2.0
```

9.3.1 Benchmark

Since the routine for calculating the determinant is pretty much the same as for finding the *inverse of a matrix*, the execution times are similar:

```
# code to be run in micropython

@timeit
def matrix_det(m):
    return linalg.inv(m)

m = np.array([[1, 2, 3, 4, 5, 6, 7, 8], [0, 5, 6, 4, 5, 6, 4, 5],
              [0, 0, 9, 7, 8, 9, 7, 8], [0, 0, 0, 10, 11, 12, 11, 12],
             [0, 0, 0, 0, 4, 6, 7, 8], [0, 0, 0, 0, 0, 5, 6, 7],
             [0, 0, 0, 0, 0, 0, 7, 6], [0, 0, 0, 0, 0, 0, 0, 2]])

matrix_det(m)
```

```
execution time:  294  us
```

9.4 eig

`numpy`: https://docs.scipy.org/doc/numpy/reference/generated/numpy.linalg.eig.html

The `eig` function calculates the eigenvalues and the eigenvectors of a real, symmetric square matrix. If the matrix is not symmetric, a `ValueError` will be raised. The function takes a single argument, and returns a tuple with the eigenvalues, and eigenvectors. With the help of the eigenvectors, amongst other things, you can implement sophisticated stabilisation routines for robots.

```
# code to be run in micropython

import ulab as np
from ulab import linalg

a = np.array([[1, 2, 1, 4], [2, 5, 3, 5], [1, 3, 6, 1], [4, 5, 1, 7]], dtype=np.uint8)
x, y = linalg.eig(a)
print('eigenvectors of a:\n', y)
print('\neigenvalues of a:\n', x)
```

```
eigenvectors of a:
 array([[0.8151560042509081, -0.4499411232970823, -0.1644660242574522, 0.
↪3256141906686505],
       [0.2211334179893007, 0.7846992598235538, 0.08372081379922657, 0.
↪5730077734355189],
```

(continues on next page)

(continued from previous page)

```
       [-0.1340114162071679, -0.3100776411558949, 0.8742786816656, 0.
↪3486109343758527],
       [-0.5183258053659028, -0.292663481927148, -0.4489749870391468, 0.
↪6664142156731531]], dtype=float)

eigenvalues of a:
 array([-1.165288365404889, 0.8029365530314914, 5.585625756072663, 13.77672605630074],
↪ dtype=float)
```

The same matrix diagonalised with `numpy` yields:

```
# code to be run in CPython

a = array([[1, 2, 1, 4], [2, 5, 3, 5], [1, 3, 6, 1], [4, 5, 1, 7]], dtype=np.uint8)
x, y = eig(a)
print('eigenvectors of a:\n', y)
print('\neigenvalues of a:\n', x)
```

```
eigenvectors of a:
 [[ 0.32561419  0.815156    0.44994112 -0.16446602]
 [ 0.57300777  0.22113342 -0.78469926  0.08372081]
 [ 0.34861093 -0.13401142  0.31007764  0.87427868]
 [ 0.66641421 -0.51832581  0.29266348 -0.44897499]]

eigenvalues of a:
 [13.77672606 -1.16528837  0.80293655  5.58562576]
```

When comparing results, we should keep two things in mind:

1. the eigenvalues and eigenvectors are not necessarily sorted in the same way
2. an eigenvector can be multiplied by an arbitrary non-zero scalar, and it is still an eigenvector with the same eigenvalue. This is why all signs of the eigenvector belonging to 5.58, and 0.80 are flipped in `ulab` with respect to `numpy`. This difference, however, is of absolutely no consequence.

9.4.1 Computation expenses

Since the function is based on Givens rotations and runs till convergence is achieved, or till the maximum number of allowed rotations is exhausted, there is no universal estimate for the time required to find the eigenvalues. However, an order of magnitude can, at least, be guessed based on the measurement below:

```
# code to be run in micropython

import ulab as np
from ulab import linalg

@timeit
def matrix_eig(a):
    return linalg.eig(a)

a = np.array([[1, 2, 1, 4], [2, 5, 3, 5], [1, 3, 6, 1], [4, 5, 1, 7]], dtype=np.uint8)

matrix_eig(a)
```

```
execution time:  111  us
```

9.5 Cholesky decomposition

`numpy`: https://docs.scipy.org/doc/numpy-1.17.0/reference/generated/numpy.linalg.cholesky.html

`cholesky` takes a positive definite, symmetric square matrix as its single argument, and returns *square root matrix* in the lower triangular form. If the input argument does not fulfill the positivity or symmetry condition, a `ValueError` is raised.

```
# code to be run in micropython

import ulab
from ulab import linalg

a = ulab.array([[25, 15, -5], [15, 18,  0], [-5,  0, 11]])
print('a: ', a)
print('\n' + '='*20 + '\nCholesky decomposition\n', linalg.cholesky(a))
```

```
a:  array([[25.0, 15.0, -5.0],
     [15.0, 18.0, 0.0],
     [-5.0, 0.0, 11.0]], dtype=float)

====================
Cholesky decomposition
 array([[5.0, 0.0, 0.0],
     [3.0, 3.0, 0.0],
     [-1.0, 1.0, 3.0]], dtype=float)
```

9.6 norm

`numpy`: https://numpy.org/doc/stable/reference/generated/numpy.linalg.norm.html

The function takes a vector or matrix without options, and returns its 2-norm, i.e., the square root of the sum of the square of the elements.

```
# code to be run in micropython

import ulab
from ulab import linalg

a = ulab.array([1, 2, 3, 4, 5])
b = ulab.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print('norm of a:', linalg.norm(a))
print('norm of b:', linalg.norm(b))
```

```
norm of a: 7.416198487095663
norm of b: 16.88194301613414
```

9.7 trace

`numpy`: https://docs.scipy.org/doc/numpy-1.17.0/reference/generated/numpy.linalg.trace.html

The `trace` function returns the sum of the diagonal elements of a square matrix. If the input argument is not a square matrix, an exception will be raised.

The scalar so returned will inherit the type of the input array, i.e., integer arrays have integer trace, and floating point arrays a floating point trace.

```
# code to be run in micropython

import ulab
from ulab import linalg

a = ulab.array([[25, 15, -5], [15, 18,  0], [-5,  0, 11]], dtype=ulab.int8)
print('a: ', a)
print('\ntrace of a: ', linalg.trace(a))

b = ulab.array([[25, 15, -5], [15, 18,  0], [-5,  0, 11]], dtype=ulab.float)

print('='*20 + '\nb: ', b)
print('\ntrace of b: ', linalg.trace(b))
```

```
a:  array([[25, 15, -5],
     [15, 18, 0],
     [-5, 0, 11]], dtype=int8)

trace of a:  54
====================
b:  array([[25.0, 15.0, -5.0],
     [15.0, 18.0, 0.0],
     [-5.0, 0.0, 11.0]], dtype=float)

trace of b:  54.0
```

CHAPTER 10

Numerical

Function in the `numerical` sub-module can be called by importing the sub-module first.

10.1 min, argmin, max, argmax

`numpy`: https://docs.scipy.org/doc/numpy/reference/generated/numpy.min.html

`numpy`: https://docs.scipy.org/doc/numpy/reference/generated/numpy.argmax.html

`numpy`: https://docs.scipy.org/doc/numpy/reference/generated/numpy.max.html

`numpy`: https://docs.scipy.org/doc/numpy/reference/generated/numpy.argmax.html

WARNING: Difference to `numpy`: the `out` keyword argument is not implemented.

These functions follow the same pattern, and work with generic iterables, and `ndarrays`. `min`, and `max` return the minimum or maximum of a sequence. If the input array is two-dimensional, the `axis` keyword argument can be supplied, in which case the minimum/maximum along the given axis will be returned. If `axis=None` (this is also the default value), the minimum/maximum of the flattened array will be determined.

`argmin/argmax` return the position (index) of the minimum/maximum in the sequence.

```
# code to be run in micropython

import ulab as np

a = np.array([1, 2, 3])
print(a)
print(a[-1:-1:-3])
try:
    sa = list(a[-1:-1:-3])
    la = len(sa)
except IndexError as e:
    sa = str(e)
    la = -1
```

(continues on next page)

(continued from previous page)

```
print(sa, la)

a[-1:-1:-3] = np.ones(0)
print(a)

b = np.ones(0) + 1
print(b)
# print('b', b.shape())
```

```
array([1.0, 2.0, 3.0], dtype=float)
array([], dtype=float)
[] 0
array([1.0, 2.0, 3.0], dtype=float)
array([], dtype=float)
```

```
# code to be run in micropython

import ulab as np
a = np.array([1, 2, 3])
print(a[0:1:-3])
```

```
0, 1, -3array([], dtype=float)
```

```
# code to be run in CPython

l = list(range(13))

l[0:10:113]
```

```
[0]
```

```
# code to be run in CPython

a = np.array([1, 2, 3])
np.ones(0, dtype=uint8) / np.zeros(0, dtype=uint16)
np.ones(0).shape
```

```
(0,)
```

```
# code to be run in micropython

import ulab as np
from ulab import numerical

a = np.array([1, 2, 0, 1, 10])
print('a:', a)
print('min of a:', numerical.min(a))
print('argmin of a:', numerical.argmin(a))

b = np.array([[1, 2, 0], [1, 10, -1]])
print('\nb:\n', b)
print('min of b (flattened):', numerical.min(b))
```

(continues on next page)

(continued from previous page)

```
print('min of b (axis=0):', numerical.min(b, axis=0))
print('min of b (axis=1):', numerical.min(b, axis=1))
```

```
a: array([1.0, 2.0, 0.0, 1.0, 10.0], dtype=float)
min of a: 0.0
argmin of a: 2

b:
 array([[1.0, 2.0, 0.0],
     [1.0, 10.0, -1.0]], dtype=float)
min of b (flattened): -1.0
min of b (axis=0): array([1.0, 2.0, -1.0], dtype=float)
min of b (axis=1): array([0.0, -1.0], dtype=float)
```

10.2 sum, std, mean

numpy: https://docs.scipy.org/doc/numpy/reference/generated/numpy.sum.html

numpy: https://docs.scipy.org/doc/numpy/reference/generated/numpy.std.html

numpy: https://docs.scipy.org/doc/numpy/reference/generated/numpy.mean.html

These three functions follow the same pattern: if the axis keyword is not specified, it assumes the default value of `None`, and returns the result of the computation for the flattened array. Otherwise, the calculation is along the given axis.

```
# code to be run in micropython

import ulab as np
from ulab import numerical

a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print('a: \n', a)

print('sum, flat array: ', numerical.sum(a))

print('mean, horizontal: ', numerical.mean(a, axis=1))

print('std, vertical: ', numerical.std(a, axis=0))
```

```
a:
 array([[1.0, 2.0, 3.0],
     [4.0, 5.0, 6.0],
     [7.0, 8.0, 9.0]], dtype=float)
sum, flat array:  45.0
mean, horizontal:  array([2.0, 5.0, 8.0], dtype=float)
std, vertical:  array([2.44949, 2.44949, 2.44949], dtype=float)
```

10.3 roll

numpy: https://docs.scipy.org/doc/numpy/reference/generated/numpy.roll.html

The roll function shifts the content of a vector by the positions given as the second argument. If the `axis` keyword is supplied, the shift is applied to the given axis.

```
# code to be run in micropython

import ulab as np
from ulab import numerical

a = np.array([1, 2, 3, 4, 5, 6, 7, 8])
print("a:\t\t\t", a)

numerical.roll(a, 2)
print("a rolled to the left:\t", a)

# this should be the original vector
numerical.roll(a, -2)
print("a rolled to the right:\t", a)
```

```
a:                      array([1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0], dtype=float)
a rolled to the left:        array([3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 1.0, 2.0],␣
↪dtype=float)
a rolled to the right:       array([1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0],␣
↪dtype=float)
```

Rolling works with matrices, too. If the `axis` keyword is 0, the matrix is rolled along its vertical axis, otherwise, horizontally.

Horizontal rolls are faster, because they require fewer steps, and larger memory chunks are copied, however, they also require more RAM: basically the whole row must be stored internally. Most expensive are the `None` keyword values, because with `axis = None`, the array is flattened first, hence the row's length is the size of the whole matrix.

Vertical rolls require two internal copies of single columns.

```
# code to be run in micropython

import ulab as np
from ulab import numerical

a = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print("a:\n", a)

numerical.roll(a, 2)
print("\na rolled to the left:\n", a)

numerical.roll(a, -1, axis=1)
print("\na rolled up:\n", a)

numerical.roll(a, 1, axis=None)
print("\na rolled with None:\n", a)
```

```
a:
 array([[1.0, 2.0, 3.0, 4.0],
     [5.0, 6.0, 7.0, 8.0]], dtype=float)

a rolled to the left:
 array([[3.0, 4.0, 5.0, 6.0],
     [7.0, 8.0, 1.0, 2.0]], dtype=float)

```

(continues on next page)

(continued from previous page)

```
a rolled up:
 array([[6.0, 3.0, 4.0, 5.0],
     [2.0, 7.0, 8.0, 1.0]], dtype=float)

a rolled with None:
 array([[3.0, 4.0, 5.0, 2.0],
     [7.0, 8.0, 1.0, 6.0]], dtype=float)
```

10.3.1 Simple running weighted average

As a demonstration of the conciseness of `ulab/numpy` operations, we will calculate an exponentially weighted running average of a measurement vector in just a couple of lines. I chose this particular example, because I think that this can indeed be used in real-life applications.

```
# code to be run in micropython

import ulab as np
from ulab import numerical
from ulab import vector

def dummy_adc():
    # dummy adc function, so that the results are reproducible
    return 2

n = 10
# These are the normalised weights; the last entry is the most dominant
weight = vector.exp([1, 2, 3, 4, 5])
weight = weight/numerical.sum(weight)

print(weight)
# initial array of samples
samples = np.array([0]*n)

for i in range(n):
    # a new datum is inserted on the right hand side. This simply overwrites whatever␣
↪was in the last slot
    samples[-1] = dummy_adc()
    print(numerical.mean(samples[-5:]*weight))
    print(samples[-5:])
    # the data are shifted by one position to the left
    numerical.roll(samples, 1)
```

```
array([0.01165623031556606, 0.03168492019176483, 0.08612854033708572, 0.
↪234121635556221, 0.6364086270332336], dtype=float)
0.2545634508132935
array([0.0, 0.0, 0.0, 0.0, 2.0], dtype=float)
0.3482121050357819
array([0.0, 0.0, 0.0, 2.0, 2.0], dtype=float)
0.3826635211706161
array([0.0, 0.0, 2.0, 2.0, 2.0], dtype=float)
0.3953374892473221
array([0.0, 2.0, 2.0, 2.0, 2.0], dtype=float)
0.3999999813735485
array([2.0, 2.0, 2.0, 2.0, 2.0], dtype=float)
```

(continues on next page)

(continued from previous page)

```
0.3999999813735485
array([2.0, 2.0, 2.0, 2.0, 2.0], dtype=float)
0.3999999813735485
array([2.0, 2.0, 2.0, 2.0, 2.0], dtype=float)
0.3999999813735485
array([2.0, 2.0, 2.0, 2.0, 2.0], dtype=float)
0.3999999813735485
array([2.0, 2.0, 2.0, 2.0, 2.0], dtype=float)
0.3999999813735485
array([2.0, 2.0, 2.0, 2.0, 2.0], dtype=float)
```

10.4 flip

`numpy`: https://docs.scipy.org/doc/numpy/reference/generated/numpy.flip.html

The `flip` function takes one positional, an `ndarray`, and one keyword argument, `axis = None`, and reverses the order of elements along the given axis. If the keyword argument is `None`, the matrix' entries are flipped along all axes. `flip` returns a new copy of the array.

```
# code to be run in micropython

import ulab as np
from ulab import numerical

a = np.array([1, 2, 3, 4, 5])
print("a: \t", a)
print("a flipped:\t", np.flip(a))

a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]], dtype=np.uint8)
print("\na flipped horizontally\n", numerical.flip(a, axis=1))
print("\na flipped vertically\n", numerical.flip(a, axis=0))
print("\na flipped horizontally+vertically\n", numerical.flip(a))
```

```
a:   array([1.0, 2.0, 3.0, 4.0, 5.0], dtype=float)
a flipped:   array([5.0, 4.0, 3.0, 2.0, 1.0], dtype=float)

a flipped horizontally
 array([[3, 2, 1],
     [6, 5, 4],
     [9, 8, 7]], dtype=uint8)

a flipped vertically
 array([[7, 8, 9],
     [4, 5, 6],
     [1, 2, 3]], dtype=uint8)

a flipped horizontally+vertically
 array([[9, 8, 7],
     [6, 5, 4],
     [3, 2, 1]], dtype=uint8)
```

10.5 diff

numpy: https://docs.scipy.org/doc/numpy/reference/generated/numpy.diff.html

The `diff` function returns the numerical derivative of the forward scheme, or more accurately, the differences of an `ndarray` along a given axis. The order of derivative can be stipulated with the `n` keyword argument, which should be between 0, and 9. Default is 1. If higher order derivatives are required, they can be gotten by repeated calls to the function. The `axis` keyword argument should be -1 (last axis, in `ulab` equivalent to the second axis, and this also happens to be the default value), 0, or 1.

Beyond the output array, the function requires only a couple of bytes of extra RAM for the differentiation stencil. (The stencil is an `int8` array, one byte longer than `n`. This also explains, why the highest order is 9: the coefficients of a ninth-order stencil all fit in signed bytes, while 10 would require `int16`.) Note that as usual in numerical differentiation (and also in `numpy`), the length of the respective axis will be reduced by `n` after the operation. If `n` is larger than, or equal to the length of the axis, an empty array will be returned.

WARNING: the `diff` function does not implement the `prepend` and `append` keywords that can be found in `numpy`.

```
# code to be run in micropython

import ulab as np
from ulab import numerical

a = np.array(range(9), dtype=np.uint8)
print('a:\n', a)

print('\nfirst derivative:\n', numerical.diff(a, n=1))
print('\nsecond derivative:\n', numerical.diff(a, n=2))

c = np.array([[1, 2, 3, 4], [4, 3, 2, 1], [1, 4, 9, 16], [0, 0, 0, 0]])
print('\nc:\n', c)
print('\nfirst derivative, first axis:\n', numerical.diff(c, axis=0))
print('\nfirst derivative, second axis:\n', numerical.diff(c, axis=1))
```

```
a:
 array([0, 1, 2, 3, 4, 5, 6, 7, 8], dtype=uint8)

first derivative:
 array([1, 1, 1, 1, 1, 1, 1, 1], dtype=uint8)

second derivative:
 array([0, 0, 0, 0, 0, 0, 0], dtype=uint8)

c:
 array([[1.0, 2.0, 3.0, 4.0],
     [4.0, 3.0, 2.0, 1.0],
     [1.0, 4.0, 9.0, 16.0],
     [0.0, 0.0, 0.0, 0.0]], dtype=float)

first derivative, first axis:
 array([[3.0, 1.0, -1.0, -3.0],
     [-3.0, 1.0, 7.0, 15.0],
     [-1.0, -4.0, -9.0, -16.0]], dtype=float)

first derivative, second axis:
 array([[1.0, 1.0, 1.0],
```

(continues on next page)

(continued from previous page)

```
      [-1.0, -1.0, -1.0],
      [3.0, 5.0, 7.0],
      [0.0, 0.0, 0.0]], dtype=float)
```

10.6 median

`numpy`: https://docs.scipy.org/doc/numpy/reference/generated/numpy.median.html

The function computes the median along the specified axis, and returns the median of the array elements. If the `axis` keyword argument is `None`, the arrays is flattened first. The `dtype` of the results is always float.

```
# code to be run in micropython

import ulab as np

a = np.array(range(12), dtype=np.int8).reshape((3, 4))
print('a:\n', a)
print('\nmedian of the flattened array: ', np.median(a))
print('\nmedian along the vertical axis: ', np.median(a, axis=0))
print('\nmedian along the horizontal axis: ', np.median(a, axis=1))
```

```
a:
 array([[0, 1, 2, 3],
        [4, 5, 6, 7],
        [8, 9, 10, 11]], dtype=int8)

median of the flattened array:  5.5

median along the vertical axis:  array([4.0, 5.0, 6.0, 7.0], dtype=float)

median along the horizontal axis:  array([1.5, 5.5, 9.5], dtype=float)
```

10.7 sort

`numpy`: https://docs.scipy.org/doc/numpy/reference/generated/numpy.sort.html

The sort function takes an ndarray, and sorts its elements in ascending order along the specified axis using a heap sort algorithm. As opposed to the `.sort()` method discussed earlier, this function creates a copy of its input before sorting, and at the end, returns this copy. Sorting takes place in place, without auxiliary storage. The `axis` keyword argument takes on the possible values of -1 (the last axis, in `ulab` equivalent to the second axis, and this also happens to be the default value), 0, 1, or `None`. The first three cases are identical to those in *diff*, while the last one flattens the array before sorting.

If descending order is required, the result can simply be `flip`ped, see *flip*.

WARNING: `numpy` defines the `kind`, and `order` keyword arguments that are not implemented here. The function in `ulab` always uses heap sort, and since `ulab` does not have the concept of data fields, the `order` keyword argument would have no meaning.

```
# code to be run in micropython

import ulab as np
```

(continues on next page)

(continued from previous page)

```
from ulab import numerical

a = np.array([[1, 12, 3, 0], [5, 3, 4, 1], [9, 11, 1, 8], [7, 10, 0, 1]], dtype=np.
↪float)
print('\na:\n', a)
b = numerical.sort(a, axis=0)
print('\na sorted along vertical axis:\n', b)

c = numerical.sort(a, axis=1)
print('\na sorted along horizontal axis:\n', c)

c = numerical.sort(a, axis=None)
print('\nflattened a sorted:\n', c)
```

```
a:
 array([[1.0, 12.0, 3.0, 0.0],
     [5.0, 3.0, 4.0, 1.0],
     [9.0, 11.0, 1.0, 8.0],
     [7.0, 10.0, 0.0, 1.0]], dtype=float)

a sorted along vertical axis:
 array([[1.0, 3.0, 0.0, 0.0],
     [5.0, 10.0, 1.0, 1.0],
     [7.0, 11.0, 3.0, 1.0],
     [9.0, 12.0, 4.0, 8.0]], dtype=float)

a sorted along horizontal axis:
 array([[0.0, 1.0, 3.0, 12.0],
     [1.0, 3.0, 4.0, 5.0],
     [1.0, 8.0, 9.0, 11.0],
     [0.0, 1.0, 7.0, 10.0]], dtype=float)

flattened a sorted:
 array([0.0, 0.0, 1.0, ..., 10.0, 11.0, 12.0], dtype=float)
```

Heap sort requires $\sim N \log N$ operations, and notably, the worst case costs only 20% more time than the average. In order to get an order-of-magnitude estimate, we will take the sine of 1000 uniformly spaced numbers between 0, and two pi, and sort them:

```
# code to be run in micropython

import ulab as np
from ulab import vector
from ulab import numerical

@timeit
def sort_time(array):
    return numerical.sort(array)

b = vector.sin(np.linspace(0, 6.28, num=1000))
print('b: ', b)
sort_time(b)
print('\nb sorted:\n', b)
```

10.8 argsort

numpy: https://docs.scipy.org/doc/numpy/reference/generated/numpy.argsort.html

Similarly to *sort*, argsort takes a positional, and a keyword argument, and returns an unsigned short index array of type ndarray with the same dimensions as the input, or, if axis=None, as a row vector with length equal to the number of elements in the input (i.e., the flattened array). The indices in the output sort the input in ascending order. The routine in argsort is the same as in sort, therefore, the comments on computational expenses (time and RAM) also apply. In particular, since no copy of the original data is required, virtually no RAM beyond the output array is used.

Since the underlying container of the output array is of type uint16_t, neither of the output dimensions should be larger than 65535. If that happens to be the case, the function will bail out with a ValueError.

```
# code to be run in micropython

import ulab as np
from ulab import numerical

a = np.array([[1, 12, 3, 0], [5, 3, 4, 1], [9, 11, 1, 8], [7, 10, 0, 1]], dtype=np.
↪float)
print('\na:\n', a)
b = numerical.argsort(a, axis=0)
print('\na sorted along vertical axis:\n', b)

c = numerical.argsort(a, axis=1)
print('\na sorted along horizontal axis:\n', c)

c = numerical.argsort(a, axis=None)
print('\nflattened a sorted:\n', c)
```

```
a:
 array([[1.0, 12.0, 3.0, 0.0],
     [5.0, 3.0, 4.0, 1.0],
     [9.0, 11.0, 1.0, 8.0],
     [7.0, 10.0, 0.0, 1.0]], dtype=float)

a sorted along vertical axis:
 array([[0, 1, 3, 0],
     [1, 3, 2, 1],
     [3, 2, 0, 3],
     [2, 0, 1, 2]], dtype=uint16)

a sorted along horizontal axis:
 array([[3, 0, 2, 1],
     [3, 1, 2, 0],
     [2, 3, 0, 1],
     [2, 3, 0, 1]], dtype=uint16)

flattened a sorted:
 array([3, 14, 0, ..., 13, 9, 1], dtype=uint16)
```

Since during the sorting, only the indices are shuffled, argsort does not modify the input array, as one can verify this by the following example:

```
# code to be run in micropython
```

(continues on next page)

(continued from previous page)

```
import ulab as np
from ulab import numerical

a = np.array([0, 5, 1, 3, 2, 4], dtype=np.uint8)
print('\na:\n', a)
b = numerical.argsort(a, axis=1)
print('\nsorting indices:\n', b)
print('\nthe original array:\n', a)
```

```
a:
 array([0, 5, 1, 3, 2, 4], dtype=uint8)

sorting indices:
 array([0, 2, 4, 3, 5, 1], dtype=uint16)

the original array:
 array([0, 5, 1, 3, 2, 4], dtype=uint8)
```

CHAPTER 11

Polynomials

Functions in the polynomial sub-module can be invoked by importing the module first.

11.1 polyval

numpy: https://docs.scipy.org/doc/numpy/reference/generated/numpy.polyval.html

polyval takes two arguments, both arrays or other iterables.

```
# code to be run in micropython

import ulab as np
from ulab import poly

p = [1, 1, 1, 0]
x = [0, 1, 2, 3, 4]
print('coefficients: ', p)
print('independent values: ', x)
print('\nvalues of p(x): ', poly.polyval(p, x))

# the same works with one-dimensional ndarrays
a = np.array(x)
print('\nndarray (a): ', a)
print('value of p(a): ', poly.polyval(p, a))
```

```
coefficients:  [1, 1, 1, 0]
independent values:  [0, 1, 2, 3, 4]

values of p(x):  array([0.0, 3.0, 14.0, 39.0, 84.0], dtype=float)

ndarray (a):  array([0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
value of p(a):  array([0.0, 3.0, 14.0, 39.0, 84.0], dtype=float)
```

11.2 polyfit

`numpy`: https://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html

polyfit takes two, or three arguments. The last one is the degree of the polynomial that will be fitted, the last but one is an array or iterable with the `y` (dependent) values, and the first one, an array or iterable with the `x` (independent) values, can be dropped. If that is the case, `x` will be generated in the function, assuming uniform sampling.

If the length of `x`, and `y` are not the same, the function raises a `ValueError`.

```
# code to be run in micropython

import ulab as np
from ulab import poly

x = np.array([0, 1, 2, 3, 4, 5, 6])
y = np.array([9, 4, 1, 0, 1, 4, 9])
print('independent values:\t', x)
print('dependent values:\t', y)
print('fitted values:\t\t', poly.polyfit(x, y, 2))

# the same with missing x
print('\ndependent values:\t', y)
print('fitted values:\t\t', poly.polyfit(y, 2))
```

```
independent values:  array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0], dtype=float)
dependent values:    array([9.0, 4.0, 1.0, 0.0, 1.0, 4.0, 9.0], dtype=float)
fitted values:               array([1.0, -6.0, 9.000000000000004], dtype=float)

dependent values:    array([9.0, 4.0, 1.0, 0.0, 1.0, 4.0, 9.0], dtype=float)
fitted values:               array([1.0, -6.0, 9.000000000000004], dtype=float)
```

11.2.1 Execution time

`polyfit` is based on the inversion of a matrix (there is more on the background in https://en.wikipedia.org/wiki/Polynomial_regression), and it requires the intermediate storage of `2*N*(deg+1)` floats, where `N` is the number of entries in the input array, and `deg` is the fit's degree. The additional computation costs of the matrix inversion discussed in *inv* also apply. The example from above needs around 150 microseconds to return:

```
# code to be run in micropython

import ulab as np
from ulab import poly

@timeit
def time_polyfit(x, y, n):
    return poly.polyfit(x, y, n)

x = np.array([0, 1, 2, 3, 4, 5, 6])
y = np.array([9, 4, 1, 0, 1, 4, 9])

time_polyfit(x, y, 2)
```

```
execution time:  153  us
```

CHAPTER 12

Universal functions

Standard mathematical functions are defined in the `vector` sub-module, and can be calculated on any scalar, scalar-valued iterable (ranges, lists, tuples containing numbers), and on `ndarrays` without having to change the call signature. In all cases the functions return a new `ndarray` of typecode `float` (since these functions usually generate float values, anyway). The functions execute faster with `ndarray` arguments than with iterables, because the values of the input vector can be extracted faster.

At present, the following functions are supported:

`acos`, `acosh`, `arctan2`, `around`, `asin`, `asinh`, `atan`, `arctan2`, `atanh`, `ceil`, `cos`, `degrees`, `erf`, `erfc`, `exp`, `expm1`, `floor`, `tgamma`, `lgamma`, `log`, `log10`, `log2`, `radians`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`.

These functions are applied element-wise to the arguments, thus, e.g., the exponential of a matrix cannot be calculated in this way. The functions can be invoked by importing the `vector` sub-module first.

```python
# code to be run in micropython

import ulab as np
from ulab import vector

a = range(9)
b = np.array(a)

# works with ranges, lists, tuples etc.
print('a:\t', a)
print('exp(a):\t', vector.exp(a))

# with 1D arrays
print('\nb:\t', b)
print('exp(b):\t', vector.exp(b))

# as well as with matrices
c = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print('\nc:\t', c)
print('exp(c):\t', vector.exp(c))
```

```
a:   range(0, 9)
exp(a):      array([1.0, 2.718282, 7.389056, 20.08554, 54.59816, 148.4132, 403.4288,␣
↪1096.633, 2980.958], dtype=float)

b:   array([0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0], dtype=float)
exp(b):      array([1.0, 2.718282, 7.389056, 20.08554, 54.59816, 148.4132, 403.4288,␣
↪1096.633, 2980.958], dtype=float)

c:   array([[1.0, 2.0, 3.0],
     [4.0, 5.0, 6.0],
     [7.0, 8.0, 9.0]], dtype=float)
exp(c):      array([[2.718282, 7.389056, 20.08554],
     [54.59816, 148.4132, 403.4288],
     [1096.633, 2980.958, 8103.084]], dtype=float)
```

12.1 Computation expenses

The overhead for calculating with micropython iterables is quite significant: for the 1000 samples below, the difference is more than 800 microseconds, because internally the function has to create the `ndarray` for the output, has to fetch the iterable's items of unknown type, and then convert them to floats. All these steps are skipped for `ndarrays`, because these pieces of information are already known.

Doing the same with `list` comprehension requires 30 times more time than with the `ndarray`, which would become even more, if we converted the resulting list to an `ndarray`.

```
# code to be run in micropython

import ulab as np
from ulab import vector
import math

a = [0]*1000
b = np.array(a)

@timeit
def timed_vector(iterable):
    return vector.exp(iterable)

@timeit
def timed_list(iterable):
    return [math.exp(i) for i in iterable]

print('iterating over ndarray in ulab')
timed_vector(b)

print('\niterating over list in ulab')
timed_vector(a)

print('\niterating over list in python')
timed_list(a)
```

```
iterating over ndarray in ulab
execution time:  441  us

```

(continues on next page)

(continued from previous page)

```
iterating over list in ulab
execution time:  1266  us

iterating over list in python
execution time:  11379  us
```

12.2 Vectorising generic python functions

numpy: https://numpy.org/doc/stable/reference/generated/numpy.vectorize.html

The examples above use factory functions. In fact, they are nothing but the vectorised versions of the standard mathematical functions. User-defined python functions can also be vectorised by help of vectorize. This function takes a positional argument, namely, the python function that you want to vectorise, and a non-mandatory keyword argument, otypes, which determines the dtype of the output array. The otypes must be None (default), or any of the dtypes defined in ulab. With None, the output is automatically turned into a float array.

The return value of vectorize is a micropython object that can be called as a standard function, but which now accepts either a scalar, an ndarray, or a generic micropython iterable as its sole argument. Note that the function that is to be vectorised must have a single argument.

```
# code to be run in micropython

import ulab as np
from ulab import vector

def f(x):
    return x*x

vf = vector.vectorize(f)

# calling with a scalar
print('{:20}'.format('f on a scalar: '), vf(44.0))

# calling with an ndarray
a = np.array([1, 2, 3, 4])
print('{:20}'.format('f on an ndarray: '), vf(a))

# calling with a list
print('{:20}'.format('f on a list: '), vf([2, 3, 4]))
```

```
f on a scalar:       array([1936.0], dtype=float)
f on an ndarray:     array([1.0, 4.0, 9.0, 16.0], dtype=float)
f on a list:         array([4.0, 9.0, 16.0], dtype=float)
```

As mentioned, the dtype of the resulting ndarray can be specified via the otypes keyword. The value is bound to the function object that vectorize returns, therefore, if the same function is to be vectorised with different output types, then for each type a new function object must be created.

```
# code to be run in micropython

import ulab as np
from ulab import vector

l = [1, 2, 3, 4]
```

(continues on next page)

(continued from previous page)

```
def f(x):
    return x*x

vf1 = vector.vectorize(f, otypes=np.uint8)
vf2 = vector.vectorize(f, otypes=np.float)

print('{:20}'.format('output is uint8: '), vf1(l))
print('{:20}'.format('output is float: '), vf2(l))
```

```
output is uint8:     array([1, 4, 9, 16], dtype=uint8)
output is float:     array([1.0, 4.0, 9.0, 16.0], dtype=float)
```

The `otypes` keyword argument cannot be used for type coercion: if the function evaluates to a float, but `otypes` would dictate an integer type, an exception will be raised:

```
# code to be run in micropython

import ulab as np
from ulab import vector

int_list = [1, 2, 3, 4]
float_list = [1.0, 2.0, 3.0, 4.0]
def f(x):
    return x*x

vf = vector.vectorize(f, otypes=np.uint8)

print('{:20}'.format('integer list: '), vf(int_list))
# this will raise a TypeError exception
print(vf(float_list))
```

```
integer list:        array([1, 4, 9, 16], dtype=uint8)

Traceback (most recent call last):
  File "/dev/shm/micropython.py", line 14, in <module>
TypeError: can't convert float to int
```

12.2.1 Benchmarks

It should be pointed out that the `vectorize` function produces the pseudo-vectorised version of the `python` function that is fed into it, i.e., on the C level, the same `python` function is called, with the all-encompassing `mp_obj_t` type arguments, and all that happens is that the `for` loop in `[f(i) for i in iterable]` runs purely in C. Since type checking and type conversion in `f()` is expensive, the speed-up is not so spectacular as when iterating over an `ndarray` with a factory function: a gain of approximately 30% can be expected, when a native `python` type (e.g., `list`) is returned by the function, and this becomes around 50% (a factor of 2), if conversion to an `ndarray` is also counted.

The following code snippet calculates the square of a 1000 numbers with the vectorised function (which returns an `ndarray`), with `list` comprehension, and with `list` comprehension followed by conversion to an `ndarray`. For comparison, the execution time is measured also for the case, when the square is calculated entirely in `ulab`.

```
# code to be run in micropython

import ulab as np
```

(continues on next page)

(continued from previous page)

```
from ulab import vector

def f(x):
    return x*x

vf = vector.vectorize(f)

@timeit
def timed_vectorised_square(iterable):
    return vf(iterable)

@timeit
def timed_python_square(iterable):
    return [f(i) for i in iterable]

@timeit
def timed_ndarray_square(iterable):
    return np.array([f(i) for i in iterable])

@timeit
def timed_ulab_square(ndarray):
    return ndarray**2

print('vectorised function')
squares = timed_vectorised_square(range(1000))

print('\nlist comprehension')
squares = timed_python_square(range(1000))

print('\nlist comprehension + ndarray conversion')
squares = timed_ndarray_square(range(1000))

print('\nsquaring an ndarray entirely in ulab')
a = np.array(range(1000))
squares = timed_ulab_square(a)
```

```
vectorised function
execution time:  7237  us

list comprehension
execution time:  10248  us

list comprehension + ndarray conversion
execution time:  12562  us

squaring an ndarray entirely in ulab
execution time:  560  us
```

From the comparisons above, it is obvious that `python` functions should only be vectorised, when the same effect cannot be gotten in `ulab` only. However, although the time savings are not significant, there is still a good reason for caring about vectorised functions. Namely, user-defined `python` functions become universal, i.e., they can accept generic iterables as well as `ndarrays` as their arguments. A vectorised function is still a one-liner, resulting in transparent and elegant code.

A final comment on this subject: the `f(x)` that we defined is a *generic* `python` function. This means that it is not required that it just crunches some numbers. It has to return a number object, but it can still access the hardware in the meantime. So, e.g.,

```
led = pyb.LED(2)

def f(x):
    if x < 100:
        led.toggle()
    return x*x
```

is perfectly valid code.

12.3 around

numpy: https://docs.scipy.org/doc/numpy-1.17.0/reference/generated/numpy.around.html

numpy's around function can also be found in the vector sub-module. The function implements the decimals keyword argument with default value 0. The first argument must be an ndarray. If this is not the case, the function raises a TypeError exception. Note that numpy accepts general iterables. The out keyword argument known from numpy is not accepted. The function always returns an ndarray of type mp_float_t.

```
# code to be run in micropython

import ulab as np
from ulab import vector

a = np.array([1, 2.2, 33.33, 444.444])
print('a:\t\t', a)
print('\ndecimals = 0\t', vector.around(a, decimals=0))
print('\ndecimals = 1\t', vector.around(a, decimals=1))
print('\ndecimals = -1\t', vector.around(a, decimals=-1))
```

```
a:             array([1.0, 2.2, 33.33, 444.444], dtype=float)

decimals = 0          array([1.0, 2.0, 33.0, 444.0], dtype=float)

decimals = 1          array([1.0, 2.2, 33.3, 444.4], dtype=float)

decimals = -1         array([0.0, 0.0, 30.0, 440.0], dtype=float)
```

12.4 arctan2

numpy: https://docs.scipy.org/doc/numpy-1.17.0/reference/generated/numpy.arctan2.html

The two-argument inverse tangent function is also part of the vector sub-module. The function implements broadcasting as discussed in the section on ndarrays. Scalars (micropython integers or floats) are also allowed.

```
# code to be run in micropython

import ulab as np
from ulab import vector

a = np.array([1, 2.2, 33.33, 444.444])
print('a:\t\t', a)
print('\narctan2(a, 1.0)\t', vector.arctan2(a, 1.0))
```

(continues on next page)

(continued from previous page)

```
print('\narctan2(1.0, a)\t', vector.arctan2(1.0, a))
print('\narctan2(a, a): \t', vector.arctan2(a, a))
```

```
a:           array([1.0, 2.2, 33.33, 444.444], dtype=float)

arctan2(a, 1.0)      array([0.7853981633974483, 1.14416883366802, 1.5408023243361, 1.
↪568546328341769], dtype=float)

arctan2(1.0, a)      array([0.7853981633974483, 0.426627493126876, 0.
↪02999400245879636, 0.002249998453127392], dtype=float)

arctan2(a, a):       array([0.7853981633974483, 0.7853981633974483, 0.
↪7853981633974483, 0.7853981633974483], dtype=float)
```

CHAPTER 13

Programming ulab

Earlier we have seen, how `ulab`'s functions and methods can be accessed in `micropython`. This last section of the book explains, how these functions are implemented. By the end of this chapter, not only would you be able to extend `ulab`, and write your own `numpy`-compatible functions, but through a deeper understanding of the inner workings of the functions, you would be able to see what the trade-offs are at the `python` level.

13.1 Code organisation

As mentioned earlier, the `python` functions are organised into sub-modules at the C level. Functions in module `x` always begin with the `x_` prefix, so it is relatively easy to navigate the code. Sub-modules are all in their respective folder. E.g., the `filter` sub-module is in `./ulab/code/filter/`, with two files, `./ulab/code/filter/filter.h`, and `./ulab/code/filter/filter.c`. `filter.c` contains two functions, `filter_convolve`, and `filter_sosfilt`, which are bound to the name space either in `ulab_filter_globals_table[]`, or, if `numpy`-compatibility is required, at the top level, in `ulab.c`.

13.2 The `ndarray` object

13.2.1 General comments

`ndarrays` are efficient containers of numerical data of the same type (i.e., signed/unsigned chars, signed/unsigned integers or `mp_float_t`s, which, depending on the platform, are either C `float`s, or C `double`s). Beyond storing the actual data in the void pointer `*array`, the type definition has eight additional members (on top of the `base` type). Namely, `dense`, which tells us, whether the array is dense or sparse (more on this later), the `dtype`, which tells us, how the bytes are to be interpreted. Moreover, the `itemsize`, which stores the size of a single entry in the array, `boolean`, an unsigned integer, which determines, whether the arrays is to be treated as a set of Booleans, or as numerical data, `ndim`, the number of dimensions (`uint8_t`), `len`, the length of the array, the shape (`*size_t`), the strides (`*int32_t`). The length is simply the product of the numbers in `shape`.

The type definition is as follows:

```
typedef struct _ndarray_obj_t {
    mp_obj_base_t base;
    uint8_t dense;
    uint8_t dtype;
    uint8_t itemsize;
    uint8_t boolean;
    uint8_t ndim;
    size_t len;
    size_t shape[ULAB_MAX_DIMS];
    int32_t strides[ULAB_MAX_DIMS];
    void *array;
} ndarray_obj_t;
```

13.2.2 Memory layout

The values of an `ndarray` are stored in a contiguous segment in the RAM. The `ndarray` can be dense, meaning that all numbers in the linear memory segment belong to a linar combination of coordinates, and it can also be sparse, i.e., some elements of the linear storage space will be skipped, when the elements of the tensor are traversed.

In the RAM, the position of the item $M(n_1, n_2, ..., n_{k-1}, n_k)$ in a dense tensor of rank k is given by the linear combination

:raw-latex:‘\begin{equation} P(n_1, n_2, ..., n_{k-1}, n_k) = n_1 s_1 + n_2 s_2 + ... + n_{k-1}s_{k-1} + n_ks_k = \sum_{i=1}^{k}n_is_i \end{equation}‘ where s_i are the strides of the tensor, defined as

:raw-latex:‘\begin{equation} s_i = \prod_{j=i+1}^k l_j \end{equation}‘

where l_j is length of the tensor along the jth axis. When the tensor is sparse (e.g., when the tensor is sliced), the strides along a particular axis will be multiplied by a non-zero integer. If this integer is different to ± 1, the linear combination above cannot access all elements in the RAM, i.e., some numbers will be skipped. Note that $|s_1| > |s_2| > ... > |s_{k-1}| > |s_k|$, even if the tensor is sparse. The statement is trivial for dense tensors, and it follows from the definition of s_i. For sparse tensors, a slice cannot have a step larger than the shape along that axis. But for dense tensors, $s_i/s_{i+1} = l_i$.

When creating a *view*, we simply re-calculate the `strides`, and re-set the `*array` pointer.

13.3 Iterating over elements of a tensor

The `shape` and `strides` members of the array tell us how we have to move our pointer, when we want to read out the numbers. For technical reasons that will become clear later, the numbers in `shape` and in `strides` are aligned to the right, and begin on the right hand side, i.e., if the number of possible dimensions is `ULAB_MAX_DIMS`, then `shape[ULAB_MAX_DIMS-1]` is the length of the last axis, `shape[ULAB_MAX_DIMS-2]` is the length of the last but one axis, and so on. If the number of actual dimensions, `ndim < ULAB_MAX_DIMS`, the first `ULAB_MAX_DIMS - ndim` entries in `shape` and `strides` will be equal to zero, but they could, in fact, be assigned any value, because these will never be accessed in an operation.

With this definition of the strides, the linear combination in $P(n_1, n_2, ..., n_{k-1}, n_k)$ is a one-to-one mapping from the space of tensor coordinates, $(n_1, n_2, ..., n_{k-1}, n_k)$, and the coordinate in the linear array, $n_1 s_1 + n_2 s_2 + ... + n_{k-1} s_{k-1} + n_k s_k$, i.e., no two distinct sets of coordinates will result in the same position in the linear array.

Since the `strides` are given in terms of bytes, when we iterate over an array, the void data pointer is usually cast to `uint8_t`, and the values are converted using the proper data type stored in `ndarray->dtype`. However, there might be cases, when it makes perfect sense to cast `*array` to a different type, in which case the `strides` have to be re-scaled by the value of `ndarray->itemsize`.

13.3.1 Iterating using the unwrapped loops

The following macro definition is taken from vectorise.h, and demonstrates, how we can iterate over a single array in four dimensions.

```
#define ITERATE_VECTOR(type, array, source, sarray) do {
    size_t i=0;
    do {
        size_t j = 0;
        do {
            size_t k = 0;
            do {
                size_t l = 0;
                do {
                    *(array)++ = f(*((type *)(sarray)));
                    (sarray) += (source)->strides[ULAB_MAX_DIMS - 1];
                    l++;
                } while(l < (source)->shape[ULAB_MAX_DIMS-1]);
                (sarray) -= (source)->strides[ULAB_MAX_DIMS - 1] * (source)->
↪shape[ULAB_MAX_DIMS-1];
                (sarray) += (source)->strides[ULAB_MAX_DIMS - 2];
                k++;
            } while(k < (source)->shape[ULAB_MAX_DIMS-2]);
            (sarray) -= (source)->strides[ULAB_MAX_DIMS - 2] * (source)->shape[ULAB_
↪MAX_DIMS-2];
            (sarray) += (source)->strides[ULAB_MAX_DIMS - 3];
            j++;
        } while(j < (source)->shape[ULAB_MAX_DIMS-3]);
        (sarray) -= (source)->strides[ULAB_MAX_DIMS - 3] * (source)->shape[ULAB_MAX_
↪DIMS-3];
        (sarray) += (source)->strides[ULAB_MAX_DIMS - 4];
        i++;
    } while(i < (source)->shape[ULAB_MAX_DIMS-4]);
} while(0)
```

We start with the innermost loop, the one recursing `l`. `array` is already of type `mp_float_t`, while the source array, `sarray`, has been cast to `uint8_t` in the calling function. The numbers contained in `sarray` have to be read out in the proper type dictated by `ndarray->dtype`. This is what happens in the statement `*((type *)(sarray))`, and this number is then fed into the function `f`. Vectorised mathematical functions produce *dense* arrays, and for this reason, we can simply advance the `array` pointer.

The advancing of the `sarray` pointer is a bit more involving: first, in the innermost loop, we simply move forward by the amount given by the last stride, which is `(source)->strides[ULAB_MAX_DIMS - 1]`, because the `shape` and the `strides` are aligned to the right. We move the pointer as many times as given by `(source)->shape[ULAB_MAX_DIMS-1]`, which is the length of the very last axis. Hence the the structure of the loop

```
size_t l = 0;
do {
    ...
    l++;
} while(l < (source)->shape[ULAB_MAX_DIMS-1]);
```

Once we have exhausted the last axis, we have to re-wind the pointer, and advance it by an amount given by the last but one stride. Keep in mind that in the the innermost loop we moved our pointer `(source)->shape[ULAB_MAX_DIMS-1]` times by `(source)->strides[ULAB_MAX_DIMS - 1]`, i.e., we re-wind it by moving it backwards by `(source)->strides[ULAB_MAX_DIMS`

`- 1] * (source)->shape[ULAB_MAX_DIMS-1]`. In the next step, we move forward by `(source)->strides[ULAB_MAX_DIMS - 2]`, which is the last but one stride.

```
(sarray) -= (source)->strides[ULAB_MAX_DIMS - 1] * (source)->shape[ULAB_MAX_DIMS-1];
(sarray) += (source)->strides[ULAB_MAX_DIMS - 2];
```

This pattern must be repeated for each axis of the array, and this is how we arrive at the four nested loops listed above.

13.3.2 Re-winding arrays by means of a function

In addition to un-wrapping the iteration loops by means of macros, there is another way of traversing all elements of a tensor: we note that, since $|s_1| > |s_2| > ... > |s_{k-1}| > |s_k|$, $P(n1, n2, ..., n_{k-1}, n_k)$ changes most slowly in the last coordinate. Hence, if we start from the very beginning, ($n_i = 0$ for all i), and walk along the linear RAM segment, we increment the value of n_k as long as $n_k < l_k$. Once $n_k = l_k$, we have to reset n_k to 0, and increment n_{k-1} by one. After each such round, n_{k-1} will be incremented by one, as long as $n_{k-1} < l_{k-1}$. Once $n_{k-1} = l_{k-1}$, we reset both n_k, and n_{k-1} to 0, and increment n_{k-2} by one.

Rewinding the arrays in this way is implemented in the function `ndarray_rewind_array` in ndarray.c.

```
void ndarray_rewind_array(uint8_t ndim, uint8_t *array, size_t *shape, int32_t␣
↪*strides, size_t *coords) {
    // resets the data pointer of a single array, whenever an axis is full
    // since we always iterate over the very last axis, we have to keep track of
    // the last ndim-2 axes only
    array -= shape[ULAB_MAX_DIMS - 1] * strides[ULAB_MAX_DIMS - 1];
    array += strides[ULAB_MAX_DIMS - 2];
    for(uint8_t i=1; i < ndim-1; i++) {
        coords[ULAB_MAX_DIMS - 1 - i] += 1;
        if(coords[ULAB_MAX_DIMS - 1 - i] == shape[ULAB_MAX_DIMS - 1 - i]) { // we are␣
↪at a dimension boundary
            array -= shape[ULAB_MAX_DIMS - 1 - i] * strides[ULAB_MAX_DIMS - 1 - i];
            array += strides[ULAB_MAX_DIMS - 2 - i];
            coords[ULAB_MAX_DIMS - 1 - i] = 0;
            coords[ULAB_MAX_DIMS - 2 - i] += 1;
        } else { // coordinates can change only, if the last coordinate changes
            return;
        }
    }
}
```

and the function would be called as in the snippet below. Note that the innermost loop is factored out, so that we can save the `if(...)` statement for the last axis.

```
size_t *coords = ndarray_new_coords(results->ndim);
for(size_t i=0; i < results->len/results->shape[ULAB_MAX_DIMS -1]; i++) {
    size_t l = 0;
    do {
        ...
        l++;
    } while(l < results->shape[ULAB_MAX_DIMS - 1]);
    ndarray_rewind_array(results->ndim, array, results->shape, strides, coords);
} while(0)
```

The advantage of this method is that the implementation is independent of the number of dimensions: the iteration requires more or less the same flash space for 2 dimensions as for 22. However, the price we have to pay for this convenience is the extra function call.

13.4 Iterating over two ndarrays simultaneously: broadcasting

Whenever we invoke a binary operator, call a function with two arguments of `ndarray` type, or assign something to an `ndarray`, we have to iterate over two views at the same time. The task is trivial, if the two `ndarrays` in question have the same shape (but not necessarily the same set of strides), because in this case, we can still iterate in the same loop. All that happens is that we move two data pointers in sync.

The problem becomes a bit more involving, when the shapes of the two `ndarrays` are not identical. For such cases, `numpy` defines so-called broadcasting, which boils down to two rules.

1. The shapes in the tensor with lower rank has to be prepended with axes of size 1 till the two ranks become equal.
2. Along all axes the two tensors should have the same size, or one of the sizes must be 1.

If, after applying the first rule the second is not satisfied, the two `ndarrays` cannot be broadcast together.

Now, let us suppose that we have two compatible `ndarrays`, i.e., after applying the first rule, the second is satisfied. How do we iterate over the elements in the tensors?

We should recall, what exactly we do, when iterating over a single array: normally, we move the data pointer by the last stride, except, when we arrive at a dimension boundary (when the last axis is exhausted). At that point, we move the pointer by an amount dictated by the strides. And this is the key: *dictated by the strides*. Now, if we have two arrays that are originally not compatible, we define new strides for them, and use these in the iteration. With that, we are back to the case, where we had two compatible arrays.

Now, let us look at the second broadcasting rule: if the two arrays have the same size, we take both `ndarrays`' strides along that axis. If, on the other hand, one of the `ndarrays` is of length 1 along one of its axes, we set the corresponding strides to 0. This will ensure that that data pointer is not moved, when we iterate over both `ndarrays` at the same time.

Thus, in order to implement broadcasting, we first have to check, whether the two above-mentioned rules can be satisfied, and if so, we have to find the two new sets strides.

The `ndarray_can_broadcast` function from ndarray.c takes two `ndarrays`, and returns `true`, if the two arrays can be broadcast together. At the same time, it also calculates new strides for the two arrays, so that they can be iterated over at the same time.

```
bool ndarray_can_broadcast(ndarray_obj_t *lhs, ndarray_obj_t *rhs, uint8_t *ndim,␣
↪size_t *shape, int32_t *lstrides, int32_t *rstrides) {
    // returns True or False, depending on, whether the two arrays can be broadcast␣
↪together
    // numpy's broadcasting rules are as follows:
    //
    // 1. the two shapes are either equal
    // 2. one of the shapes is 1
    memset(lstrides, 0, sizeof(size_t)*ULAB_MAX_DIMS);
    memset(rstrides, 0, sizeof(size_t)*ULAB_MAX_DIMS);
    lstrides[ULAB_MAX_DIMS - 1] = lhs->strides[ULAB_MAX_DIMS - 1];
    rstrides[ULAB_MAX_DIMS - 1] = rhs->strides[ULAB_MAX_DIMS - 1];
    for(uint8_t i=ULAB_MAX_DIMS; i > 0; i--) {
        if((lhs->shape[i-1] == rhs->shape[i-1]) || (lhs->shape[i-1] == 0) || (lhs->
↪shape[i-1] == 1) ||
        (rhs->shape[i-1] == 0) || (rhs->shape[i-1] == 1)) {
            shape[i-1] = MAX(lhs->shape[i-1], rhs->shape[i-1]);
            if(shape[i-1] > 0) (*ndim)++;
            if(lhs->shape[i-1] < 2) {
                lstrides[i-1] = 0;
            } else {
                lstrides[i-1] = lhs->strides[i-1];
```

(continues on next page)

(continued from previous page)

```
            }
            if(rhs->shape[i-1] < 2) {
                rstrides[i-1] = 0;
            } else {
                rstrides[i-1] = rhs->strides[i-1];
            }
        } else {
            return false;
        }
    }
    return true;
}
```

A good example of how the function would be called can be found in vectorise.c, in the `vectorise_arctan2` function:

```
mp_obj_t vectorise_arctan2(mp_obj_t y, mp_obj_t x) {
    ...
    uint8_t ndim = 0;
    size_t *shape = m_new(size_t, ULAB_MAX_DIMS);
    int32_t *xstrides = m_new(int32_t, ULAB_MAX_DIMS);
    int32_t *ystrides = m_new(int32_t, ULAB_MAX_DIMS);
    if(!ndarray_can_broadcast(ndarray_x, ndarray_y, &ndim, shape, xstrides,␣
↪ystrides)) {
        mp_raise_ValueError(translate("operands could not be broadcast together"));
        m_del(size_t, shape, ULAB_MAX_DIMS);
        m_del(int32_t, xstrides, ULAB_MAX_DIMS);
        m_del(int32_t, ystrides, ULAB_MAX_DIMS);
    }

    uint8_t *xarray = (uint8_t *)ndarray_x->array;
    uint8_t *yarray = (uint8_t *)ndarray_y->array;

    ndarray_obj_t *results = ndarray_new_dense_ndarray(ndim, shape, NDARRAY_FLOAT);
    mp_float_t *rarray = (mp_float_t *)results->array;
    ...
```

After the new strides have been calculated, the iteration loop is identical to what we discussed in the previous section.

13.5 Contracting an `ndarray`

There are many operations that reduce the number of dimensions of an `ndarray` by 1, i.e., that remove an axis from the tensor. The drill is the same as before, with the exception that first we have to remove the `strides` and `shape` that corresponds to the axis along which we intend to contract. The `numerical_reduce_axes` function from numerical.c does that.

```
static void numerical_reduce_axes(ndarray_obj_t *ndarray, int8_t axis, size_t *shape,␣
↪int32_t *strides) {
    // removes the values corresponding to a single axis from the shape and strides␣
↪array
    uint8_t index = ULAB_MAX_DIMS - ndarray->ndim + axis;
    if((ndarray->ndim == 1) && (axis == 0)) {
        index = 0;
        shape[ULAB_MAX_DIMS - 1] = 0;
```

(continues on next page)

(continued from previous page)

```
        return;
    }
    for(uint8_t i = ULAB_MAX_DIMS - 1; i > 0; i--) {
        if(i > index) {
            shape[i] = ndarray->shape[i];
            strides[i] = ndarray->strides[i];
        } else {
            shape[i] = ndarray->shape[i-1];
            strides[i] = ndarray->strides[i-1];
        }
    }
}
```

Once the reduced `strides` and `shape` are known, we place the axis in question in the innermost loop, and wrap it with the loops, whose coordinates are in the `strides`, and `shape` arrays. The `RUN_STD` macro from numerical.h is a good example. The macro is expanded in the `numerical_sum_mean_std_ndarray` function.

```
static mp_obj_t numerical_sum_mean_std_ndarray(ndarray_obj_t *ndarray, mp_obj_t axis,␣
↪uint8_t optype, size_t ddof) {
    uint8_t *array = (uint8_t *)ndarray->array;
    size_t *shape = m_new(size_t, ULAB_MAX_DIMS);
    memset(shape, 0, sizeof(size_t)*ULAB_MAX_DIMS);
    int32_t *strides = m_new(int32_t, ULAB_MAX_DIMS);
    memset(strides, 0, sizeof(uint32_t)*ULAB_MAX_DIMS);

    int8_t ax = mp_obj_get_int(axis);
    if(ax < 0) ax += ndarray->ndim;
    if((ax < 0) || (ax > ndarray->ndim - 1)) {
        mp_raise_ValueError(translate("index out of range"));
    }
    numerical_reduce_axes(ndarray, ax, shape, strides);
    uint8_t index = ULAB_MAX_DIMS - ndarray->ndim + ax;
    ndarray_obj_t *results = NULL;
    uint8_t *rarray = NULL;
    ...
```

Here is the macro for the three-dimensional case:

```
#define RUN_STD(ndarray, type, array, results, r, shape, strides, index, div) do {
    size_t k = 0;
    do {
        size_t l = 0;
        do {
            RUN_STD1((ndarray), type, (array), (results), (r), (index), (div));
            (array) -= (ndarray)->strides[(index)] * (ndarray)->shape[(index)];
            (array) += (strides)[ULAB_MAX_DIMS - 1];
            l++;
        } while(l < (shape)[ULAB_MAX_DIMS - 1]);
        (array) -= (strides)[ULAB_MAX_DIMS - 2] * (shape)[ULAB_MAX_DIMS-2];
        (array) += (strides)[ULAB_MAX_DIMS - 3];
        k++;
    } while(k < (shape)[ULAB_MAX_DIMS - 2]);
} while(0)
```

In `RUN_STD`, we simply move our pointers; the calculation itself happens in the `RUN_STD1` macro below. (Note that this is the implementation of the numerically stable Welford algorithm.)

```
#define RUN_STD1(ndarray, type, array, results, r, index, div)
({
    mp_float_t M, m, S = 0.0, s = 0.0;
    M = m = *(mp_float_t *)((type *)(array));
    for(size_t i=1; i < (ndarray)->shape[(index)]; i++) {
        (array) += (ndarray)->strides[(index)];
        mp_float_t value = *(mp_float_t *)((type *)(array));
        m = M + (value - M) / (mp_float_t)i;
        s = S + (value - M) * (value - m);
        M = m;
        S = s;
    }
    (array) += (ndarray)->strides[(index)];
    *(r)++ = MICROPY_FLOAT_C_FUN(sqrt)((ndarray)->shape[(index)] * s / (div));
})
```

13.6 Upcasting

When in an operation the `dtype`s of two arrays are different, the result's `dtype` will be decided by the following upcasting rules:

1. Operations with two `ndarray`s of the same `dtype` preserve their `dtype`, even when the results overflow.
2. if either of the operands is a float, the result automatically becomes a float
3. otherwise
    - `uint8` + `int8` => `int16`,
    - `uint8` + `int16` => `int16`
    - `uint8` + `uint16` => `uint16`
    - `int8` + `int16` => `int16`
    - `int8` + `uint16` => `uint16` (in numpy, the result is a `int32`)
    - `uint16` + `int16` => `float` (in numpy, the result is a `int32`)
4. When one operand of a binary operation is a generic scalar `micropython` variable, i.e., `mp_obj_int`, or `mp_obj_float`, it will be converted to a linear array of length 1, and with the smallest `dtype` that can accommodate the variable in question. After that the broadcasting rules apply, as described in the section *Iterating over two ndarrays simultaneously: broadcasting*

Upcasting is resolved in place, wherever it is required. Notable examples can be found in ndarray_operators.c

13.7 Slicing and indexing

An `ndarray` can be indexed with three types of objects: integer scalars, slices, and another `ndarray`, whose elements are either integer scalars, or Booleans. Since slice and integer indices can be thought of as modifications of the `strides`, these indices return a view of the `ndarray`. This statement does not hold for `ndarray` indices, and therefore, the return a copy of the array.

13.8 Extending ulab

The `user` module is disabled by default, as can be seen from the last couple of lines of ulab.h

```
// user-defined module
#ifndef ULAB_USER_MODULE
#define ULAB_USER_MODULE                (0)
#endif
```

The module contains a very simple function, `user_dummy`, and this function is bound to the module itself. In other words, even if the module is enabled, one has to `import`:

```
import ulab
from ulab import user

user.dummy_function(2.5)
```

which should just return 5.0. Even if `numpy`-compatibility is required (i.e., if most functions are bound at the top level to `ulab` directly), having to `import` the module has a great advantage. Namely, only the user.h and user.c files have to be modified, thus it should be relatively straightforward to update your local copy from github.

Now, let us see, how we can add a more meaningful function.

13.9 Creating a new ndarray

In the *General comments* sections we have seen the type definition of an `ndarray`. This structure can be generated by means of a couple of functions listed in ndarray.c.

13.9.1 ndarray_new_ndarray

The `ndarray_new_ndarray` functions is called by all other array-generating functions. It takes the number of dimensions, `ndim`, a `uint8_t`, the `shape`, a pointer to `size_t`, the `strides`, a pointer to `int32_t`, and `dtype`, another `uint8_t` as its arguments, and returns a new array with all entries initialised to 0.

Assuming that `ULAB_MAX_DIMS > 2`, a new dense array of dimension 3, of `shape` (3, 4, 5), of `strides` (1000, 200, 10), and `dtype` `uint16_t` can be generated by the following instructions

```
size_t *shape = m_new(size_t, ULAB_MAX_DIMS);
shape[ULAB_MAX_DIMS - 1] = 5;
shape[ULAB_MAX_DIMS - 2] = 4;
shape[ULAB_MAX_DIMS - 3] = 3;

int32_t *strides = m_new(int32_t, ULAB_MAX_DIMS);
strides[ULAB_MAX_DIMS - 1] = 10;
strides[ULAB_MAX_DIMS - 2] = 200;
strides[ULAB_MAX_DIMS - 3] = 1000;

ndarray_obj_t *new_ndarray = ndarray_new_ndarray(3, shape, strides, NDARRAY_UINT16);
```

13.9.2 ndarray_new_dense_ndarray

The functions simply calculates the `strides` from the `shape`, and calls `ndarray_new_ndarray`. Assuming that `ULAB_MAX_DIMS > 2`, a new dense array of dimension 3, of `shape` (3, 4, 5), and `dtype` `mp_float_t` can be generated by the following instructions

```
size_t *shape = m_new(size_t, ULAB_MAX_DIMS);
shape[ULAB_MAX_DIMS - 1] = 5;
shape[ULAB_MAX_DIMS - 2] = 4;
shape[ULAB_MAX_DIMS - 3] = 3;

ndarray_obj_t *new_ndarray = ndarray_new_dense_ndarray(3, shape, NDARRAY_FLOAT);
```

13.9.3 ndarray_new_linear_array

Since the dimensions of a linear array are known (1), the `ndarray_new_linear_array` takes the `length`, a `size_t`, and the `dtype`, an `uint8_t`. Internally, `ndarray_new_linear_array` generates the `shape` array, and calls `ndarray_new_dense_array` with `ndim = 1`.

A linear array of length 100, and `dtype` `uint8` could be created by the function call

```
ndarray_obj_t *new_ndarray = ndarray_new_linear_array(100, NDARRAY_UINT8)
```

13.9.4 ndarray_new_ndarray_from_tuple

This function takes a `tuple`, which should hold the lengths of the axes (in other words, the `shape`), and the `dtype`, and calls internally `ndarray_new_dense_array`. A new `ndarray` can be generated by calling

```
ndarray_obj_t *new_ndarray = ndarray_new_ndarray_from_tuple(shape, NDARRAY_FLOAT);
```

where `shape` is a tuple.

13.9.5 ndarray_new_view

This function crates a *view*, and takes the source, an `ndarray`, the number of dimensions, an `uint8_t`, the `shape`, a pointer to `size_t`, the `strides`, a pointer to `int32_t`, and the offset, an `int32_t` as arguments. The offset is the number of bytes by which the void `array` pointer is shifted. E.g., the `python` statement

```
a = np.array([0, 1, 2, 3, 4, 5], dtype=uint8)
b = a[1::2]
```

produces the array

```
array([1, 3, 5], dtype=uint8)
```

which holds its data at position `x0 + 1`, if `a`'s pointer is at `x0`. In this particular case, the offset is 1.

The array `b` from the example above could be generated as

```
size_t *shape = m_new(size_t, ULAB_MAX_DIMS);
shape[ULAB_MAX_DIMS - 1] = 3;

int32_t *strides = m_new(int32_t, ULAB_MAX_DIMS);
strides[ULAB_MAX_DIMS - 1] = 2;

int32_t offset = 1;
uint8_t ndim = 1;
```

(continues on next page)

(continued from previous page)

```
ndarray_obj_t *new_ndarray = ndarray_new_view(ndarray_a, ndim, shape, strides,␣
↪offset);
```

13.9.6 ndarray_copy_array

The `ndarray_copy_array` function can be used for copying the contents of an array. Note that the target array has to be created beforehand. E.g., a one-to-one copy can be gotten by

```
ndarray_obj_t *new_ndarray = ndarray_new_ndarray(source->ndim, source->shape, source->
↪strides, source->dtype);
ndarray_copy_array(source, new_ndarray);
```

Note that the function cannot be used for forcing type conversion, i.e., the input and output types must be identical, because the function simply calls the `memcpy` function. On the other hand, the input and output `strides` do not necessarily have to be equal.

13.9.7 ndarray_copy_view

The `ndarray_obj_t *new_ndarray = ...` instruction can be saved by calling the `ndarray_copy_view` function with the single `source` argument.

13.10 Accessing data in the ndarray

Having seen, how arrays can be generated and copied, it is time to look at how the data in an `ndarray` can be accessed and modified.

For starters, let us suppose that the object in question comes from the user (i.e., via the `micropython` interface), First, we have to acquire a pointer to the `ndarray` by calling

```
ndarray_obj_t *ndarray = MP_OBJ_TO_PTR(object_in);
```

If it is not clear, whether the object is an `ndarray` (e.g., if we want to write a function that can take `ndarrays`, and other iterables as its argument), we find this out by evaluating

```
MP_OBJ_IS_TYPE(object_in, &ulab_ndarray_type)
```

which should return `true`. Once the pointer is at our disposal, we can get a pointer to the underlying numerical array as discussed earlier, i.e.,

```
uint8_t *array = (uint8_t *)ndarray->array;
```

If you need to find out the `dtype` of the array, you can get it by accessing the `dtype` member of the `ndarray`, i.e.,

```
ndarray->dtype
```

should be equal to `B`, `b`, `H`, `h`, or `f`. The size of a single item is stored in the `itemsize` member. This number should be equal to 1, if the `dtype` is `B`, or `b`, 2, if the `dtype` is `H`, or `h`, 4, if the `dtype` is `f`, and 8 for `d`.

13.11 Boilerplate

In the next section, we will construct a function that generates the element-wise square of a dense array, otherwise, raises a `TypeError` exception. Dense arrays can easily be iterated over, since we do not have to care about the `shape` and the `strides`. If the array is sparse, the section *Iterating over elements of a tensor* should contain hints as to how the iteration can be implemented.

The function is listed under user.c. The `user` module is bound to `ulab` in ulab.c in the lines

```
#if ULAB_USER_MODULE
    { MP_ROM_QSTR(MP_QSTR_user), MP_ROM_PTR(&ulab_user_module) },
#endif
```

which assumes that at the very end of ulab.h the

```
// user-defined module
#ifndef ULAB_USER_MODULE
#define ULAB_USER_MODULE                (1)
#endif
```

constant has been set to 1. After compilation, you can call a particular `user` function in `python` by importing the module first, i.e.,

```
import ulab
from ulab import user

user.some_function(...)
```

This separation of user-defined functions from the rest of the code ensures that the integrity of the main module and all its functions are always preserved. Even in case of a catastrophic failure, you can easily clone `ulab` anew, and start over.

And now the function:

```
static mp_obj_t user_square(mp_obj_t arg) {
    // the function takes a single dense ndarray, and calculates the
    // element-wise square of its entries

    // raise a TypeError exception, if the input is not an ndarray
    if(!MP_OBJ_IS_TYPE(arg, &ulab_ndarray_type)) {
        mp_raise_TypeError(translate("input must be an ndarray"));
    }
    ndarray_obj_t *ndarray = MP_OBJ_TO_PTR(arg);

    // make sure that the input is a dense array
    if(!ndarray_is_dense(ndarray)) {
        mp_raise_TypeError(translate("input must be a dense ndarray"));
    }

    // if the input is a dense array, create `results` with the same number of
    // dimensions, shape, and dtype
    ndarray_obj_t *results = ndarray_new_dense_ndarray(ndarray->ndim, ndarray->shape,␣
↪ndarray->dtype);

    // since in a dense array the iteration over the elements is trivial, we
    // can cast the data arrays ndarray->array and results->array to the actual type
    if(ndarray->dtype == NDARRAY_UINT8) {
```

(continues on next page)

(continued from previous page)

```
        uint8_t *array = (uint8_t *)ndarray->array;
        uint8_t *rarray = (uint8_t *)results->array;
        for(size_t i=0; i < ndarray->len; i++, array++) {
            *rarray++ = (*array) * (*array);
        }
    } else if(ndarray->dtype == NDARRAY_INT8) {
        int8_t *array = (int8_t *)ndarray->array;
        int8_t *rarray = (int8_t *)results->array;
        for(size_t i=0; i < ndarray->len; i++, array++) {
            *rarray++ = (*array) * (*array);
        }
    } else if(ndarray->dtype == NDARRAY_UINT16) {
        uint16_t *array = (uint16_t *)ndarray->array;
        uint16_t *rarray = (uint16_t *)results->array;
        for(size_t i=0; i < ndarray->len; i++, array++) {
            *rarray++ = (*array) * (*array);
        }
    } else if(ndarray->dtype == NDARRAY_INT16) {
        int16_t *array = (int16_t *)ndarray->array;
        int16_t *rarray = (int16_t *)results->array;
        for(size_t i=0; i < ndarray->len; i++, array++) {
            *rarray++ = (*array) * (*array);
        }
    } else { // if we end up here, the dtype is NDARRAY_FLOAT
        mp_float_t *array = (mp_float_t *)ndarray->array;
        mp_float_t *rarray = (mp_float_t *)results->array;
        for(size_t i=0; i < ndarray->len; i++, array++) {
            *rarray++ = (*array) * (*array);
        }
    }
    // at the end, return a micropython object
    return MP_OBJ_FROM_PTR(results);
}
```

To summarise, the steps for *implementing* a function are

1. If necessary, inspect the type of the input object, which is always a `mp_obj_t` object
2. If the input is an `ndarray_obj_t`, acquire a pointer to it by calling `ndarray_obj_t *ndarray = MP_OBJ_TO_PTR(arg);`
3. Create a new array, or modify the existing one; get a pointer to the data by calling `uint8_t *array = (uint8_t *)ndarray->array;`, or something equivalent
4. Once the new data have been calculated, return a `micropython` object by calling `MP_OBJ_FROM_PTR(...)`.

The listing above contains the implementation of the function, but as such, it cannot be called from `python`: it still has to be bound to the name space. This we do by first defining a function object in

```
MP_DEFINE_CONST_FUN_OBJ_1(user_square_obj, user_square);
```

`micropython` defines a number of `MP_DEFINE_CONST_FUN_OBJ_N` macros in obj.h. `N` is always the number of arguments the function takes. We had a function definition `static mp_obj_t user_square(mp_obj_t arg)`, i.e., we dealt with a single argument.

Finally, we have to bind this function object in the globals table of the `user` module:

```
STATIC const mp_rom_map_elem_t ulab_user_globals_table[] = {
    { MP_OBJ_NEW_QSTR(MP_QSTR___name__), MP_OBJ_NEW_QSTR(MP_QSTR_user) },
    { MP_OBJ_NEW_QSTR(MP_QSTR_square), (mp_obj_t)&user_square_obj },
};
```

Thus, the three steps required for the definition of a user-defined function are

1. The low-level implementation of the function itself
2. The definition of a function object by calling MP_DEFINE_CONST_FUN_OBJ_N()
3. Binding this function object to the namespace in the `ulab_user_globals_table[]`

```
# code to be run in CPython
```

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index