4 Array Computing and Curve Plotting

4.1 Compulsory Exercises

Exercise 4.1 *Fill lists with function values.*

A function with many applications in science is defined as

$$h(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2} \tag{1}$$

Fill lists `xlist` and `hlist` with x and $h(x)$ values for uniformly spaced x coordinates in $[-4, 4]$. You may adapt the example in Chapter 4.2.1[1]. Name of program file: `fill_lists.py`.

Exercise 4.2 *Fill arrays; loop version.*

The aim is to fill two arrays `x` and `h` with x and $h(x)$ values, where $h(x)$ is defined in (1). Let the x values be uniformly spaced in $[-4, 4]$. Create two arrays of zeros and fill both arrays with values inside a loop. Name of program file: `fill_arrays_loop.py`.

Exercise 4.3 *Fill arrays; vectorized version.*

Vectorize the code in Exercise 4.2 by creating the x values using the `linspace` function and by evaluating $h(x)$ for an array argument. Name of program file: `fill_arrays_vectorized.py`.

Exercise 4.4 *Apply a function to a vector.*

Given a vector $v = (2, 3, -1)$ and a function $f(x) = x^3 + xe^x + 1$, apply f to each element in v. Then calculate $f(v)$ as $v^3 + v * e^v + 1$ using the vector computing rules. Show that the two results are equal.

Exercise 4.10 *Plot the trajectory of a ball.*

The formula for the trajectory of a ball is given by

$$f(x) = x\tan(\theta) - \frac{1}{2v_0^2}\frac{gx^2}{\cos^2\theta} + y_0. \tag{2}$$

In a program, first read the input data y_0 , θ, and v_0 from the command line. Then compute where the ball hits the ground, i.e., the value x_g for which $f(x_g) = 0$. Plot the trajectory $y = f(x)$ for $x \in [0, x_g]$, using the same scale on the x and y axes such that we get a visually correct view of the trajectory. Name of program file: `plot_trajectory.py`.

4.2 Advanced Exercises

Exercise 4.21 *Plot functions from the command line.*

For quickly get a plot a function $f(x)$ for $x \in [x_{\min}, x_{\max}]$ it could be nice to a have a program that takes the minimum amount of information from the command line and produces a plot on the screen and a hardcopy `tmp.eps`. The usage of the program goes as follows:

[1]See the Appendix A.1

Code 1: Commands on the command-line/terminal

```
plotf.py "f(x)" xmin xmax
```

A specific example is

Code 2: Commands on the command-line/terminal

```
plotf.py "exp(-0.2*x)*sin(2*pi*x)" 0 4*pi
```

Hint: Make x coordinates from the 2nd and 3rd command-line arguments and then use `eval` (or `StringFunction` from Chapters 3.1.4 and 4.4.3[2]) on the first 1st argument. Try to write as short program as possible (we leave it to Exercise 4.22 to test for valid input). Name of program file: `plotf_v1.py`.

Exercise 4.22 *Improve the program from Exericse 4.21.*

Equip the program from Exericse 4.21 with tests on valid input on the command line. Also allow an optional 4th command-line argument for the number of points along the function curve. Set this number to 501 if it is not given. Name of program file: `plotf.py`.

Exercise 4.28 *Extend Exercise 4.4 to a rank 2 array.*

Let A be a two-dimensional array with two indices:

$$\begin{bmatrix} 0 & 12 & -1 \\ -1 & -1 & -1 \\ 11 & 5 & 5 \end{bmatrix}$$

Apply the function f from Exercise 4.4 to each element in A. Thereafter, calculate the result of the array expression $A**3 + A*e^A + 1$ and demonstrate that the end result of the two methods are the same.

A Appendix

A.1 Example from Chapter 4.2.1:

Suppose we have a function $f(x)$ and want to evaluate this function at a number of x points $x_0, x_1, \ldots, x_{n-1}$. We could collect the n pairs $(x_i, f(x_i))$ in a list, or we could collect all the x_i values, for $i = 0, \ldots, n-1$, in a list and all the associated $f(x_i)$ values in another list. We learned how to create such lists in Chapter 2, but as a review, we present the relevant program statements here:

Code 3: Example Code from Chapter 4.2.1

```
def f(x):
    return x**3          # sample function
n = 5                    # no of points along the x axis
```

[2]see Appendix A.2 and A.3

```
dx = 1.0/(n-1)          # spacing between x points in [0,1]
xlist = [i*dx for i in range(n)]
ylist = [f(x) for x in xlist]
pairs = [[x, y] for x, y in zip(xlist, ylist)]
```

Here we have used list comprehensions for achieving compact code. Make sure that you understand what is going on in these list comprehensions (you are encouraged to write the same code using standard for loops and appending new list elements in each pass of the loops).

A.2 Chapter 3.1.4: Turning String Expressions into Functions

An example of the `StringFunction`:

Code 4: Example Code from Chapter 3.1.4 (`StringFunction`)

```
>>> from scitools.StringFunction import StringFunction
>>> formula = 'exp(x)*sin(x)'
>>> f = StringFunction(formula) # turn formula into function f(x)
```

The `f` object now behaves as an ordinary Python function of `x`:

```
>>> f(0)
0.0
>>> f(pi)
2.8338239229952166e-15
>>> f(log(1))
0.0
```

Expressions involving other independent variables than `x` are also possible. Here is an example with the function $g(t) = Ae^{-at}\sin(\omega x)$:

```
g = StringFunction('A*exp(-a*t)*sin(omega*x)',
                   independent_variable='t',
                   A=1, a=0.1, omega=pi, x=0.5)
```

The first argument is the function formula, as before, but now we need to specify the name of the independent variable (`'x'` is default). The other parameters in the function (A, a, ω, and x) must be specified with values, and we use keyword arguments, consistent with the names in the function formula, for this purpose. Any of the parameters `A`, `a`, `omega`, and `x` can be changed later by calls like

```
g.set_parameters(omega=0.1)
g.set_parameters(omega=0.1, A=5, x=0)
```

Calling `g(t)` works as if `g` were a plain Python function of `t`, which "remembers" all the parameters `A`, `a`, `omega`, and `x`, and their values. You can use pydoc to bring up more documentation on the possibilities with StringFunction. Just run pydoc scitools.StringFunction.StringFunction.

A final important point is that `StringFunction` objects are as computationally efficient as hand-written Python functions.

A.3 Chapter 4.4.3: Vectorizing StringFunction Objects

The `StringFunction` object described in Chapter 3.1.4 does unfortunately not work with array arguments unless we explicitly tell the object to do so. The recipe is very simple. Say `f` is some `StringFunction` object. To allow array arguments we must first call `f.vectorize(globals())` once:

Code 5: Example Code from Chapter 4.4.3 (StringFunction)

```
f = StringFunction(formula)
x = linspace(0, 1, 30)
# f(x) will in general not work
from numpy import *
f.vectorize(globals())
# now f works with array arguments:
values = f(x)
```

It is important that you import everything from `numpy` or `scitools.std` *before* you call `f.vectorize`. We suggest to take the `f.vectorize` call as a magic recipe.

Even after calling `f.vectorize(globals())` a `StringFunction` object may face problems with vectorization. One example is a piecewise constant function as specified by a string expression `'1 if x > 2 else 0'`. One remedy is to use the vectorized version of an `if` test: `'where(x > 2, 1, 0)'`. For an average user of the program this construct is not at all obvious so a more user-friendly solution is to apply `vectorize` from `numpy`:

```
f = vectorize(f) # vectorize a StringFunction f
```

The line above is therefore the most general (but also the slowest) way of vectorizing a `StringFunction` object. After that call, `f` is no more a `StringFunction` object, but `f` behaves as a (vectorized) function. The `vectorize` tool from `numpy` can be used to allow any Python function taking scalar arguments to also accept array arguments.

To get better speed, one can use `vectorize(f)` only in the case the formula in `f` contains an inline `if` test (e.g., recognized by the string `' else '` inside the formula). Otherwise, we use `f.vectorize`. The formula in `f` is obtained by `str(f)` so we can test

```
if ' else ' in str(f):
    f = vectorize(f)
else:
    f.vectorize(globals())
```