Lecture Notes PX281: Computational Physics

Y.A. Ramachers, University of Warwick

Contents

Preface

Python environment

This module uses the Python programming language. You should have full access to the ITS supplied **Anaconda** package application on every desktop machine on campus.

In case you wish to have Python on your own machine, it is recommended to install **Anaconda**. It contains all the Python scientific libraries, the Python language itself and a full coding environment in the form of the **Spyder** application with a smart editor and terminal and help system etc. **MS Visual Studio Code** is another popular development environment and it plays nicely with an installed Anaconda package, i.e. finds and uses it if required.

We will be running Python on Jupyter notebooks, see first section, for assignments. This will be handled by the **Notable** system and you get to that from the PX281 Moodle page. You could in principle pass all assignments on that remote computing system from Notable, except for the final project assignment. It is therefore still recommended setting up your own Python environment such that you can work independently from the cloud solution.

1 First lecture: Warm up

Physics and programming: what does one have to do with the other and why should you bother?

1.1 Motivation

The first year programming workshop was a core module for all students of the Physics Department since programming is considered to be an essential skill for all physicists. This Computational Physics module, PX281, is optional. It will assume all the skills and knowledge taught on the first year module, in particular the Python programming language to the level of the first year module. Forcing a skill upon unsuspecting students can be a double-edged sword. The academics know how essential the skill is for you but not all students necessarily agree or see the point. This optional module does not enforce anything but should serve to allow you to get interested in the skill of programming on the basis of **enabling exciting Physics**.

PX281 consists of 10 lectures and 20 workshop session. Clearly, the workshops are the place to be since learning by doing is the principle for this module.

Trying to get to exciting physics on the basis of first year taught material is challenging. It is expected that the current selection of topics will be refined over time. The broad topics we will cover on this module are

1. Reminders from the First Year module
2. Data analysis: inference from numbers
3. Integration and Differentiation: signal processing
4. Random numbers, Monte Carlo applications
5. Ordinary Differential Equations, dynamics on the computer
6. Speed-up Python code and what is a compiler?

Obviously, reviewing some of the first year module is a requirement to move forward on this module. After all, a lot has happened since you finished that core Python module and quite naturally, you will have forgotten some of it. Skills, any skills, are retained by practice after all. If you didn't program in the last few months then of course something will be lost. Nothing to worry about. It will all come back very swiftly if you let it.

Data analysis already sounds quite useful for any scientist, and it is. Again, in itself this is more a skill than a new exciting Physics topic. In order to make this palatable for your tastes we will attempt to give it some Astronomy and Particle Physics flavour.

The next section includes a classic of programming, i.e. how to let the computer calculate complicated **integrals** we often encounter in Physics. Most, if not all textbooks deal with this topic and I can't stress enough how off-putting and boring it is in case one isn't interested in numerical algorithms themselves. If you set out to do interesting physics on the computer, calculus should simply happen, clean, clear and reliable. It is what you can do with that calculus that ought to be exciting.

Therefore, we will boldly ignore the textbooks and go straight to direct applications of integration and(!) differentiation in the form of digital signal processing. That is all the stuff happening in practically every physics experiment as well as right now in your various gadgets like mobile phones. It also allows to sneak-in a beyond first year maths topic, the Fourier transform, well before you are meant to learn about it at the end of the second year. Your future supervisors of projects and PhD's will be well pleased to hear that you actually know what to do with all those waveforms your new Warp drive controller is spitting out.

Random numbers are a big topic for doing physics on the computer as they are an essential ingredient in modelling physics on the computer. There is already a sizeable section on random numbers in physics in the first year module and we will build on that. Therefore, please review that section in the first year script. It already contains all the important elements we need to use here like the random walk and all the important Python random number functions. Standard examples for computational physics always contain the random walk and diffusion in a box. Here the task is to lead you to more interesting random numbers.

Finally, ordinary differential equations (ODEs) describe Physics, quite a lot of it. You don't know that yet but most fundamental laws in modern physics are actually described by partial differential equations (PDE). These we will leave for the computational third year module. ODEs, however, play a very important role in Physics regardless and show up at surprising places. Particle dynamics for instance can often we described successfully with classical mechanics even though we deal with microscopic physics. That would be plain Newton laws but an awful lot of them, hence the computer. Likewise, very complicated processes in Nano-systems can be described successfully in space and time with so called rate equations, again several coupled ODEs, impossible to solve analytically.

The collection of the above four areas of computational physics are really only the tip of the iceberg. Each one alone is worth its own textbook with very

sophisticated physics and science. Each one plays an important role all the way to the current frontiers of science, not even to mention all the applications outside of academia (big data, machine learning, etc).

Here you get a taster of the skill with the help of a healthy dose of motivation from Physics but in the end this is all about programming skills. Ignoring programming skills is simply not a viable option anymore. Scientific programming is now a discipline in itself.

One aspect we will cover at the end of this module is not directly relevant for Physics code but prepares you to some extent for the third-year computing module and enhances your programming skills. This is about a first glance at optimizing code such that it runs faster. There are plenty more skills required to actually optimize code, see paragraph below, but speeding up Python code is a notoriously re-occurring topic. We tackle this issue for a bit in order to prepare you conceptually for your next step in the skills stream. Changing programming language is often the most efficient way to speed up Python code and you will learn one of the fastest next year, the C-language. However, that comes with quite a bit of additional effort you would not have seen before coming from the Python world. One keyword here is 'compiler'. We need to have a look at that concept. The benefit for this module is that you gain in skill, Python programs that really, really take too much time can be made to run a lot faster and you will be less confused initially should you decide to take the third year module.

There is a lot more to it than what we can touch upon here. Key areas to look up if you are interested would be all the techniques to avoid making mistakes as a programmer. For instance, good programming style, unit tests, documentation, version control, etc. are all topics you should become familiar with if you wish to take your expertise forward.

1.2 Essential reminders on First Year material

For all those having missed or forgotten the first year module, PX150, please download the script from the Moodle site for PX281. It is listed under Resources. Several computational physics concepts beyond the plain Python language are explained in there for the first time and will be used again during this module.

The essential list of key concepts introduced already in PX150 is:

- **Graphics:** PX150 used **import** `pylab as plt`. This is deprecated and we will switch to **import** `matplotlib.pyplot as plt`. Plenty of examples for plots are in chapter 4, Figs. 4–7 with their plot commands. More on graphics later throughout the module.

- **Data fitting:** Likewise in chapter 4, a quite non-trivial example is shown for a dedicated data fitting function in SciPy, called `curve_fit()`, listing 17 in the PX150 script. It is given again below, see Marking criteria Sect. 1.4, with some amendments. A further, simpler, data fitting function from NumPy was used in the assessment questions for week 3 in PX150: `polyfit()`.

- **Data analysis:** some exercises in PX150 already asked for standard deviation, mean value and root mean square calculations. We also discussed the Gaussian or 'Normal' distribution function quite a bit. Lastly, the script also contains an example in chapter 6 for a principal component analysis routine as part of the linear algebra section. All these are good topics to keep in mind when it comes to analysing data in addition to data fitting.

- **Data files:** More often than not you will be required to load data from a file in order to work with it, say for exercises. Possibilities to do that are almost endless. However, we will stick to one of the simplest, the reading of tabular data in comma-separate values files (so called CSV files). The reason for that is that NumPy offers a particularly convenient function to load such data, see PX150 script section 4, page 41ff. The `alldata = np.loadtxt(filename, type)` function returns the content of CSV data files directly as a two-dimensional array, a table (for instance alldata in this example). The rows would be the individual data items (say, N data points = N rows) while the columns are the individual observables contained in the file (say for x, y, z points, find three columns, one for x one for y and one for z values). We would exclusively read numbers from a file hence the type in the command should always be **float**. Just be aware that other possibilities exist and that `loadtxt()` can do quite a bit more if you check the manual. Might come in handy at some point.

- **Random numbers:** The curve fitting example first introduced another important topic in PX150, random numbers, properly explored in chapter 5 of that script. We will mostly use that material for applications rather than explore random numbers in any greater depth themselves. We used the plain **from** `random` **import** `random` random number function to obtain uniformly distributed floating point (numbers with a decimal point) numbers between 0 and 1. More general uniform distributions were obtained with the `random.uniform()` function while random integer numbers were used when simulating games of dice with the `random.randint()` function. Even non-uniformly distributed random numbers such as `random.expovariate()` were demonstrated. All this will be useful for this current module.

- **Integration and Differentiation:** Section 6.2 for PX150 explains both topics in sufficient depth and detail to connect to this module quite easily. The

key items to look up are probably all visible on page 71 of the script including an example on numerical integration, listing 24. Likewise the usefulness of numerical differentiation and integration is mentioned at the end of the script when digital signal processing is mentioned. This will in fact be our main focus for this current module.

- Note that ODE's and dynamics on the computer is a genuinely new topic, not introduced in PX150.

1.3 Jupyter and Assessments

Assignments on this module will use Jupyter notebooks. *Jupyter* is a well documented (follow the link) system for running and distributing Python (and other languages) code embedded with smart text and graphics and so forth. Such files are called **notebooks** and are essentially executable documents.

Anaconda comes with the Jupyter software . Its dashboard (after launching) allows you to browse files and pick notebook files (ending on .ipynb). These are the executable documents. It will display them and you can edit them and execute all or parts of them containing code, here Python code obviously. This makes exchanging code with documentation and whatever else you want very easy as these notebooks can run on any machine with the Jupyter software installed.

Assignments and workshop exercises will consist of Jupyter notebooks. Assignments (see Preface) will all be hosted and run on the **Notable** system. It is recommended to download them (and the teaching notebooks from the Moodle page) once available and work with the notebooks on your computer running Jupyter. Note that this is all independent of Moodle. Copy final solutions to the appropriate cells in the assignment notebook, follow the advice in the header (reset, run all, see that all works as expected) and submit at your convenience.

You can code directly in the notebooks but as programs become longer and more complex, use Spyder to create and run and test your code before inserting it into the notebook. You can then run your code in the notebook and see what happens.

All submissions will run separately on the Notable system after the deadline passed. The automatic tests have to pass for marks and the plots will be assessed and marked manually, in addition to any sanity-checks.

1.4 Marking criteria

The assignments using Jupyter notebooks have the following elements:

- Pass the program tests: Some tests will be (partially) visible in the notebook, most will be hidden. Where applicable, that allows you to test your own code in your own time. Note that these parts of the notebook are 'locked'. You can still go in and disable the tests for your own running of the notebook but for the assessment on Notable, your edits of locked parts would be overwritten and all tests are being conducted anyway.
- If graphics output is required, the plot(s) will be assessed for correctness. All graphics should have labels on axes, and if it is only tenuously related to the axis values, something has to be there or marks will be subtracted.

The marking scheme will be visible at the bottom of the notebook (total marks for tests and graphics).

1.5 Working with the script

This script should contain all the teaching material you would require on this module. It is important to point out that this script in fact contains **more material** than you strictly need to learn. Much of the theory on display merely serves to explain teaching material for anyone with an interest to understand a particular point.

This module is a fully assessed module which means **no exam** assessment. That also means, no book-work questions, no memorizing of derivations or physics arguments. I can therefore deliberately introduce material into the script that will never(!) feature in any assignment hence you can skip it without immediate consequences. Please feel free to do so if you wish.

The assignments are all programming-based. None will require you to do any theoretical work using the content of the script. It is however there if you want to look up arguments and discussions around the topic of interest. The script also contains all the programming examples, explicitly written as source code and resulting pictures from the lectures. If nothing else then you should at least seriously consider these code examples in the script.

2 Data Analysis

Physics data and how to work with it is an essential topic in Physics and yet one that can appear to sit between highly abstract and highly trivial, depending on who you ask. The abstract starts somewhere in the region of casting data into equations in symbolic form, for instance to describe the mathematical procedure of 'fitting', see below. The trivial simply considers data as numbers on a computer, to do with as one pleases.

Let's not get into this but look at the ground rules. Data always has context, without which data really is trivial, just numbers meaning nothing. Context is what in experimental physics is called meta-data. For instance, every data set has time and location of origin (when and where taken), a list of conditions under which the data was obtained (settings like apparatus context, environmental conditions, operator - who took it, etc.). All that is also data and gives meaning to a data set. Meta-data is what physicists typically put into their logbooks.

Data on a computer has become so ubiquitous that it is often forgotten how it got there and hence what data actually consists of in order to process it on a computer. There are two related mechanisms, quite simple, no magic. Either data is put there by hand, i.e. typing, or electronically. In either case, a process called Analog-to-Digital conversion (ADC) has taken place. For the former process, the human is the ADC, for the latter, an electronic circuit has converted a bias value (or sometimes a current value) at a specific time (acquisition time) into a digital value with fixed resolution. The conversion of anything you wish to measure, the observable, to either a voltage or current value is done by a device class called sensor. That is what data in Physics really consists of. Clearly, such conversions require context, a list of complete conditions or otherwise they are simply useless.

That is data but what to do with it? That depends on what you would like to find out and what information the data can in principle deliver. This is the point of departure where data analysis can become abstract. Data never tells you a story, you have to tease abstract information out and compose the story yourself. For simple analysis procedures the two processes are one and the same. The more complex the analysis, the more **justification, calibration and validation** you have to deliver in order to support your 'story'. That is when data analysis becomes its own area of research. On this module, we will not go that far but you are likely to get to that in your Final Year project on a Physics degree.

2.1 Inference from fitting data using Python

Inference is the technical term for teasing out information from data. It involves necessarily statistical methods since you reduce the full content, the entire data set, to something smaller, often just a single number, a summary which ideally characterizes the information in the data as best you can.

2.1.1 Fitting data in general

There are now many different, often related methods for fitting data to model expectations. What does it mean to fit data? Every model with free parameters, numbers, can be compared to data. That comparison can yield better or worse parameter values depending on the comparison. Is it any good or should one choose different parameter values to adapt the model better to data. A fit is the process by which one determines the best parameter values for the model to describe the data.

The classic comparison methods are called **least-squares** and **maximum likelihood** and are described in many textbooks. In words, the least-squares method minimizes the sum of squared differences between data items and model expectations for those data items. The differences take uncertainties of the model expectation (or more often, the data items) into account as weighting factors in the sum.

The maximum likelihood method is different (but with simplifying assumptions can be shown to be identical to least-squares) in that one attempts to maximize the probability of obtaining the data items given the model expectations. Note the subtle but important difference here to the above: explain the model given the data (concept of fitting) as opposed to explain the data given the model (likelihood).

Subtle as that difference in wording might seem, it sparked an ongoing turf war in Physics, originating from Statistics, between 'Bayesians' and 'Frequentists', something that sounds like from a Victorian novel. Bottom line for you is, I will not teach the maximum likelihood method of fitting. It is more general and often more robust than least-squares and leads to the fascinating world of Bayesian statistics but there is no time on this module to go anywhere near that. In practical terms you hardly loose anything at all. All data we will analyse with the method of fitting works perfectly fine using least-squares fitting. Once you got the idea and some practice, exploring the wider world of statistical methods will seem a lot easier.

2.1.2 Least-squares fitting in Python

Let's walk through a very typical and non-trivial example you might encounter many times in various disguises as a scientist.

First make some artificial data to illustrate the point. Try this:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> xvalues = np.linspace(0, 10, 5000)
>>> tempdata = 12.0 * np.ones(5000)
>>> noisy = tempdata + tempdata * 0.2 * np.random.normal(size =
    len(tempdata))
```

Consider some meta-data for this artificial set. Let's say the data represents readings from a temperature sensor in units of degree Celsius as function of time. Taking 5000 data points within 10 seconds would correspond to a sampling rate of 500 Hz. The temperature values seem to jump about quite a bit but always around a central value. If you simply plot the data you don't really learn more than that. We can do better. First the drawing: continue typing at the Python prompt

```
>>> plt.title('Artificial␣temperature␣values␣over␣time', size=12)
>>> plt.xlabel('Time␣[s]', size=12)
>>> plt.ylabel('Temperature␣[C]', size=12)
>>> plt.plot(xvalues, noisy)
>>> plt.show()
```

Now find the mean value and its uncertainty, all in one go. And since we are already at it, let's simultaneously check that the fluctuations are really about a central value since we will have to justify that assumption at some point. There are several ways how to achieve the above but by far the easiest is to simply fit the data to an assumption, a suitable model.

Fitting with NumPy polyfit() The fitting in Python in this case couldn't be easier: use the `np.polyfit()` function. This function will attempt to fit a simple polynomial model (a polynomial of a degree you specify) to data and hand back the optimal result for the polynomial coefficients describing your data. Here, let's fit a polynomial of degree one, i.e. $y = ax + b$ with two coefficients, a and b for slope and intercept of a line, respectively. Why degree one if the data looks flat in y? Shouldn't that just be fitting a constant function?

Well, no, not if you want to get the value of the constant, the mean value and show that there is no trend in the data, no slope. If the best slope is consistent with zero then you have shown flat data and your interpretation of the constant value as a mean value is justified. Remember, you need all the support you can get to tell your story about the data convincingly.

```
>>> bestfit, covariance = np.polyfit(xvalues, noisy, 1, cov=True)
>>> print(bestfit)
```

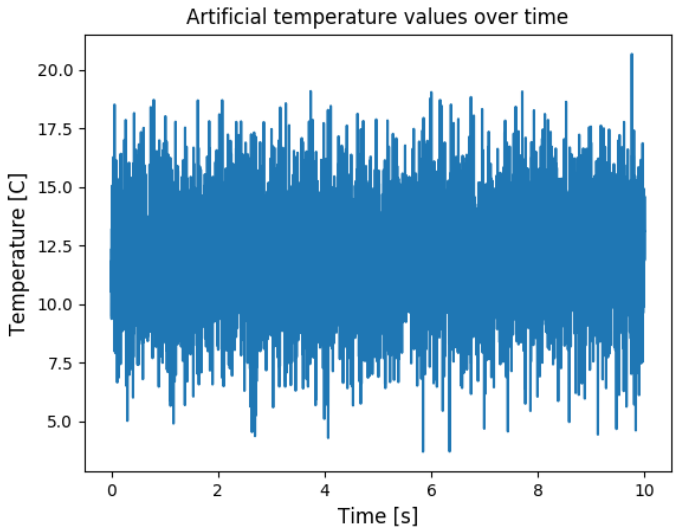

Figure 1: Artificial temperature data with 20% Gaussian fluctuations.

```
[ 5.98921712e-04 1.20453826e+01]
>>> print(covariance)
[[ 0.00013952 -0.00069762]
 [-0.00069762 0.00465126]]
```

What does the above mean? Apart from you getting slightly different values at your machine (these fluctuations are random after all), this says: Your best fit values for a line give a tiny slope, $a = 6.0 \times 10^{-4} \pm \sqrt{1.4 \times 10^{-4}}$ consistent with zero within the fit uncertainty, see below.

Therefore the constant is a mean value at $b = 12.045 \pm \sqrt{4.6 \times 10^{-3}}$. The square root on the uncertainty comes from reading off the diagonal elements of the covariance matrix since these are the best fit value variances. The standard deviation or uncertainty of the best fit values is then the square root of the corresponding variance.

That is the inference on the best fit values, just a mean and its uncertainty, really. However, we can do even more. What is the noise distribution or how 'big' is the noise? The mean is very well determined, with a tiny error as it should be after 5000 measurements. Still, the picture might provoke some feeling of cheating somehow. How can such a noisy data set give such a precise mean temperature reading? In fact, this is a different question to the data with a different answer.

Simple histograms Let's make a histogram plot of the same data, i.e. plot how many temperature readings fall into predetermined intervals of temperature, so called bins. We used a histogram plot once in the first year module if you wish to look that up. So, forget about the time axis, just plot temperature values in a histogram and store the histogram data for later:

```
1 >>> H = plt.hist(noisy, 20)
2 >>> values = H[0]
3 >>> bins = H[1]
4 >>> plt.show()
```

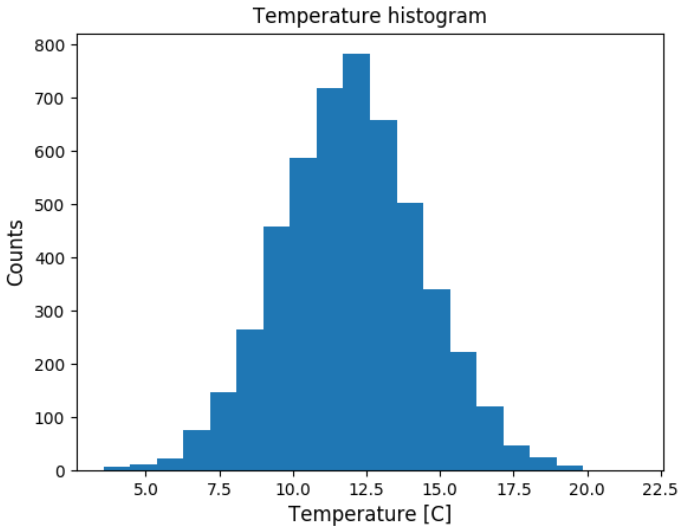

Figure 2: Temperature histogram from noisy data.

That looks quite like a Gaussian distribution, something we have met a few times during the first year module. Of course, this shouldn't come as a surprise since that is what we instructed the computer to use when we made the artificial data in the first place, see line 5 using `np.random.normal()`. Still, nice to see that it worked.

What can you do with a Gaussian distribution? Ideally you can perfectly characterize it by fitting a Gaussian model expectation to it. That will tell you the three parameters of a Gaussian distribution as the best fit values. Let's do that then. No, polyfit will not do that. A Gaussian is not a polynomial. This is the moment when we need a more general fitting function from SciPy: `curve_fit()`.

More general: Fitting with SciPy curve_fit() As you know, using `curve_fit()` requires the definition of a proper fitting function, our model. Let's therefore make a proper Gaussian fit function:

```
>>> # define a Gaussian function for fitting
>>> def gaussian(data, total, position, width):
        ''' Gaussian function, working with numpy arrays
        Parameter: total = integral of the Gaussian
                   position = mean position of the peak
                   width = standard deviation of the Gaussian curve
        '''
        term = -0.5 * ((data - position)**2 / width**2)
        return total / (math.sqrt(2 * math.pi) * width) *
           np.exp(term)
```

Also, the data we wish to hand to the fit function are not the histogram bins. They come as the bin edges, i.e. for 20 values, there are 21 entries in the bins array. These are not the x-values we wish to use but rather the bin centres! For instance, get these from

```
>>> binwidth = bins[1] - bins[0]
>>> bclist = [bins[0] + 0.5 * binwidth] # first element set
>>> for idx in range(1,len(bins)-1):
        bclist.append(bclist[-1] + binwidth) # insert previous + step
>>>> bincenters = np.array(bclist)
```

Now we can fit and plot the result

```
>>> fitParams, fitCovariances = curve_fit(gaussian, bincenters,
   values)
>>> print (fitParams)
[ 4.23937484e+03 1.20123170e+01 2.41733470e+00]
>>> print (fitCovariances)
[[ 2.14003702e+03 -3.45963619e-05 8.13642508e-01]
 [ -3.45963619e-05 9.27682619e-04 -3.64762444e-08]
 [ 8.13642508e-01 -3.64762444e-08 9.27877897e-04]]
>>> H = plt.hist(noisy,20)
>>> plt.plot(bincenters, gaussian(bincenters, fitParams[0],
   fitParams[1], fitParams[2]), 'r-')
>>> plt.show()
```

Look at the best fit values. The mean, again sits at around 12.0 as expected and the standard deviation, the Gaussian width, is at around 2.4. Remember that we fixed that width when creating the artificial data at 20% of the mean, i.e. the 2.4 is

exactly what we would expect. So, that worked fine. The uncertainties on the best fit values are also quite small indeed.

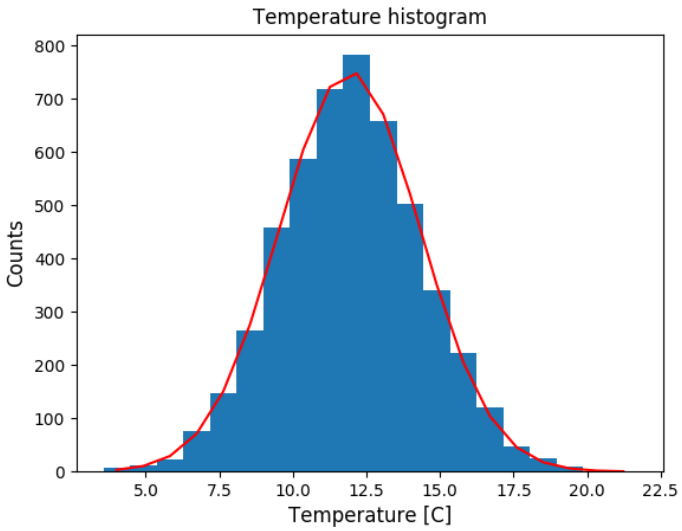

Figure 3: Temperature histogram with Gaussian best fit in red.

Still, there remains at least one unanswered question or rather untested assumptions. Are our models any good? They fit fine but does that mean anything?

That is the moment when probability and statistics enter the analysis. They can't be ignored any further. The questions above have no definite answer other than to a certain degree of confidence and the closer you look at that, not even that is true strictly speaking since there are hidden layers upon layers of assumptions.

Fit quality, goodness of fit Never mind, let's keep it as simple as we can and hope to get away with it. There is one standard qualifier, a number, which can summarize a fit quality, a goodness of fit: the χ^2 value

$$\chi^2 = \sum_{i=1}^{n} \frac{(y_i - \langle y_t(x_i) \rangle)^2}{\sigma_{t,i}^2}.$$

This is the sum of squared differences between data and model expectation. Each difference is divided by the variance of the model expectation. The latter is the fly in the ointment, the challenge one might say. We don't have the variance of the model expectation at every point, no idea what that is and no way of finding out. All we have is a model, a function.

That is the first short-cut then, one typically uses the variance of the data as a substitute and gets away with it, just. If you have the variance of the data points, that is. We didn't even mention any error bars on data points so far. If there were error bars, what does the noise mean or is it all the same? No, it isn't. You can have super precise individual measurements and still collect a noisy data set with fluctuations much larger than the error bar of individual data items. The distribution of data items in a set is not the same as the distribution of individual measurements which you show as an error bar. The devil is in the detail.

In short, noise is an additional contribution to the total uncertainty of a data set. The uncertainty of individual measurements is merely one of many contributors to the total uncertainty.

For the polynomial fit though, what counts is the total uncertainty. All these fluctuations determine the best fit values, here for our line. Error bars on individual data items, temperature values in our example, would only matter if they were the dominant or at least similar contributors to the total uncertainties.

That is the clue from which we can get the correct data-driven uncertainty to be used for the goodness of fit. We do have the variance of the fluctuations! It is the best estimate for the width or standard deviation, squared, of the Gaussian which describes the total data distribution.

The χ^2 value and distribution If you read up on the χ^2 distribution, you note that the expectation value is determined by the number of degrees of freedom of the data set. That is a technical term for something really simple, i.e. number of data items minus number of fit free parameter. In our example, 5000 data points minus 2 fit parameter for the line fit. It is always a good idea to hence calculate the reduced χ^2 distribution value, i.e. $\chi^2/(n-m)$, where n is the number of data points, m the number of free parameter for the fit. The reduced χ^2 has then always the expectation value of 1 and probability statements become quite easy since you only deal with that one, standard distribution.

In Python that looks for example like the following:

```
>>> # chi^2 calculation for line fit
>>> model = best[0] * xvalues + best[1] # line model
>>> sqresiduals = (noisy - model)**2
>>> S = sqresiduals.sum()
>>> print (S / ((len(noisy)-2) * fitParams[2]**2))
0.986215115834
```

Very close to 1, a very good fit quality indeed, as expected. Note that such χ^2 tests can go both ways, neither too bad (big χ^2 values) nor too good (tiny χ^2 values close

to zero) are acceptable. Staying close to 1 is the golden middle.

What about the histogram fit? That is fortunately quite a bit easier to deal with. Histogram data represents a separate data set in itself. As always, the truth is a little more complicated than that but in most cases the statement is fine. The consequence is that for histograms in general, any histogram, we know the dominant uncertainty distribution for each data item.

A data item for histogram data is the entry in a bin. That entry has a well defined uncertainty by the very nature of histogram creation. The idea is that each entry into a bin follows a Poisson distribution which has the nice property that the mean and the variance are identical! The error on the entry in each bin is hence the square root of the entry. Couldn't be any easier. The Poisson distribution is well worth reading about and plays a big role in Physics.

For a fit to a histogram therefore, the χ^2 follows from a subtly different formula:

$$\chi^2 = \sum_{i=1}^{n} \frac{(y_i - \langle y_t(x_i) \rangle)^2}{y_i}.$$

Note the data value in the denominator. That is the result of the Poisson distribution assumption for histograms. In principle, this should be the variance of the model value but again, assuming the data variance represents the model variance (which we don't know), the above formula works in practice.

By eye, see Fig. 3, the Gaussian fit works well. Numerically, we get

```
>>> # chi^2 calculation for Gauss fit
>>> model = gaussian(bincenters, fitParams[0], fitParams[1],
    fitParams[2])
>>> sqresiduals = (values - model)**2 / values
>>> S = sqresiduals.sum()
>>> print (S / (len(bincenters)-3))
1.4595128961621007
```

which is still a very good fit. Let's check with Python:

```
>>> from scipy.stats import chi2
>>> print (chi2.sf(S, len(bincenters)-3))
0.0990039177018
```

Note that SciPy deals with the full χ^2 distribution hence requires the degrees of freedom as input and the non-reduced value. The result is also worth noting. The roughly 0.1 output means: you can reject the hypothesis of a failed fit model with 10% probability. Not great it seems, maybe our fit wasn't that good after all.

That is until you have done a good number of fits and χ^2 tests and realize the variability of test results. The 90% failure means effectively nothing, is as good as any successful fit. Just change the number of bins and you'll see that results start shifting noticeably. Then realize that the bin number is arbitrary by definition and your respect for the test indicator diminishes. The χ^2 test is a nice indicator and used consistently can give you some quantification of the quality of your data description but not more.

2.2 Event data in Physics

Now let's use fitting in Physics for something more intricate, i.e. event data in Physics. What could that possibly be? In its shortest form, event data consists of a single or several sampled data items obtained in a fixed time interval. Sampled data items mean a set of individual measurements acquired in time.

Complex Physics experiments have almost always more than one measurement device. Particle Physics experiments are typical examples, having many detectors that all work together to produce one measurement in typically a fixed time frame, called an event. Imagine lots of measurements going off in a burst of activity and then all stops until the next burst. That data obtained in such a collection of sampled data would form an event.

A lot of different and complex data can be hosted in an event and each needs individual analysis. Many such events then form a data set and analysis would proceed in a chain: process the sampled data, summarize the individual event data one by one, summarize the event data set as a whole and try to learn some physics from it. Other technical terms often mentioned in this context are **data reduction** and **reconstruction** which both describe the first stage of the process. The term **analysis** in that case is then meant to describe the final stage process.

Note that there is a clear story line here from making simple averages on a single sampled observable to the above. It is fundamentally always the same except that you add processes to the story, make it longer with many more opportunities for making mistakes.

Say you gather a full spectrum with a spectrometer and take several pictures of a star and its neighbourhood with a camera and some filters and all that for several nights over the course of a year. You will wish to know how your spectrometer works at any moment in time and that the filters stay as expected and the camera works as expected. That's not all though. You will also want to be certain that the spectrum you measure so nicely still belongs to the star you take a picture of, i.e. that several devices work well together as expected and remain to do so.

Note the term 'as expected' above. That results from a process called **calibration**, a measurement prepared and controlled by the experimenter under special circumstances which needs data analysis. Your devices will all change in time, they all do. That is fine as long as you know about the change to the precision required. That is calibration. Calibration also serves to validate your analysis. If you know what you put in and have a well justified expectation for the outcome of a thus prepared measurement then you can validate your analysis. You know what the result should be.

Let's cut to the chase then and set up a more complex data analysis chain with event data without overdoing it.

2.2.1 Particle Dark Matter search example

Take a simple dark matter detection experiment. Looking for particle dark matter often involves the setting up of several identical solid-state detectors or a single big liquid noble gas detector in an underground laboratory, well shielded from any external influences. These machines would then nominally measure the energy deposition in their volume due to particle dark matter amongst many other forms of energy depositions, radioactivity for instance. Those unwanted signals are called background.

Of course, they don't measure the energy observable at all but an electronic signal, voltage as function of time. With a lot of work and ingenuity one can analyse the waveform, the detector pulse, and relate such analysis results to the energy observable. That is event analysis in Physics, so let's try that without getting into the mess of having to consider many different detectors working together to extract many more observables.

A waveform or pulse represents sampled data, many individual voltage measurements in time, just like the picture on an oscilloscope screen. Each detector pulse is either part of an event or like in this example, forms the event. As discussed at the beginning of the chapter, an event representing data should have meta-data for context. At the very least, events should also contain a time-stamp to give the x-axis, the time-axis a meaning, an origin. If one isn't concerned with absolute times then that can be abandoned. In that case one would only wish to record as a bare minimum the start and end of event collection, often called a data 'run'.

Well, with technical terms out of the way, how does a pulse actually look like and what can you do with it? Considering the teaching material in this chapter, one can fit the pulse, of course. That requires a model though and that in turn requires that you know your machine rather well. Let's have a look at an artificial dark matter detector pulse, see Fig. 4.

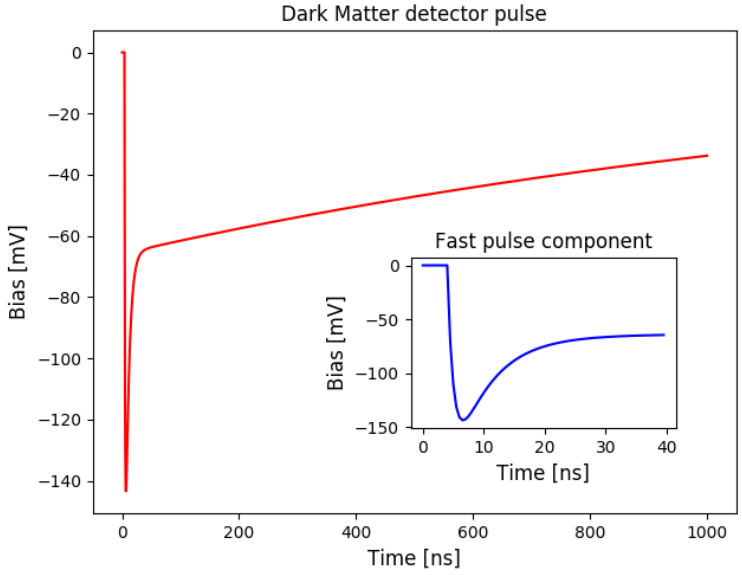

Figure 4: Artificial dark matter detector pulse. Note the characteristic property of the light pulse model featuring two different decay times with the insert figure showing a short time interval demonstrating the short decay time more clearly.

The model for our little experiment shall be a big tank of liquid noble gas, say Xenon (this is close to existing experiments currently running and planned for the future). Such a tank would feature many light detectors capturing light emitted from the liquid after energy deposition by any particle, be it electrons, photons, alpha-particles or for that matter, unknown dark matter particles. The pulse would represent a light pulse measurement from a single light detector. Most if not all of the many light detectors would observe some light from the same energy deposition, i.e. there should be many of such pulses in a single event. For now, let's stick to a single pulse.

The model for such a light pulse is

$$V(t) = A\left[\exp\left(-\frac{t-t_o}{\tau_{rise}}\right) - \left(\frac{R}{R+1}\exp\left(-\frac{t-t_o}{\tau_{short}}\right) + \frac{1}{R+1}\exp\left(-\frac{t-t_o}{\tau_{long}}\right)\right)\right] + B$$

with $R = \frac{I_{short}}{I_{long}}$ the ratio of light intensity of short decay time emissions to long decay time emissions. The reason for that is that liquid noble gas light emitters have two processes that produce light emission but with different intensities. The characteristic times for light emission are described by τ_{short} and τ_{long} respectively while τ_{rise} describes the characteristic time of light arriving at the detector at all. The term A is simply the scale factor, related to the amplitude of the pulse while t_o is the time of onset, the start of the pulse. Before onset there should be only noise

in the sampled data around a mean value called the baseline, here parameter B. In total that gives us 7 free parameters to describe the pulse.

That is quite an ambitious model to fit. Fitting anything to data with more than 3 to 4 parameters is difficult! What it means is that you will need very good initial guesses for the values of the parameters. Otherwise fits will fail! This ambitious example is chosen for that very purpose. Fits fail in Physics, more often than not. You wouldn't know from studying Physics or reading research articles. It is one of the routine challenges everyone faces in research. So now you know.

Of course, we will try not to suffer from failed fits too often. Given a good initial guess, all will be well. Here for instance, let's make that pulse and fit it, see listing 1.

Listing 1: Light pulse model calculation and plot.

```
'''
Detector pulses script
'''
import math
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

def plotpulse(xval,yval):
    ''' plot with insert '''
    fig = plt.figure()
    axis1 = fig.add_axes([0.12, 0.1, 0.85, 0.85]) # main axes
    axis2 = fig.add_axes([0.54, 0.25, 0.35, 0.3]) # inset axes
    axis1.plot(xval, yval, 'r-')
    axis1.set_title('Dark␣Matter␣detector␣pulse', size=12)
    axis1.set_xlabel('Time␣[ns]', size=12)
    axis1.set_ylabel('Bias␣[mV]', size=12)

    axis2.plot(xval[:80], yval[:80], 'b-') # zoom in to 40 ns
       maximum
    axis2.set_title('Fast␣pulse␣component', size=12)
    axis2.set_xlabel('Time␣[ns]', size=12)
    axis2.set_ylabel('Bias␣[mV]', size=12)
    plt.show()
    return


```

```
27 def pulse(t, scale, onset, baseline, ratio, taurise, taushort,
       taulong):
28     ''' pulse model function to work with numpy.'''
29     denominator = ratio + 1
30     temp1 = ratio / (denominator) * np.exp(-(t - onset) / taushort)
31     temp2 = 1.0 / (denominator) * np.exp(-(t - onset) / taulong)
32     decay = temp1 + temp2
33     pulse = scale * (np.exp(-(t - onset) / taurise) - decay) +
           baseline
34     pulse[np.where(t < onset)] = 0.0 # not defined before onset
           time, set 0
35     return pulse
36
37 # make a pulse, consider times in nano seconds [ns]
38 timevalues = np.linspace(0, 1000, 2001) # 0.5 unit step size
39 taurise = 1.0 # fast sensor rise time
40 taushort = 6.0 # realistic short decay time for Xenon
41 taulong = 1500.0 # realistic decay time for Xenon 1500 ns
42 scale = 210.0 # some scale factor giving reasonable values
43 onset = 4.0 # start on step 4, here 2 ns in the sample
44 baseline = 0.0 # no baseline offset
45 ratio = 2.2 # more in short intensity than long
46 pp = pulse(timevalues, scale, onset, baseline, ratio, taurise,
       taushort, taulong)
47
48 plotpulse(timevalues, pp)
49
50 # fit the pulse, try similar values to construction
51 initguess = (20.0, 4.0, 0.0, 2.0, 1.0, 5.0, 1000.0)
52 fitParams, fitCovariances = curve_fit(pulse, timevalues, pp, p0 =
       initguess)
53 print (fitParams)
54 print (fitCovariances)
```

```
1 >>> import trypulses
2 [ 2.09999998e+02 4.00000000e+00 -1.13898171e-06 2.20000003e+00
3    1.00000000e+00 6.00000001e+00 1.49999996e+03]
4 [[ 6.24248339e-15 -1.30892213e-17 -3.94414232e-16 1.15151406e-16
5     4.80398422e-17 -2.04441694e-16 -1.43464324e-14]
6  [ -1.30892213e-17 2.45654054e-19 -2.11997336e-18 -8.46279325e-20
7    -2.59123366e-19 -3.12977054e-20 -6.94698474e-17]
```

```
[ -3.94414232e-16 -2.11997336e-18 1.17085759e-16 -1.22210192e-17
  -8.34733594e-19 2.26302630e-17 3.96188638e-15]
[ 1.15151406e-16 -8.46279325e-20 -1.22210192e-17 2.39517301e-18
   7.69883889e-19 -4.27584317e-18 -4.21777098e-16]
[ 4.80398422e-17 -2.59123366e-19 -8.34733594e-19 7.69883889e-19
   5.24449520e-19 -1.18471068e-18 -3.51120352e-17]
[ -2.04441694e-16 -3.12977054e-20 2.26302630e-17 -4.27584317e-18
  -1.18471068e-18 8.43926268e-18 8.11853837e-16]
[ -1.43464324e-14 -6.94698474e-17 3.96188638e-15 -4.21777098e-16
  -3.51120352e-17 8.11853837e-16 1.46021415e-13]]
```

The fit results nicely reproduce the construction as they should. The errors are also very small for this ideal pulse. Real pulses will have noise added and that should increase the fit errors drastically even if the fit results are good.

The secret to difficult fits are the initial guesses as mentioned above. Any parameter value you can keep close to the anticipated truth will help. Have a close look at listing 1, line 51. The onset value and baseline are numbers you can find out quite well without fitting, simply by the way you set up the measurement. Putting in a good value to the guess is therefore justified.

Similarly for the rise time constant. Remember that you calibrated your apparatus many times before and hence know the rise time well enough to give a decent guess for the fit. It is after all mostly apparatus specific.

The physics in the model will also give you a good idea about the characteristic decay time constants. Somewhere in the single nano seconds for the fast time and micro seconds for the slow is a good guess you can get from calibrations and previous measurements. So, educated guesses are not magic, just good work and they make all the difference between good or failed fits.

The scale and intensity ratio are of course undetermined since you can't know how much light arrives and in which proportion for which emission mechanism. That is the whole point of making the measurement in the first place. The pulse amplitude relates to the energy deposition, something to be measured. Likewise the ratio parameter links to the physics causing the energy deposition. That's where things start to become interesting and why this dark matter detector is built like this.

Have a look at Fig. 5. The caption shows the details but the bottom line is that the free parameter we called ratio allows to discriminate between what is called nuclear recoil events from electron recoil events. Each population of events has quite different ratios if you compare the top figure with the bottom figure.

This is important since most background in dark matter experiments is made

of electron recoil events while the signal would show up as nuclear recoil events. The discrimination therefore helps a lot to suppress background.

I quote the figure on purpose since making such a picture would simply be too much work at this point even just making idealized pulses and fitting them. We would need to make thousands of pulses, fit each of them, account for the energy dependence of parameters and apparatus response and then collect all fit results to create a plot like Fig. 5 from our artificial experiment. I rather show you the real thing and hope that you get the idea of how to make that analysis in principle.

Next, you will learn what else can and is being done to waveforms other than fitting. Those will be computational processes in everyday use in Physics experiments everywhere but even more prominently outside academia. Well worth knowing about.

2.3 Summary

Data analysis is a ubiquitous topic in physics and science in general. The focus in this lecture is on the process called **fitting** which means comparing data to model expectations in a precisely defined mathematical procedure. We used **least-squares** fitting as the chosen method for that comparison.

The second important topic in data analysis introduced in this lecture is the process of analysis, i.e. the requirements to form a full analysis.

Every analysis consists of a chain of processes in order to support a result. Every step has to have a justification, data has to have meta-data and a calibration which allows to have the best possible understanding of data uncertainties and every process has to be validated.

Each step in the chain contains these three ingredients and an analysis consists of the full chain. A first glimpse of an analysis requiring such a chain was discussed in form of an event analysis. Raw data first has to be processed before final results can be obtained, where these processing steps are known as **data reduction** or **reconstruction**. These are typical examples for elements in a full analysis chain.

What we did not cover in this lecture is a discussion of the required statistics inherent in analysing data, whether the analysis includes model expectations or not. Bayesian statistics would be the main topic left out here. That discussion would feature the **maximum likelihood** method as a prominent ingredient. Discussing the latter in isolation without the former has caused many misunderstandings among physicists hence shall not be repeated here.

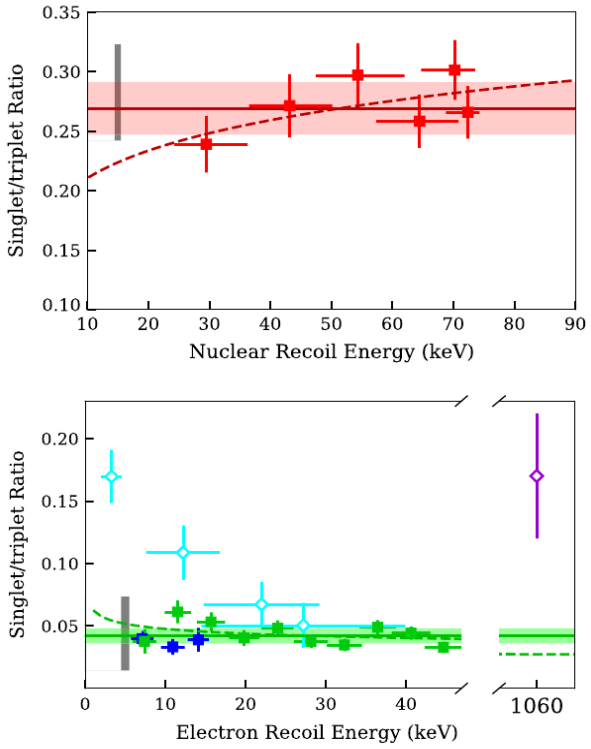

FIG. 7. Singlet/triplet ratio ($C_1\tau_1/C_3\tau_3$) measured for nuclear recoils (top) and electron recoils (bottom) using LUX calibration data. Only statistical uncertainties of the data are shown. Calibration sources are DD neutrons (red), tritium (blue), and ^{14}C (green). Measurements in different energy bins are shown by the square points, and the best fit constant model by the solid line. The shaded region indicates the statistical uncertainty on the constant model. The shaded gray bar indicated the systematic uncertainty of the constant model. A power law is also fit to the data and is presented by the dashed line. We also show measurements of the ER singlet/triplet ratio at zero field from Ref. [11] (cyan diamonds), and a measurement using a ^{207}Bi internal conversion source at 4 kV/cm from Ref. [8] (purple diamond). In Ref. [11], the singlet fraction (denoted F_1) is given rather than the singlet/triplet ratio. For direct comparison to this work we make the conversion $(C_1\tau_1)/(C_3\tau_3) = F_1/(1+F_1)$.

Figure 5: From D.S. Akerib et al, Phys. Rev. D97, 112002 (2018).

3 Integration and Differentiation in Physics

It was mentioned previously that all data on a computer is sampled data, coming in almost all cases from something called an Analog-to-Digital converter in order to be digestible by a computer. The question for this chapter is: what does that sampling thing have to do with Integration and Differentiation in computational physics?

The technical term sampling is the key to answering that question. Motivating integration and differentiation on the computer usually involves learning about numerical algorithms or simply methods on **how to integrate** (and differentiate but that is so easy, it is mostly left out). Strictly speaking, algorithms are what you are meant to learn in the year 3 module, not here!

Once you ditch the textbooks on integration algorithms, what are the interesting integrals in Physics? You look into that and quickly realize that you solve differential equations (by integration) since that is where all the Physics is. We got a separate chapter on ODE's, so not here.

What is left is, amongst other interesting items but I had to make a choice, a profoundly important topic in experimental physics which is taught and talked about far too rarely in undergraduate modules, mostly because you would be expected to pick that up easily during a PhD or in a later job if required. Yet, it fits the bill for this section of the module rather nicely hence you will learn about digital signal processing (DSP).

For all those who expect numerical integration and differentiation at this point, let me remind you again to check section 1.2. Numerical integration as well as differentiation using SciPy and suitable formulae is something we have done previously in the first-year module PX150. No need to repeat those lessons here.

3.1 What integrals?

What does signal processing have to do with integrals and with Physics for that matter? For the latter point, your data arrives in electronic format ready for processing in order to do Physics hence you better know what to do with that data. Even if that step is done for you, in order to pass the usual scrutiny on new Physics results, in your thesis for instance, you should be in a position to understand what that service has done to your data before you got it.

For the former part of the question the technical term of sampling provides a first answer. Here is a first incling of what is going on and it works like this: do you

remember Fourier series from last year?

$$a_n = \frac{1}{T} \int_0^T x(t)\, \exp\left(-i\,\omega_n\, t\right)\, dt$$

with the reconstruction (the inverse)

$$x(t) = \sum_n a_n\, \exp\left(i\,\omega_n\, t\right)$$

there is an integral already. Not the one we are interested in you will be pleased to know but it has started. Have a closer look at the integral. It calculates a discrete value, the a_n, from the entire function $x(t)$. This process does not(!) sample the function $x(t)$ at a single point to get a_n. Sampling takes the function, this signal or waveform to be more precise, at single points and one tries to deduce the function from those single points. Also, the Fourier series representation of functions assumes that they are periodic. This is another disturbing condition when dealing with something arbitrary like data.

It would be better in the argument above using the Fourier series if we could map the signal, $x(t)$, not onto a single point (coefficient a_n) at a frequency, the ω_n, but onto a complete function, say $X(\omega)$, for all frequencies.

That way we could relate individual sample points in time to hopefully limited structures in frequency such that a full sample, i.e. lot's of points in time correspond to something similar in frequency and we can learn something about sampling from that result.

You can guess that after this long introduction, such a mapping does exist. You just haven't met it yet. It will be new to you, **the Fourier transformation**.

$$X(\omega) = \int_{-\infty}^{\infty} x(t)\, \exp\left(\,-i\,\omega\, t\right)\, dt$$

with the inverse

$$x(t) = \frac{1}{2\,\pi} \int_{-\infty}^{\infty} X(\omega)\, \exp\left(\,i\,\omega\, t\right)\, d\omega$$

You will learn about the Fourier transformation properly at the end of the second year in your maths module. Suffices to say here that you can derive it from the Fourier series by the limiting process $\lim_{T\to\infty} a_n$ for the Fourier series coefficients, i.e. let the period tend to infinity and hence frequency intervals to zero.

Here is the crucial part of this lecture then: the **Discrete Fourier transform (DFT)**. The integrals above can not be solved analytically in most cases. Worst of all, the function, $x(t)$, might not even be known analytically, for instance because

it consists merely of sampled data points. Where is the formula for that? This challenge can only be solved numerically.

Fortunately, the solution is rather simple when re-writing the Fourier transformation as a discrete sum of terms for either the forward as well as the inverse transformation. Let

$$c_k = \sum_{n=0}^{N-1} y_n \exp\left(-i\,\frac{2\,\pi\,k\,n}{N}\right)$$

be the discrete set of numbers mapping on the Fourier transformed function, $X(\omega)$ while the inverse

$$y_n = \frac{1}{N}\sum_{k=0}^{N-1} c_k \exp\left(i\,\frac{2\,\pi\,k\,n}{N}\right)$$

returns the sampled values, y_n, if there are N sampled points in total. This might look like an approximation to the Fourier transformation formulae from above but for sampled data this is in fact an exact solution!

The price you pay for the change from the exact Fourier transformation to the DFT is a consequence of the initial sampling, not the DFT method. Sampling after all means to take only some function evaluations at discrete points, disregarding the infinite number of other function values. That is a loss of information right at the start and information loss always has a price which we look at in a moment, see the sampling theorem discussion below.

For now, let's put the DFT into practice, pedestrian style. Make a Python function for later use:

```
>>> import numpy as np
>>> from cmath import exp, pi
>>> def dft(y):
        ''' simple discrete fourier transform '''
        N = len(y)
        c = np.zeros(N//2 + 1, complex)
        for k in range(N//2 + 1):
            for n in range(N):
                c[k] += y[n] * exp(-2j * pi * k * n / N)
        return c
```

The odd expression 'N//2' still requires a little explanation. Even if you know the Python operator '//' for integer division, why dividing the number of samples by two? The DFT deals with real as well as complex number functions. It is quite a general result. Our data, y_n, represents real number though and real numbers only. That has a practical consequence for all computations: Consider the value of c_k for

some k that is less than N but greater than $N/2$, with $k = N - r$ with $1 \leq r < N/2$.

$$c_{N-r} = \sum_{n=0}^{N-1} y_n \exp\left(-i\frac{2\,\pi\,(N-r)\,n}{N}\right) = \sum_{n=0}^{N-1} y_n \exp\left(-i\,2\,\pi\,n\right) \exp\left(i\frac{2\,\pi\,r\,n}{N}\right)$$

$$= \sum_{n=0}^{N-1} y_n \exp\left(i\frac{2\,\pi\,r\,n}{N}\right) = c_r^*$$

using $\exp\left(-i\,2\,\pi\,n\right) = 1$ for all integer n and the y_n all real. In other words, the DFT on real values gives you a symmetric reply for half the number of sampled values N where the second half consists of a copy of the first half but with complex values conjugated. We are interested only in the real values (the complex values display the phase if you are privately interested - of no concern on this module) hence we only look at half the response from the DFT.

Here is an example then on how that can be put into practice. Make some artificial data and run the DFT function over it.

```
>>> import matplotlib.pyplot as plt
>>> fsampling = 64 # sampling frequency
>>> Nsamples = 256 # N samples at sampling frequency
>>> # data time series
>>> duration = float(Nsamples) / float(fsampling)
>>> time = np.arange(0.0, duration, 1 / float(fsampling))
>>> # 10 Hz signal + smaller 22 Hz
>>> signal = np.sin(2 * np.pi * 10.0 * time) + 0.2 * np.sin(2 *
    np.pi * 22.0 * time)
>>> c = dft(signal)
>>> freq = np.arange(0.0, fsampling/2.0, float(fsampling) /
    float(Nsamples))
>>> plt.title('Frequency space', size=12)
>>> plt.xlabel('Frequency [Hz]', size=12)
>>> plt.ylabel('Amplitude', size=12)
>>> plt.plot(freq, abs(c[:128]))
>>> plt.show()
```

Note how manual the process is, i.e. you have to think about the right length of arrays, the timings and the frequencies. That is no bad thing but can become tedious after a while. Anyway, the picture of the Fourier transformed signal can be inspected in Fig. 6.

We can Fourier transform now, solving very complicated and very important integrals in Physics in a spectacularly simple way, at least when it comes to sampled

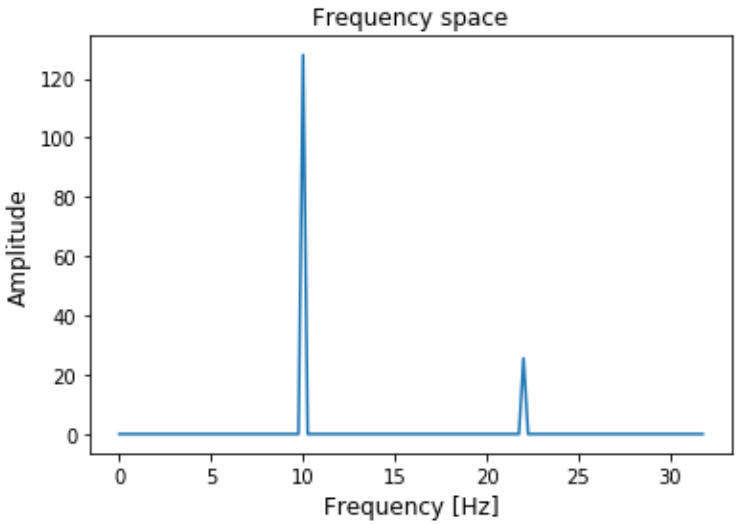

Figure 6: Artificial signal with two frequencies in Fourier space.

data as the signal 'function'. What about the price to pay for sampling as mentioned earlier in the text?

3.2 Sampling Theorem in practice

The **sampling theorem** says that a continuous (analog) signal can be represented in samples at discrete points in time and be recovered back if the sampling frequency, f_s, is greater than or equal twice the highest frequency component contained in the continuous signal.

So, now we got interesting integrals, let's mop up the remaining technical terms in signal processing before we can get going on some computational physics.

Consider a single sampling point $x(t_0) = x(t) \times \delta(t - t_0)$, where $\delta(t)$ shall be an impulse function that is

$$\delta\,(t) = 1 \quad \forall\,[-1/2 \leq t \leq 1/2]$$

and zero elsewhere, i.e. something to represent chopping a continuous signal into discrete, finite width sampling points. That representation allows to consider purely the Fourier transform of $\delta(t)$ to learn about sampling. Plugging in gives

$$X(\omega) = \int_{-1/2}^{1/2} \exp\,(\,-i\,\omega\,t)\;dt = -\,\frac{1}{i\,\omega}\;(\exp\,(\,-i\,\omega\,1/2) - \exp\,(\,i\,\omega\,1/2)) = \frac{\sin(\pi\,f)}{\pi\,f}$$

and $\omega = 2\,\pi\,f$. Let's take a look, see Fig. 7.

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> impulse = np.zeros(512)
>>> for i in range(256):
        impulse[128+i] = 1
>>> X = np.fft.rfft(impulse) # Fourier transformation in NumPy
>>> n = impulse.size
>>> freq = np.fft.rfftfreq(n) # corresponding frequencies
>>> plt.title('Frequency space', size=12)
>>> plt.xlabel('normalized Frequency [$\pi$/T]', size=12)
>>> plt.ylabel('Amplitude', size=12)
>>> plt.plot(freq, X.real)
>>> plt.show()
```

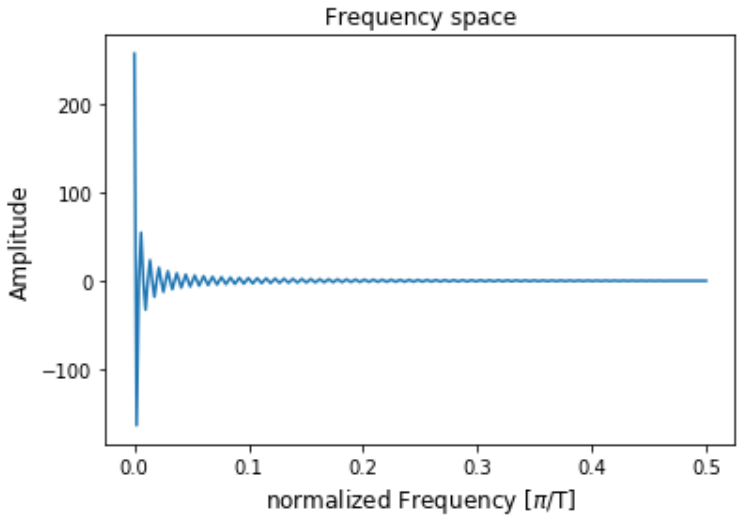

Figure 7: Impulse response in Fourier space. Check the code to learn about Fourier transforming with NumPy.

The recovery or rather reconstruction of the continuous signal after discrete sampling is conditioned in the sampling theorem. It says the sampling frequency f_s must be greater than or equal twice the highest frequency component in the signal. Phrased equivalently, one could state the condition as requiring for the discrete time intervals between measurements to be less than or equal to half the inverse sampling frequency $T \leq 1/(2\,f_s)$.

The reconstruction of the original signal then proceeds according to the sam-

pling theorem as

$$x(t) = \sum_{n=-\infty}^{\infty} x(n\,T)\,\frac{\sin(\pi\,(t/T - n))}{\pi\,(t/T - n)}$$

since our sampled signal consists of a sum of many of the impulses discussed above, one per measurement, which each has a Fourier transform in the form of this $\sin(t)/t$ function, see Fig. 7. The limiting sampling frequency then arises from the condition that these oscillating functions don't overlap too much such that reconstruction is still possible. If they do, violating the sampling limit, then that is called **aliasing** and would be a mistake.

In addition to aliasing, i.e. reconstructing something that can't fundamentally be reconstructed anymore, there is the more practical question of **resolution**. What are the conditions for resolving two features in frequency space? Again, the sampling theorem limit comes into play.

If you collect a thousand measurements, $N = 1000$ then half of those will give you distinct Fourier transform information since the transform is symmetric around zero. If the sampling rate to acquire those thousand points is 1 kHz, i.e. we measure for one second then the sampling theorem limit (also called Nyquist frequency) states that no frequency content higher than 500 Hz should be contained (otherwise suffer from aliasing). Putting those numbers together gives then a single frequency response every $500/500 = 1$ Hz / frequency interval.

How to improve the resolution? Increase the sampling time, i.e. take more measurements. The main learning outcome from these considerations is that the Fourier transformation shows the **duality** of time and frequency. If you want high frequency resolution, best knowledge gain on frequency, then your time resolution, your knowledge about the timing of the measurement becomes bad since you have to measure for longer. Conversely, if you measure for a shorter duration, your frequency resolution suffers. Let's see all this more visually.

Listing 2: Script demonstrating frequency resolution limits.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  '''
5  Script demonstrating Fourier frequency resolution limits
6  and dependencies on sampling conditions.
7  '''
8
9  def quickplot(freq, sigfft, duration):
10     ''' simple plotting of Fourier pictures '''
```

```
11      n = 2 * freq.size / duration
12      plt.title('Frequency␣space', size=12)
13      plt.xlabel('Frequency␣[Hz]', size=12)
14      plt.ylabel('Amplitude', size=12)
15      plt.plot(n * freq, sigfft.real, '-o')
16      plt.show()
17
18  # Nsamples / fsampling = sampling time
19  Nsamples = 64 # N samples at sampling frequency
20  fsampling = 64 # sampling frequency
21  fsignal = 10 # signal frequency
22
23  # data time series
24  duration = float(Nsamples) / float(fsampling)
25  time = np.arange(0.0, duration, 1 / float(fsampling))
26
27  # 2 Hz signal frequency difference
28  x = np.cos(2 * np.pi * fsignal * time) + np.cos(2 * np.pi *
        (fsignal+2) * time)
29  X = np.fft.rfft(x, Nsamples)
30  n = x.size
31  freq = np.fft.rfftfreq(n)
32  quickplot(freq, X, duration)
33
34  # 1 Hz signal frequency difference
35  x = np.cos(2 * np.pi * fsignal * time) + np.cos(2 * np.pi *
        (fsignal+1) * time)
36  X = np.fft.rfft(x, Nsamples)
37  n = x.size
38  freq = np.fft.rfftfreq(n)
39  quickplot(freq, X, duration)
40
41  # 1 Hz signal frequency difference, 2 s sampling
42  Nsamples *= 2
43  duration = float(Nsamples) / float(fsampling)
44  time = np.arange(0.0, duration, 1 / float(fsampling))
45  x = np.cos(2 * np.pi * fsignal * time) + np.cos(2 * np.pi *
        (fsignal+1) * time)
46  n = x.size
47  X = np.fft.rfft(x, Nsamples)
48  freq = np.fft.rfftfreq(n)
```

```
49 quickplot(freq, X, duration)
```

The script in listing 2 produces three figures. Fig. 8 shows a measurement for 1 s duration of 64 samples hence a sampling frequency of 64 Hz. The two signal frequencies are at 10 Hz and 12 Hz. You can just about separate the two frequencies with a difference of 2 Hz.

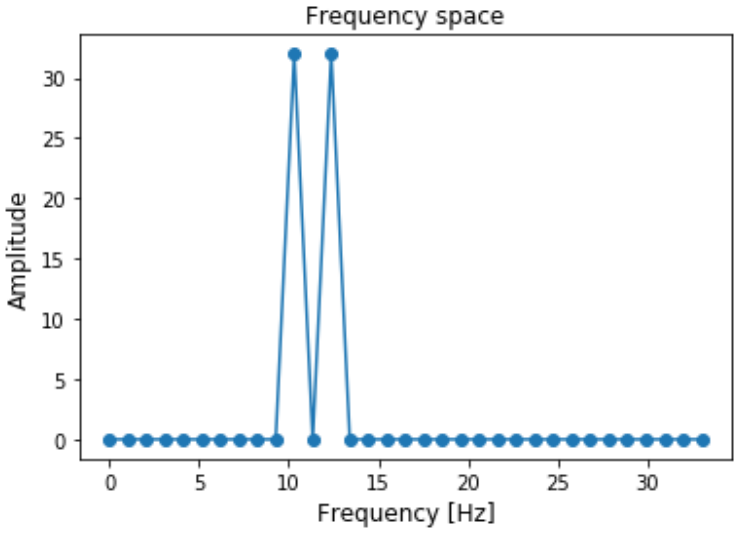

Figure 8: Separating a 2 Hz difference signal after a 1 s measurement at $f_s = 64$ Hz.

If the signals are closer together in frequency, here by just 1 Hz, then they can not be recovered anymore, see Fig. 9.

That tallys with the discussion above, 32 samples give unique Fourier transform frequency information, taken at a sampling frequency of 64 Hz of which half at best can be resolved according to the Nyquist limit. You hence get intervals of 1 Hz in frequency space and that is what is on the pictures. Two peaks at a distance of 1 Hz can therefore not be distinguished anymore while a difference of 2 Hz can.

For Fig. 10 we increased the signal duration to 2 s, take more samples and frequency resolution is recovered for a 1 Hz difference.

3.3 Convolution and filtering

This section covers the most important concept for digital signal processing. While Fourier transforming and sampling are fundamental methods and concepts to it all, convolution is the most important concept for signal processing as such.

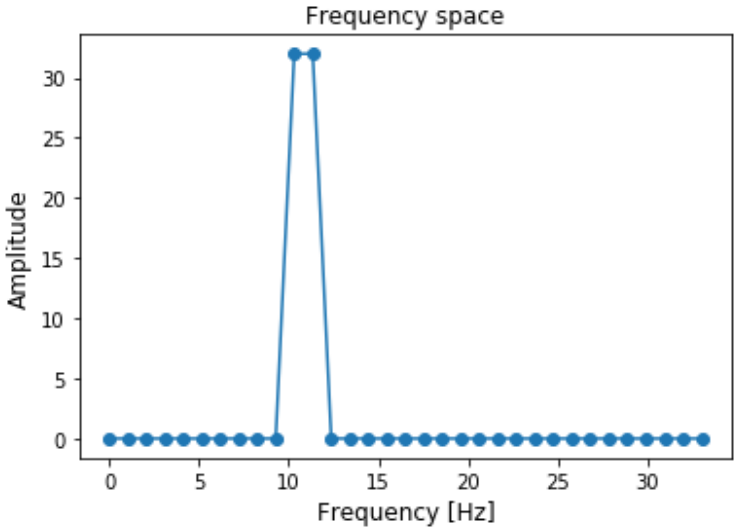

Figure 9: Separating a 1 Hz difference signal after a 1 s measurement at $f_s = 64$ Hz.

Fourier transforming a signal gives you a **frequency spectrum** and that alone is already an excellent diagnostic tool but it doesn't represent signal processing. You will wish to do something useful with that signal, transformed or not.

3.3.1 Convolution operation

In comes **convolution** which is at the heart of it simply a mathematical operation like addition or multiplication, just that it works on discrete sets rather than single numbers. The convolution of two sets of numbers gives a new, third set of numbers. That's it. The procedure works according to

$$y[n] = h[n] * x[n] = \sum_{k=-\infty}^{\infty} h[k]\, x[n-k]$$

where that bracket notation should remind you of number arrays like in Python (index notation is also used) and the asterisk $*$ denotes the convolution operation. If you combine a number array $h[n]$ with another array $x[n]$ like in that formula above then $y[n]$ is the result of the convolution of h and x.

Mathematically, convolution is rather an integral transform with many diverse applications of which signal processing is just one. However, again we are effectively solving difficult integrals even though it does not appear to be so as long as we work with data in physics.

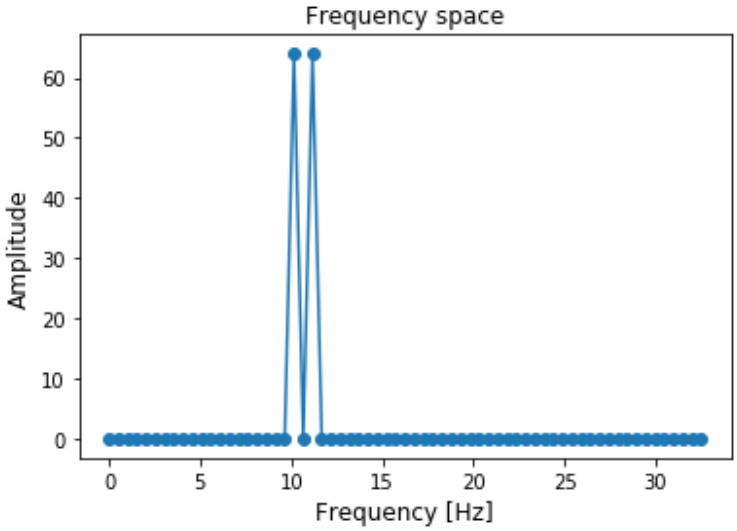

Figure 10: Separating a 1 Hz difference signal after a 2 s measurement at $f_s = 64$ Hz.

This should be easy to program:

```
>>> import numpy as np
>>> def convolution(x, h):
        ''' simple convolution operation'''
        y=[]
        # equal array lengths for indexing
        xextension = np.zeros(len(h)) # zero padding
        xextend = np.concatenate([x, xextension])
        hextension = np.zeros(len(x))
        hextend = np.concatenate([h, hextension])
        # convolution loops
        for n in range(len(xextend)):
            entry = 0
            for k in range(len(xextend)):
                if (n - k) >= 0: # only for overlap
                    entry += hextend[k] * xextend[n - k]
            y.append(entry)
        return np.array(y)
>>> x=np.array([0,0,1,1,1,0,0]) # square pulse on
>>> h=np.array([1,1,1]) # square kernel gives triangle
>>> print (convolution(x, h))
```

```
[ 0. 0. 1. 2. 3. 2. 1. 0. 0. 0.]
```

or more conveniently and faster

```
>>> import numpy as np
>>> from scipy import signal
>>> sig = np.repeat([0.0,1.0,0.0],3)
>>> window = np.array([1,1,1])
>>> print (signal.convolve(sig, window))
[ 0. 0. 0. 1. 2. 3. 2. 1. 0. 0. 0.]
```

Well, that seems convenient from a computational point of view. The convolution of two square pulses is a triangle pulse. This example is inspired by the Wikipedia page on convolution, from one of its impressive animations. Just one more property to learn and applications can commence. The final missing piece in this story is the connection between Fourier transforms and convolution. That turns out to be again rather nice.

The **convolution theorem** proves that a Fourier transformation translates between convolution and multiplication of functions. If

$$y(t) = (x * h)(t)$$

then the Fourier transform results from multiplication

$$\hat{y}(\omega) = \hat{x}(\omega)\,\hat{h}(\omega)$$

with $\hat{y}(\omega)$ the Fourier transform of $y(t)$. The reverse statement is also true, i.e. if the time-domain signal can be split into a product of two functions then the Fourier transform of that product is the convolution of the Fourier transformed parts.

3.3.2 Applications: digital filtering

This final section on digital signal processing should teach you the most important types of digital filters and their use. We will not go into filter design which is a huge specialised area in itself. Once you got some practice on different filter types and know what they can and cannot do for you then you'll be very well equipped on digital filtering already. Filtering means preserving certain favoured signal frequencies or features without distorting them while simultaneously suppressing others.

1. There are two conceptually different types of signal filters: finite impulse response (FIR) and infinite impulse response (IIR) filters. We will use examples of both.

2. There are two computationally different procedures to apply filters: the convolution algorithm and the direct method. The direct method is faster and simpler to program. Again, both are very much related but might well appear very different.

3. There are three optimal filters where optimal means you can't do better than these. The issue though is what optimal means, i.e. optimal for what? Learning a bit about these optimal filters and their meaning is hence important.

4. Frequency cut-off filters correspond to integration and differentiation. This link is the original motivation for the lecture title.

FIR and IIR filters are best illustrated visually, see Fig. 7. There you see the real Fourier transform of a square signal in time, this oscillating response in frequency space. The converse is also true. If you want a square window in frequency space, its Fourier transform in time looks like Fig. 7 but with a time axis instead of frequency.

Why a window in frequency? If you want to remove, filter, noise frequencies from your data, cutting out frequencies above and below some cut-off values seems like a good idea. The response you get on your data however is rather unpleasant. You get these nasty oscillations multiplied to your data so that the result is typically not what you want. Note that this oscillating response carries on to infinity, in principle, i.e. you made an IIR filter, the ideal band-pass filter, just that it doesn't work. Perfect in frequency space means rubbish in the time-domain. That duality is an important example for many areas in physics exhibiting dual behaviour.

Not all is lost, however. A FIR filter could be made from this by truncating the oscillating function from Fig. 7 at some point. That is indeed what happens in practice when you design a filter. Most FIR filters truncate an impossible but theoretically ideal filter with a window function. The art of design is then shifted to making the window function. SciPy for instance has plenty of classic window functions pre-defined. Let's not pursue this design business any further. It would be an entire teaching module in itself.

Instead, consider the second point in the list above. We discussed and implemented convolution, a convenient procedure since there is this `signal.convolve()` function working for us. So what is this direct method?

It derives from convolution but takes into account that most of the terms in the sum can be disregarded once you consider a fully defined filter (with window and all). That results from such a filter turning into (that is the short-cut we take by disregarding filter design!) a typically short set of **filter coefficients**. The direct method then implements digital filtering each sample point at n simply as a sum of

data points in time following:

$$y[n] = \sum_{i=0}^{M} a_i\, x[n-i] + \sum_{j=1}^{N} b_j\, y[n-j]$$

where $x[n]$ is the input signal and $y[n]$ is the output after filtering. The constants a_i and b_j are the filter coefficients. Note that the filter can now use 'past' output samples, $y[n-1]$ for instance, while the first term on the right is nothing other than convolution again. The equation above is also called **recursion equation**. The use of past results is nothing other than **feedback**. Most filters with feedback coefficients are IIR filters, bypassing the infinite response of the filter. Filters without feedback are most often FIR filters.

What the above means is that digital filtering with the direct method is very fast and super-simple IF you know the right filter coefficients for what you want to do. It isn't always possible to filter using the recursion equation, see below, but it certainly beats anything else if you can.

Optimal filters Before delving into examples of recursive filters, let's list and discuss the optimal filters [3]:

Moving Average filter: the marvellous moving average filter is the simplest and most practical filter you will come across. Best of all, it is even an optimal filter. No other filter is better at preserving rising or falling edge features in data while suppressing noise. Edges are a highly relevant feature as they typically signal a rapid change of something, say a pulse arriving or a collapse of the stock market.

Matched filter: this optimal filter is best at retrieving a known(!) structure from data. This filter corresponds to a search function for data when looking for anything in your data you know or suspect is there. This filter is the best at finding it. However, it isn't very good at approximately finding something. You really need to know what you are looking for.

Wiener or Optimal filter: this filter is optimal for maximising the signal to noise ratio. That being one of the most important numbers for experimental data, the Wiener filter is often also known as simply 'the' optimal filter. It can be a little temperamental on occasion but if it works fine then achieving a maximum signal to noise ratio is pretty much the best you can do with data.

The Wiener and matched filters must be carried out by convolution and both are completely determined by the characteristic of the problem, a specific apparatus or

experiment for instance. They will not be considered further but are well worth reading about if you are interested.

The more general but also far more practical optimal filter up for discussion is the moving average filter. In short, the moving average filter consists of the convolution of data with a rectangular pulse of length M and area 1, an M-point filter. Ideally, the rectangular pulse in centred around the data point of interest hence require M to be an odd number. Then you would always average at point $y[n]$ as

$$y[n] = \frac{1}{M} \sum_{i=-(M-1)/2}^{(M-1)/2} x[n+i]$$

with $x[n]$ the input data and M the filter width. That can be turned into a very fast recursive filter with the coefficients (see the recursion equation above): $b_1 = 1$, $a_{-(M-1)/2} = -1/M$ and $a_{(M-1)/2+1} = 1/M$ [3]. Let's try that with a quick function, see listing 3.

Listing 3: Moving average recursive function.

```
'''
Moving average filter implementation for use with NumPy arrays.
'''
import numpy as np

def moving_average(inputdata, filterorder):
    ''' implement a moving average filter of odd order
        inputdata: to be filtered, NumPy array
        filterorder: integer number as filter order,
                     will be made odd if even.
    '''
    if not filterorder % 2: # even order
        filterorder += 1 # make it odd for symmetric window

    response = 1 / filterorder # a float normalization factor
    output = []

    # padding the input for averaging data borders
    leftextension = np.flip(inputdata[:filterorder // 2], 0)
    rightextension = np.flip(inputdata[-filterorder // 2 + 1:], 0)
    padded = np.concatenate([leftextension, inputdata,
       rightextension])

```

```
23      output.append(np.sum(padded[:filterorder]) * response) # first
            average
24      n = filterorder // 2 + 1
25      for idx in range(n, len(inputdata) + n - 1):
26          term = output[-1] # previous
27          term += padded[idx + filterorder // 2] * response
28          term -= padded[idx - filterorder // 2 - 1] * response
29          output.append(term)
30      return np.array(output)
```

If you then make some noisy data and filter it, you can see how the filter smoothes the data while preserving sharp edges.

```
1  >>> import numpy as np
2  >>> import matplotlib.pyplot as plt
3  >>> from moving_average import moving average
4  >>> def simple_pulse(length, amplitude, risetime, decaytime):
5          ''' pulse model function to work with numpy.'''
6          time = np.linspace(0, length, length + 1)
7          onset = 0.3 * length # start 30% into the length
8          pulse = np.exp(-(time - onset) / risetime) - np.exp(-(time -
               onset) / decaytime)
9          pulse[np.where(time < onset)] = 0.0 # not defined before
               onset time, set 0
10         return -amplitude * pulse
11 >>> amp = 10.0
12 >>> risetime = 3.0
13 >>> decaytime = 300.0
14 >>> data = simple_pulse(2000, amp, risetime, decaytime)
15 >>> noisy = data + 0.4 * amp * np.random.normal(size = len(data))
16 >>> smooth = moving_average(noisy, 13)
17 >>> plt.plot(noisy)
18 >>> plt.plot(smooth, 'r-')
19 >>> plt.title('pulse␣with␣noise␣and␣smoothing')
20 >>> plt.ylabel('amplitude', fontsize = 14)
21 >>> plt.xlabel('samples', fontsize = 14)
22 >>> plt.show()
```

So much for smoothing by digital filtering. The moving average filter is strongly recommended for such tasks. However, when it comes to removing specific frequencies from a signal, the moving average is really, really bad. Such challenges are best left to frequency filters.

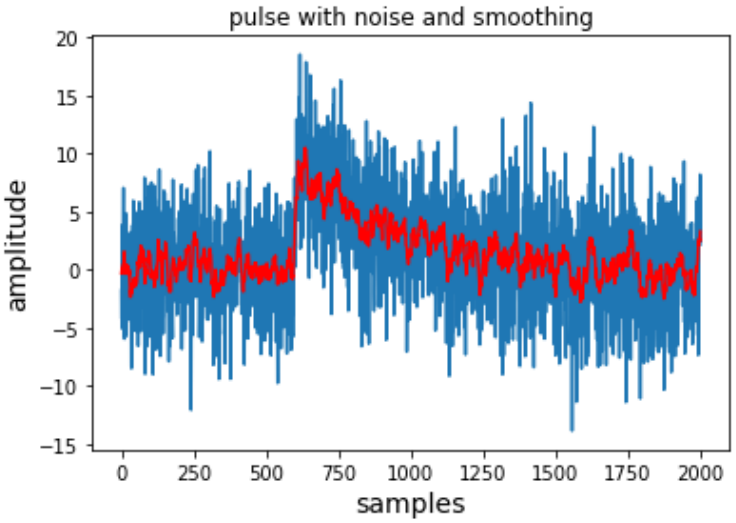

Figure 11: Overlay of artificial pulse with noise (blue) and smoothed pulse (red). Note how the moving average filter preserves the sharp rising edge of the pulse while suppressing random noise fluctuations.

Frequency filters Let's therefore end the discussion of optimal filters and come to filters specifically made to manipulate the frequency spectrum of any signal. It is for these filters that the recursion equation plays to its strength when implementing such filters. First have the implementation of the recursion equation in Python.

Listing 4: Recursion equation as Python function.

```
'''
Recursion equation implementation for use with NumPy arrays.
'''
import numpy as np

def recursive(inputdata, acoeff, bcoeff):
    ''' implement the recursion equation '''
    if len(inputdata) <= len(acoeff):
            return np.array([]) # more coefficients than data
    output = []
    la = len(acoeff)
    lb = len(bcoeff)
    ldata = len(inputdata)
    convolvedata = np.flip(inputdata,0)

```

```
16      output.append(acoeff[0] * inputdata[0]) # y[0]
17      for n in range(1, ldata):
18          if n+1 <= la: # border case 1
19              term1 = np.sum(acoeff[:n+1] * convolvedata[-n-1:])
20          else:
21              term1 = np.sum(acoeff * convolvedata[-n-1:-n-1+la])
22
23          if (n-lb) < 0: # border case 2
24              term2 = np.sum(bcoeff[:n] * output[:n])
25          else:
26              term2 = np.sum(bcoeff * output[:lb])
27
28          output.insert(0, term1 + term2) # prepend to list
29
30      return np.flip(np.array(output), 0) # normal order
```

First of all, the **low-pass filter**. Chances are you heard that technical term before. It comes up fairly often in electronics where it is also called an RC-filter owing to the way how to build it from a resistor and a capacitor. The name low-pass however is much more apt as it describes what this filter does with frequencies. Low frequencies are left unchanged up to a critical frequency and higher frequencies are removed.

As discussed above, any sharp cut-off in frequency space will destroy any time-domain signal after Fourier transform. Simply setting all frequencies higher than a cut-off value to zero is therefore not a good idea. A smoother transition would be good and that is what a low-pass filter will do.

Using the recursion equation, the low-pass filter is implemented by merely two non-zero coefficients [3]:

$$a_0 = 1 - f_c \quad ; \quad b_1 = f_c$$

where f_c is defined as the critical or cut-off parameter of the filter according to

$$f_c = \exp\left(-2\,\pi\, f_{cut} \,/\, f_{Nyquist}\right)$$

where $f_{Nyquist}$ is the limiting frequency for sampling from the sampling theorem: $f_{nyquist} = 1/\left(2\,\triangle t\right)$ and $\triangle t$ the time interval between sampling points. Hence the cut-off parameter can only have well defined values below the Nyquist frequency and the factor f_c is limited to the interval between zero and one which keeps the filter operation stable and well defined. Time to test our recursion equation function.

Say we recycle the simple pulse example from above, testing the moving average filter, see Fig. 11 then taking that noisy pulse and run two different low pass filter

operations on it, you get Fig. 12 and 13 from the listing below, then plot as before for the moving average example.

```
>>> # have noisy pulse as a numpy array, then
>>> from recursive import recursive
>>> from math import exp, pi
>>> fnyq = 1.0 / (2.0 * 1.0) # time steps in units of 1
>>> frc = 1.0 / 100.0 # also try 1.0 / 10.0
>>> fcut = exp(-2.0 * pi * frc / fnyq)
>>> acoeff = np.array([1-fcut])
>>> bcoeff = np.array([fcut])
>>> filtered = recursive(noisy, acoeff, bcoeff)
```

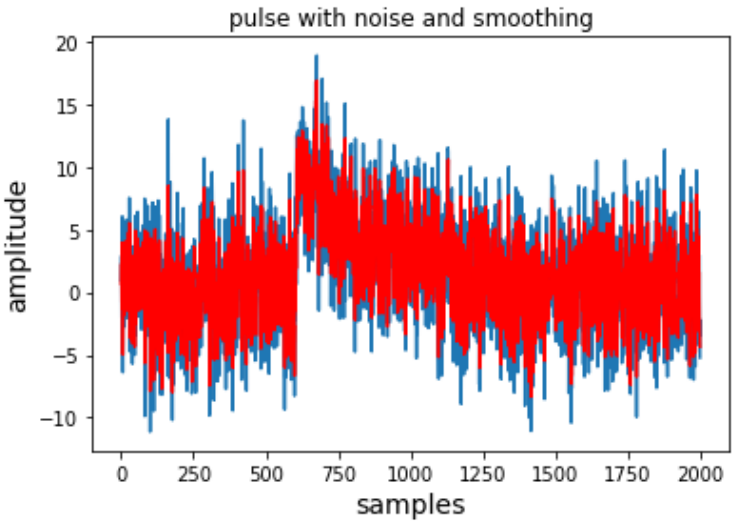

Figure 12: Overlay of artificial pulse with noise (blue) and low-pass filtered pulse (red). Note that here the cut-off frequency is set to 1/10 for unit interval sampling hence only few high-frequencies are being suppressed hence the filtered pulse still looks quite noisy.

A **high-pass filter** is characterized by the following coefficients:

$$a_0 = \frac{1}{2}\,(1+f_c) \quad ; \quad a_1 = -\frac{1}{2}\,(1+f_c) \quad ; \quad b_1 = f_c$$

with f_c defined as above. Other interesting frequency filters can be made this way. When working with pulses you would quite often read the technical term **quasi-Gaussian shaping**. That translates into a variety of possibilities, none of which is well defined but for practical purposes, this simply means: Apply a series of

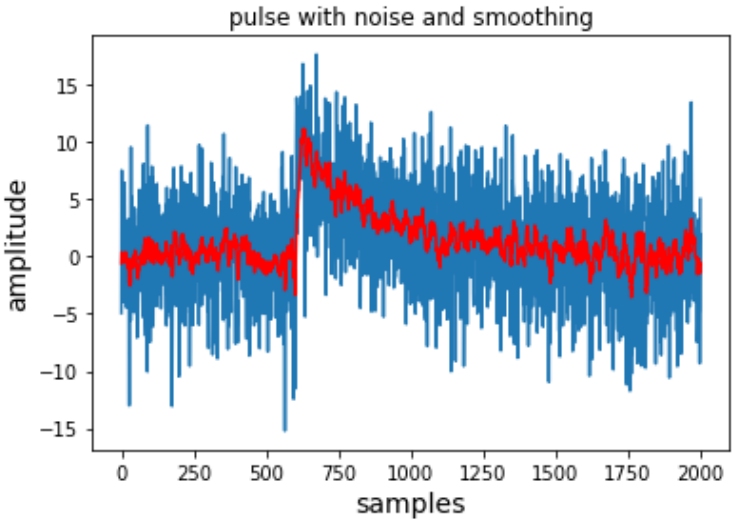

Figure 13: Overlay of artificial pulse with noise (blue) and low-pass filtered pulse (red). Note how the lower cut-off frequency at 1/100 suppresses random noise fluctuations as these are mostly composed of higher frequencies. The pulse however is still nicely retained without suffering too much distortion.

frequency filters to a pulse. For instance a series of (HP, $4\times$ LP) with HP, high-pass, and LP, low-pass, performs sufficiently well not to be distinguishable from other quasi-Gaussian filter networks. Here the $4\times$ means literally, run the low-pass filter four-times in series over the pulse, i.e. the output of one filter becomes the input of the next.

Combining a low-pass and a high-pass filter at different frequency cut-offs can construct a band-pass filter (only frequencies between the cut-off's are passed) or its opposite, a notch filter. These are implemented as bi-directional filters such that they preserve the phases of the signal and hence don't introduce unnecessary distortions beyond cutting frequencies. A bidirectional filter runs twice over the data, once as normal, the second time over the time-inverted or mirrored signal (use for instance `np.flip()` as seen in listing 3). The band-pass coefficients are a little more complicated since you need to hand over two parameter instead of just the cut-off frequency [3].

$$a_0 = 1 - k \quad ; \quad a_1 = 2\,k\,r\,\cos\left(2\,\pi\,f_{centre}\right) \quad ; \quad a_2 = r^2 - k$$

and

$$b_1 = 2\,r\,\cos\left(2\,\pi\,f_{centre}\right) \quad ; \quad b_2 = -r^2$$

where $f_{centre} = f/f_{Nyquist}$ is now the normalized centre frequency f around which the symmetric band-pass interval extends in frequency. The constants k and r are

defined as

$$k = \frac{1 - 2\,r\,\cos\left(2\,\pi\,f_{centre}\right) + r^2}{2 - 2\,\cos\left(2\,\pi\,f_{centre}\right)}$$

and $r = 1 - 3\,b$ with $b = f_{band}/f_{Nyquist}$ the normalized bandwidth frequency interval. Note that the normalization of f_{centre} and b means each can only be a number between zero and $\frac{1}{2}$. The notch filter uses the same system of constants but the coefficients look like

$$a_0 = k \quad ; \quad a_1 = -2\,k\,\cos\left(2\,\pi\,f_{centre}\right) \quad ; \quad a_2 = k$$

and

$$b_1 = 2\,r\,\cos\left(2\,\pi\,f_{centre}\right) \quad ; \quad b_2 = -r^2$$

where the centre frequency and the bandwidth describes the range of frequencies you want to cut out of the signal, i.e. the opposite of the band-pass filter. Notch filters are often used to remove prominent noise frequencies like 50 Hz noise from the domestic power supply.

Integration and differentiation? Remains the question what this filtering has to do with differentiation and integration. There is an easy, electric circuit explanation and a long but proper explanation. For those who don't consider electric circuits as easy, all explanations are quite involved. Naturally, this is again a part of the script you don't really need to know anything about in order to pass this second year module.

Let's keep the explanations as short as possible. The low-pass filter corresponds to integration in the sense that the capacitor integrates (collects) electric charge and hence the output voltage of a RC-filter, the low-pass filter, is the integral of the input charge as a function of time. Naturally, this type of integration applies to sampled data, not mathematical functions. For the latter, we'd still use the `scipy.integrate.quad()` function.

For the differentiator, the CR-filter or high-pass filter, the output voltage is taken across the resistor instead of the capacitor and then Ohm's law gives

$$V_{out} = R\,I = R\,\frac{dq}{dt}$$

hence the signal is proportional to the derivative of the charge on the capacitor as a function of time rather than the integral of the charge.

The point here is that the electric circuit explanation makes a lot of hidden assumptions before you even get started. The real reason why the filters to a certain extent correspond to integration or differentiation respectively is that the respective

filter transfer functions correspond to Laplace transforms of the integration and differentiation operators respectively.

You see, the longer explanation promises to be a really long one. What is a filter transfer function and what is a Laplace transform? In truth, I wrote this part of the script so that you have something to study but it simply became too long even for this script. The electronics explanation must suffice for now. Just to say that studying the Laplace transform is worthwhile as it has pride of place for many Physics applications.

3.4 Summary

Numerical integration with the `scipy.integrate.quad()` function was assumed to be known for this lecture. Instead the focus was on more unusual but ubiquitous integrals in computational physics: the **Fourier transformation** and **convolution** as the main methods in digital signal processing.

The key concept here is that a large part of computational physics deals with integration (to a lesser extent with differentiation) over discrete data rather than functions. The second most prominent integration application in physics, solving differential equations for physics modelling, will feature in the final lecture.

This lecture on digital signal processing covers a large amount of ground. The Fourier transformation itself is often an indispensable tool for data analysis while the discussion of digital filters introduces quite practical solutions to most requirements physicists will encounter.

What was kept short and would be worth extending is the topic of convolution. The lecture only considered convolution on data in time in order to lead to digital filtering. The most prominent applications of convolution however are with multi-dimensional data, for instance images in 2D and 3D. In that context convolution rose to fame mostly because of its use in image processing and hence now with machine learning, for instance in medical imaging applications.

4 Modelling Physics with Random Numbers

So far in this lecture, random numbers already feature when adding random noise to data. That is an important application of modelling physics with random numbers, used without much discussion. We can consolidate what we know about random numbers in Python at this point a little more systematically and then go from there.

Two main Python libraries for random numbers have been used on occasion, the **import** `random` Python standard library and the NumPy `np.random` library. The former produces single random numbers on request using its many functions while the latter produces entire arrays of random numbers using its many functions.

We have used quite a few of those functions: `random.random()` returning float numbers between zero and one, `random.uniform(a, b)` returning float numbers from a to b, and `random.randint(a, b)` returning integer numbers from a to b. Similar functions exist in NumPy. These are all random numbers **sampled from a uniform distribution**. That means, all values have equal probability to be picked. The uniform distribution is short for uniform probability distribution function (PDF). Welcome back to statistics and data analysis and all that.

It is quite crucial for modelling physics to realise the motivation for and foundations of random numbers. The connection to probability density functions is the key. Any function or distribution that can be integrated, discrete or continuous, analytically or numerically, can be normalized to an integral of value 1. Once that is done, whichever way, then you can draw random numbers from such a construct. A uniform distribution is just the simplest possible PDF. We could use almost anything.

Plenty of standard PDF's are already predefined in Python. We used the exponential distribution `random.expovariate(mean)` before, also the normal (Gaussian) distribution `random.gauss(mean, std)` for noise for instance. However, we could make our own PDF if required. Such sampling from non-uniform PDF's is often at the heart of modelling physics with random numbers (and other uses of random numbers). For instance the exponential distribution was useful for modelling radioactive decays because that is how you describe the process in time. Variations of individual measurements, data points, are often very well described by the Gaussian PDF, so much so that you define error bars with the help of the Gaussian PDF (standard deviation definition).

With this sizeable stock of Python experience on random numbers, we should start using them with pre-defined distributions and do some physics. That method to 'do some physics' using random numbers is in fact what is considered as **Monte-Carlo simulation** method. After that, you should see how to make your own, bespoke PDF to solve a modelling problem before you learn a little bit about the

non-trivial variations of the Monte-Carlo method of modelling physics. This again, is a general method described in big textbooks and can at best be lightly skimmed in a single lecture.

4.1 Modelling with standard probability distribution functions

The key point for modelling anything with random numbers is that you need a lot of them, whatever you do. Drawing a single random number only makes sense in the context of drawing many before coming to any conclusion, no matter the setting or task.

Only a, preferably, large set of random numbers can even be considered as random. A single random number is merely a number like any other, standing well on its own. Therefore, any Physics simulation contains necessarily a method that draws a large set of random numbers, all in one with NumPy or in series with the standard library, whatever is more appropriate or faster.

Time to give it a try:

```
>>> import numpy as np
>>> from random import gauss
>>> def gravity(height, time):
        ''' constant acceleration g calculation from s=0.5*g*t^2 '''
        return 2 * height / time**2
>>> def fallsim(attempts, height, heightError, time, timeError):
        ''' simulate Galileo's tower of Pisa experiment '''
        collector = []
        for _ in range(attempts): # no counter variable needed
            distance = gauss(height, heightError)
            watch = gauss(time, timeError)
            collector.append(gravity(distance, watch))
        return np.array(collector)
>>> pisa = 58 # [m]
>>> falltime = 3.4 # [s] in standard Earth gravity
>>> herror = 0.5 # [m] uncertainty from where to drop to the floor
>>> werror = 0.5 # [s] watches were rather uncertain
>>> measurements = fallsim(1000, pisa, herror, falltime, werror)
>>> plt.hist(measurements, 21)
>>> plt.title('Gravity␣constant␣measurements')
>>> plt.ylabel('measurements', fontsize = 14)
>>> plt.xlabel('g␣[ms$^{-2}$]', fontsize = 14)
```

```
23 >>> plt.show()
```

This little simulation shows the variations one has to expect when determining the gravity constant from measurements of height of the tower of Pisa and time measurements. This assumes first that you know the model, the function to calculate the constant from height and time. Secondly, that you know the error you make in your distance and time measurements. It also assumes that your error bars originate from a Gaussian PDF.

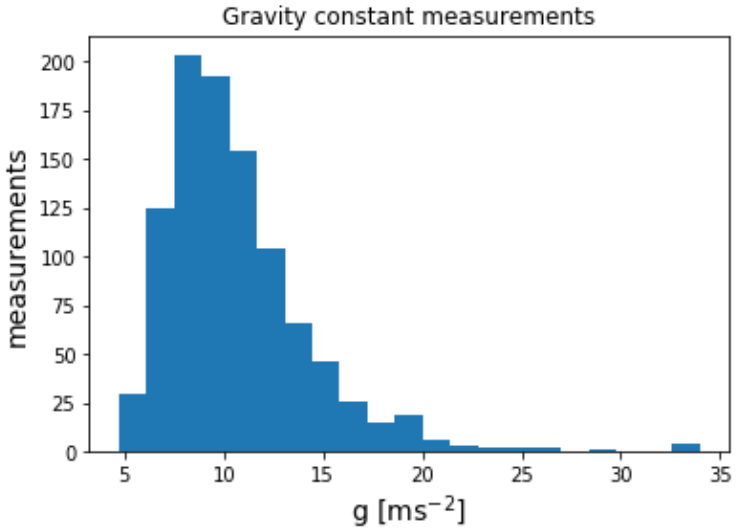

Figure 14: Simulation of measurements of the gravity constant by falling from the tower of Pisa.

You then draw many random numbers for time and height values in the permitted error bar range with the right distribution and calculate g for each one. The histogram, Fig. 14, nicely summarises your many simulated measurements. As you can see from the histogram, this isn't an easy averaging to get the value of g. The distribution of measurements is far from Gaussian, rather tailed to larger values.

Often the relations between observables are too complicated or not even known, or the processes to get to observables is so complicated or numerous that you simply can't propagate errors or do anything analytically. That is the point when experiment simulation is inevitably an essential tool. Ideally, you would always use that tool when planning and designing an experiment.

In this case for instance, planning the experiment would mean that you determine requirements on height and time uncertainties from a target uncertainty, also called precision, in determining g. The simulation would allow to vary the uncer-

tainties on height and time measurements such that you hit the required precision on g and then and only then would you perform the experiment, assuming you have devices to measure height and times with sufficient precision.

Here is one more experiment simulation showing you another way to utilize random numbers from known PDF's. Say you have a complete measurement and summarize your results in a histogram, see Fig. 15.

```
>>> signal = np.random.normal(15.0, 2.1, 420)
>>> background = np.random.uniform(0,40,400)
>>> measurements = np.concatenate([signal, background])
>>> hist, bin_edges = np.histogram(measurements, 40)
```

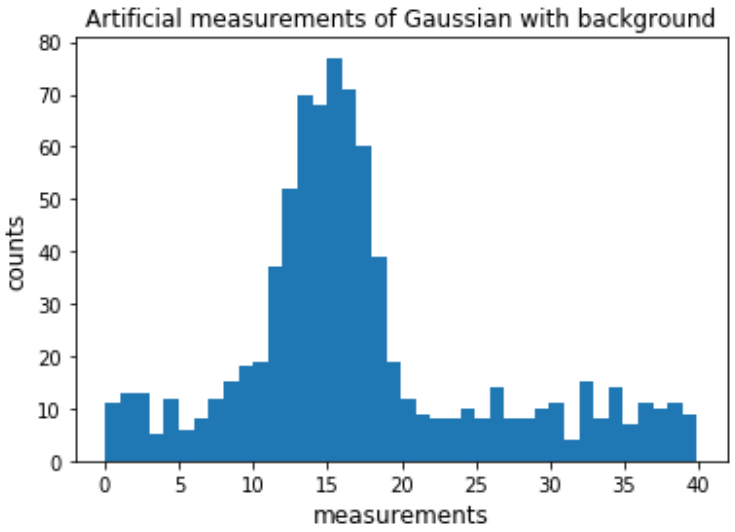

Figure 15: One random data set of a Gaussian with background.

You can certainly fit the Gaussian structure in that histogram even though it looks maybe a bit messy due to the non-negligible background but still, should work. However, also histograms have uncertainties. Is the Gaussian structure robust enough to survive despite the possible fluctuations of individual bin contents? That would determine again the precision to which you can measure the characteristics of this Gaussian structure. If we use Monte-Carlo simulation in order to get that precision, you can appreciate how non-trivial that process is, see Fig. 16.

Listing 5: Monte-Carlo simulation of Gaussian fits with background in histogram.

```
import numpy as np
import math
```

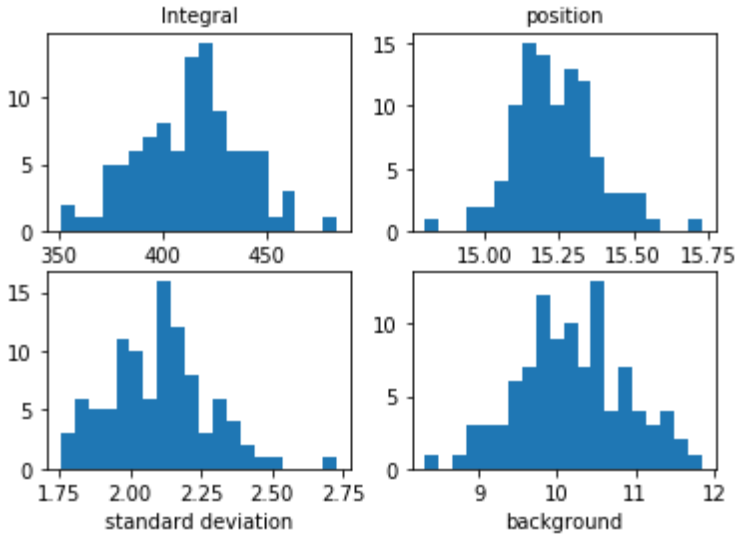

Figure 16: Distribution of fit results from Monte-Carlo simulation of histogram data containing a Gaussian with background.

```
from scipy.optimize import curve_fit
import matplotlib.pyplot as plt

# define a Gaussian function for fitting
def gaussian(data, total, position, width, baseline):
    ''' Gaussian function, working with numpy arrays
    Parameter: total = integral of the Gaussian
               position = mean position of the peak
               width = standard deviation of the Gaussian curve
               baseline = flat background level
    '''
    term = -0.5 * ((data - position)**2 / width**2)
    return total / (math.sqrt(2 * math.pi) * width) * np.exp(term) +
       baseline

# make the histogram
signal = np.random.normal(15.0, 2.1, 420)
background = np.random.uniform(0, 40, 400)
measurements = np.concatenate([signal, background])
hist, bin_edges = np.histogram(measurements, 40)

# used this before to fit to the centre of bins
```

```
24 binwidth = bin_edges[1] - bin_edges[0]
25 bclist = [bin_edges[0] + 0.5 * binwidth] # first element set
26 for idx in range(1, len(bin_edges)-1):
27     bclist.append(bclist[-1] + binwidth)
28 bincenters = np.array(bclist)
29
30 # first guess, should remain the same for each fit
31 initguess = (300.0, 15.0, 2.0, 5.0)
32
33 # start the simulation
34 nsims = 100
35 storage = []
36 data = hist.copy() # preserve original histogram
37 for _ in range(nsims):
38     bestfit, fitcov = curve_fit(gaussian, bincenters, data, p0 =
          initguess)
39     storage.append(bestfit) # list of lists for later analysis
40     # vary the data according to its PDF
41     data = np.array([np.random.poisson(value) for value in hist])
42 results = np.array(storage) # nsims times 4 array
43
44 # analyze results
45 plt.figure()
46 plt.subplot(221)
47 plt.hist(results[:,0], 20)
48 plt.title('Integral', fontsize = 10)
49 plt.subplot(222)
50 plt.hist(results[:,1], 20)
51 plt.title('position', fontsize = 10)
52 plt.subplot(223)
53 plt.hist(results[:,2], 20)
54 plt.xlabel('standard␣deviation', fontsize = 10)
55 plt.subplot(224)
56 plt.hist(results[:,3], 20)
57 plt.xlabel('background', fontsize = 10)
58 plt.show()
```

The listing 5 shows the Monte-Carlo code. We recycle the Gaussian fit function (from line 7) as well as the method to get bin centres from histograms for fitting (from line 23). Before that we make a measurement as above. The simulation starts from line 34 with the crucial line 41. There we allow each histogram bin to fluctuate

according to the Poisson distribution (as appropriate for histogram data) and fit again in each single simulation. Note that the fluctuations always take the original data set as their base, not the latest modified data set.

The fit results are all stored and then plotted. As can be seen in Fig. 16, the variation on the four fit parameters is non-trivial, i.e. not exactly Gaussian and in absolute value could not have been predicted easily in any other way. This is a typical case for using Monte-Carlo simulations as the tool of choice.

4.2 Make your own PDF

You might be forgiven to look at all the pre-defined distribution functions for random numbers in Python (for instance from the *random module* and *numpy.random*, check the links to see the manual pages) and think that surely you can cover all possible cases with those. Why bother thinking about making your own random numbers from a custom PDF?

You'll find that quite often you will need to draw random numbers from something complicated that isn't already defined. That something could be a result from theory or geometry or simply a previous measurement used for another measurement.

In practice, there are two quite straightforward ways to create bespoke random numbers for all those cases. Distributions from theory or geometry can often be handled with the **accept/reject** method if they describe some result with boundaries, i.e. nothing in that result tends to infinity. Measurement results (as well the other cases of interest) can often also be cast directly into distributions in form of a histogram (could be multi-dimensional, of course). Then those custom-made histograms can serve directly as the basis for drawing random numbers from them.

4.2.1 Accept / Reject method

Conceptually, this method couldn't be easier. Whatever complicated shape in parameters you want to draw random numbers from, you can wrap that shape in a simple volume, usually a box (in N-dimensions). Of course, there are more complex and efficient ways to do this but for us here, a box will suffice perfectly well.

That image can sometimes be a little confusing as it should be read as an abstract concept. However, in practice this often can be visualised literally as a 3-D box or a 2-D rectangle simply because in almost all cases, we are rarely dealing with more than 2 or 3 variables, abstract or not.

Therefore, in an attempt to be exhaustive at this point, when visualising a

box wrapped around something complicated like a function or a difficult geometric shape you wish to draw random numbers from, consider the boundaries of the box really as the separate limits of a uniform distribution (equal probabilities between lower and upper limit) for each one of the abstract parameters. These could be space coordinates in the case of geometry descriptions but they could just as well be abstract parameters of a complicated function.

Once you wrapped the object distribution in your box you simply draw uniform random numbers, one for each dimension separately, within the limits of the box and check whether a complete point in that abstract space belongs to the function (geometry shape) or not, i.e. you either accept it or reject it. If accepted, you got yourself a random point from that difficult shape, otherwise try again. That's it, the accept/reject method [1].

The easiest way to get started with that method is a geometry case study. Say you need random points originating from a circle then any two number passing the condition $X^2 + Y^2 \leq r^2$, where X and Y shall be random numbers and r the fixed circle radius, will form a random point (X, Y) on the circle. We have therefore transformed(!) the more challenging distribution (the circle) to the easiest distribution (two uniform distribution numbers). The price you pay for this simplification is the efficiency. Not all points drawn from the two calls to the uniform distribution will result in a circle point. You wasted computing effort. That can be a problem at times and you'd have to smarter in your choice of 'box' but that should not bother us here.

When would you ever want random numbers from a geometric shape for doing physics? I've used the circle random numbers for instance to simulate light emission from the Sun, or radioactive particle emission from the surface of a glass hemisphere or a steel wire. You never know.

Random numbers from complicated functions are dealt with the same way but clearly appear to be more abstract and in need of an example. Wrap your function in a box and use uniform random numbers for each dimension separately, no change to the geometry example. However, wrapping a function in a box of course includes the function values, not just the function parameters. If you got a function $f(x, y)$ in 2-D then you need a 3-D box to wrap it. The condition to check for is in that case $Z \leq f(X, Y)$, with X, Y, Z all uniform random numbers.

Objection, I hear, that way you draw random numbers from the area of the

[1] As an aside: This is also the basis for **Monte-Carlo integration**. You count the number of hits against the total number of attempts and estimate the volume of the unknown shape, i.e. its integral since you know the integral of your box by definition. Quite trivial, really, which is why I didn't teach you that method in the integration chapter and anyway, we've done that in PX150 exercises.

function, i.e. its integral, not from the function itself. Well, only if we wanted the integral and add up all the accepted counts. What this procedure also enables us is to count the accepts at the given parameter values. The more counts we got for a small interval of parameter values, the higher the value of the function for that parameter interval.

The phrase 'parameter interval' is your clue. We can build the function distribution from the histogram(!) of accepted counts. Each bin then corresponds to the function values integrated over only a small interval. More informally, by drawing the histogram, we draw the derivative of the integral, i.e. the function.

Try this interesting distribution you get from the classic double-slit experiment in optics, see Fig. 17. It is the intensity distribution for Fraunhofer diffraction.

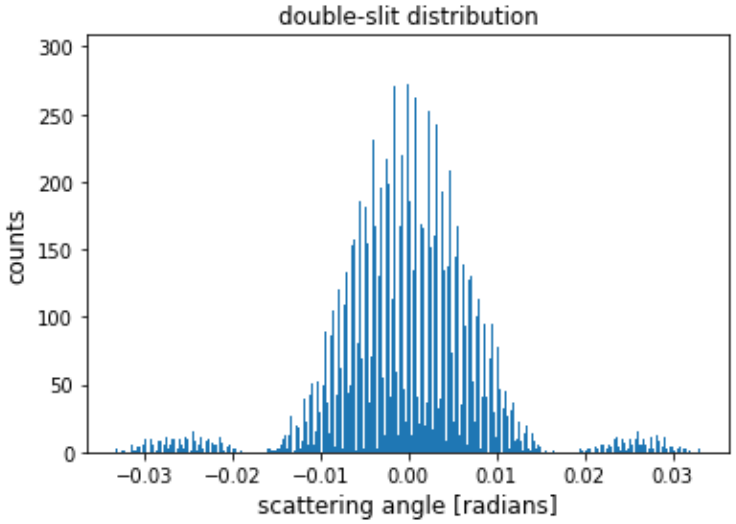

Figure 17: Distribution of the double-slit interference pattern for green light $\lambda = 550$ nm on slits with width 0.01 mm and slit distance of 0.7 mm.

Listing 6: Double-slit interference pattern distribution.

```
import numpy as np
import math
import random
import matplotlib.pyplot as plt

def doubleSlit(theta, width, distance, wavelength):
    ''' calculate the intensity distribution on screen after double
        slit.
```

```
8           Needs theta in radians, wavelength in [nm],
9           distance and width in [nm]
10      '''
11      if wavelength < 1.0: # safeguard against nonsense
12              return 0.0
13      par1 = math.sin(theta) * math.pi * distance / wavelength
14      term1 = math.cos(par1)
15      par2 = math.sin(theta) * math.pi * width / wavelength
16      term2 = np.sinc(par2)
17      return term1**2 * term2**2
18
19  nsims = 200000
20  wvlength = 550 # green light [nm]
21  distance = 7.0e5 # 0.7 mm slit distance
22  width = 1.0e4 # 0.01 mm slit
23  values = []
24  for _ in range(nsims):
25      theta = random.uniform(-math.pi/90.0, math.pi/90.0) # range of
            theta
26      ftry = random.uniform(0.0, 1.0) # max function =1 at theta = 0
27      value = doubleSlit(theta, width, distance, wvlength) # get
            f(theta)
28      if ftry <= value: # accept/reject
29          values.append(theta)
30
31  distribution = np.array(values)
32  plt.hist(distribution, 721)
33  plt.xlabel('scattering␣angle␣[radians]', fontsize=12)
34  plt.ylabel('counts', fontsize=12)
35  plt.title('double-slit␣distribution', fontsize=12)
36  plt.show()
```

The Monte-Carlo loop to create these bespoke random numbers starts in line 24 in listing 6 with the accept/reject condition in line 28.

4.2.2 Random numbers from histograms

If you had the histogram in Fig. 17 as a distribution from which to draw random numbers, i.e. not the set of accepted angle values like you get from listing 6, you would require a new way to draw bespoke random numbers.

Note that drawing random numbers from arbitrary histograms happens quite often in physics since they represent concise summaries of measurement results, easily communicated and utilized for further analysis. Fortunately, drawing random numbers from histograms is rather simple. They are already distributions by definition. All we need to do is to normalize them to an integral of 1 in order to be able to call them a **probability distribution**. Any (finite) distribution of values, normalized to area 1 is a probability distribution for a single parameter.

It might be a bit alien to think of measurement results in histograms to be turned into bespoke random number distributions. Why would anyone ever want that? You've done the measurement after all, what is there to simulate?

A great deal, to be honest, would be an answer to that. Most measurements in physics are in fact chains of measurements. Most of the time, these chains are well hidden by simply quoting an error bar and using that further up the chain in your error propagation or analysis. That's all neat and fine as long as you can trust the error bar to be the result of a Gaussian distribution, tried and checked by someone else, usually a manufacturer.

That pretty picture, however, doesn't always work. In fact, in physics research this actually works very rarely. For instance, you could spend a lot of effort to measure the precise wavelength spectrum of a new and expensive lamp which you would like to use in an optics experiment. Lamps normally do not emit light as a function of wavelength in a Gaussian distribution around a mean value. Far from it! Most lamps show a spectrum with lumps and bumps and tails and are anything but clean emitters if you make the effort of recording their emission spectrum. If you then want to use that lamp in your optics experiment, you would wish to simulate what all these other wavelengths would do to your experiment.

Here is the recipe for turning that lamp spectrum (or any other histogram distribution for that matter) into random numbers:

1. Normalize the histogram, i.e. integrate and divide every bin by that total integral. The total area is now equal to one and hence the distribution is a probability distribution.

2. Create the cumulative distribution of the histogram, i.e. create a new histogram with the same bins, varying widths and all. Then calculate the new entries for the bins as the sum of areas up to the bin under consideration. The first bin remains unchanged, the second is the sum of the first and the second, etc.

3. If the normalization worked correctly, the cumulative distribution must reach the final value equal to one at its final bin entry.

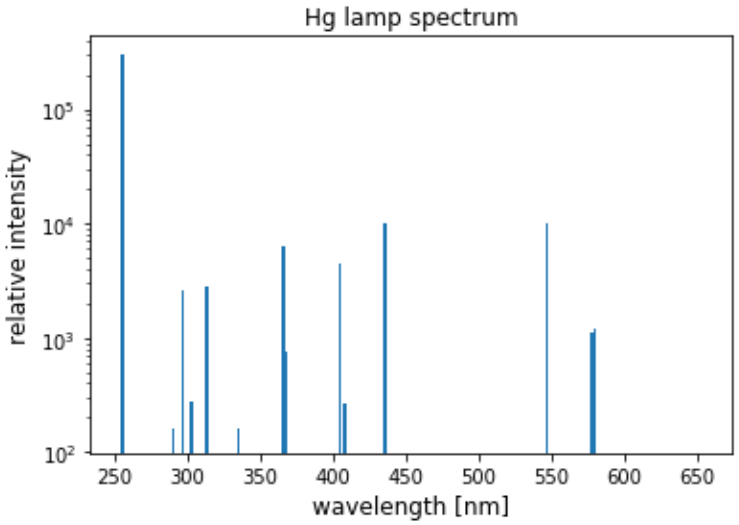

Figure 18: Mercury lamp emission spectrum as basis for bespoke random numbers.

4. Draw a uniform random number Y (yes, that again).
5. Find the first bin from the cumulative distribution with $Y \leq$ bin content.
6. Draw a uniform random number X in the chosen bin interval which gives the bespoke random number you seek.

Having a recipe means we can program that, see listing 7.

Listing 7: Drawing bespoke random numbers from the Mercury emission spectrum.

```
1  import numpy as np
2  from random import random, uniform
3  import matplotlib.pyplot as plt
4
5  bin_edges = np.array([250.0, 253.0, 254.0, 255.0, 288.0, 289.0,
       290.0,
6                        295.0, 296.0, 297.0, 301.0, 302.0, 303.0, 311.0,
7                        312.0, 313.0, 314.0, 333.0, 334.0, 335.0, 364.0,
8                        365.0, 366.0, 367.0, 403.0, 404.0, 405.0, 406.0,
9                        407.0, 408.0, 434.0, 435.0, 436.0, 545.0, 546.0,
10                       547.0, 576.0, 577.0, 578.0, 579.0, 580.0,
                            600.0]) # [nm]
11 spectrum = np.array([0, 0, 3.0e5, 0, 0.0, 160.0, 0, 0.0, 2600.0,
12                       0, 0.0, 280.0, 0, 2800.0, 0.0, 4700.0, 0,
```

```
13                      0.0, 160.0, 0, 0.0, 6270.0, 760.0, 0, 0.0,
14                      4400.0, 0, 0.0, 270.0, 0, 0, 1.0e4, 0,
15                      0, 1.0e4, 0, 0.0, 1100.0, 0.0, 1200.0, 0]) #
                            relative intensity
16
17  # step 1
18  bindifferences = np.diff(bin_edges)
19  norm = np.sum(spectrum * bindifferences)
20  normalized = spectrum / norm
21
22  # step 2
23  cumulative = np.cumsum(normalized * bindifferences)
24
25  store = []
26  nsims = 1000
27  for _ in range(nsims):
28      # step 4
29      trial = random()
30
31      # step 5
32      idx = np.where(cumulative >= trial)[0] # all indices
33      upperbin = bin_edges[1 + idx[0]] # take only the first index
34      lowerbin = bin_edges[idx[0]]
35
36      # step 6
37      value = uniform(lowerbin, upperbin)
38      store.append(value)
39
40  data = np.array(store)
41  plt.hist(data, bins=bin_edges, log=True)
42  plt.xlabel('wavelength␣[nm]', fontsize=12)
43  plt.ylabel('relative␣intensity', fontsize=12)
44  plt.title('Random␣draw␣from␣Hg␣lamp␣spectrum', fontsize=12)
45  plt.show()
```

Note from line 5 in listing 7 that a histogram can very well contain unequal bin widths, no problem with that. This has to be taken account of for the integral, of course, as demonstrated in line 19 and also when adding up the cumulative distribution, line 23.

So much on random numbers and their use in Monte-Carlo simulations, bespoke or not, these methods are clearly very powerful in modelling Physics.

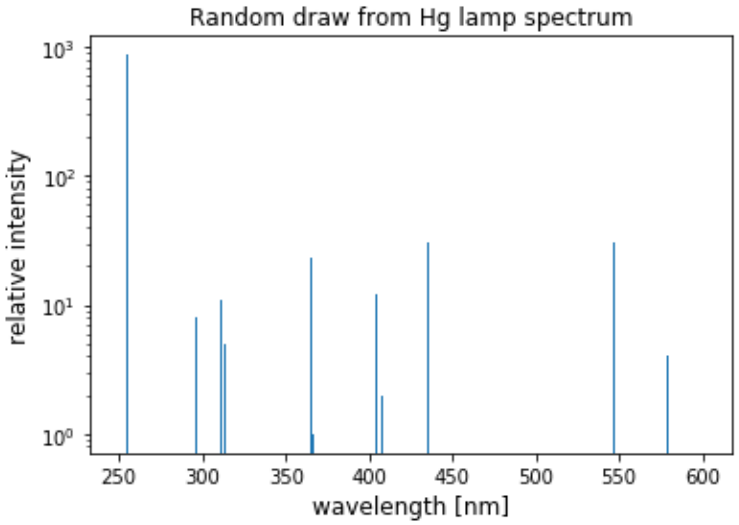

Figure 19: Distribution of 1000 random numbers drawn from the lamp spectrum.

4.3 Simulation in statistical physics

Statistical physics deserves a separate section not only since it is a huge topic in itself but also because Monte-Carlo simulations are an essential tool for this subject, more so than for any other part of physics. Clearly we can only have the briefest of glimpses but at least a little insight should be attempted.

Statistical mechanics deals with intrinsic random motion of many individual parts of a physical system. Motion determines the word mechanics and random implies the word statistical. What do you do if the basic laws of motion and interaction are all simple and well known but the system, say a liquid, has so many parts that you can't keep track of all of them? Your thermal lecture will have made it clear that you then look for averages if they exist. Just discount all the individual bits and pieces and try to learn about the averages over them all.

Monte-Carlo simulation of many-element systems like liquids or gases can be separated into two related but conceptually quite different methods: first, the direct simulation of each part, bit by bit for the whole system. The second method involves simulating configurations of all the parts of a system and see how they change stochastically (randomly under conditions), given some simple assumptions.

The latter method you can find under the keyword of Markov Chain Monte Carlo or MCMC. It is a big research area in computational physics and far too much to be covered here. In particular, this method requires physics concepts from your second thermal lecture, the one you didn't have yet, so MCMC will not be explained

here.

The direct simulation method however is straightforward and can be linked to your first thermal lecture. We simply have to trust (or check) that simulating a (relatively) tiny number of particles and building averages from such an ensemble is good enough to show some thermal physics averages.

We tackled the direct simulation method on the first year module in section 5.4, page 60ff. There we discussed a random walk example. A few hundred particles are allowed to randomly move about without checks and balances (rules and boundary conditions). That is not terribly helpful when trying to get some physics out of it.

If we try that exercise again but this time get some relevant physics out then the most fundamental would be to demonstrate the second law of thermodynamics with a random walk in a closed box. Such a system would for instance demonstrate the expansion of a gas in an empty (vacuum) vessel from a central point. Entropy, $S = -k_B \sum_i^N P_i \log P_i$, should on average only increase until equilibrium is reached and it does, see Fig. 20.

What this means is that despite not using any force laws or external influences, simply by random walking in a very simple fashion (no collisions for instance), we still get a very fundamental physical law demonstrated. The particles move all around the box after a time and remain doing so. The system has changed from a very orderly state, i.e. all particles in one place at the origin to a highly dis-organised state in equilibrium. The measure of dis-order is entropy.

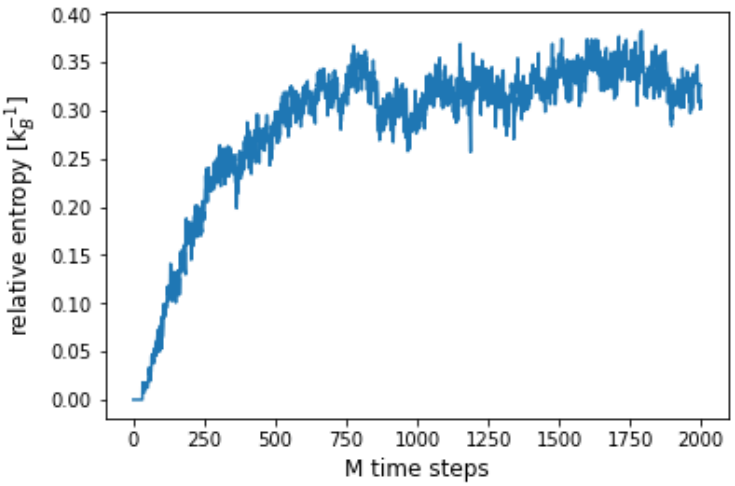

Figure 20: Entropy of 1000 particles in a closed box of side length 40 units after 2000 time steps.

The code is a little longer, see listing 8, but the Monte-Carlo element is as simple as it gets. Merely random integers are required to determine the unit step size changes to each particle. Every loop iteration corresponds to a single time step and all particles move one unit in space in an iteration. The function `random_walk_2D` moves all particles, they are all in position arrays, and checks the boundaries.

The function `probability` calculates the entropy for each hard-coded grid cell, where the box is subdivided into 5×5 cells. In statistical physics that is called coarse graining. Again, the counting is all vectorized and NumPy arrays are used for easy logical 'AND' operation.

The `box_entropy` function runs the Monte-Carlo loop where line 63 requests the random integer numbers where each of the integers 1 to 4 corresponds to movements in space by one unit as in `random_walk_2D`. The entropy calculation is on line 70 where all 25 cell entropies are added up for the total entropy of the gas in the box.

Listing 8: Random walk in a closed box.

```
1  '''
2  Entropy in a closed box.
3  Calculate entropy in a closed box of +- side length as given
4  with N particles for M time steps.
5  '''
6  import numpy as np
7  import matplotlib.pyplot as plt
8  import math
9
10 def random_walk_2D(xpositions, ypositions, move, box):
11     ''' random walk action with boundary check '''
12     # ease readability by naming constants sensibly
13     NORTH = 1; SOUTH = 2; WEST = 3; EAST = 4 # constants
14
15     # updating positions in vectorized form
16     ypositions += np.where(move == NORTH, 1, 0)
17     ypositions -= np.where(move == SOUTH, 1, 0)
18     xpositions += np.where(move == EAST, 1, 0)
19     xpositions -= np.where(move == WEST, 1, 0)
20
21     # boundary check, hard boundaries
22     # move one unit away from boundary
23     ypositions += np.where(ypositions == -box, 1, 0)
24     ypositions -= np.where(ypositions == box, 1, 0)
```

```
25      xpositions += np.where(xpositions == -box, 1, 0)
26      xpositions -= np.where(xpositions == box, 1, 0)
27
28      return xpositions, ypositions
29
30
31  def probability(xpositions, ypositions, box, which):
32      ''' coarse grain box in 25 cells, i.e. 5 by 5 '''
33      pos = which % 5 # modulus 5 [0-4]
34      row = which // 5 # multiples of 5 [0-4]
35      # first cell is bottom left
36      left = -1.0 + 0.4 * pos * box
37      right = left + 0.4 * pos * box
38      bottom = -1.0 + 0.4 * row * box
39      top = bottom + 0.4 * row * box
40
41      regionx = xpositions > left
42      regionx = np.logical_and(regionx, xpositions < right)
43      regiony = ypositions > bottom
44      regiony = np.logical_and(regiony, ypositions < top)
45
46      region = np.logical_and(regionx, regiony)
47      region = region[region == True]
48
49      densityA = float(len(region))/float(len(xpositions))
50
51      return densityA
52
53
54  def box_entropy(nparticles, msteps, box_side_length):
55      ''' calculate entropy in a closed box '''
56      # starting position at origin
57      xpositions = np.zeros(nparticles) # x-position vector
58      ypositions = np.zeros(nparticles) # y-position vector
59
60      Sdevelop = []
61      for _ in range(msteps):
62          # request four random integer numbers per particle
63          moves = np.random.randint(1, 4+1, size=nparticles)
64          xpositions, ypositions = random_walk_2D(xpositions,
65                                      ypositions, moves, box_side_length)
```

```
66          entropy = 0.0
67          for i in range(25):
68              prob = probability(xpositions, ypositions,
                    box_side_length, i)
69              if prob>0.0:
70                  entropy += -prob * math.log(prob)
71          Sdevelop.append(entropy)
72      return np.array(Sdevelop)
73
74  # main
75  allentropy = box_entropy(1000, 2000, 20)
76  plt.plot(allentropy)
77  plt.xlabel('M␣time␣steps', fontsize=12)
78  plt.ylabel('relative␣entropy␣[k$_{B}^{-1}$]', fontsize=12)
79  plt.show()
```

As you can guess, this type of Monte-Carlo for statistical physics could be used for many more applications but this should suffice for now.

4.4 Summary

The focus in this lecture is on using simulations on the basis of random numbers to model physics. This can serve to specify the obtainable precision of results given uncertainties or to model expected raw data as simulated input to a full data analysis, again to specify requirements on either the analysis or the raw data.

Often, pre-defined probability distributions for random numbers are not sufficient to model physics and two popular methods on creating bespoke random numbers are discussed. The **accept/reject** method is useful for Monte-Carlo integration but also to produce bespoke random numbers from geometrical shapes and functions. Obtaining bespoke random numbers from histograms instead is often required if a starting point for modelling originates from already independently processed data. Each method is shown with examples.

Simulation in **statistical physics** is mentioned in passing with a random walk example, alluding to the ubiquitous use of random numbers and simulation in statistical physics. Little else on this topic can be done at this stage but it is too important to leave it out completely.

5 Ordinary Differential Equations in Physics

This final section of the lecture will bring us back to numerical integration. Solving ordinary differential equations (ODE's) is often called integrating ODE's. However, it is not quite integration as you know it. There are many different, more or less sophisticated methods to solve, integrate, an ODE.

Initially, we will focus on a 'black-box' solver provided by SciPy. General solver modules for differential equations are very difficult to write. Do not expect a general solver to work in all cases. The challenge for anyone using general ODE solvers is not so much 'how' to use them but understand when 'not' to use them.

ODE's play an important role in physics. One of the most prominent laws of physics is an ODE, Newton's third law, the $F = m\,a$. Unfortunately, all the other fundamental equations that you might well look forward to learning about this year and later are not ODE's: the Maxwell equations for electromagnetism, the Schroedinger equation for quantum mechanics, Einsteins's field equations of general relativity, all are partial differential equations, not ODE's.

Dealing with them on a computer is significantly more complicated than what we can attempt here. As a somewhat weak apology, all I can offer is the prospect that if you can master the computational physics on this module then you will be in a much better position to understand in later modules, in year three for instance, how to deal with the above fundamental computational physics challenges.

5.1 Single and multi-variable ODE's

Just to clarify, dealing with ODE's means there is exactly **one independent variable** in contrast to partial differential equations where you can have many. This does not exclude multi-variable ODE's! The word 'variable' does not imply automatically the word 'independent'. Consider movement under a force: the movement description will depend on location variables, say three coordinates in 3D space, which in turn will all depend on time as the independent variable. Any set of differential equations in this case can always be written as ODE's with respect to time and hence are proper ODE's, i.e. one independent time variable.

One-dimensional problems in physics aren't always particularly interesting hence let's go to multi-variable ODE's almost immediately and rather see some simpler examples from that perspective. Write first-order ODE's in general form as

$$\frac{d\boldsymbol{r}\,(t)}{dt} = \boldsymbol{f}\,(\boldsymbol{r}\,(t)\,,\,t)$$

and second order ODE's similarly as

$$\frac{d^2\,\boldsymbol{r}\,(t)}{dt^2} = \boldsymbol{f}\left(\boldsymbol{r}\,(t)\,, \frac{d\,\boldsymbol{r}\,(t)}{dt}, t\right)$$

or rather

$$\frac{d\,\boldsymbol{r}\,(t)}{dt} = \boldsymbol{s}\,(t) \quad ; \quad \frac{d\,\boldsymbol{s}\,(t)}{dt} = \boldsymbol{f}\,(\boldsymbol{r}\,(t)\,, \boldsymbol{s}\,(t)\,, t)$$

where bold symbols denote vectors, either vectors of variables $\boldsymbol{r}\,(t)$ or vectors of functions $\boldsymbol{f}\,(\boldsymbol{r}\,(t)\,, t)$. Note how the second-order ODE has been reduced to two first-order ODE's. You might have seen that already in your maths modules. If not, convince yourselves by simple substitution.

The point of this very brief display is that all we need to focus on here is how to solve systems of first-order ODE's with initial conditions. That is what this section will be about and what is conveniently covered by the SciPy solver.

5.2 Using solve_ivp()

Well, what then is this SciPy solver and how to use it? It is part of the SciPy integration library hence it should come as no surprise that its usage is similar to the `quad()` integration function from last year. The first argument it takes must again be a function to solve, here a first order ODE. Fortunately, `solve_ivp()` also allows an array of first-order functions as first argument. Therefore, solving second order ODE's like shown above is also possible.

The second argument is a tuple of two floats, stating the time interval for the solution. It is important here that the start time as given must be the time value for which the initial conditions are given. The initial conditions are the third argument to the `solve_ivp()` function. The optional argument `t_eval` can take an array of values for the independent variable at which we wish the solutions to be computed. Otherwise the solver chooses its own set of values for time. The next optional argument to consider is `args`, a tuple of arguments to the ODE function which would typically be parameter values. For more arguments to the function check the link to the *manual page*.

5.2.1 The simplest example

Time to practice. Solve the simplest first-order ODE with exponential decay as the known solution:

$$\frac{dy(t)}{dt} = -\mu\, y(t)$$

with μ a constant parameter. That could for instance be a single radioactive decay with a decay constant μ as an example. The ODE is already in a format that we can use the solver directly hence we define the right-hand side of the ODE:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from scipy.integrate import solve_ivp
>>> def exponential(t, y, mu):
    ''' dy(t)/dt = this function expression '''
    return -mu * y[0]
```

As you can see, the function definition is quite literally along the line of the maths definition of our ODE, i.e. we need to integrate the right-hand side of the first-order ODE and that is what the function calculates. The reason to write `y[0]` in the function is that `solve_ivp()` not only permits an array of functions to be solved for, it expects to receive that array but here we only deal with a single variable ODE, our `y[0]`. Next come the parts to engage the solver (and plot):

```
>>> mu = 0.1 # decay time constant parameter
>>> time = np.linspace(0.0, 100.0, 1001) # must contain t=0 for y(0)
>>> init = np.array([1.0]) # y(0)=1.0
>>> ysolve = solve_ivp(exponential, (0.,100.), init, t_eval=time,
    args=(mu,)) # solve with parameter
>>> plt.plot(time, ysolve.y[0])
>>> plt.ylabel('y(t)',fontsize=12)
>>> plt.xlabel('t',fontsize=12)
>>> plt.show()
```

We set the constant parameter to its value, create the array with time values at which we wish to have solutions and create the initial condition $y(0) = 1$ as the expected array even though there is just a single condition. It is important to include the independent variable value corresponding to the initial condition as the first(!) value of the array, here for time values.

The call to the `solve_ivp()` function follows the manual, the name of the function to solve as the first argument, time interval next, then initial condition, the array of time values to evaluate the solution and the optional parameters of the function as a tuple (even if it is just one as in this example). The plot of the results in Fig. 21 should come as no surprise to anyone, a fine exponential decay curve.

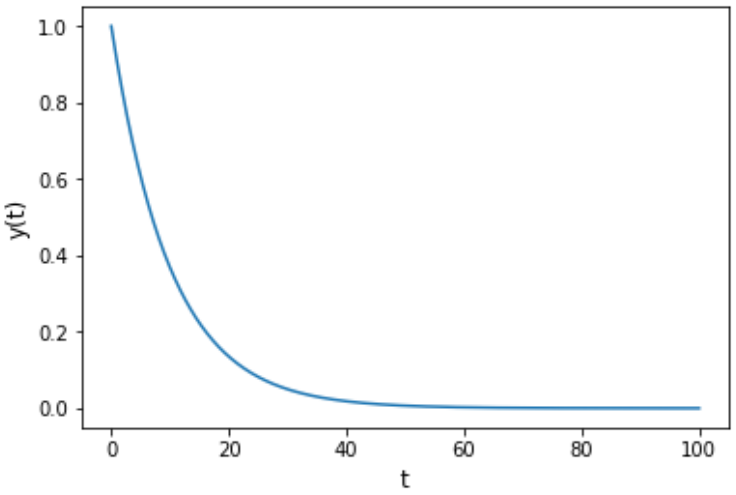

Figure 21: Exponential decay as the solution to $y'(t) = -\mu\, y(t)$ with $\mu = 0.1$.

5.2.2 A second-order example

That's the mechanics of the function dealt with and we can start demonstrating the ODE solver with some physics. Due to the predominance of Newton's ODE in physics, most examples show second-order ODE's and we will do the same. However, there are quite a few relevant and important first-order ODE's in physics.

You might wonder why bothering with first-order since they are most likely all solved analytically. Radioactive decay chains, i.e. coupled first-order ODE's are indeed solved analytically since a long time. Those solutions are called Bateman equations in case you come across those. However, there are quite a lot of **coupled, non-linear** first-order ODE's. Those are certainly not all solved analytically, quite the opposite.

Still, dealing with coupled first-order ODE's is much better demonstrated by solving second-order ODE's since for that we can use the comforting familiarity of classical mechanics. For the sake of avoiding boredom though let's take a look at a rather unusual pendulum [4], one hanging on a spring rather than a wire. The

differential equations describing that system would be[2]

$$\begin{aligned}\frac{d^2x(t)}{dt^2} &= (x_0 + x(t))\left(\frac{d\theta(t)}{dt}\right)^2 - \frac{k}{m}x(t) + g\cos\theta(t)\\ \frac{d^2\theta(t)}{dt^2} &= -\frac{1}{x_0 + x(t)}\left[g\sin\theta(t) + 2\frac{dx(t)}{dt}\frac{d\theta(t)}{dt}\right].\end{aligned}$$

This is a coupled oscillator system. The pendulum oscillation with the angle variable θ interacts with the spring oscillator in its length. Define the angle $\theta(t)$ as the angle with respect to the vertical and $x(t)$ as the change of length (stretch or compression) of the spring with respect to its rest length x_0.

As we know already, each second-order ODE has to be converted into two first-order ODE's in order to work with our SciPy solver. The corresponding function hence looks like

```
>>> from math import sin, cos
>>> def spring_pendulum(t, y, l_0, k, m, g):
    ''' 2 coupled 2nd order ODE for a spring pendulum'''
    f_0 = y[1] # this is s1
    f_1 = (l_0 + y[0]) * y[3] * y[3] - y[0] * k / m + g * cos(y[2])
    f_2 = y[3] # this is s2
    f_3 = -(g * sin(y[2]) + 2 * y[1] * y[3]) / (l_0 + y[0])
    # return all four variables (s1(t), ds1/dt, s2(t), ds2/dt) in
       this order
    return np.array([f_0, f_1, f_2, f_3])
```

The order of the `y[i]` is determined as coordinate followed by the time derivative of the coordinate for each variable (here two) separately. Our initial conditions have to respect that order. Let's fix the constant parameters, set initial conditions and solve the coupled ODE's

```
>>> # function constants
>>> l_0 = 1.0
>>> k = 3.5
>>> m = 0.2
>>> g = 9.8
>>> # initial conditions
>>> x_0 = 1.0 # initial stretch
>>> v_0 = 0.0 # no speed
>>> theta_0 = 0.3 # radians
```

[2]If you want to derive these, by far the easiest way is to study the Hamiltonian Mechanics module, PX267, and use Lagrangian methods explained in that lecture.

```
>>> dtheta_0 = 0.0 # from angular rest
>>> init = np.array([x_0, v_0, theta_0, dtheta_0])
>>>
>>> time = np.linspace(0.0, 25.0, 1000)
>>> answer = solve_ivp(spring_pendulum, (0.,25.), init, t_eval=time,
    args=(l_0, k, m, g))
```

Instead of displaying the oscillations as a function of time, they are best viewed differently. If we plot the results for the two variables in polar representation in 2D then what we get to see is the **trajectory** of solutions, i.e. how the stretch changes with the angle. In such a plot, consider the independent variable as marker on the trajectory, labelling each point drawn.

```
>>> # plot variables in 2D
>>> xdata = (l_0 + answer.y[0]) * np.sin(answer.y[2])
>>> ydata = (l_0 + answer.y[0]) * np.cos(answer.y[2])
>>> plt.plot(xdata, ydata, 'r-')
>>> plt.xlabel('Horizontal position')
>>> plt.ylabel('Vertical position')
>>> plt.show()
```

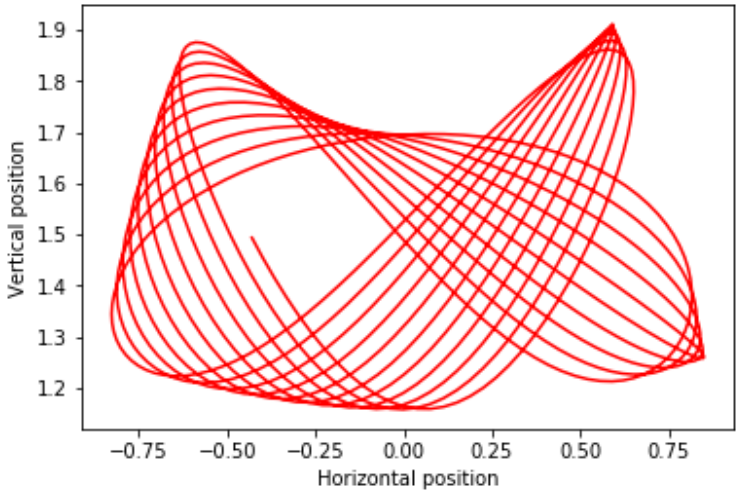

Figure 22: Trajectory of the coupled spring pendulum following [4].

So far, we never actually used the independent variable, time, explicitly in the ODE but the `solve_ivp()` allows for that, no problem. Let's try another pendulum, more conventional but realistic and learn a few more technical terms about ODE's and their use in physics.

5.2.3 A time-dependent example: Chaos

Consider a standard pendulum without small angle approximation, i.e. we keep the $\sin\theta$ term, but with friction and also allow for a driving force. That gets us [4]

$$\frac{d^2\theta(t)}{dt^2} = -\frac{g}{L}\sin\theta(t) - \beta\frac{d\theta(t)}{dt} + A\cos(\omega t)$$

with the ODE function

```
>>> def realistic_pendulum(t, y, g, length, beta, torque, omega):
    ''' ODE for a realistic pendulum'''
    f_0 = y[1] # this is s1
    f_1 = -(g / length) * sin(y[0]) - beta * y[1] + torque *
        cos(omega * t)
    return np.array([f_0, f_1])
```

The four constants describe the pendulum and the forces acting on it with g the gravitational acceleration, L the pendulum length, β the friction coefficient, A the torque due to the driving force and ω the drive frequency.

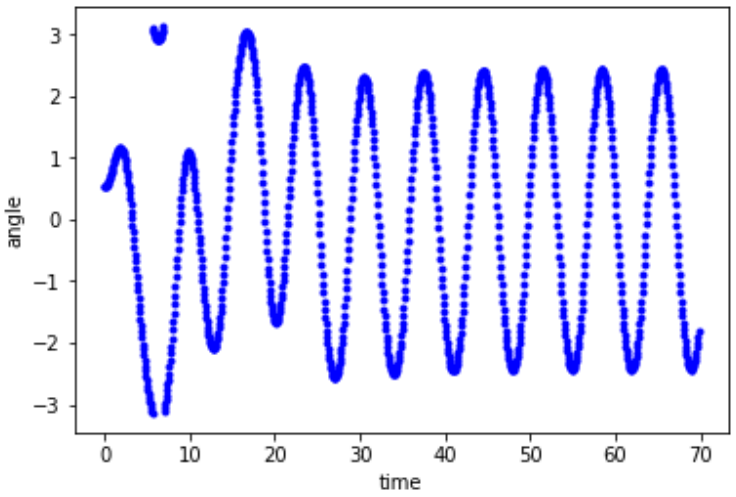

Figure 23: Swing of a pendulum with driving force and friction and angle range of $[-\pi\,;\,\pi\,]$, see text.

```
>>> g = 9.8
>>> length = g
>>> beta = 0.5
>>> A = 1.5
```

```
>>> om = 0.9
>>> N = 10
>>> init = np.array([pi / 6.0, 0.0]) # 30 degree out and at rest
>>> steps = 100
>>> tstep = 2 * pi / (om * steps)
>>> tend = N * (2 * pi) / om
>>> time = np.arange(0.0, tend, tstep)
>>> answer = solve_ivp(realistic_pendulum, (0.,tend), init,
    t_eval=time, args=(g, length, beta, A, om))
>>> # wrap to [-pi, pi]
>>> swing = answer.y[0]
>>> swing -= np.where(swing > pi, 2*pi, 0)
>>> swing += np.where(swing < -pi, 2*pi, 0)
>>> plt.plot(time, swing, 'b.')
>>> plt.xlabel('time')
>>> plt.ylabel('angle')
>>> plt.show()
```

An example to run this pendulum is shown above. Note the wrapping procedure from line 15. Initially the pendulum is driven around a little wild but soon settles into a regular swing pattern, say from around time unit 30, see Fig. 23. This initial settling period is typical for a driven, dynamic system.

In fact, this system is very suitable to introduce a few key concepts of dynamic systems. That is the research area dealing with a lot of real world physics, for instance it covers the big topic of **chaos** and ODE's are ideal to gain insights in this area.

The first concept to mention here is **phase space** which in general shows values of the coordinate and the change of the coordinate (location and speed for instance) on the axes. This is finally an opportunity to use the full return array from `solve_ivp()` since that is exactly what we got. Therefore, just change the plot command to show the phase space of the pendulum, see Fig. 24.

```
>>> plt.plot(swing, answer.y[1], 'b.')
>>> plt.xlabel('angle')
>>> plt.ylabel('angular␣velocity')
>>> plt.show()
```

The regular swing pattern in phase space then corresponds to a stable orbit, here visible as the thicker line circling around the centre. Such a stable orbit (could be any pattern as long as its stable) is called an **attractor**, something the whole system tends towards as it dynamically develops. The whole fun of researching

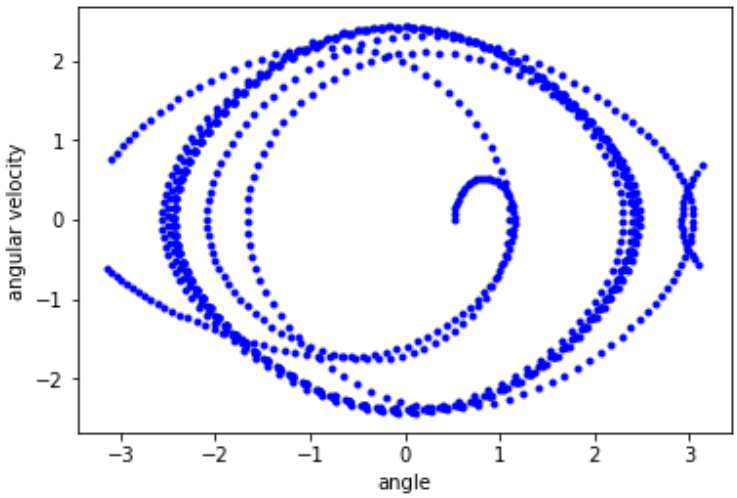

Figure 24: Phase space of a pendulum with driving force and friction and angle range of $[-\pi\,;\,\pi\,]$.

dynamical system now starts once you observe what happens to phase space patterns if you change system parameter, here for instance the drive frequency.

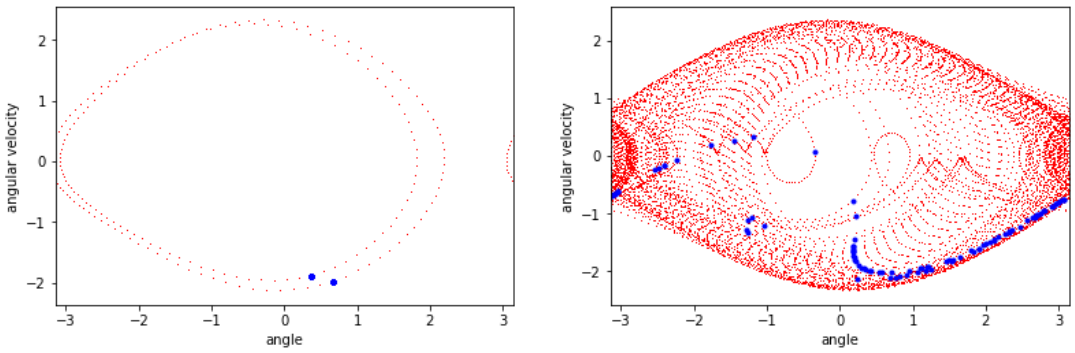

Figure 25: Phase space of a pendulum with period doubling (left) and chaotic behaviour (right).

One way to display a summary of the potentially messy behaviour that a dynamic system can show is the **Poincare plot** for which you pick a single snapshot, a photo, per swing and show that state of the oscillator in phase space, i.e. a single point in phase space. If you do that at always the same phase of the periodic behaviour then you can see by eye how that phase changes, if at all. For the code that means only a few short additions to collect those snapshot moments and display

them on top of the full phase space picture.

```
>>> # one snapshot per lap for Poincare plot
>>> skip = 100 # cut this initial behaviour
>>> N = 300 # give the system more time, more swings
>>> swing = answer.y[0][skip * steps:] # insert before line 14
>>> angvel = answer.y[1][skip * steps:] # insert before line 14
>>> # insert this before previous plot commands
>>> offset = 50
>>> max_index = (N - skip) * steps - offset
>>> poincare1 = []
>>> poincare2 = []
>>> for j in range(offset, max_index, steps):
        poincare1.append(swing[j])
        poincare2.append(angvel[j])
>>> plt.plot(swing, angvel, 'r,')
>>> plt.plot(poincare1, poincare2, 'b.')
```

where line 2 and 3 should replace the previous line 6 declaration and give the system more time to swing, mostly because you want to use lines 4+ directly replacing the previous line 14 in order to skip the initial dynamics before settling into a semi-stable behaviour.

What this Poincare curve shows is then either stable or chaotic motion of the entire system in phase space, see Fig. 25. After changing the drive frequency to a slightly smaller value than before, on the left of the figure you got strange behaviour, so called period doubling. There are two concentric attractors visible and hence two Poincare points in the plot. The system hence changes at that drive frequency between two response frequencies.

If that were not strange enough, on the right you can see what happens at another, lower drive frequency. The system changes now chaotically. There is a whole Poincare curve instead of a few fixed points. The Poincare curve for chaotic motion is in general a fractal, i.e. the greater the magnification of the curve, the more detail emerges, and helps to define chaotic behaviour.

5.3 Multi-particle dynamics

For this final topic on ODE's we will open the black-box a little and use a more direct approach to solving ODE's. The numerical algorithm is of no further consequence beyond the comments when introduced below. The real reason to stop working with SciPy is that the ODE's become very simple but the challenge is the huge number of ODE's to solve simultaneously.

That is what multi-particle dynamics is about, just Newton's law but for a huge number of constituents. One prominent example for this approach is molecular dynamics, another would be cosmological N-body simulations on large-scale structure formation. Both require better physics than Newtonian mechanics, true, but if you understand multi-particle dynamics with Newton's force law then changing that to new physics is not too onerous anymore from a programming point of view.

5.3.1 The Verlet method for equations of motion

There are plenty of numerical algorithms to solve ODE's, some well hidden in that black-box we were using. Now that we need one explicitly, we go for a nice and friendly method, one that inherently preserves energy conservation. Clearly a good property to have. The family of methods is called Verlet algorithms and includes the leap frog algorithm in case you look for keywords to study these algorithms[3].

Solving Newton's $F = m\,a$ could not be any easier. Once the F is specified, you get **equations of motion**. The leap frog algorithm can be adapted to solving equations of motion very well. It calculates locations and speeds using accelerations and that's it. Once there is a force, the rest follows. Well, at least as long as the force is velocity independent but most are in the areas of interest here.

That kind of approach sets the tone for what we have to do, supply the accelerations and hand them over to the fixed machinery of the leap frog algorithm, then repeat, step by step. Oh, and we have to supply the step, here the time step since we do kinematic.

To the method then in detail: In the year one script on pages 72/73 we briefly mention numerical differentiation with the message that taking a derivative at a point makes the lowest mistake if taken as a central difference. That is also what the SciPy function `scipy.misc.derivative()` does. Solving an equation of motion ODE with the Verlet or closely related leap frog algorithm means just that. Replace all differentiation symbols with finite, small differences, here time steps $\triangle t$, and write the difference equations explicitly (bold font symbols are vectors):

$$\boldsymbol{r}(t + \triangle t) = \boldsymbol{r}(t) + \triangle t\, \boldsymbol{v}(t + \triangle t/2)$$

and

$$\boldsymbol{v}(t + \triangle t) = \boldsymbol{v}(t + \triangle t/2) + \frac{1}{2}\triangle t\, \frac{\boldsymbol{F}(\boldsymbol{r}(t + \triangle t),\, t + \triangle t)}{m}$$

The first equation corresponds to the first step of forming first-order ODE's from a second order ODE, i.e. $\frac{dx(t)}{dt} = v(t)$. The second equation then is the derivative of

[3]The Verlet algorithms preserve quite a bit more than just energy; they are so called symplectic algorithms which is indeed a very nice property to feature.

$v(t)$ which has on the right hand side the force expression to form the equation of motion.

The half-steps in time intervals dotted around the expressions then originate from the central difference idea. The clue here is that central differences for the derivative of $x(t)$ and $v(t)$ are out of step by half a $\triangle t$ (that's the leaping frog) and then brought back together. That is important in order to calculate locations and speeds at the same time points if you want to use them for instance to calculate energy as a function of time.

One expression is still missing and that is the starter

$$\boldsymbol{v}(t + \triangle t/2) = \boldsymbol{v}(t) + \frac{1}{2}\triangle t\,\frac{\boldsymbol{F}(\boldsymbol{r}(t),\, t)}{m}$$

and the preparation step for the following iteration is also good to have

$$\boldsymbol{v}(t + 3\,\triangle t/2) = \boldsymbol{v}(t + \triangle t/2) + \triangle t\,\frac{\boldsymbol{F}(\boldsymbol{r}(t + \triangle t),\, t + \triangle t)}{m}.$$

In code, this is ideally a separate function, see listing 9. The easiest test to devise is checking the standard harmonic motion, for instance of a spring. If a clean oscillation results from the calculation and energy is conserved then we can start trusting the code in the current version, see Fig. 26.

Note that this is merely a convenient example on Newton's force law. In general, the Verlet method simply solves second-order ODE's which do not depend explicitly on first-order differentials. For such cases, of course, the 'force' function **F** should not be divided by a mass in the Verlet algorithm code.

Listing 9: Verlet ODE solver test for harmonic oscillator.

```
'''
Test the Verlet algorithm for solving equations of motion
from Newton's law.
The Force expression stands in a separate function, here
for the Hooke law giving standard harmonic oscillator motion.
The Verlet function is one-dimensional and needs adapting
for multi-particle, multi-variable applications.
'''
import numpy as np
import matplotlib.pyplot as plt

def force(x):
    ''' force law function '''
    return -0.5 * x # harmonic oscillator
```

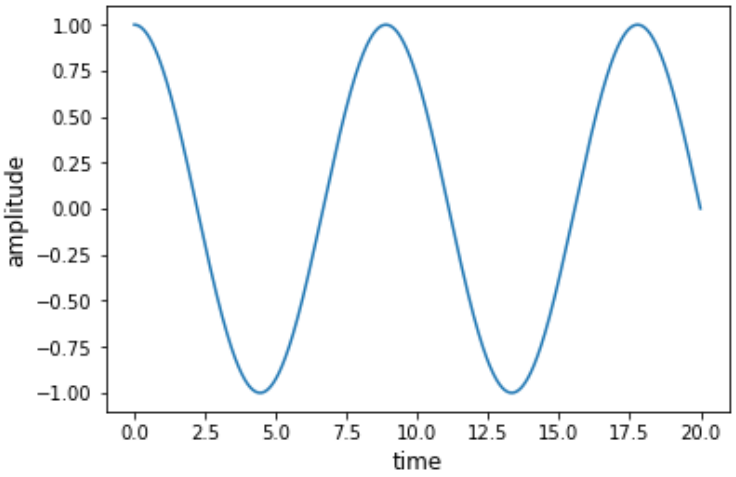

Figure 26: Standard harmonic motion as calculated with the Verlet method test.

```
15
16
17 def verlet(F, x0, v0, N, timestep, par=()):
18     ''' Equations of motion solver using 1-D Verlet algorithm '''
19     # set up
20     x = np.zeros(N)
21     v = np.zeros(N)
22     x[0] = x0
23     v[0] = v0
24     m = par[0]
25
26     # starter
27     vhalf = v0 + 0.5 * timestep * F(x0) / m
28
29     # loop
30     for i in range(N-1):
31         x[i+1] = x[i] + timestep * vhalf
32         k = timestep * F(x[i+1]) / m
33         v[i+1] = vhalf + 0.5 * k
34         vhalf += k
35     return x, v
36
37 N = 2000
38 step = 0.01
```

```
39 m = 1.0
40 x0 = 1.0
41 v0 = 0.0
42
43 xval, speedval = verlet(force, x0, v0, N, step, par=(m,))
44
45 time = np.arange(0.0, step * N, step)
46 plt.plot(time, xval)
47 plt.xlabel('time', fontsize=12)
48 plt.ylabel('amplitude', fontsize=12)
49 #plt.plot(xval, speedval)
50 #plt.xlabel('amplitude', fontsize=12)
51 #plt.ylabel('speed', fontsize=12)
52 plt.show()
```

The code in listing 9 doesn't yet give us anything more than what we used already with `solve_ivp()`. However, adapting it to multi-variable (for instance in 3D space) and multi-particle is fairly straightforward.

Currently, NumPy arrays are used to store the results as a sequence in time. It would be more useful to use the arrays to store N variables for M particles as $N \times M$ arrays and deliver results for each time-step either to be stored separately or discarded periodically merely to inspect final states after some time. This scheme would benefit from the in-built vectorization of NumPy arrays and require only one explicit Python loop for time steps. This is implemented in listing 10.

5.3.2 N-body dynamics for gravity-like forces

Inspecting the example code in listing 10 shows immediately, that is not the Verlet algorithm that is the real challenge for these N-body problems. It is rather the force law. Once particle positions have several coordinates one has to work with vectors (or here NumPy arrays), which is fine. Unfortunately, all particle–particle interactions and corresponding forces depend on particle distances. If everything can interact with everything else then one gets a lot of distances to keep track of.

Listing 10: Verlet N-body solver test with inverse square law force.

```
1 '''
2 Multi-variable, multi-particle Verlet method
3 equation of motion solver.
4 '''
5 import numpy as np
6 from scipy.spatial import distance
```

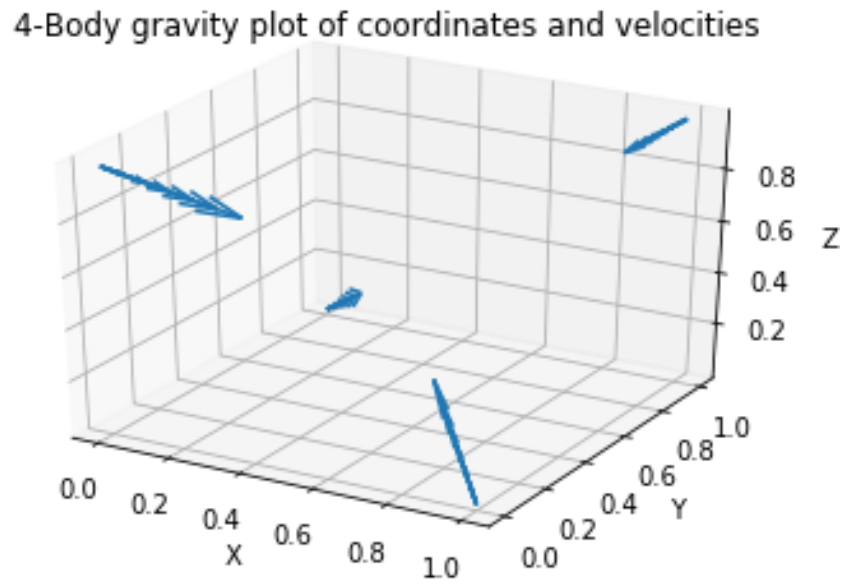

Figure 27: 4-body inverse square law test in 3D showing velocity vectors of all bodies after a short time evolution. All bodies start at rest and develop different speeds as they fall, mutually influenced by all others.

```
import matplotlib.pyplot as plt

def force(x, coupling):
    ''' force law function for particle i
        F_i = normed r_i * Sum_j(j != i) m_ij / r_ij^2
    '''
    allforces = []
    rcoord = []
    nparticles = x.shape[0]

    # mutual point distance matrix
    dij = distance.squareform(distance.pdist(x))
    for entry in dij:
        msk = entry>0 # all other particles, not itself
        invmsk = np.logical_not(msk) # only itself
        # inverse distance square law * couplings
        weight = np.sum(coupling[msk] / np.square(entry)[msk])
        for i in range(3): # 3D coordinates fixed
            coords = x[:,i] # all rows, i column
```

```
26              # mutual distance vectors from one to all other particles
27              rcoord.append(np.sum(coords[msk]) - (nparticles-1) *
                    coords[invmsk])
28          rijvec = np.array(rcoord).T[0] # reorder horizontally as
                array
29          allforces.append(weight / np.linalg.norm(rijvec) * rijvec)
30          rcoord = [] # next particle
31      return np.array(allforces)
32
33
34  def verlet(F, x0, v0, nsteps, timestep, par=()):
35      ''' Equations of motion solver using Verlet algorithm
36          for 3D, multi-particle states, x and v, no time-step
37          recording. Only final states are returned.
38      '''
39      x = x0 # initial states of NxM arrays
40      couplings = par[0]
41
42      # starter velocity
43      vhalf = v0 + 0.5 * timestep * F(x0, couplings)
44
45      # loop, no time step results recorded
46      for _ in range(nsteps - 1):
47          x += timestep * vhalf
48          k = timestep * F(x, couplings)
49          v = vhalf + 0.5 * k
50          vhalf += k
51      return x, v
52
53  # Calculation set up
54  nsteps = 2 # just a few for testing
55  step = 0.01
56  dim = 3 # x, y,z
57  M = 4 # number of particles
58
59  # Place particles in 3D with initial velocities
60  positions = [[1.0,0.0,0.0],[0.0,1.0,0.0],[0.0,0.0,1.0],[1.0,1.0,1.0]]
61  x0 = np.array(positions) # M particles in rows
62  v0 = np.zeros((M, dim)) # at rest
63
64  # Force law set up
```

```
65 coupling_constant = 1.0 # could be Newton G or electric charge
66 # just some test values
67 mvalues = coupling_constant * np.array([1.0, 2.0, 3.0, 10.0])
68
69 # Solve N-body problem and plot
70 # 3D plot set up
71 fig = plt.figure()
72 ax = fig.add_subplot(111, projection='3d')
73 ax.set_xlabel('X')
74 ax.set_ylabel('Y')
75 ax.set_zlabel('Z')
76 ax.set_title('4-Body␣gravity␣plot␣of␣coordinates␣and␣velocities')
77
78 # Try a few iterations and plot frequently on the same canvas
79 for _ in range(10):
80     xval, speedval = verlet(force, x0, v0, nsteps, step,
        par=(mvalues, ))
81     x0 = xval
82     v0 = speedval
83     ax.quiver(xval[:,0], xval[:,1], xval[:,2], speedval[:,0],
        speedval[:,1],
84               speedval[:,2]) # 3D vector plot
85
86 plt.show()
```

As exemplified in listing 10 from line 20, we need to harness all NumPy has to offer to get the mutual inverse distance squares right for every particle relative to every other and sort out the distance vectors in order to get the force directions right. The point–distance matrix in line 18 is also rather important and we have to resort to asking SciPy for that. For many particles, it is likely that matrix and its memory consumption which will limit this example code.

5.3.3 Molecular dynamics, many ODE's

Anyway, since we entered advanced territory already, we might as well go for a final spin and swap the force law from gravity–like to molecular dynamics. Inter-atomic classical interactions are often well described by a force law that you might or might not have seen before, the force derived from the Lennard-Jones potential: The force

$\boldsymbol{F}_{ij}$ exerted on particle i by particle j is

$$\boldsymbol{F}_{ij}(\boldsymbol{r}_{ij}(t)) = \left[\frac{A}{r_{ij}^{13}} - \frac{B}{r_{ij}^{7}}\right] \frac{\boldsymbol{r}_{ij}}{r_{ij}}$$

with constants A and B, r_{ij} the distance between particles i and j and $\boldsymbol{r}_{ij}$ the position vector between the particles. The main difference to the previous force we considered is that here you got the structure of a repulsion term and an attraction term. The repulsion term is called Pauli repulsion and the attraction is due to the van der Waals force. There are many other inter-atomic or inter-molecular force law expression but this is a classic. The good thing for this application is that the repulsive term will prevent numerically catastrophic collisions to occur unlike for the gravity example.

Note that in order to implement this force law instead of the gravity example we did all the hard work already. The mutual, particle to particle distances and vectors are required just as before. The force direction is also already sorted as the sum of all vectors contributing. Changes to the code in 10 hence are minimal.

Let's have a final specific example: colliding micro-droplets of water. They shall all have the same mass which we hence set to unity and get for the force term an expression as approximation to a Lennard-Jones potential [5]:

$$\frac{d^2\boldsymbol{r}_i}{dt^2} = \sum_{j=1;j\neq i}^{N} \left(-\frac{16.5}{r_{ij}^3} + \frac{158.6}{r_{ij}^5}\right) \frac{\boldsymbol{r}_{ij}}{r_{ij}}.$$

How to get that into code? There are subtle changes to the previous code apart from the obvious inverse distance dependencies, see listing 11.

Listing 11: Verlet N-body solver with Lennard-Jones–like force.

```
'''
Multi-variable, multi-particle Verlet method
equation of motion solver.using the Lennard-Jones force
'''
import numpy as np
from scipy.spatial import distance
import matplotlib.pyplot as plt

def force(x, coupling):
    ''' force law function for particle i
        F_i = normed r_i * Sum_j(j != i) (-A / r_ij^3 + B / r_ij^5)
        Regularization required against exploding force values.
        Simple minded cut off minimum distance implemented,
```

```
14          not a parameter but hard wired.
15      '''
16      allforces = []
17      rcoord = []
18      nparticles = x.shape[0]
19
20      # mutual point distance matrix
21      dij = distance.squareform(distance.pdist(x))
22      for entry in dij: # distances are r_ij length numbers already
23          msk = entry > 0 # all other particles, not itself
24          invmsk = np.logical_not(msk) # only itself
25
26          # strongly repulsive core, hard-limit
27          msk2 = entry < 0.8
28          regularize = np.logical_and(msk, msk2)
29          entry[regularize] = 0.8 # replace tiny distances with minimum
30
31          # attraction term
32          weightA = np.sum(- coupling[0] / entry[msk]**3)
33          # repulsion term
34          weightB = np.sum(coupling[1] / entry[msk]**5)
35          weight = weightA + weightB
36
37          # make r_ij connection vector
38          for i in range(3): # 3D coordinates fixed
39              coords = x[:,i] # all rows, i column
40              # mutual distance vectors from one to all other particles
41              rcoord.append(np.sum(coords[msk]) - (nparticles-1) *
                    coords[invmsk])
42          rijvec = np.array(rcoord).T[0] # reorder horizontally as
                array
43
44          allforces.append(weight / np.linalg.norm(rijvec) * rijvec)
45          rcoord = [] # next particle
46      return np.array(allforces)
47
48
49  def verlet(F, x0, v0, nsteps, timestep, par=()):
50      ''' Equations of motion solver using Verlet algorithm
51          for 3D, multi-particle states, x and v, no time-step
52          recording. Only final states are returned.
```

```
    '''
    x = x0 # initial states of NxM arrays
    couplings = par[0]
    cutoff = par[1]

    # starter velocity
    vhalf = v0 + 0.5 * timestep * F(x0, couplings)

    # loop, no time step results recorded
    for _ in range(nsteps - 1):
        x += timestep * vhalf
        k = timestep * F(x, couplings)
        k[np.where(k>cutoff)] = cutoff
        v = vhalf + 0.5 * k
        vhalf += k
    return x, v


def setup(gridlength, dim, initspeed):
    ''' Place particles in 3D with initial velocities
    as two same size clouds opposite each other, shifted in x.
    Cloud particle size increases with increasing grid size.
    '''
    # force zero at r_ij = 3.1
    unit = 3.1
    radius2 = (unit * (gridlength // 2))**2
    bound = gridlength // 2
    xp = unit * np.linspace(-bound, bound, gridlength+1)
    positions = []
    for i in range(gridlength):
        for j in range(gridlength):
            for k in range(gridlength):
                parr = np.array([xp[i], xp[j], xp[k]])
                if np.sum(np.square(parr)) <= radius2: # spherical
                    condition
                    positions.append(parr)

    parr = positions[0]
    for which in range(1, len(positions)):
        parr = np.vstack((parr, positions[which])) # particle list
            shape
```

```
92
93      # shift in x
94      shiftx = np.zeros_like(parr)
95      shiftx += np.array([bound * unit/2, unit/4, 0]) # shift in x,y
            coordinate
96
97      # for two droplets
98      x0 = np.vstack((parr-shiftx, parr+shiftx)) # double particles in
            rows
99      print ('Number␣of␣particles', len(x0))
100
101     # prepare velocities
102     M = x0.shape[0] # number of particles fitting in condition
            sphere
103     v0 = np.zeros((M//2, dim)) # prepared at rest
104
105     # now move the droplets towards each other
106     xspeed = initspeed[0]
107     yspeed = initspeed[1]
108     zspeed = initspeed[2]
109     speed = np.zeros_like(parr)
110     speed += np.array([xspeed, yspeed, zspeed])
111     vel0 = np.vstack((v0 + speed, v0 - speed)) # double particles in
            rows
112     return x0, vel0
113
114
115 def runcalculation(nsteps, step, x0, vel0):
116     ''' Steer the calculation with fixed coupling constants and
            plot setup
117     '''
118     # Force law set up, two coupling constants, all masses equal
119     couplings = np.array([16.5, 158.6])
120     cutoff = 10.0
121     M = x0.shape[0] # number of particles fitting in condition
            sphere
122
123     # Solve N-body problem and plot
124     # 3D plot set up
125     fig = plt.figure()
126     ax = fig.add_subplot(111, projection='3d')
```

```
127     ax.set_xlabel('X [Angstroem]')
128     ax.set_ylabel('Y [Angstroem]')
129     ax.set_zlabel('Z [Angstroem]')
130     ax.set_title('Microdroplets')
131     ax.set_xlim(-30, 30)
132     ax.set_ylim(-25, 25)
133     ax.set_zlim(-25, 25)
134 
135     # Try a few iterations and plot final distribution
136     xval, speedval = verlet(force, x0, vel0, nsteps, step,
            par=(couplings, cutoff))
137     ax.scatter(xval[:M//2, 0], xval[:M//2, 1], xval[:M//2, 2]) # 3D
            scatter plot
138     ax.scatter(xval[M//2:, 0], xval[M//2:, 1], xval[M//2:, 2],
            c='r', marker='^')
139     plt.show()
140 
141 
142 ''' main script
143     Note: a grid of 7 gives 320 particles in total, a grid of 5
            gives 112.
144     Running 320 particles for a few thousand time steps takes
            several
145     minutes. This is not a fast code. Observing what happens to
            particles
146     is much better on a grid of 3 (8 particles) of even 2 (4
            particles),
147     with small step numbers. Those are done in seconds. Value prints
148     also become possible that way if you insert some print commands
149     in the functions.
150 '''
151 # Calculation set up
152 nsteps = 3000 # number of time steps
153 step = 0.0002 # time step size
154 dim = 3 # x, y,z
155 grid = 7 # 7x7x7 grid to fit particles; gives 320 in total
156 initspeed = (20.0 , 0.0, 10.0) # (vx, vy, vz)
157 
158 x0, v0 = setup(grid, dim, initspeed)
159 runcalculation(nsteps, step, x0, v0)
```

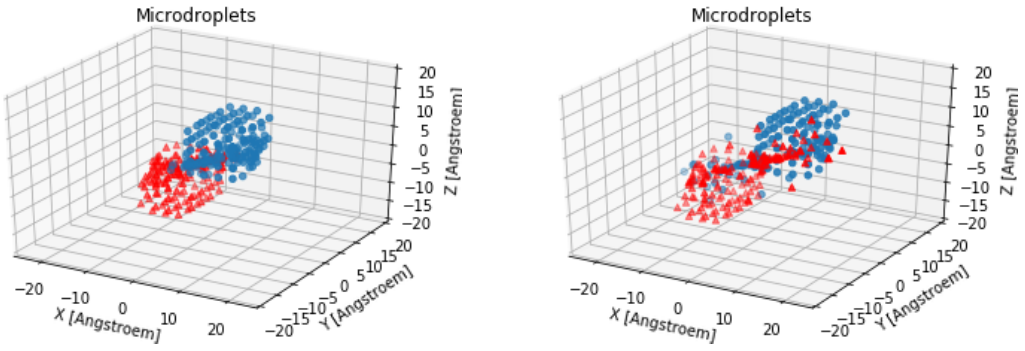

Figure 28: Droplet collision calculation using the Verlet solver from listing 11 with initial speeds at $v_x = 20$ and $v_z = 10$ for 320 particles. Snapshots after 3000 and 4000 time steps are shown.

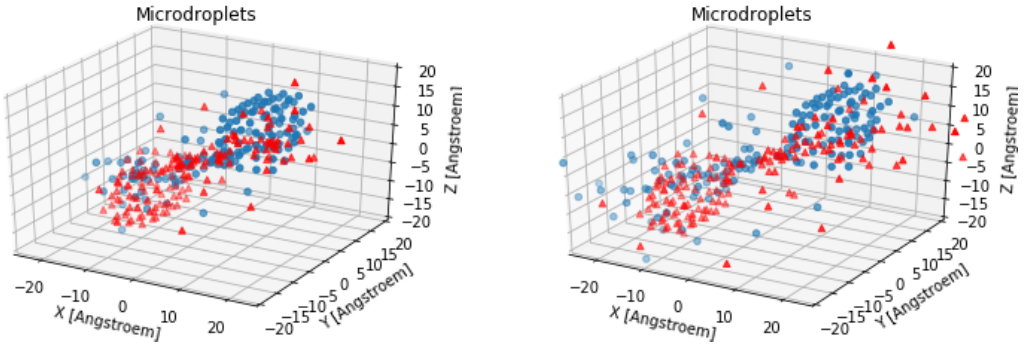

Figure 29: Same as Fig. 28 but for time steps 5000 and 6000. Note how the droplets have sheared off each other.

There is a force regularization built in. That has become necessary since the repulsion term becomes very large very quickly. In fact, in the literature, such regularization of strong repulsive forces is quite normal and smart methods of doing that are known. The simplest, a cut off, is shown here. The reasoning is that molecules have a finite size and should not approach each other too closely, force law or not. Therefore the code sets and enforces a minimum distance, from line 27. Some effort has to be spent on setting the scene, see the function `setup()`. The Verlet solver has not been changed, save for one safety cut off, line 65 (should not be triggered with the distance cap in the force function) to maximum forces.

This code allows us to play with many particles but for speed reasons we should be limited to a few hundred in total. This is then the point to realise that high-performance computing (HPC) has a good reason to exist. Structure formation

calculations employ many millions of 'particles' as do plasma calculations or proper molecular dynamics codes. That is a little beyond Python on its own. Your next year computing module will hence pave the way for you to get your HPC training and open an exciting world for you.

5.4 Summary

This lecture deals with ordinary differential equations with given initial conditions. A convenient solver function from SciPy, `scipy.integrate.solve_ivp()`, is introduced to solve ODE's and systems of coupled ODE's with several examples. One example delves into the prominent topic of chaos in dynamical systems.

The dynamics of multiple particles instead required to pick a manually programmed ODE solver method, here the Verlet method, specialised on solving equations of motion from Newton's law. In contrast to the previous treatment of ODE, here the challenge is not on solving ODE's as such but on solving a lot of them when introducing applications like molecular dynamics or N-body problems in gravity. Again, two examples display how that can be attempted.

Owing to the richness of this topic in physics, much has been left out despite the length of this lecture. The most obvious gap is the treatment of boundary conditions and ODE's in contrast to initial conditions and ODE's. Note that the `solve_ivp()` function will not work with boundary conditions. Their treatment requires more work and different strategies.

Likewise, the large number of different solver methods was not mentioned. The Verlet solver is not necessarily the method of choice for all. Far from it. Also, there are many more types of ODE's of interest in physics that can not be treated as discussed here, for instance stochastic and delay differential equations. The announcement that we don't touch partial differential equations has been made several times but it is worthwhile to refer you to the third year module where you will do that. A lot of physics can be explored after that has been covered.

6 Speeding-up Python and what is a compiler?

This final section of the module should be considered separate from the preceding material since it has got nothing to do with Physics as such. However, more often than not, code modelling a physics problem can become time-consuming to an unreasonable degree. We've had already a few examples that took minutes to run. Such situations can arise quite easily and the question on how to speed up the code becomes more urgent and relevant. Waiting for minutes is quite acceptable in a working day. Waiting for days is a different story. A factor of ten in speed-up can often make a significant difference.

The most important message for this topic, however, is that the code must be working correctly first(!), optimization (aka improving the performance) comes second.

The second most important message is displayed in a nice graphics in the source of most of the material in this section [6]. Improving code performance often follows a curve that rises swiftly by picking up easy gains with easy methods and then saturates at a near constant level when finding more speed gains becomes hard. They label that region quite rightly as 'do not bother'.

Python as a programming language has its limitations when it comes to optimization. At some point, the more efficient decision to take is to switch programming language. Note that by that time, you are supposed to have a working Python code, problem solved but too slow to work with. Translating Python code into another language is quite straightforward and easy once you have a working code. Yes, that other, faster (of course) language is unlikely to feature convenient libraries comparable to Python (some do) but the routines your solution uses can likely be found in that new language and the translation process can continue.

This module will pick up on mostly two convenient methods, (a) enabling parallel execution of your Python program on your multi-CPU (called 'cores') computer and (b) compiling your Python program before running it. There are many more methods and I would recommend looking into the book [6] if you want to learn more. It is full of very valuable advice even for advanced Python programmers. For this short section, however, the two methods mentioned above should suffice. They exemplify important concepts which apply not only to Python but to any programming task.

Finally, it is worth mentioning a standard Python tool at this point, i.e. the `time` library. We will need to measure whether some piece of code is fast or slow or has changed to run faster. There are better, more detailed ways doing that than the standard library, see [6], but simply measuring time will be good enough for us.

6.1 NumPy and vectorization: speed you already use

Vectorization is an important concept in programming, one that many different languages offer since it so incredibly useful and efficient. You've used vectorization since quite some time, NumPy is built on the concept. Translated, vectorization merely means that you apply an operation on all elements of a container automatically. The keyword here is 'automatically'.

In Python, this means the loop which takes one element of the container, applies the operation and stores the result is hidden from you in very fast, compiled code on which NumPy is built (in the C-language, in case you wonder). This process can even go all the way down to the silicon chip level (machine-level code) by using properties of the CPU chip your computer uses. Nevertheless, that is quite irrelevant for us here. Point is, vectorization already makes operations on NumPy container really fast without you having to do anything other than using NumPy.

This is the second time this word, compiler, features. In short, more below, it means a computer program that translates your code into machine-code, something your CPU chip, the heart of your computer, actually understands. Yes, your computer does not speak Python (or C), all it knows are bits, how to manipulate them and move them between areas of silicon which process bits and areas which store them. Bits are the basic on/off pieces of information, switches in form of transistors, on which all computing is based. Fortunately, you don't really need to know much about that since you are highly unlikely to ever needing to write a compiler (translator) yourself [4].

With the keyword 'compiler' out of the way, it is only fair to state what Python actually is, an 'interpreted' language. The program 'interpreting' Python allows you to type single Python commands, press enter and something happens, or write a code and run it and the interpreter program reads and executes each of your lines of code one after the other. This is the key difference to a compiled code. The interpreter effectively 'runs' your program, you run your program inside another. Clearly, this renders execution slower than your instructions reaching the processor immediately, already in a form it can digest directly.

The program running your Python program features the GIL, the Global Interpreter Lock. The GIL checks all your Python commands and essentially makes Python code safe but slow. The NumPy library disables the GIL for all its operations and containers. However, whenever it needs to come back to Python code, the GIL is on again. One reason why it isn't advised to mix Python and NumPy too

[4] All the fuss about quantum computers results from the basic unit being replaced from bits to quantum bits which offers a completely new conceptual foundation for computing. PX447 will teach you more about that.

much. Rather work with NumPy for as much as possible and only get back using Python once the number crunching is over. Functions in SciPy behave similarly but are more tolerant when it comes to mixing Python and NumPy in using SciPy functions. That is not helpful for optimization but a price worth paying in most cases.

Ok, enough text, let's have a look at an example.

Listing 12: Demonstration of different vector loops and their speed.

```
import numpy as np

def norm_square_numpy_dot(vector):
    '''
    914 mus +- 42.7 mus per loop (mean +- std. dev. of 7 runs, 1000
        loops each)

    '''
    return np.dot(vector, vector)

def norm_square_numpy(vector):
    '''
    2.91 ms +- 444 mus per loop (mean +- std. dev. of 7 runs, 100
        loops each)

    '''
    return np.sum(vector * vector)

def norm_square_list(vector):
    '''
    106 ms +- 2.84 ms per loop (mean +- std. dev. of 7 runs, 10
        loops each)

    '''
    norm = 0
    for entry in vector:
        norm += entry * entry
    return norm

def norm_square_list_comprehension(vector):
    '''
    145 ms +- 953 mus per loop (mean +- std. dev. of 7 runs, 10
        loops each)
```

```
30
31      '''
32      return sum([v * v for v in vector])
33
34  vector = list(range(1000000))
35  vector_np = np.arange(1000000)
36
37  # %timeit norm_square_list_comprehension(vector)
```

As you can see in listing 12, several different function, all doing the same sum of square operation, are called with a container with 10^6 entries. Two functions use pure Python, lines 17 and 27. The list comprehension, line 27, is the slowest way to calculate the sum of many squared numbers. The reason is that each squared number requires its own storage operation (in a list) and only then the 'sum' operation is called.

In line 17 you can see how that can be made a little faster in Python by summing 'in memory', i.e. drop the individual storage of numbers but square and add immediately, called 'in-place'. Admittedly the speed-up isn't all that much but still, a gain of 37%. The NumPy functions using vectorization are considerably faster. A speed-up of a factor of 36.5 is possible. That would turn an hour of run-time into 99 s, so a really rather useful speed-up. The integrated 'dot-product' operation, line 8, handles squaring and summing internally (compiled code!) and 'in-place' and is consequently even faster. Now you look at a speed-up factor beyond 100. Now that is useful in all circumstances.

Note that this detailed timing information was obtained using the ipython 'magic' timeit command in Spyder. This is a special case and not used below but nice if you can.

In summary, use NumPy as much as possible and keep in mind that in-place operations are faster than storing and operating subsequently.

6.2 The Python multiprocessing library

Parallel processing is ubiquitous in High-Performance Computing (HPC) and you'll learn a lot about it in the fourth-year computing module. However, as a programming paradigm, it applies only to a subset of problems.

Some problem solutions simply can not be easily divided in pieces that can be processed in parallel. This is typically the case if one computation depends on the previous computation as input like a recursive algorithm. Often the effort required turning a problem solution into another that can be implemented in parallel is not

worth it and serial computation is the way forward.

If, however, a problem is amenable to parallel implementation then the speed-up can be significant. Your PC most likely features two to four CPUs, called cores. If each core can work on your problem independently, you could gain up to a factor of four. Useful but not revolutionary. Large HPC facilities can give you hundreds of cores and that clearly changes the game.

You might also have heard of GPU (graphics processing units) processing. These chips usually run the graphics on your PC. They tend to offer massive parallel processing units, thousands, even for single PC machines. Nevertheless, the price one pays for using them is that these many cores can only do extremely simple operations and the main effort has to go to shuffle the data to and from the GPU, i.e. input to GPU and output back to CPU is the main bottleneck. Likewise, phrasing your problem solution such that it can be processed by a GPU core is not easy and often impossible. In short, we won't look at using GPUs. That effort would require a formidable justification.

Python doesn't use multiple cores (CPU's) by default. The multiprocessing library enables you to instruct Python to use as many processors as your machine offers at once. You should not exceed that number when telling multiprocessing how many cores you wish to use. One fundamental observation to keep in mind is that requesting the use of n-cores can only give you up to a factor n gain in speed. The reason is the communication overhead of any parallel processing system. That overhead renders parallel programming complex in the sense that you have to figure out whether the effort is worth it. Not every problem solution benefits from parallel execution.

The best possible case is a problem that is called *embarrassingly parallel*. If we can have multiple Python programs running, all solving the same problem without communicating with each other, i.e. independently(!), then such a code is called embarrassingly parallel. Communication between separate processes, i.e. parallel running programs, is called sharing state and is to be avoided if at all possible. All the complexities of parallel programming revolve around sharing state.

Sharing state is possible and often required. The multiprocessing library offers tools to do just that but even with such simple tools, sharing is to be avoided at all cost. In short, it slows down speed gains considerably, renders fault finding very difficult and can even crash your program for reasons that require more research effort than it's worth using them without a lot of experience. This module will stay away from sharing state and only consider embarrassingly parallel tasks.

As a final preliminary, we need to clarify the terms 'process' and 'thread'. Your single CPU chip can launch calculations on separate threads, setting up a form of parallel processing. These threads are not running on independent silicon chips but

share resources on a single chip, i.e. memory, internal and access to external. That's not good, not what you want when setting up parallel Python code. The reason is, again, the GIL, see above. One process, many threads doesn't help if the GIL runs your Python program since the GIL is one process(!). The threads all talk to one and only one GIL on that single processor.

If you spawn (launch) multiple processes, however, each starts its own GIL! In this case, you really run several Python programs in parallel hence telling the multiprocessing library to launch processes is the way to go.

The easiest way to do that is the `Pool` in multiprocessing. The simplest example is in the multiprocessing module *manual*.

Listing 13: Simple Pool example.

```
1  """
2  simple Pool example
3  """
4
5  from multiprocessing import Pool
6
7  def f(x):
8      ''' square input '''
9      return x*x
10
11 if __name__ == '__main__':
12     ''' This main clause is important, see text '''
13     datalist = [1,2,3,4]
14     with Pool(4) as pool:
15         print(pool.map(f, datalist))
```

Line 5 imports `Pool` from multiprocessing. All the action to run in parallel is the function `f(x)` in lines 7-9. Line 13 sets up the data to be processed. Note that for using the **map** function of a pool, the data input must be an iterable container like a Python List. In line 14 the Pool is created safely, in a conditional `with` statement, i.e. inside that block, the Pool exists and quits automatically (and safely), once the block ends. Here the Pool is created for a number of 4 processes, i.e. one for each CPU my little PC has access to. Line 15 then starts the action. It tells the pool to start processes which each takes one(!) entry in the `datalist` and calls the function `f(x)`.

Once all(!) the entries in the list have been processed, the result is printed to screen. This is important to note, a property of the **map** function. There are other function that can return results earlier once a single process finished. For large data

to process, this is often a faster way to run processes.

Line 11 is important for all multiprocessing operations. The reason is that many independent processes are being started. That means they start as main programs like starting any Python program from a terminal prompt by importing your code. Line 11 takes care of setting up the program on each processor correctly, each gets its own GIL (interpreter) to run the code from the start with all the right environment. If you leave out line 11, anything can happen. Usually, the code hangs on the last process and doesn't know how to end. Worst case, you need to reboot your entire machine. Let's not try that.

One classic example for an embarrassingly parallel problem is a Monte-Carlo simulation or integration. Let's calculate π as a Monte Carlo integration example. Have a look at the pure Python code for this to remind yourself how that can be done. It is not efficient as given but instructive. This an educational text after all.

Listing 14: Calculate Pi by M-C integration.

```
1  """
2  Calculate Pi by M-C integration
3  """
4
5  from random import random
6  from math import sqrt
7  from timeit import default_timer as timer
8
9
10 def pi(n):
11     '''
12     Parameters
13     ----------
14     n : int
15         number of simulations.
16
17     Returns
18     -------
19     float
20         estimate for pi.
21     Takes about 85 s.
22     '''
23     count = 0
24
25     for _ in range(n):
```

```
26
27          x, y = random(), random()
28
29          r = sqrt(pow(x, 2) + pow(y, 2))
30
31          if r < 1:
32              count += 1
33
34      return 4 * count / n
35
36
37  start = timer()
38  pi_est = pi(100_000_000)
39  end = timer()
40
41  print(f'elapsed time: {end - start}')
42  print(f'pi estimate: {pi_est}')
```

and here is one version that solves the same task but with the help of parallel processing.

Listing 15: Calculate Pi by M-C integration in parallel.

```
1   """
2   Calculate Pi by M-C integration in parallel
3   """
4
5   import random
6   from multiprocessing import Pool, cpu_count
7   from math import sqrt
8   from timeit import default_timer as timer
9
10
11  def pi_part(n):
12      '''
13      Parameters
14      ----------
15      n : int
16          number of simulation.
17
18      Returns
19      -------
```

```
20      count : int
21          number of hits in quarter-circle.
22
23      This function can be called in parallel with a fraction
24      of the total number of required simulations.
25      '''
26      print(n)
27
28      count = 0
29
30      for _ in range(int(n)):
31
32          x, y = random.random(), random.random()
33
34          r = sqrt(pow(x, 2) + pow(y, 2))
35
36          if r < 1:
37              count += 1
38
39      return count
40
41
42  def main():
43      '''
44      Returns
45      -------
46      None.
47
48      Main function to run simulation
49      With 4 cores: 26 s
50      '''
51      start = timer()
52
53      np = cpu_count()
54      print(f'You␣have␣{np}␣cores')
55
56      n = 100_000_000
57
58      part_count = [n/np for i in range(np)]
59
60      with Pool(processes=np) as pool:
```

```
61
62          count = pool.map(pi_part, part_count)
63          pi_est = sum(count) / (n * 1.0) * 4
64
65          end = timer()
66
67          print(f'elapsed time: {end - start}')
68          print(f'pi estimate: {pi_est}')
69
70  if __name__=='__main__':
71      # Note the requirement to run as an executable, not in the
            interpreter
72      # for multiprocessing to work properly.
73      main()
```

In listing 15, note how line 58 sets up a short list of numbers (with the number of processors as number of entries) as input data for parallel processes spawned by the Pool. The function called for each process is the `pi_part(n)` function in line 62, taking the number of Monte-Carlo attempts to simulate. Line 53 can be useful if you don't know or don't wish to assume the number of CPU the PC might have. The multiprocessing function `cpu_count()` finds out that number. Execution time for code (on my machine) is down from 85 s to 26 s, a quite typical factor 3.3 gain in speed.

Well, so much for that. Now let's speed up the integration properly. Find below the Monte-Carlo integration using NumPy.

Listing 16: Calculate Pi by M-C integration using NumPy.

```
1   """
2   Calculate Pi by M-C integration with NumPy
3   """
4
5   import numpy as np
6   from timeit import default_timer as timer
7
8
9   def pi(n):
10      '''
11      Parameters
12      ----------
13      n : int
14          number of simulations.
```

```
15
16      Returns
17      -------
18      float
19          estimate for pi.
20      Takes about 6 s.
21      '''
22      np.random.seed()
23      xs = np.random.uniform(0,1,n)
24      ys = np.random.uniform(0,1,n)
25      count = np.count_nonzero(np.sqrt(xs*xs+ys*ys)<=1)
26      return 4 * count / n
27
28
29  start = timer()
30  pi_est = pi(100_000_000)
31  end = timer()
32
33  print(f'elapsed time: {end - start}')
34  print(f'pi estimate: {pi_est}')
```

and then execute that NumPy function in parallel.

Listing 17: Calculate Pi by M-C integration with NumPy in parallel.

```
1   """
2   Calculate Pi by M-C integration with NumPy in parallel
3   """
4
5   from multiprocessing import Pool, cpu_count
6   import numpy as np
7   from timeit import default_timer as timer
8
9
10  def pi_part(n):
11      '''
12      Parameters
13      ----------
14      n : int
15          number of simulation.
16
17      Returns
```

```
18      -------
19      count : int
20          number of hits in quarter-circle.
21
22      This function can be called in parallel with a fraction
23      of the total number of required simulations.
24      '''
25      print(n)
26      np.random.seed()
27      xs = np.random.random(n)
28      ys = np.random.random(n)
29      count = np.count_nonzero(np.sqrt(xs*xs+ys*ys)<=1)
30
31      return count
32
33
34  def main():
35      '''
36      Returns
37      -------
38      None.
39
40      Main function to run simulation
41      With 4 cores: 2 s
42      '''
43      start = timer()
44
45      np = cpu_count()
46      print(f'You␣have␣{np}␣cores')
47
48      n = 100_000_000
49
50      part_count = [int(n/np) for i in range(np)]
51
52      with Pool(processes=np) as pool:
53
54          count = pool.map(pi_part, part_count)
55          pi_est = sum(count) / (n * 1.0) * 4
56
57          end = timer()
58
```

```
59          print(f'elapsed time: {end - start}')
60          print(f'pi estimate: {pi_est}')
61
62  if __name__=='__main__':
63      # Note the requirement to run as an executable, not in the
            interpreter
64      # for multiprocessing to work properly.
65      main()
```

As you can see, we reduce the execution time from 26 s to 6 s in a simple step, just by programming our function using NumPy throughout and then to 2 s by executing that in parallel. In total, this is a gain in speed from 85 s to 2 s, a factor of over 40, all with relatively little effort.

One detail worth pointing out is line 26 in listing 17. Calling the seed function as `np.random.seed()` is important at the start of the function so that each of the independent processes creates its own random number sequence. Listing 15 did not need to do that since the standard Python random module seeds itself automatically for every process, whereas `np.random` does not do that. Securing independent random number sequences for parallel processes is important and missing that is a notorious mistake to make when starting with parallel programming.

6.3 Compiling Python

As mentioned in the first subsection, a compiler is a program that translates your program directly into machine code which then runs without invoking the Python interpreter. The potential gains can be substantial, an order of magnitude in gain of speed wouldn't be unusual when compiling pure Python code. Calls to external libraries like SciPy and NumPy are unlikely to see any speed-up. These are already compiled, mostly, as are many other external libraries in use for Python.

Invoking a compiler for an interpreted language like Python is always a bit of a compromise and can bring substantial complexities. It will not always work and can be quite frustrating at times. Where it does work though, it works very well indeed and is well worth the effort.

One way to compile Python is summarised by a Python package called `Cython` (short for compiling to the C-language - and from there to machine code). We will not do that since it requires knowledge of the C-language. Why C? The Python language is written in C, would be the short answer.

Instead, we will have a brief look at a project called `Numba`, a Just-In-Time (JIT) compiler. The benefit of a JIT is that you as the user don't need to start and

run the compiler program but Python itself will do that for you. That is different to compiling a C program where you write the code, compile separately, and run separately. Using a JIT, your normal use of Python stays nearly the same. JIT means that for the very first time that you run your Python program with compile instructions, see below, the run time is longer since it takes time to compile. Already the next run of your program will then show the full speed of the compiled code, assuming the compilation actually worked.

Now to using *Numba*. This is a Python package in development. It has not yet reached a version 1.0 but reached a level of stability that it can be used. It comes together with the Anaconda package as standard but could be installed quite easily, see the link, in case you don't have it yet.

Not having reached version 1.0 means that not all intended features are yet available. As a package, the compiler aims at allowing to compile all standard Python code and recognize all of NumPy. The latter isn't quite ready yet but most is. It also allows in a very minimal but useful way to include parallel execution of compiled code. Let's inspect a first example.

Listing 18: Demo Numba compilation.

```
"""
Numba starter example
"""

from numba import jit
import numpy as np
import time

@jit(nopython=True)
def go_fast(a): # Function is compiled and runs in machine code
    trace = 0.0
    for i in range(a.shape[0]):
        trace += np.tanh(a[i, i])
    return a + trace

x = np.arange(100).reshape(10, 10)

# DO NOT REPORT THIS... COMPILATION TIME IS INCLUDED IN THE
    EXECUTION TIME!
start = time.time()
go_fast(x)
end = time.time()
```

```
22 print("Elapsed␣(with␣compilation)␣=␣%s" % (end - start))
23
24 # NOW THE FUNCTION IS COMPILED, RE-TIME IT EXECUTING FROM CACHE
25 start = time.time()
26 go_fast(x)
27 end = time.time()
28 print("Elapsed␣(after␣compilation)␣=␣%s" % (end - start))
```

Scan the listing 18 and note that there are only 2 instructions worth mentioning, line 5 and 9. The rest is all straightforward. Line 5 imports the `jit` compiler from Numba and line 9 uses it. That usage in line 9 looks a little unusual. The 'at'-sign in front and the placement directly above a function declaration is called a decorator in Python. This instructs Numba to compile the function adorned with that decorator. The effect of the decorator does not extend any further than the function. It stops where the function block ends.

So, there is a `@jit(nopython=True)`. The manual says, quite rightly, that compiling with the option 'nopython' means compile all the following to C and thus to machine code. Don't retain any Python code. This instruction happens so often that it has its own decorator `@njit` which we could have imported and used instead.

Nevertheless, it is useful to show the `jit` decorator with options since there are many more options, some very useful and important indeed. Given how powerful compilation can be for Python codes and the already rich features of Numba, I'd recommend to start reading into the topic with examples and tips from the manual *here*. Use the final example in this section below as a guide, yet another version of the calculate Pi example, compiled and running in parallel. As you can see, on my machine I am now down to a run-time of under 0.5 s.

Listing 19: Calculate Pi by M-C integration with Numba and parallel loop.

```
1 """
2 Calculate Pi by M-C integration, compiled
3 """
4
5 from random import random
6 from math import sqrt
7 from timeit import default_timer as timer
8 from numba import njit, int32, float64, prange
9
10 @njit(float64(int32), parallel=True)
11 def pi(n):
12     '''
13     Parameters
```

```
14      ----------
15      n : int
16          number of simulation.
17
18      Returns
19      -------
20      float
21          estimate for pi.
22
23      Compiled version, takes about 1.8 s
24      Compiled, parallel, takes about 0.5 s
25      '''
26      count = 0
27
28      for _ in prange(n):
29
30          x, y = random(), random()
31
32          r = sqrt(pow(x, 2) + pow(y, 2))
33
34          if r < 1:
35              count += 1
36
37      return 4 * count / n
38
39  start = timer()
40  pi_est = pi(100_000_000)
41  end = timer()
42
43  print(f'elapsed time: {end - start}')
44  print(f'pi estimate: {pi_est}')
```

As you can see in listing 19, all it took was to import from Numba and decorate the original pure Python function. When doing that, run-time reduces to just under 2 s, very similar to the NumPy result and that is no coincidence since the Python code does effectively the same thing as the NumPy code but now compiled and hence similarly fast as the NumPy version.

The final change is then the keyword option in line 10 (absent for the statement above), switching on parallel processing. That parallel processing is very specific in Numba only for loops! That can be seen and must be implemented as such when you look at line 28. The Python **range()** function changes to `prange()` which is a

Numba function ('p' stands for parallel). Now the Python loop is not only compiled but also executed such that all the available cores of your machine run an about equal part of the loop. So, very specific but also very useful and easy to implement in your code.

A final observation to mention is in line 10, the keywords `int32, float64`. These are types, known to Numba and imported for use. When used directly after the decorator keyword like here, they specify the function signature and help the compiler to understand (and fix) how to compile your function. The word signature here means the output type and input types are given and not left to the compiler to guess from context. The function `pi(n)` in line 11 expects therefore a 32-bit integer number as input (don't worry about the 32-bit, it is still an integer that can become very big indeed) and outputs (returns) a long (64 bit) floating point number, a type often called 'double'. That information fully describes the function for the compiler.

You see, compiling even Python will eventually require you to understand a bit more about what is going on in a computer when you program and run your code. Python is such an efficient and productive language since you can mostly ignore all that and nearly just 'talk' to the computer. It'll sort out all these types and signature itself. The closer you get to other languages, however, by compilation or switching language, you'll learn that such details are important and have heir advantages (and plenty of disadvantages). Speed is certainly one aspect that greater detailed knowledge brings as compilation clearly shows. Numba tries to shield you from that complexity quite well. Nevertheless, don't expect simply decorating what you like and it'll sort itself out to be a viable strategy. The Numba compiler is bound to throw many errors even when you don't expect them and making sense out of compiler warnings is a life-long occupation for masochists.

A final tip before ending this lecture module then. Most NumPy functions are being understood by Numba and that can only get better. There is at this moment in time, no way to use Numba and SciPy functions! A first, very preliminary project in that direction has started but discard it for now. SciPy and Numba don't understand each other and Numba will throw unpleasant error messages. Apart from that, if you have a pure Python function, even a small one or a Python loop to speed up, Numba is an easy and very efficient way to speed up your code considerably and is highly recommended.

A Quick Start

It is expected that most of your work will take directly on the Notable cloud solution system which hosts almost all assignments (not the final projects). You can edit and run your solutions on that solution without any Python on your own machine. Final solutions must be submitted from Notable hence working there directly makes sense. Nevertheless, working on your own machine (downloading from and uploading to Notable is possible) enables somewhat more flexibility.

Working on an ITS supplied PC running Windows, you should find an application folder called **Anaconda**. The application to run with double-click is called **Spyder**. Another application of relevance is called **Jupyter-notebook**. This is the application you'll need to test and submit assignment. This application runs in your Browser and allows easy file exchange in form of complete '.ipynb' type files. You will have to download files of that format from Moodle for your assignments and upload such files for your answers.

Running the Spyder application, what you want to see on your screen is the **Editor** (default location is on the left of your Spyder window) and the **Console** (on the right).

Next step is to set your work directory location so that Python knows where to look for your own programs and where to save what you want such that you can find it again later. You can set the work directory either with the preferences menu or with a button on the top right of the Spyder window. Easiest is to create a new folder. Call it 'workdir' or something else recognisable and set the default location to that folder.

That is all you need to get started. You can download the existing source codes from the Moodle page (for the final project assignment) or Notable system and save them in your work directory for instance or type them all yourself and save them there as you work through the script.

If you work with your private machine and have Python installed on it then the same preparations apply. You'll need a comfortable editor program of your choice, ideally one that understands Python, i.e. automatically takes care of spaces and maybe even colouring of source code. Secondly you'll need a console or terminal window from where to launch or run the Python programs and see the output. A popular alternative to Spyder is the **MS Visual Studio Code** application.

References

[1] M. Newman, **Computational Physics**, CreateSpace Independent Publishing Platform, 2nd ed., 2013.

[2] Ch. Hill, **Learning scientific programming with Python**, Cambridge University Press, Cambridge, 2016.

[3] S.W. Smith, **The scientist and engineer's guide to digital signal processing**, Cal. Tech. Publ., 1999, `http://www.dspguide.com/`

[4] E. Ayars, **Computational Physics with Python**, Cal. State Univ. Chico, 2013, `http://www.fizika.unios.hr/rf/wp-content/uploads/sites/67/2011/02/CPwP.pdf`

[5] D. Greenspan, **Numerical solution of ordinary differential equations**, Wiley-VCH, Weinheim, Germany, 2006

[6] M. Gorelick and I. Ozsvald, **High Performance Python**, O'Reilly, 2020, 2nd. ed.

[7] Python documentation: `http://www.python.org/doc/`

[8] The Python Tutorial: `https://docs.python.org/2/tutorial/introduction.html`

[9] Scientific Python documentation: `http://docs.scipy.org/doc/scipy/reference/`

[10] Scientific Python tutorial: `http://scipy-lectures.github.io/index.html`