AIM manual

Kim E. van Adrichem

May 10, 2022

Preface

Before you lies the manual to AIM, version 1.0. This program is designed to generate a time-dependent Hamiltonian and dipoles from trajectories calculated using Molecular Dynamics (MD) software. These files contain all information required for calculating the FTIR and two-dimensional infrared (2D-IR) spectrum of the system modelled in the MD simulation. Note that it does NOT generate the entire spectral range, but only part of it. This part can be specified by the user. The generated output files have the correct format to immediately be used as input for the NISE software[1], which is the intended use of this program.

This program is written with multiple types of users in mind. One of the goals was to make it as accessible and easy-to-use as possible, without any unnecessary details. However, at the same time, it was always supposed to be a very powerful, flexible and versatile tool, with more functionality than one could ever wish for. Therefore, do not be scared by all the options in this manual: you absolutely do not need them all right away. Simultaneously, do not be disappointed by the simplicity of the first calculation: there is much more functionality for you to discover!

The original goal of this program was to calculate the IR spectrum of the amide-I stretch for proteins. Therefore, this still makes up the core of the program. However, due to interest shown by the community, the program has been extended beyond both the amide-I stretch and proteins: with the use of the extra maps functionality (see section 5), a wide range of IR-active vibrations can be added without delving into the core code itself.

If you want to get started on your first calculation immediately, section 1 is just right for you. It walks you through the installation process and helps with performing the first calculations. Section 2 contains the documentation on the input files used to specify the details of the calculation. Section 3 is similar, giving all details on the output files. Advanced users might want to look through section 4, as it explains the source files used by AIM. These files, among other things, contain the parameters of a variety of methods, including maps. The theory behind the method is covered in section 6, and section 7 helps with commonly occurring problems.

Contents

Chapter 1

Getting started

1.1 Basic installation

The program consists of multiple files and folders in a main folder titled AIM, and some extra files in another folder named support_files. Without changing the internal structure, the entire AIM folder can be placed in any desirable location. Recommended is a location with a short path that contains no spaces. No further installation or setup is required for the basic version itself: just having the folder with all files on your computer is sufficient. The code can run on both Windows and MacOS/Linux (here referred to as Unix).

As the program is written in python, a python 3 installation is required in order for the program to run. It also requires installation of the 3rd party modules 'MDanalysis'[2,3], 'numba'[4] and 'numpy'[5]. Instructions on how to install those are present on their websites. The code has been tested using the following versions: python - 3.8.3, MDanalysis - version 1.0.0 and 2.0.0, numba - version 0.50.1, numpy - version 1.18.5.

Considering AIM depends on this python installation, a script has been included to confirm the installation was successful (and to confirm all modules are of a recent enough version). It is called python_test.py, and can be found in the support_files folder. To run it, open (in the case of Windows) a command prompt, or (in the case of Unix) a terminal. There, navigate to the support files folder, such that when using a command like `dir` (windows) or `ls` (Unix), the script python_test.py is listed. Once there, run the following command: `python python_test.py`. The script will run and check whether your environment has been set-up successfully. If there are any issues, the script will report them. You should fix any reported problems before attempting to run AIM. If everything went well, the script will print the message that you can now go ahead and use AIM.

Having the python environment set up using the previously mentioned script, following the rest of this subsection isn't necessary (it's the manual way of doing what that script did). It is here for those who would also like to check manually.

Open (in the case of Windows) a command prompt, or (in the case of Unix) a terminal, in case you closed it. Using that terminal/command prompt, navigate to the installation folder of AIM, such that when using a command like `dir` (windows) or `ls` (Unix), the script AIM.py is listed. Before running the program, the python environment can be tested by typing `python`. This should open a python environment. Most important is to make sure it lists a python version starting with 3.x.x, like 3.8.6. If it starts with a 2 (like 2.7.16), there are two options:

- For your OS, look up how to change this default setting. Depending on OS, it will take either a few mouse-clicks, or commands. Although slightly more difficult, this is the recommended way.
- Every time the manual prescribes using the command `python`, use `python3` instead. While this does work for running the initial script, more advanced functionality requires that `python` refers to a python-3 installation. Therefore, this option might be simpler, but it is not recommended.

In both cases, you can leave the python environment (indicated with `>>>` at the beginning of the command line) by typing `quit()`.

Similarly, if one wants to check whether the 3rd party modules 'MDanalysis', 'numba' and 'numpy' have been installed correctly, one can open the python environment by typing `python`, followed by

`import MDanalysis`, `import numba`, or `import numpy`. When installed correctly, nothing will be printed. However, when not installed correctly, the environment will return the error message "ImportError: No module named ...".

1.2 Basic installation - plan B

As the title says, this is plan B. For technical reasons, the method above is preferred. But if you cannot get AIM to run without throwing an error, give this a go!

In rare occasions, running the script python_test.py works, but you get errors when trying to run the demo mode of AIM. The most likely explanation is the installed versions of modules. While, if you passed the test, all individual modules have functioning versions, it can happen that certain versions of modules do not go together. The easiest solution to this, is to take a set of versions that are known to work together. This can be done through the following steps:

Firstly, make sure you have the correct version of python installed. To check which version you have, you can either check the output of the python_test.py script, or type 'python' in the command line. For more details on both of these options, check section 1.1.

In the support files folder (you've encountered this one in section 1.1), there is one or more files starting with 'requirements', and ending in '.txt'. Ideally, you want your entire 3-part version number of your python installation to match that of one of the requirements files, to make absolutely sure you are running the exact same tested version. In most cases, matching the first 2 parts (so, for example, using 3.8.6 instead of 3.8.3) should also work. But if not, make sure to match it perfectly!

If you've got a matching python version and requirements file, it is time to install the modules. I'd recommend using the python module venv (this comes with the python installation), as it makes experimenting with module versions much easier. It is not a must, it can work without, too.

When you're ready to install the modules, navigate to the location of the AIM files, so that the `ls` (unix) or `dir` (windows) command lists the requirements file. Then, run the following command: `python -m pip install -r requirements3-x-x.txt` Make sure to replace the x's with the version you want to use! This will install the exact versions listed in the file. If the installation gives an error, check if you get the (usually yellow-colored) message that there is a new version of pip available. Use their suggestion to upgrade pip, and try again. If you get an error still (either when installing, or when running the AIM demo mode), please, let us know!

1.3 File structure

Inside the main AIM folder, three folders are present: extra_maps, sourcecode and sourcefiles. The extra_maps folder stores all maps that the user wants to use. They can be added/removed at will, and add to the modularity of the program. More information on them can be found in section 5. The sourcecode folder contains all python scripts used by the program. For more information, read the dev manual. Finally, the sourcefiles folder contains all data the program needs to run. More information can be found in section 4.

Besides these three folders, the main folder also contains a few files: The most important one of these is the main python script AIM.py. This is the script that is executed when calling the program. It also contains an example input file called input_parameters_example.txt, on which more information can be found in section 2.2.

Finally, apart from the main AIM folder, there is another folder, called support_files. This folder contains files that may be useful when using AIM, but are not vital to run the program (for example, this manual).

1.4 Running the first calculation

After installation, all files required for the first calculation are present. In general, a calculation requires 3 main files: An input file to tell the program what to calculate exactly, and two data files. These two data files are generated using molecular dynamics software, like GROMACS, CHARMM, or AMBER. One of these files is the topology file, which contains information on all atoms present in the system. The other is the trajectory, which contains the position and velocity of each atom at each calculated frame.

To start, open (in the case of Windows) a command prompt, or (in the case of Unix) a terminal. There, navigate to the installation folder of AIM, such that when using a command like `dir` (windows) or `ls` (Unix), the script AIM.py is listed.

After having verified that the python environment functions correctly (see section 1.1), the very first calculation can be run by simply calling the main script as follows: `python AIM.py`. The calculation should take under a minute. A few messages are printed to the command prompt/terminal, all of which will be explained later. Most important to check here is the very last message. It should read "Demo Mode: If you read this, the program encountered no problems, and ran successfully!".

It can happen very occasionally that instead, python throws up an error. This most likely has to do with the versions of the modules used. If you get an error, please take another look at section 1.1, to make sure that the test script is happy with your versions. If it is, but you do get the error, have a look at the procedure in section 1.2. If that still doesn't work, please, feel free to contact us!

If all went well, you'll see that all printed lines start with the text "Demo Mode:". The very first line shows you why: no input parameter file was given. The next line indicates whether or not a functioning c-library was found. This will become relevant soon (see section 1.7.1), but can be ignored for now. Next, the main header is printed. This means that the actual calculation is starting. The header is followed by a print of the version number of the program.

As the program tries to identify the system itself, it is wise to check whether it has been interpreted correctly. The files that came with the installation contain the trajectory and topology of a short MD run on the lysosome protein (code 1AKI on RCSB.org). Details on how these were generated (and how to generate your own) can be found in section 2.1. For now, it is important to know that this protein consists of a single (linear) chain of 129 residues. The program confirms this: in total, only 1 chain has been detected, and it should have been identified to be a linear chain of 129 residues.

After having identified the properties of the system, for each frame, the Hamiltonian and transition dipole moments of the amide groups are calculated. The given trajectory consists of 51 frames (labelled 0 through 50), so the program gives fourteen updates: once every frame until frame 10, then once every 10 frames. These updates allow the user to keep track of the program, and give the user an estimate how much longer the calculation will take. This estimation uses that typically, all frames are computationally equally costly. The exception to this rule is the first frame. This frame takes longer than any following it, resulting in an overestimation of remaining time. This estimate quickly decreases within the first frames - after a few dozen frames, the estimate is usually quite accurate.

When the calculation is done, the program prints some information on the system. First, it mentions how many protein chains were identified (the intended use case of AIM are proteins, although it can also treat systems without them), and then, for each chain, prints how many of it's groups were considered during the calculations. Then, it sums up all these numbers, and lists the groups not associated with a protein. Finally, for the entire system, it prints how many groups there are, and how many were considered for the calculation. Next, it prints how many frames were available for calculation, and how many were actually used. After that, the program prints how long the calculation took, as walltime. Last, all references corresponding to the maps used for that specific calculation are printed. This list is automatically updated/changed with each calculation.

1.5 Looking at the first results

This first run should have generated 5 files: one in the log folder, and four in the output folder. The location and name of these files can be changed, but that was not done this time. First, lets take a look at the log file. Its first section, "Input file" is empty, as the calculation ran without an input file. "Checking parameters" is dedicated to reporting anything noteworthy on the supplied parameters. If any parameter is wrong, missing, or contradicting another, it is mentioned here. As there is no input file used, there is not much that contradicts, so there is only one noteworthy message. It is the same one that got printed to the command prompt/terminal earlier: about a c library. Don't worry about it, that'll come later.

As the program solves quite some input parameter problems by changing/adding parameters (see section 2.2), the log-file also prints the final set of parameters. The meaning of any of these can be found in section 2.2. Then, the same messages that were printed to the command prompt/terminal are listed in the log file, plus some more. If ever you need to see these messages again, this is where to find them.

Of the four files in the output folder, the one ending in "Parameters.txt" is easiest to understand. This file contains all the input parameters as used in the calculation (after fixing any problems with the initial set), and can therefore be used for redoing the calculation if so desired. The other three files in the

output folder, those ending in "AtomPos.txt", "Dipoles.txt" and "Hamiltonian.txt", contain the actual data computed by the program. Details on the output files can be found in section 3. For now, the results can be checked against the following:

- The first few items in Hamiltonian.txt are:
  `0 1621.07502 0.49422 0.258176 0.197284 0.194376`
- The first few items in Dipoles.txt are:
  `0 1.655915 -1.889241 2.422885 -1.224684 0.414298`
- The first few items in AtomPos.txt are:
  `0 41.77 41.84 41.810001 42.000004 41.830002`

Due to internal rounding by the computer, the last digit(s) could differ, but the first 4 decimals should always match. Do not worry, the spectral calculations are not sensitive to these rounding errors.

1.6 Specifying parameters

In order for this demo to work, a lot of settings have been pre-selected. In order to use the program, you will have to learn how to set them yourself. This is actually pretty easy: make an empty text file using your favourite text editor, and save it. The name and location of the file do not matter. For easy description, lets say the file is named "input_parameters.txt", and saved in the same folder as the python script called earlier. An example of such a file is included with the program. In the command line, the command used earlier will now get an extra argument: `python AIM.py input_parameters.txt`. When the files are in different folders, make sure that they are referenced correctly. It does not matter in which folder the command prompt/terminal operates.

The generated output in the terminal is very similar to before. Again, the program displays it is running in Demo Mode. In fact, it will always do this when no trajectory/topology files are specified. There are, unlike the first time, a few extra messages before we see the logo: now there are more issues with the input file. It is good to know that whenever a parameter is missing from the input file, the program will load a default. More on these default parameters can be found in section 4.2, but for now, you should know one thing: some parameters are more important than others. Therefore, the program will specifically raise warnings for some parameters, but not all. Others might still be changed, and one should check the log-file to see what parameters the program ran with.

There are 5 warnings for missing parameters: map_choice, Dipole_choice, coupling_choice, NN_coupling_choice and SphereSize. The final message before the calculation header is the same one as before, whether a c-library was used succesfully. What each of the parameters means can be found in section 2.2, here, we will just see how to set these settings.

To specify a setting yourself, open the input file again with your favourite text editor. On the first line, add `map_choice  Skinner`. Notice the capitalization! The amount of spaces does not matter, as long as there is at least one. Also add the following lines (order does not matter):

```
Dipole_choice  Torii
coupling_choice  TDCTasumi
NN_coupling_choice  TDCTasumi
SphereSize  20
```

Save the file, and re-run the command from before: `python AIM.py input_parameters.txt`. The output should again be very similar, still in Demo Mode, but the five warnings mentioned before should now have disappeared. Have a look at the log file of this last run. Under the header "Input file" you should now see the five lines you added before. If lines are missing (which only happens if not all errors in the command prompt/terminal disappeared), the most likely cause is a typo in the first word on that line.

Further down, in the "Final input parameters" section, under "run parameters" the selection should be visible. Our map_choice is listed under "Electrostatic map", the other three choices are underneath: dipole method, coupling method, and NN coupling method. The SphereSize parameter is further down, under "Cutoff distance for electrostatic effects". When you check the log-file of the run with the empty input parameter file, you can see that a few of these parameters are different.

To know what parameters are available, you can either look in section 2.2, or look at one of the generated output parameter files. These files contain all parameters that can be specified using the input file, but to know other options and what they mean, check this manual!

The last important thing for using the software are the parameters trjfile and topfile. This is where you specify which data to compute the Hamiltonian and Dipole files for. To start calculating on your data, specify the trajectory file location and name using trjfile, and the topology file location and name using topfile. The paths to these files can either be absolute, or relative compared to the input file itself. For this reason, it might be a useful practice to save the input parameter file close to your data files. The trajectory file is usually the larger of the two, and common extensions for both files can be found in the table.

MD package	topology extension (preferred type in **bold**)	trajectory extension (preferred type in **bold**)
GROMACS	.tpr (NOT .gro or .itp)	**.xtc** or .gro or .trr
CHARMM	.psf (.crd untested)	.dcd (.crd untested)
NAMD	.psf	.dcd (.coor and .namdbin untested)
AMBER	.top (.parm7 and .prmtop untested)	**.nc** or .mdcrd (NOT .crd!) (.crdbox, .ncdf, .trj untested)

Note: we see no reason why the extensions marked as 'untested' wouldn't work, but they have not been tested. In general, the preferred options are chosen as they tend to have the smallest file sizes.

1.7 Advanced installation

Consider the steps below as an added bonus: AIM can run without them, but they are incredibly useful. Especially the c library is incredibly useful, for basically every type of user. Who doesn't like their calculations finishing a lot sooner? The profile grapher is only useful for developers trying to make (parts of) the program run faster.

1.7.1 Installing the c library manually

During the demo mode runs, the same message kept turning up about the use of a c library. Maybe it was used successfully, maybe it was not. The use of one is preferred, as it speeds up the calculation time considerably. Therefore, the program makes multiple different attempts to load such a library. If all these fail, either your system is too different from the ones used for testing (so none of the precompiled c-libraries work), or for some reason, it blocked the use of such a file. In these cases, it could be useful to compile the c code yourself.

The following instructions explain how to compile the c library yourself, and how to hook it up. If you're already familiar with compiling c scripts, most of this might already be familiar - just make sure to compile it into a dynamic library (.dll for Windows, .so for Linux, .dylib for MacOS). If that doesn't mean anything to you, just read on, we'll go through it step by step.

The instructions for compiling the c code yourself differ between operating systems (OS). Regardless of your OS, you might want to open the raw file using a text editor, to check it out. This is mainly for those of you that (rightfully so) don't blindly trust a downloaded .dll/.so/.dylib file, and want to see the code and compile it yourself.

When you trust the code, you can go ahead and compile it. Please note that the methods listed here work for the tested systems, but you might prefer a different method yourself. The presented methods here are not the only ones available. For (Linux) computer clusters in particular, you may want to check with your system administrator to help set up the optimal compilation.

Windows

In order to compile correctly, it is important to know what type of installation you have. This depends on the windows version itself, but also on your python installation. Therefore, just as during the basic installation, in your command prompt, type `python`. Have a good look at the first line (the one starting

with the version number 3.x.x). Close to the end, take note on the part inside square brackets: it either says "32 bit" or "64 bit". Don't worry about the "win32" at the very end: 64-bit systems also list win32. You can leave the python interface by typing `quit()`.

Regardless of the method you use, windows requires the AIMVx-x.cpp file to contain an extra block of code, which UNIX systems can't understand. Therefore, you should open the file in any text editor, and make a small change. You will see that the file starts with a short explanation, and the lines below that start with `#include`. The line below those is what we're interested in: it says `/*extern "C" {`. You should remove the two first characters on this line; the `/*`. Then, you get a lot of lines starting with `__declspec(dllexport)`. Leave these the way they are. Then, the final line of this block contains three characters: `}*/`. Remove the last two, so only the `}` remains. The four characters you've just removed mean something like "ignore everything between us". While UNIX systems should indeed ignore it (therefore, they don't edit the file like that), Windows shouldn't - hence why we removed it.

The following is the **RECOMMENDED** way of compiling the script. It is easier, creates fewer extra files, and less prone to mistakes. If you insist on using a graphical interface, there is an alternative method below.

To compile the script using windows, you will need a copy of Microsoft Visual Studio. When using this method, it is sufficient to only install the command line build tools. When on the download page of visual studio (https://www.visualstudio.microsoft.com/downloads/), you do not have to click the obvious download button. Instead, scroll down the page, and, under 'All Downloads', open the drop down menu titled 'Tools for Visual Studio 20xx', where xx can be any year. At the time of writing, 2022 has been downloaded, but 2017 and 2019 have also been tested. In that menu, look for the 'Build Tools for Visual Studio 20xx' line, and click the download button next to it.

After you downloaded the installer, open it. You have the option to select one of many packages, so-called 'workloads'. You do not need these (although a package for building c++ code is offered there, too - it'll just install 2GB more than you need, but you do get the correct tools as well). Instead, in the top bar, select 'individual components'. You are presented with a long list of components. You need to select two boxes. Firstly, you want one of the build tools. Depending on the year of your visual studio installation, numbers may differ, but it should look something like this: 'MSVC v1xx - VS 20xx C++ x64/x86 build tools (latest)'. Again, at the time of writing, this was v143 VS 2022, but other years (and the corresponding versions) will work, too. The second box you need to select is for the so-called SDK. Again, the one you need depends on your installation of windows. Here, the most recent version for windows 10 was installed (as my windows PC currently runs windows 10). The text next to the box was called 'Windows 10 SDK (10.0.20348.0)'.

After double checking you've selected two components (they should show up in the list on the right), a build tool and the SDK, you can click 'download' in the bottom right. The program will start downloading and installing. Once installed, in the start menu, scroll through the list of installed programs. Under "V", you'll find a folder named "Visual studio 20xx". The used version for the development this program is 2017, but 2019 and 2022 are known to work, and if your (different) version doesn't, please let us know!

Inside the folder, you will find a few special Command Prompts. Which one you need, depends on the bits of your python installation: if you run 64 bit python, you need the x64 Native Tools Command Prompt for VS 20xx, if you run 32 bit python, you need the x86 Native Tools Command Prompt for VS 20xx. Click it, and a command prompt opens.

Inside this special VS command prompt, navigate to the folder where the .cpp script resides. By default, this is in the sourcefiles folder of the AIM installation. When inside this folder, run the following command: `cl.exe /LD AIMVx-x.cpp`. Make sure to replace x-x with the actual number of the script. The command will generate four files. All have the same name as the .cpp script, but will end in .dll, .exp, .lib and .obj. AIM only requires the .dll, not the rest.

Finally, open the default_settings.txt file in sourcefiles. Here, find the line starting with 'libfile', and enter the name of the file there. If the freshly made .dll lives in the sourcefiles folder, the line can just be `libfile  AIM_Vx-x.dll`, as the file path is relative to the listed sourcefiles folder. Save the default settings file, and try to run a basic calculation. Now, the c speedup should be available!

A short reminder that this method is **NOT** the recommended way. But if you really prefer a visual interface, you can open the Visual Studio program. You should be presented with the start page "Get Started", and under "New project", click "Create new project". Here, under "Visual C++", in "Windows Desktop", there is an option "Dynamic-Link Library (DLL)". Click it, enter a file name and location, and a new document opens. Replace all text in this document with the 'text' from the .cpp script in the installation folder. Then, at the top of the screen, make sure the first drop-down box after the save logo

says "Debug". The second should list x86 for 32-bit python, and x64 for 64 bit python. Then, under "Build" (to the right of "File" and "Edit"), click "Rebuild (yourfilename)". At the bottom of the screen, it should say there are no errors.

In File Explorer, navigate to the save folder of the project. Here should be a file with the chosen project name, ending in .sln. Navigate to the compiled .dll file: for x64, go to x64 -¿ Debug. For x86, click the Debug folder. Inside the correct folder is a file with your project name, ending in .dll. Copy it, and paste it into the sourcefiles folder of your AIM installation.

Finally, just as with the other method, open the "default_input.txt". Replace the line 'libfile (something)' with 'libfile (yourfilename.dll)'. Now, when running the program, it should be able to utilize the c library!

Linux

To compile the .cpp script, you will need the cc command. If it is not yet installed, look up the instructions on how to get it on your distribution.

Open a terminal, and navigate to the AIM installation folder. Then, go to the sourcefiles folder, such that 'ls' lists the .cpp script. Then, enter the following command:
`cc -fPIC -shared -o AIMVx-x.so AIMVx-x.cpp`
Make sure that x-x matches your version number. Then, open the file "default_input.txt" using your favourite text editor, and make sure that the line starting with 'libfile' lists your freshly made file. As long as it lives inside the sourcefiles folder, listing its name (including .so) is enough. To test, run a simple calculation, and confirm that the program recognises the file and is able to use it!

MacOS

To compile the .cpp script, you will need the cc command. It comes with the app "Xcode" from the app store. If it is not yet installed, go to the app store, and install it. If a restart is requested, do so.

Open a terminal, and navigate to the AIM installation folder. Then, go to the sourcefiles folder, such that 'ls' lists the .cpp script. Then, enter the following command:
`cc -fPIC -dynamiclib -o AIMVx-x.dylib AIMVx-x.cpp`
Make sure that x-x matches your version number. Then, open the file "default_input.txt" using your favourite text editor, and make sure that the line starting with 'libfile' lists your freshly made file. As long as it lives inside the sourcefiles folder, listing its name (including .dylib) is enough. To test, run a simple calculation, and confirm that the program recognises the file and is able to use it!

1.7.2 Installing the profile grapher

Please note: this functionality is only interesting for those who want to tinker with the python code of the script. If you just use AIM and do not develop for it, you should never need this!

The python code has usage of the cProfile library built-in. If the input parameter file contains the line `profiler True`, the run is profiled using this cProfile library. This results in creation of a second file in the log folder, ending in .pstats. Generating this file requires nothing but the already present python installation, and available cProfile tutorials will explain ways of looking at the raw data inside.

There is, however a better way: if the input parameter file also contains the line "pngout True", a branching diagram will be created, which indicates using colored nodes which parts of the calculation are most expensive. Creation of this, however, requires some 3rd party programs:

- Look up the gprof2dot program. It has a very nice github page with further instructions. It is enough to download the file "gprof2dot.py" from here, and save it inside the main AIM install folder, right next to the main "AIM.py" file. This program reads the .pstats file, sorts through the data, and decides what nodes to export. If you want more control over the output graph, you can also manually run the program with the generated .pstats file. However, it does not produce the image on its own. This is also indicated on the github page.
- On the gprof2dot github page, follow the link to the Graphviz program. This is what actually converts the plot data to an image. The web page has clear instructions on how to install it. If installed correctly, AIM can automatically call this program, and return the .png with the colored branch diagram.

Now, you have a very powerful tool for optimizing the speed of the code. Further information on how to use it can be found in the dev section.

1.8 Simplifying calling the program

While this section is written intended for Unix users, there should be an alternative for Windows.

Say you've placed the AIM folder in a handy location (for example /home). While this is the shortest path possible, calling the program can still be painful:

```
python /home/AIM/AIM.py yourinputfile.txt
```

This can be simplified by editing the .bashrc (linux), or .bash_profile (MacOS) file. Add the following lines:

```
AIM () {
if [ -z "$1" ]; then
   python /home/AIM/AIM.py
else
   python /home/AIM/AIM.py "$1"
fi
}
```

Make sure to replace the location of the AIM.py file with the actual location on your system! After saving, you have to prompt the system to reload this file. To do so, type `source ~/.bashrc`. From then on, you can call the simple Demo Mode version by typing `AIM`, and the full version by appending your input parameter file name.

Chapter 2

Input files

In normal usage, the AIM program requires three input files: The data from the preceding MD calculation (in the form of a trajectory and topology file), and an input parameter file. In this section, you will find all info on how to correctly make these.

2.1 Generating usable MD output

While in theory any MD file could be used, the physical world exerts some restraints. Firstly, for AIM to calculate the frequency of certain IR-active modes, these modes have to be present in the MD simulation. Originally intended for the Amide I vibration, the program performs best if the MD simulation contains a protein. However, when using non-amide groups exclusively (through using custom extra maps, see section 5), there is no reason for the MD simulation to contain a protein. Secondly, the included maps are designed for atomistic simulations. Sometimes also referred to as all-atom, these model the behaviour of every atom in the system. This is the opposite of course-grained simulations, where one so-called bead mimics the behaviour of a group of atoms. The one exception here are hydrogens on carbon atoms; if the forcefield is designed to omit these, their absence is not a problem.

There exists a hard-coded limit on the amount of atoms in the system: at most, there can be 999999999 (just under a billion) atoms. This should be sufficient for most users. Similarly, there is a limit on the amount of frames: The program does not consider frames with a number larger than 999999999 (just under a billion). This, too, should be sufficient for most users, as the calculation of 2D-IR spectra requires a minimum of about 50000 frames. If it is not sufficient, split the MD simulation in multiple parts to consider all frames, and append the output data.

While the size of the time step does not matter for the program to function (besides perhaps use of the NSA parameters, see section 2.2), there are physical limitations. Typically, the dynamics are sufficiently fast to require a frame to be saved every 10-20 fs. The time step between computed frames must be even smaller: with a frequency of roughly 1600 cm^{-1}, one oscillation is expected to last almost 21 fs. To model an oscillation properly in MD, one needs at least 10 datapoints per oscillation, so calculated frames should at most be 2 fs apart.

On a final note, the program is thoroughly tested using input files generated using GROMACS. Therefore, when using the .tpr and .xtc files from a GROMACS simulation, one only has to worry about a sufficiently small time step; atom names and the molecule properties are no issue. CHARMM, AMBER and NAMD files have also been tested, albeit not as much. For other MD software, please get in touch with us! We'd love to fully accommodate a wider range of MD packages! Please do note that you select the correct atomnames file as an input parameter. For more information, see section 2.2.

2.2 Writing an input parameter file

If you ran any successful calculation (including using Demo Mode), the program has returned an output file. These output files are a nice template for future input files: they contain all possible parameters, grouped by theme. However, while using them, keep the following things in mind:

- Every line in the file can specify at most one parameter. The line starts with the parameter name, followed by some whitespaces and the chosen option for that parameter. Even if the chosen option is long (for example filenames), it must remain on a single line.

- As demonstrated in section 1.4, not all parameters have to be specified. In fact, the file may even be empty. Most users will want to specify their data files, and a handful of important parameters. If a parameter is not specified, it will be read from the "default_input.txt" file.
- After the program reads a '#' on a line, it will ignore anything after. This is a useful feature for labeling or explaining settings, or for easily disabling one.
- While the created files are nicely formatted, this is no requirement. The order of the items may differ. Furthermore, there may be any amount of spaces before a parameter name, between the parameter name and parameter choice, and after the choice. There is, however, one point of caution: everything is case-sensitive!
- If anything is unclear to the program (unrecognised parameter name or setting), it will print this to the command line and log file. Pay attention for any messages there!
- Special attention has to be paid to file names. The entire address cannot contain any whitespaces or '#' characters. Furthermore, the path to a file must either be absolute, or relative from a very specific location. Which location is indicated below, at the relevant items.
- By default, the names of the generated files (log and output files) contain the date/time at which the calculation was started, and the version of the program. If you wish to change something about the name, but still include either date or program version, you can do this using the [now] and [version] text. If the output filename contains [now] (including square brackets), this will be replaced by the date and time. Similarly, [version] (including square brackets) will be replaced by the version of the script.

Some combinations of parameters do not work. In those cases, the program will automatically resolve the problem by changing one of the parameters. It will always notify the user when doing so. Parameters that can be affected have a little note on this behaviour in their specification.

Keeping these things in mind, here are the details on each of the parameters, listed in order of appearance in the output parameter file.

topfile

Used to specify the location of the topology file to be used in the program. This location can either be absolute, or relative to the location of the input parameter file. When both the topology file and trajectory file are not specified, the demo files will be used instead. To indicate this, every print statement will be preceded by the text "Demo Mode:". Topology files usually end in .tpr (Gromacs) or .psf (CHARMM).

trjfile

Used to specify the location of the trajectory file to be used in the program. This location can either be absolute, or relative to the location of the input parameter file. When both the topology file and trajectory file are not specified, the demo files will be used instead. To indicate this, every print statement will be preceded by the text "Demo Mode:". Trajectory files usually end in .xtc (Gromacs) or .dcd (CHARMM).

sourcedir

This specifies the path of the directory that contains the source files. This location can either be absolute, or relative to the input parameter file. If this parameter is specified, any of the specified mapfiles/libfiles will only be looked for in the specified folder. If they are not present there, the program raises an error and quits. It will not check the "sourcefiles" folder! This parameter is intended only for those who wish to alter the contents of the used map files.

def_parfile

This specifies the path of the file that contains the default input parameters. This location can either be absolute, or relative to the sourcedir location. If the file lives in the sourcedir folder, just enter its name (plus extension). If sourcedir is specified, the program will only look for this file relative to the specified sourcedir folder, and will ignore the default sourcedir folder. If sourcedir is not specified, it will only check the sourcefiles folder inside the main installation for a file of that name. This parameter is intended only for those who wish to alter the contents of the default parameter file, without editing the default one directly.

NN_Map

This specifies the path of the file that contains the parameters for the nearest-neighbor mapping. This location can either be absolute, or relative to the sourcedir location. If the file lives in the sourcedir folder, just enter its name (plus extension). If sourcedir is specified, the program will only look for this file relative to the specified sourcedir folder, and will ignore the default sourcedir folder. If sourcedir is not specified, it will check both the sourcedir defined in the file "default_input.txt", and the original "sourcefiles" folder with a file of that name. This parameter is intended only for those who wish to alter the contents of the nearest-neighbor map file.

Tasumi_Map

This specifies the path of the file that contains the parameters for the Tasumi mapping. This location can either be absolute, or relative to the sourcedir location. If the file lives in the sourcedir folder, just enter its name (plus extension). If sourcedir is specified, the program will only look for this file relative to the specified sourcedir folder, and will ignore the default sourcedir folder. If sourcedir is not specified, it will check both the sourcedir defined in the file "default_input.txt", and the original "sourcefiles" folder with a file of that name. This parameter is intended only for those who wish to alter the contents of the Tasumi map file.

TCC_Map

This specifies the path of the file that contains the parameters for the TCC mapping. This location can either be absolute, or relative to the sourcedir location. If the file lives in the sourcedir folder, just enter its name (plus extension). If sourcedir is specified, the program will only look for this file relative to the specified sourcedir folder, and will ignore the default sourcedir folder. If sourcedir is not specified, it will check both the sourcedir defined in the file "default_input.txt", and the original "sourcefiles" folder with a file of that name. This parameter is intended only for those who wish to alter the contents of the TCC map file.

resnamesfile

This specifies the path of the file that contains the different resname groups. See section 4.6 for more details on the contents of this file. The location of the file can either be absolute, or relative to the sourcedir location. If the file lives in the sourcedir folder, just enter its name (plus extension). If sourcedir is specified, the program will only look for this file relative to the specified sourcedir folder, and will ignore the default sourcedir folder. If sourcedir is not specified, it will check both the sourcedir defined in the default parameters file, and the original "sourcefiles" folder for a file of that name. This parameter is intended only for those who wish to alter the residue names and selection keywords recognized by the program.

atnamesfile

This specifies the path of the file that contains the different atom names. See section 4.7 for more details on the contents of this file. The location of the file can either be absolute, or relative to the sourcedir location. If the file lives in the sourcedir folder, just enter its name (plus extension). If sourcedir is specified, the program will only look for this file relative to the specified sourcedir folder, and will ignore the default sourcedir folder. If sourcedir is not specified, it will check both the sourcedir defined in the default parameters file, and the original "sourcefiles" folder for a file of that name. This parameter is intended only for those who wish to alter the atom names used by the program.

libfile

This specifies the path of the file of the compiled c script. For Windows machines, it ends in .dll, for Linux in .so, and for MacOS in .dylib. This location can either be absolute, or relative to the sourcedir location. If the file lives in the sourcedir folder, just enter its name (plus extension). If sourcedir is specified, the program will only look for this file relative to the specified sourcedir folder, and will ignore the default sourcedir folder. If sourcedir is not specified, it will check both the sourcedir defined in the file "default_input.txt", and the original "sourcefiles" folder with a file of that name.

If no file with this name can be found, the program will also attempt to open the pre-compiled file for your OS from the default sourcefiles folder. This parameter is intended only for those who wish to compile their own version of the c script.

use_c_lib

Options: True, False
This setting indicates whether you wish to use a c library to speed up the calculation. If set to True, the program will quit if it cannot find a functioning library. If set to false, it will never attempt to use any (and instead use numba functions). When omitted, the program will try its best to find and use a working c library, but if it cannot, the run will continue without.

extramapdir

This specifies the path of the directory where extra mapfiles can be found. This location can either be absolute, or relative to the input parameter file. The program reads all valid .map and .cmap files in this location, so the maps can be used during calculations. For more info on the .map and .cmap files, see section 5.

outdir

This specifies the path of the directory where the output files should be saved. This location can either be absolute, or relative to the input parameter file. If this parameter is specified, all output files will be saved relative to this folder, regardless of whether they are specified later, or not. When this parameter is not specified, the output files will be saved relative to the folder from which AIM is called (NOT where AIM is installed).

Please note: the folder called "output" in the AIM installation directory is maintained: if more than 30 files are present at the start of the calculation, they are sorted by name, and only the last 30 files are kept. This means that when using the default filenames, the oldest get removed. If you do not want this behaviour, specify a different folder. This can also be done in the "default_input.txt" file, see section 4.2.

outfilename

Used to specify the path of the Hamiltonian output file. This location can either be absolute, or relative to the folder specified using outdir. If a specific name is chosen, the calculation will overwrite any prior files with that name at that location. To ensure the name to be unique, the [now] and [version] syntax may be used! Do not add any file type/extension here. '.txt' or '.bin' will be appended automatically, depending on the choice of output datatype.

outdipfilename

Used to specify the path of the dipole output file. This location can either be absolute, or relative to the folder specified using outdir. If a specific name is chosen, the calculation will overwrite any prior files with that name at that location. To ensure the name to be unique, the [now] and [version] syntax may be used! Do not add any file type/extension here. '.txt' or '.bin' will be appended automatically, depending on the choice of output datatype.

outramfilename

Used to specify the path of the Raman tensor output file. This location can either be absolute, or relative to the folder specified using outdir. If a specific name is chosen, the calculation will overwrite any prior files with that name at that location. To ensure the name to be unique, the [now] and [version] syntax may be used! Unlike outfilename and outdipfilename, an extension is required, for example '.txt'.

outposfilename

Used to specify the path of the atom positions output file. This location can either be absolute, or relative to the folder specified using outdir. If a specific name is chosen, the calculation will overwrite any prior files with that name at that location. To ensure the name to be unique, the [now] and [version] syntax may be used! Do not add any file type/extension here. '.txt' or '.bin' will be appended automatically, depending on the choice of output datatype.

outparfilename

Used to specify the path of the parameter output file. This location can either be absolute, or relative to the folder specified using outdir. If a specific name is chosen, the calculation will overwrite any prior files with that name at that location. To ensure the name to be unique, the [now] and [version] syntax may be used! Unlike outfilename and outdipfilename, an extension is required, for example '.txt'.

logdir

This specifies the path of the directory where the log files should be saved. This location can either be absolute, or relative to the input parameter file. If this parameter is specified, all log files will be saved relative to this folder, regardless of whether they are specified later, or not. When this parameter is not specified, the log files will be saved relative to the folder from which AIM is called (NOT where AIM is installed).

Please note: the default "log" folder is maintained: if more than 30 files are present at the start of the calculation, they are sorted by name, and only the last 30 files are kept. This means that when using the default filenames, the oldest get removed. If you do not want this behaviour, specify a different folder. This can also be done in the "default_input.txt" file, see section 4.2.

logfilename

Used to specify the path of the log file. This location can either be absolute, or relative to the folder specified using logdir. If a specific name is chosen, the calculation will overwrite any prior files with that name at that location. To ensure the name to be unique, the [now] and [version] syntax may be used!

proffilename

Used to specify the path of the profiling file. This location can either be absolute, or relative to the folder specified using logdir. If a specific name is chosen, the calculation will overwrite any prior files with that name at that location. To ensure the name to be unique, the [now] and [version] syntax may be used!

pngfilename

Used to specify the path of the image of the profiling file. This location can either be absolute, or relative to the folder specified using logdir. If a specific name is chosen, the calculation will overwrite any prior files with that name at that location. To ensure the name to be unique, the [now] and [version] syntax may be used!

output_format

Options: bin, txt This setting indicates what format you want the output hamiltonian, dipole and atompos files to have. This parameter is read just like the `influencers` and `oscillators` parameters - multiple choices are allowed, and should be separated by a whitespace. `output_format bin` will result in AIM giving only binary files, while `output_format bin txt` will result in AIM giving both binary, and txt-style files. Please note that the NISE package[1] can use both (and convert both into many more other formats), but the binary files take up much less space.

output_type

Options: Ham, Dip, Ram, Pos This settings indicate what information you want AIM to return. `output_type Ham` returns the hamiltonian file(s), `output_type Dip` returns the dipole file(s), `output_type Pos` returns the position file(s), and `output_type Ham Dip Pos` returns all. Any combination of files is allowed, just make sure to specify all on a single line, separated using whitespaces.

Verbose

Options: 0, 1, 2, 3, 4

This setting indicates how many prints you want in the command prompt/terminal. A lower number indicates fewer prints. With 0, only warning and error messages are printed. 4 Prints sufficient amounts of information that it should only be used for debugging purposes.

Verbose_log

Options: 0, 1, 2, 3, 4

This setting indicates how many prints you want in the log file. A lower number indicates fewer prints. With 0, only warning and error messages are printed. 4 Prints sufficient amounts of information that it should only be used for debugging purposes.

profiler

Options: True, False

This setting indicates whether you want to have cProfile active in the background while running the calculation. Having it running slows down the calculation, so it is wise to turn it of when not needed. This parameter is intended only for those who wish to see what parts of the code take longest, so these parts can be improved. Therefore, it is only relevant for programmers, not general users. As of version 9.10, this makes the code run about 15% slower.

pngout

Options: True, False

This setting indicates whether you want to make a graphical representation of the profiling information. This information is only generated if the 'profiler' parameter is set to 'True'. Therefore, regardless of choice here, if 'profiler' is set to 'False', the program will automatically set this to 'False', too.

influencers

Options: All, Solvent*, Water*, Lipid*, Protein, Ions*, K*, Cl*, Na*

This setting indicates which types of residues should be considered when calculating the electric potential, field and gradient for any group. 'All' and 'Protein' are always available. The others (marked with an asterisk) depend on the categories specified in the resnames file. See section 4.6 for more details. By default (without changing the contents of the resnames file), 'All' selects all atoms, 'Solvent' selects all atoms that are part of a water molecule or a dissolved ion like K or Cl, 'Water' selects all atoms that are part of a water molecule, 'Lipid' selects all atoms that are part of a lipid molecule, 'Protein' selects all atoms that are a part of the protein itself, 'Ions' selects all atoms that are a dissolved ion like K or Cl, 'K' selects all potassium ions, 'Cl' selects all chlorine ions and 'Na' selects all sodium ions. Multiple groups can be selected by typing multiple choices, separated by spaces. For example: `influencers  Lipid Protein` selects all atoms which are either part of the lipid, or protein section.

oscillators

Options: AmideBB, AmideSC*, Azide_ion*

This setting indicates which types of oscillators should be included in the Hamiltonian and other output files. Only AmideBB is coded into AIM itself. All other options are present as .map files in the extramaps directory. Section 5 explains these files in detail. In short: every file contains a given name. That name can be specified here to specify that the types of groups defined in that map should be included. If multiple types are desired, separate those choices with spaces. For example, AmideSC comes with your installation of AIM. It contains all necessary info to include the amide groups in protein sidechains (think of the glutamine and asparagine amino acids) in the calculation. If you would like to run AIM for calculating the IR spectrum of amide groups in both the protein backbone and side chains, you would use `oscillators AmideBB AmideSC`.

apply_dd_coupling

Options: All, None, Same, Different

In theory, every pair of oscillating groups should couple. AIM provides multiple ways to encode very specific coupling between groups. The basic installation includes 5 different methods for calculating the coupling between amide groups. Furthermore, users can include special coupling maps for different types of groups introduced by them (more info in section 5). However, it is very much possible that not all available combinations of groups have a coupling method specified that way. This parameter then decides how to treat them. There is two types of pairs possible. Either, both groups in a given pair are of the

same kind (for example, both are amides), or of a different kind (for example, one amide, one azide). The option 'All' applies basic dipole-dipole coupling to both types of pairs, the option 'None' to neither. 'Same' only applies basic dipole-dipole coupling to pairs of the same type of group, while assuming 0 coupling between pairs of two different types. 'Different' has the opposite behaviour compared to 'Same'.

map_choice

Options: Tokmakoff, Skinner, Jansen, Cho, Hirst, other*

This setting indicates what map should be used for calculating the oscillation frequencies of amide groups. Out of the three, the Tokmakoff mapping is cheapest to calculate, but also gives the least accurate results. The Jansen mapping gives the best results, but is also the most expensive to calculate.

The Tokmakoff map only considers the electric field on the oxygen atom, while the Skinner map considers the electric field on the carbon and nitrogen atoms. The Jansen map considers the electric field and electric gradient on the carbon, nitrogen, oxygen and hydrogen atoms of the amide bond. Both Skinner and Jansen map make use of a special proline map for calculating the frequency of amide bonds just before proline residues. Only the Skinner map has a map designed especially for side chain amide groups, so the validity for side chain frequencies using other maps cannot be guaranteed.

* Other maps can be chosen as well. If you make a file for another map in the sourcefiles directory, the map can then be chosen using the same name as in the filename. See section 4.8.1 for more details.

Dipole_choice

Options: Torii, Jansen, other*

This setting indicates how the dipole moments of amide groups should be calculated. The Torii method finds the dipole moments using the positions of the carbon-oxygen and carbon-nitrogen bonds. The Jansen method finds the dipole moments using a map which uses the electric field and gradient on the carbon, nitrogen, oxygen and hydrogen atoms.

* Other maps can be chosen as well, following the same format as for the electrostatic maps (see section 2.2)

coupling_choice

Options: None, TDCKrimm, TDCTasumi, TCC

This setting indicates how to calculate the coupling between amide groups that are not nearest neighbors. Both TDCKrimm and TDCTasumi calculate this coupling using the interaction of the dipole moments on the two amide bonds. TDCKrimm assumes the dipoles to have a fixed magnitude and orientation relative to the amide group, while TDCTasumi uses Torii dipoles (see section 6.3.3). The TCC method determines the coupling through the interactions between any pair of atoms inside the amide groups. How those interactions weigh in to the total is governed by a map.

NN_coupling_choice

Options: None, TDCKrimm, TDCTasumi, TCC, Tasumi, GLDP

This setting indicates how to calculate the coupling between amide groups that are nearest neighbors. For details on TDCKrimm, TDCTasumi and TCC, see section 6.3.2. Both the Tasumi and GLDP methods make use of a map which is based on the ramachandran angles surrounding the amide bond. There is one map for the Tasumi method, while for the GLDP method, there are 7: depending on whether one of the two amide groups is followed by a proline residue, a different map is used.

AtomPos_choice

Options: C, N, O, D

This setting indicates of which atom the position should be written to the AtomPos output file. All available options are one of the atoms part of the amide bond.

start_frame

Use this to specify a different starting frame number. The first frame has number 0, so frame 10 is actually the 11th frame. Useful for dividing up the data in smaller sets for calculation. If you define start_frame and n_Frames_to_calculate and end_frame, make sure that start + ntotal = end.

nFrames_to_calculate

Use this to specify a different amount of total frames that should be calculated. When entering 10, after having calculated 10 frames, the program stops automatically. If you define start_frame and n_Frames_to_calculate and end_frame, make sure that start + ntotal = end.

end_frame

Use this to specify at which frame number the calculation should stop. The number entered here is NOT included in the calculation, so 10 will consider frame 9, but not frame 10. If you define start_frame and n_Frames_to_calculate and end_frame, make sure that start + ntotal = end.

max_time

The maximum amount of time (expressed in minutes) that the calculation may take. Useful for running on clusters with a time limit: if the time limit is reached before the last frame is reached, the program saves and stops automatically to prevent data loss. When stopping before the final frame has been considered, the log file will show at which frame to continue to calculate the remaining data.

SphereSize

The maximum distance between residues for considering their electrostatic effects, in Angstrom. If a group falls outside this range of an amide group, it will not be considered for calculating the electric potential, field and gradient on that amide group.

replicate_orig_AIM

Options: True, False

There was an earlier predecessor of this program that ran inside the GROMACS environment. It contains a few minor mistakes, which results in frequencies that deviate at most a few wave numbers. These mistakes have been fixed in this program, resulting in a slightly different output. This setting adds the mistakes, so the exact results can be replicated. This parameter is intended only for those who wish to verify the validity of the code to this earlier version, so as to prevent new mistakes. Therefore, it is only relevant for some programmers, not general users. Unless you really know what you're doing, don't use it.

NSA_toggle

Options: True, False

By default, the code spends a considerable amount of time on checking which residues are in range of which amide groups to calculate their electrostatic influence. This setting allows to speed up this process, by assuming residues only move a little between frames, allowing to divide this search up into two steps: every n^{th} frame, all residues are considered, and added to a special list if they fall within this technique's larger radius. Then, all intermittent frames, only this list is checked to see which residues fall within the smaller range. Due to the nature of the calculation, this does not work together with replicate_orig_AIM. Therefore, if replicate_orig_AIM is set to 'True', this parameter is automatically set to 'False'.

Depending on how much time lies between two frames, the temperature of the system and the mobility of residues, the generated shortlist might require more or fewer updates, and might need to contain a larger or smaller subset of all residues. Therefore, it is set to 'False' by default, and when set to 'True', the input file must contain both the parameters NSA_nframes and NSA_spheresize. To determine valid parameters, one could first run a 100 or 1000 frame calculation with this parameter set to 'False'. Then, redo the calculation with this parameter set to 'True', and see what settings for the other two parameters yield the same results.

NSA_nframes

This setting indicates how often the residue shortlists are generated. For more info, see section 2.2. As a starting point for finding the ideal setting for this parameter: A room-temperature simulation in water with a SphereSize of 20 Angstrom and 20 fs between frames gave identical results when NSA_nframes was set to 50 and NSA_spheresize was set to 25. For comparison, liquid water at room temperature has a thermal velocity of 590 m/s, or 5.9 angstrom/ps.

NSA_spheresize

This setting indicates the maximum distance to a residue (in Angstrom) for it to be included on a shortlist. For more info, see section 2.2. As a starting point for finding the ideal setting for this parameter: A room-temperature simulation in water with a SphereSize of 20 Angstrom and 20 fs between frames gave identical results when NSA_nframes was set to 50 and NSA_spheresize was set to 25. For comparison, liquid water at room temperature has a thermal velocity of 590 m/s, or 5.9 angstrom/ps.

atom_based_chainID

Options: True, False

Usually, separate protein segments can be identified using indices assigned by the MD packages. Either every chain is a new so-called segment (this is the case for CHARMM), or a new molecule (this is the case for Gromacs). However, sometimes, the MD package doesn't assign these ID's correctly. In those cases, this parameter can help! When setting it to true, AIM doesn't deduce the protein chains using those MD indices. Instead, it looks at important atom and residue names, as well as bonds between amino acids.

use_protein_specials

Options: True, False

Not all proteins are created equal. While there are 20 (naturally occuring) amino acids that can make up a protein, other molecules can join in as well. For example, the first element could be formic acid instead of an amino acid. A list of currently understood extra protein residues can be found below. This setting tells the program whether the protein chain should include these residue names. This is mainly intended for when one of these specials' names is identical to a non-protein residue in your simulation. Currently, these specials are included:

- FOR: A bonded formic acid (only H-C(=O)- remains). Can only occur at the start of a chain.
- ETA: A bonded ethanolamine (NH-CH2-CH2-OH remains). Can only occur at the end of a chain

Scale_LRCoupling

The values calculated for long-range coupling are multiplied with this value.

Use_AmGroup_selection_criteria

Options: True, False

Use this setting to indicate whether you want to include all backbone amide groups in the Hamiltonian and dipole calculation, or only a sub-selection. Please note: if you want to only include a sub-selection, setting this to 'True' is not enough: your selection has to be indicated with one of the resnum/resname blacklist/whitelist commands. Furthermore, if you include one of those selections, this parameter will automatically be set to 'True'. If you include none, this parameter will automatically be set to 'False'. This parameter is intended as an override: if you include a sub-selection, but set this to 'False', that selection will not be used!

resnum_whitelist

This is one of the selection commands for indicating which backbone amide groups to include. In this case, selection is done through residue numbers. If a group is specified, it will be included. Multiple groups can be selected by specifying multiple numbers, separated using spaces, like this: `resnum_whitelist 0 3 4 5 12`. Ranges of residues can be selected as follows: `resnum_whitelist 0 3-5 12` selects the same residues as the first command. When combined with one (or more) of the other three selection commands, these are additive: only if an amide group satisfies all selection criteria, it is included.

If the first residue of the simulation (residue number 0) is also the first residue of the protein, the first amide bond will be between the residues with residue numbers 0 and 1. This bond can be selected using resnum 0. In fact, a bond is always specified by the residue number of the left residue (the residue that contributes the C=O to the amide bond).

resnum_blacklist

This is one of the selection commands for indicating which backbone amide groups to include. In this case, selection is done through residue numbers. If a group is specified, it will not be included. Multiple groups can be selected by specifying multiple numbers, separated using spaces, like this: `resnum_blacklist 0 3 4 5 12`. Ranges of residues can be selected as follows: `resnum_blacklist 0 3-5 12` selects the same residues as the first command. When combined with one (or more) of the other three selection commands, these are additive: only if an amide group satisfies all mentioned selection criteria, it is included.

If the first residue of the simulation (residue number 0) is also the first residue of the protein, the first amide bond will be between the residues with residue numbers 0 and 1. This bond can be selected using resnum 0. In fact, a bond is always specified by the residue number of the left residue (the residue that contributes the C=O to the amide bond).

resname_whitelist

This is one of the selection commands for indicating which backbone amide groups to include. In this case, selection is done through residue names. If a group is specified, it will be included. Multiple types of groups can be selected by specifying multiple names, separated using spaces, like this: `resname_whitelist LYS GLY ASN`. When combined with one (or more) of the other three selection commands, these are additive: only if an amide group satisfies all mentioned selection criteria, it is included.

Just as with using residue numbers, it is the left residue (the residue that the C=O of the amide bond belongs to) that determines the residue name. A bond between a GLY on the left, and a PRO on the right will only be selected using resname GLY, and not using resname PRO.

resname_blacklist

This is one of the selection commands for indicating which backbone amide groups to include. In this case, selection is done through residue names. If a group is specified, it will not be included. Multiple types of groups can be selected by specifying multiple names, separated using spaces, like this: `resname_blacklist LYS GLY ASN`. When combined with one (or more) of the other three selection commands, these are additive: only if an amide group satisfies all mentioned selection criteria, it is included.

Just as with using residue numbers, it is the left residue (the residue that the C=O of the amide bond belongs to) that determines the residue name. A bond between a GLY on the left, and a PRO on the right will only be selected using resname GLY, and not using resname PRO.

TreatNN

Options: True, False

Whether to include nearest-neighbour interactions in Hamiltonian calculations. If set to 'True', some atoms of the neighbouring residues will be excluded from the electrostatic interactions calculation, and instead be accounted for using a residue-type-depended mapping using the ramachandran angles of the bond. This only makes sense if the protein should be considered for electrostatic interactions to begin with. Therefore, if the parameter 'selection' does not specify 'All' or 'Protein', this parameter will automatically be set to 'False'. Similarly, this parameter is set to 'False' when the Tokmakoff mapping is used.

Chapter 3

Output files

The AmideImaps program creates 5-7 output files. Always, it produces a file containing the calculated Hamiltonian, a file containing the calculated dipoles, a file containing the atom positions, a file containing the correct settings for redoing the exact calculation, and a log file. When requested to do so, the program produces a file with the profiling data (about what parts of the code took what amount of time), and possibly a graphical representation of this profiling data. In this section, you will find all info on what exactly is there, and what it means.

3.1 Output Hamiltonian file

Obtaining this file is one of the main reasons to run the program. It contains the time-dependent Hamiltonian, in the following format:

A Hamiltonian is computed for every single frame. In the txt format, the entire Hamiltonian lives on a single line. Therefore, the file contains a line for each frame. On a single line, the first number is the number of the frame, followed by a space. Then, the Hamiltonian itself follows. As the Hamiltonian is a diagonal matrix, only roughly half of the entries need to be written. The Hamiltonian is written row by row, starting at the diagonal entry. Therefore, after the frame number, the entry at 0,0 is written to the output file, followed by the one at 0,1, etc. For the next line, the first item is 1,1 (1,0 is skipped as it is identical to the value at 0,1), the second is 1,2, etc.

The values at the diagonal of the Hamiltonian are the expected vibrational frequencies in wavenumbers. These include electrostatic effects from the environment, but exclude any influences from coupling with neighbouring amide groups. The values are usually around 1600. The off-diagonal elements of the Hamiltonian list the coupling between amide groups. The coupling can be both positive and negative, and occasionally reach values of 30. However, virtually all values are smaller than 5, with most of them being smaller than 0.1.

The binary file is very similar, with half the hamiltonian per frame. It contains no spaces, only float (32 bit) characters. The first one indicates the frame number, and is followed by the hamiltonian entries in the same order as the text file. After one hamiltonian has been written, no spacer is used: the next float is the number of the next frame.

3.2 Output dipoles file

Obtaining this file is, besides the Hamiltonian file, the other main reason to run the program. It contains the time-dependent dipole moments, in the following format:

Just as with the Hamiltonian file, when using the txt format, there is one line per frame. The first entry on this line is the frame number, followed by a space. Then, the dipole moments follow. For each oscillator in the hamiltonian, there is a dipole moment, which has an x, y, and z component. First, all x-components are written, in the same order as the groups appear in the hamiltonian. Then, all y-components of all vectors are written, and finally, all z-components.

The binary file is similar. It contains only float (32 bit) objects. The first is the number of the first frame, and it is followed by all x-components of the dipoles. Just as with the txt, these are followed by the y- and z-components.

3.3 Output Raman tensor file

This file contains the time-dependent Raman tensors. They can be used for calculating Raman and sum-frequency-generation (SFG) spectra. The file has the following format:

Just as with the Hamiltonian file, when using the txt format, there is one line per frame. The first entry on this line is the frame number, followed by a space. Then, the Raman tensors follow. For each oscillator in the hamiltonian, there is a 3*3 symmetric Raman tensor, containing the unique xx, xy, xz, yy, yz and zz components. First, all xx-components are written, in the same order as the groups appear in the hamiltonian. Then, all xy-components are written, followed by all xz-, yy-, yz- and zz-components of the tensors.

The binary file is similar. It contains only float (32 bit) objects. The first is the number of the first frame, and it is followed by all xx-components of the Raman tensors. Just as with the txt, these are followed by the xy-, xz-, yy-, yz- and zz-components.

3.4 Output positions file

This file contains the positions of a certain type of atom selected using the `AtomPos_choice` keyword. These positions can be used in calculating circular dichroism spectra. The format of this file is the same as that of the dipole file.

3.5 Output parameters file

To make running the calculation easier, the program does a few tricks: it can obtain parameter settings from both the input parameters and default parameters files, file names depend on the location of the parameter files during calculation, and parameter settings can be changed when conflicts arise. To make sure that the calculation performed can be redone (or continued using different frames), the program creates the output parameters file. This file can immediately be used for the next calculation, and has a few nice properties:

Where necessary, all file paths are absolute. Paths to folders and the topology and trajectory files are always absolute, the paths to files expected within those folders are relative to them. Even when specified absolute, not relative, at the start of the run, the program will attempt to make them relative. If the output file names make use of the [now] and [version] syntax (see section 2.2), this is kept.

The file is nicely formatted, and has all items in the same order every time. This order does not depend on the order of the items in the input file name. This makes comparing different parameter files easy and quick. To properly do this, all output parameter files contain all parameters, even those not explicitly specified in the input parameter file.

3.6 Log file

When you want to know anything about the calculation, this file is your friend. It is written to in such a way that it can be accessed during the calculation. This allows you to see how the calculation is doing and how long it will still take, even when you cannot see the command line output immediately (as can be the case on clusters). Lets take a look at all the separate parts:

Input file: This section shows how the input file looks to the program. All '#' and the text after any '#' is removed, and empty lines are not printed either. Any unrecognised commands are ignored; a warning will be printed to the command line and log file.

Checking parameters: It can happen that important parameters are missing, or that the submitted combination of parameters does not work. The program can deal with these cases, and will report on what it did. All that information can be found here.

Final input parameters: Here, all parameters used by the program are printed. These are the same as those that can be found in the output parameters file, albeit in a more text-based version. These are the settings the program actually runs with!

AmideImaps: This indicates that the input files have been read, and the program actually starts. Depending on the 'Verbose' setting, this section could contain much information, or almost none. If the program ran without any errors, this section (and the entire log file) ends with the text: 'If you read this, the program encountered no problems, and ran normally!'

3.7 pstats file

This file only is generated when 'profiler' is set to 'True'. This file is mostly meant for those working on improving the code. If you are using the program to perform calculations, you should set 'profiler' to 'False' to improve the speed of the calculation.

This file is the direct product of the cProfile module. It is not a normal text file, and requires python to be read. The most direct way is to open it in python (use `python -m pstats file.pstats`). It can then be navigated with commands. The following set of commands creates a list of the 10 most computationally expensive parts of the program:

```
strip
sort tottime
stats 10
```

If you are using this function/file, this should get you started, and you should know where to find more info.

Creating this file can be done with a normal python installation. Converting it to a coloured branch diagram (MUCH easier way of visualizing all the information) requires extra software. This is why you can save only this file: when necessary, it can be moved to a different system/environment for creating the plot.

3.8 pstats image

This file is only generated when 'pngout' is set to 'True'. This file is mostly meant for those working on improving the code. If you are using the program to perform calculations, you should set 'profiler' to 'False' to improve the speed of the calculation.

This file is created using the Graphviz software, which reads a file created by the gprof2dot.py script, which in turn reads the .pstats file. AmideImaps runs the most basic command for gprof2dot.py, which actually has quite a few options. Check their github page for more information if you want to create a png yourself.

Chapter 4

Source files

AmideImaps requires more than just code to run: for example, it uses predetermined maps to predict one property of the system using another one which is easier to calculate. While this information could be hard-coded (included right in the python script itself), this process is avoided as much as possible, so users can more easily change this information if so desired.

It should therefore be noted that anything to do with changing these files is for the advanced user only. Changing anything in these files is at your own peril. Make sure to create regular back-ups. AIM automatically searches the source files folder for any maps - you do not have to specify the names of the files starting with 'Map-' in the input file.

4.1 AIMVx-x.???

There are multiple files with different file extensions of the name AIMVx-x. These are the c library files. The main file is the c++ script (.cpp), all others are compiled versions of it. Instructions on how to compile it yourself can be found in section 1.7.1.

These c scripts are the answer to a question some of you might have had: Why would you write a script that runs for hours using (the convenient but slow) python? The short answer is already in the question: python is convenient and easy to use. The long answer is in the c scripts: a python script does not have to be slow. Benchmarks of the python version showed that it is equally fast, if not faster, than the original version written using only c. While the calculation is heavy and takes a long time, it is actually a relatively simple and small section that takes so long. By only coding that part into c, the program becomes basically as fast as a c-only program, but keeps all the awesomeness of python: easy to read, use, write, and... all the modules!

The main reason for choosing python was actually the module MDAnalysis. It allows to easily read any MD trajectory, unlike the c-based AmideImaps, which can only deal with a specific version of GROMACS. Together with cProfile, the computationally intense sections can easily be found, and converted into c code. Yes, the python module numba can do that, too, but the human-written version is roughly twice as fast.

For writing any c code, it is important to remember that it has to interface with python, and multiple OS's. **Include sth on export "C"decldll section!**. This is why all functions use so many pointers: instead of passing huge arrays with data, a single pointer is passed to c. The numpy.ctypeslib submodule is vital here: it is the cheapest way (we could find) to make a c-friendly array, and create a pointer to it. The c-code can directly read (and alter) the arrays provided by python. To make sure that multiple instances of AIM running simultaneously can all access the c-based code, it has been compiled into a dynamic library. Downside of the dynamic library is that different OS's work with them differently. Therefore, the .cpp script actually has to look different for Windows and Unix systems. While both require the functions to be part of an export section, Windows systems want an extra export section. This section is found at the top, and all lines within it start with `__declspec(dllexport)`. For Unix systems, this section actually creates errors, and should therefore be ignored by using block-comments.

If you ever wish to alter a c function or add one of your own, read into cTypes. Not only do you have to specify data types in the c++ script, you will also need to do so in the python script itself. As a rule of thumb: the simpler a piece of code is, and the more expensive, the better it is to convert into c. Most pieces taking over 10% of total run time have been converted.

4.2 default_input.txt

This file is the reason why the program can run with incomplete input files: whenever a parameter is missing from the submitted input file, it is read from this file instead. The file can be specified in the input parameters file. If not specified, it is expected under the default name in the default location, which are hard-coded into the python script. For all other files, the default location and filename are denoted in this default input file.

In principle, this file does never need to be changed: everything can be done by specifying it in the normal input parameter file. However, as default settings had to be included anyways, they were made accessible. If the defaults are not suitable for your systems and you get tired of having to specify many parameters every single time, you could instead change the defaults to something more suitable. Be careful, however: if a parameter is missing from this file, the program cannot run. Please note that specified file locations are assumed relative to the default parameters file.

While most parameters in this file are the same as those in the input parameters file (see section 2.2), there are some differences. Firstly, the option `def_parfile` is only available in the input parameters file (and not in this default file) for obvious reasons. Furthermore, there are 4 parameters in this file that are not included in the input parameter file: the libfile location for 4 different operating systems: 32-bit Windows, 64-bit Windows, MacOS and Linux. These are part of the program's attempt to load a non-specified c library file, and one should not have to change these: change the main libfile parameter instead.

4.3 md_0_1

There are two files starting with md_0_1: md_0_1.tpr and md_0_1_short.xtc. These are the default topology and trajectory files, used for running the program in Demo Mode. They correspond to a file created by following the lysosome in water GROMACS tutorial, of the lysosome protein (1AKI in the RCSB PDB). The final (production) run was different from the tutorial: it was made using GROMACS 4.6.3 (for compatibility with the old AmideImaps program), with a simulation timestep of 2 fs, and a saved frame every 20 fs.

4.4 .md_0_1_short.xtc_offsets.npz

This file did not come with the original download of the program. Instead, it is a file created by MDanalysis. This file only gets created when the program has used the Demo Mode files. Similar files will be created for every trajectory file, in the same folder as the used trajectory file.

According to MDanalysis' extensive documentation pages, this file is used for quick access to the different frames of the trajectory files. These trajectory files can grow quite large, making it slow to read through them every single time to read a random frame. The offsets files tell MDanalysis where certain frames are stored.

4.5 references.dat

This file contains some the references used by AIM. References belonging to either frequency maps (see section 4.8) or dipole maps (see section 4.9) are stored in the same file as 'their' map. Similarly, references belonging to user-added extra maps (see section 5) are saved in that same file, too. The remaining maps go in this `reference.dat` file. These are maps belonging to the different coupling methods, and methods that are not stored in a file, but are part of the code itself.

As you might have to add your own references in (mainly intended for those who create their own extra .(c)maps, or their own frequency/dipole maps in the sourcefiles folder), the rest of this section will be a discussion of the format of these references.

The references are stored in a format heavily based on that of .bib files created by most reference manager software. It is also a commonly available format when downloading references from journal webpages. The main reason is to simplify (and reduce errors in) the process of creating a new reference. You can usually just copy the .bib file entry, and make the required minor changes. These changes are required as there are three differences between the .bib format and the one used by AIM:

Firstly, AIM ignores the first line (starting with @). It needs it to be present for automatic recognition, but only reads the @ - the reference type and keyword used are not interpreted. Similarly, AIM does not use all available bits of information. However, if the following entries are missing, AIM will report them as missing when printing references:

- author
- title
- journal
- volume
- number
- year

The following items can be used when present. When not, AIM will just not print anything regarding these.

- pages
- issn
- doi
- url

Secondly, AIM has added a new entry: AIMnotes. AIM requires the presence of this one, so make sure to add it! The format is the same as for the other entries. Inside the curly braces, information on what the reference did for the map is stored. An example can be found in the 'Map-E_Jansen.dat' file: this map is based on two different references; one for the general (and side-chain) map, and one for the proline map. If the system did not contain a proline, that reference is not printed. And vice versa, if only the proline map was used (and not the general), only the proline reference is printed.

In the case of 'Map-E' files, there's three available keywords to use: `EmapGen` for the reference(s) corresponding to the general map, `EmapPro` for those corresponding to the pre-proline map, and `EmapSC` for those of the side-chain map. Similarly, for the 'Map-D' files, there are the `DmapGen`, `DmapPro` and `DmapSC` keywords available. If a reference is relevant for more than one map, both keywords can be added, separated by a white space like this: `{EmapGen EmapSC}`.

In the case of .map files, you are more free to choose your own keywords, due to the fact that these files also require a python function dealing with the references (see section 5 for more details). It is suggested, however to use 'frequency' for maps/methods calculating the frequency, 'dipole' for the dipole maps/methods, and 'Raman' for the raman tensor maps/methods.

For .cmap files, the AIM_notes entry is not used. All references are always printed.

Finally, as the references.dat file only contain references for maps/methods AIM has in its own code, you should only be adding references here if you're changing AIM itself. If you're changing code, you just have to make sure the code can read the keywords.

The last difference should not be much of a difference if you already personally maintain/edit your bib files (which you should - often, there are misstakes, but that's a different story). Whenever you copy a new .bib entry into one of AIM's files, please do the following:

- Check the title field. Often, journals place curly braces (the `{}` characters) around parts that no software should change capitalization of, or add other 'odd' characters. Make sure to remove these. AIM does not format the text in any way, and python can interpret these characters (especially curly braces) differently.
- Check the authors. There should be the following format: author1_lastname, author1_firstname and author2_lastname1 author2_lastname2, author2_firstname1 author2_firstname2. It occasionally happens that author's first and last names are switched, for example.
- Check the journal field. This field should contain the official abbreviation of the journal, not the full name.
- Finally, you can check the other fields are present and correct, too. It just seems that the ones mentioned above are most error-prone!

4.6 resnames.dat

This file stores the accepted residue names used for the 'influencers' parameter in the input file (see section 2.2). Each line is used to define a new group, which can be done in two ways:

The first way is by defining all residue names separately. In the default resnames file, an example of this is the line `Na NA NA+`: The group called "`Na`" consists of two different residue names - "`NA`" and "`NA+`". If this group is selected for in the input parameter file, all atoms belonging to a residue with the name "`NA`" or the name "`NA+`" will be selected.

The second way is by combining existing groups. In the default resnames file, an example of this is the line `add Ions K Na Cl`. The first word, `add`, indicates that the following group will be specified in terms of previously specified groups. The next word on the line, `Ions`, is the name of this new group. The other items on the line are the groups to be merged. In this case, all residue names specified in the K, Na and Cl categories will make up the Ions group.

A few points deserve special attention:

- Not all group names can be defined: `Protein` and `All` (mind the capitalization) are already taken, and cannot be used. When attempted to do so anyways, the program throws an error and quits. They can, however, still be called to be merged with something else. So, while `Protein GLY PRO` will throw an error, `add Newgroup Protein newresidues` will work normally, as long as the group `newresidues` has been specified on a previous line.

- The group name `Protein_ext` can be used (and might be indispensable in some cases), but requires special attention!
  Usually, specified residues are environmental. They are part of the solvent, a membrane, or some other non-protein entity. However, not all proteins consist of only the 20 default residue names, and it might be desired to expand the list of residues that make up the protein. This is done using the `Protein_ext` group: items in this group will be added to the main "Protein" list in the script. This means that the added residues will be assumed to be part of the same backbone, and are assumed to have a few special properties. For example, they must contain a few atoms with very special names. Examples of residues that can be included are (de)protonated versions of functioning residues. Examples are the 'HSD', 'HSE' and 'HSP' used by CHARMM to replace 'HIS'. Some residues are too different, and must be specially included into the code itself. Examples are the 'FOR' and 'ETA' residues (see section 2.2). These kinds of residues can only be included by changing the code itself.

- The file is read sequentially. When adding groups together, the added groups have to be specified on lines higher up. If a group is redefined lower in the file, the groups depending on it will NOT be updated.

- Residue names cannot contain whitespaces, these are used as separators. Apart from that, residue names may contain numbers and special characters (but using \, ' and " should be avoided, as python uses it internally for specifying special characters). The amount of spaces separating terms is not important, and can be changed for ease of reading the file.

4.7 atnames_MDpackage.dat

This file stores the atom names used for the calculation. These names differ between MD packages, so every package requires it's own file. It currently contains the names of 7 special atoms. The first five are part of the backbone part of each residue, and required for defining the amide bonds. The other two are for atoms present only when the residue is one of the two terminal residues of the chain.

Multiple atom names can be assigned to a single category (as is done with the terminal-indicating atoms), but be careful that these names CANNOT be used for anything else, and that only one is present in each residue!

4.8 Map-E_xx.dat

These files store the maps for calculating frequencies. They contain three blocks of data of six lines each. The first line contains 'defmap' to indicate that a new map will follow. The name of the map is specified right after. Note that every electrostatic map file must contain a map for every type: 'Gen' (generic), 'Pro' for proline groups and 'SC' for side-chain groups. If there is no special map made for that instance, another map can be copied. As an example in the Jansen file, the 'SC' map is a copy of the 'Gen' map. Which of the files is used can be selected by using `map_choice xx`. For more info, see section 2.2.

The line starting with 'defmap' is followed by 7 more lines. The first of these has the omega value: the unshifted frequency. The other 6 lines are for four of the atoms of the amide group, and two connected ones: C, O, CA, N, H, and another CA. The first column is for the potential, the second, third and fourth are for the electric field, the final six are for the electric field gradient. The final frequency of a group is found by adding the product of its potential with the potential value of the map to the unshifted frequency. This happens for electric field and gradient too, and for all six atoms (although the CA's are far less useful for the amides). For more information, see section 6.3.1.

The one exeption to the defmap structure is the references. For code-related reasons, the references of the map should be preceded by the line `defmap Reference`. Multiple papers may be referenced, each of which occupies multiple lines. The first of these lines always starts with '@' - just like .bib files. For more details on the structure of the references, see section 4.5

4.8.1 Adding your own map

You might want to use other frequency maps than those included in AIM. Depending on the type of map, you should follow one of two ways:

If the map you want to use is for amides, all you need to do is to make a copy of an existing frequency map file, with your name of choice in the file name. If you want to add a map of the name 'BetterFrequencies', the file must be named 'Map-E_BetterFrequencies.dat. Then, you open this new file, and change the values inside. Make sure to not remove/add a number: there should be a line with a single value, followed by 6 lines of 10 values each. Note that this method does not allow you to change the MEANING of the numbers, just their VALUE. After changing the map, you can use it by supplying the same name as in the file name, to the map_choice parameter. For this example, that would be `map_choice BetterFrequencies` .

If you, instead, wish to make a different kind of map and/or a map for a different kind of group than amides, please see chapter 5. It contains all details on how to make your own!

4.9 Map-D_Jansen.dat

This file stores the Jansen map for calculating dipoles. It is structured quite similar to the EMap files, with a notable exception: As the dipoles have three components, each section contains three sets of 7 lines: each structured just like the EMaps, each belonging to a different direction.

Please note that you can add your own dipole maps much like you can add frequency maps. The procedure is the same, too: if you make a (correctly) formatted file with a new name, you can use that name to select it in the input parameters file!

4.10 Map-NN.dat

This file stores the mappings for the nearest neighbour interactions. There are 21 maps total: seven sets of three. The first of a set is for calculating the coupling (`NN_coupling_choice GLDP`), the second for the shift caused by the Nterm neighbour on the diagonal element of the Hamiltonian of another, the third for the shift caused by the Cterm neighbour on the diagonal element of the Hamiltonian of another (`TreatNN`).

The seven maps are for different configurations. There are two types of amide bond: those that have a N donated by a proline (denoted Pro), and those that have not (denoted Gly). There are maps for 7 possible combinations (in the same order as in the file): Gly-Gly, transGly-transPro, cisGly-transPro, transPro-transGly, transDPro-transGly, cisPro-transGly, cisDPro-transGly.

The application of these maps depends on the Ramachandran angles: the 13 lines containing 13 columns for each map are for a 360 degree rotation in 30-degree bins. Depending on the angles phi and psi, the four values surrounding these angles are bilinearly interpolated to find the correct coupling/shift due to the nearest neighbours.

4.11 Map-C_Tasumi.dat

This file stores the Tasumi mapping (`NN_coupling_choice Tasumi`). Its use is identical to that of the NN.dat map, so see section 4.10 for details. Please note that these maps have their angles swapped: between the Tasumi and GLDP coupling, phi and psi are swapped.

4.12 Map-C_TCC

This file stores the TCC mapping (`coupling_choice TCC` and `NN_coupling_choice TCC`). It loops over all six atoms of both amide groups (so, 36 pairs in total), and adds the influence of each pair to a grand total. Section 6.3.2 explains the map in more detail.

Chapter 5

Extra maps

The option to read in additional maps allows users to calculate their own preferred part of the IR spectrum, without needing the dev's of AIM for every new kind of oscillator. The creation of these maps is considered to be advanced usage of AIM, while using community-created maps should be doable for every user.

Before this section goes into detail for those wanting to create these maps, a few important things for the general users: Maps for calculating frequencies and dipoles are stored as somename.map. The filenames can be changed if desired, this has no influence whatsoever on the functioning of AIM. The extension (.map) can be changed (to .nomap for example) if you want AIM to ignore the file. Similarly, maps for coupling models are stored as somename.cmap. Again, the name itself doesn't matter, and the extension can be changed to disable the map if desired.

Another detail worth mentioning for everyone is about safety. To give users the freedom to incorporate any kind of map into AIM, they have the option to supply their own code. AIM cannot check whether that code is malicious - it can only execute. Therefore, make sure you trust any maps you download and use. The advantage of these map files is that they are simple text files, so you can open them and see for yourself how they work. We hope a warning like this isn't necessary - that everyone in the community is as nice as the people we've personally met. But you never know, and should be cautious like that about every program you find on the internet, including AIM itself. With that out of the way, lets get more technical.

While .map files and .cmap files contain instructions for computing different things, their format is very similar. Both are structured using headers. A header looks like `[ this ]`, where instead of 'this', the title is given. There are a few sections dedicated to specifying some general info, values, etc, and a section dedicated to python code. As the latter is typically the largest by far, we will treat that one in a separate part. You could see the non-code part of these files as ingredients, and the code part as the recipe, explaining how to apply the ingredients.

The sections of the .map and .cmap files are very similar, the few differences are outlined in their respective sections. Before delving into that, a few general remarks on the structure of these files:

- There can only be one header of each kind. It is not possible to specify part of the python code, for example, then something else, and then another part of python code. While the order of the sections does not matter, the manual and example files both have the same order, and we encourage users to also use this order in their own files (to make sharing easier).

- Just as with python code itself, AIM input parameter files and resnames/atnames files, the '#' character is used to add text that the program should ignore. You can use it to add some explanation to a line (for example, units for the values in your maps), or to turn lines on and off. All text on the line after the '#' character is ignored.

- Again, just as with the AIM input parameter files and resnames/atnames files, any whitespace (empty lines, extra spaces in a line, etc) is for readability. The program requires headers and other information to be on their own line, but whether there are extra empty lines in between does not matter. The same holds for spaces: sometimes, a space is required. But it does not matter whether there is one, or more, and neither does it matter if a line starts/ends with extra spaces. The one exception to this is the python code: there, follow the standard python rules.

- When in doubt about format, look at the example .map and .cmap files that came with AIM!

5.1 non-code sections of .map files

The non-code part of .map files is separated out into a few parts, each with their own format. Of these, Identifiers and Resname + Atnames must be present, Emap data and Dmap data are optional.

5.1.1 Identifiers

The identifiers are basically how to refer to the map. This section is just three lines, each consisting of two words. The format is basically identical to that of the AIM input parameter files. However, unlike those files, a map is user-made, so a default cannot be predicted/given by AIM. Therefore, all three MUST be present. They are 'name', 'ID', and 'inprot' (when in the file, don't use the quotation marks).

'name' is the most important one to the user. As mentioned in section 2.2, this is the name that the user specifies in the input parameter file, to indicate that this map should be used. A name cannot include spaces, (back)slashes or '#', but (capital) letters, numbers, dashes and underscores are allowed. A good name makes it clear what kind of group is described by a map, without being too long. Users have to type in this name every single time when calling the map. Also, all names must be unique.

Opposed to 'name', 'ID' is most important to AIM itself. It is how the map is referred to internally. A map ID must be an integer number (that can fit in an int32). Apart from creating .cmap files, users will not see or use the .map ID number.

Because its importance internally, again, ID's must be unique. Numbers 0-9 are reserved for AIM internally, where 0 is defined in UniverseMaster.py for amide backbones. Due to these groups having multiple parts in different residues, they could not be converted to a .map file. If you want to apply a different map to these, use the 'Custom' option for the map_choice keyword in the input parameter files. Amide groups in protein side chains, however, are defined as a .map file. Still, they are assigned a reserved ID of 1, as part of the coupling methods rely on them.

In order to avoid many people in the community using the same ID for their own maps, we would like to emphasize that the specific ID matters little. The ID 123456789 works equally well as 123. Probably, different researchers or research groups would like their own series of numbers - for example, all ID's between 8300 and 8400 are claimed by research group x. If everyone has their own series, ID clashes will rarely occur. If they do occur, simply changing the number to another should fix it.

Finally, 'inprot' is a bit of an oddball here. It is only used once in AIM: when printing the system at the end of a calculation. For each protein chain, AIM prints which oscillating groups in it have been taken into account. After, it does the same scan for all groups outside a protein chain. 'inprot' determines whether this group should be listed amongst the protein oscillators, or not. There are just two options: 'True' and 'False' (no quotation marks, mind the capitalization!)

5.1.2 Resname + Atnames

This section specifies what the oscillating group looks like. There can be multiple lines, each listing one or more different allowed configurations. In its simplest form, a line consists of 7 items, separated by spaces. The first item is the name of the residue. As an example, lets take the amides in protein sidechains. That .map file contains the line `ASN CG OD1 CB ND2 HD21 HD22`. The first item, ASN, refers to the name of the residue that this group can be expected in. Naturally, as there are two canonical amino acids with an amide in the side chain, the .map file contains a second line for the other residue.

The other six items in this line are the names of atoms. In all tested MD packages, the atoms of the canonical amino acids have (virtually) the same names. Therefore, only one line is needed. When creating a .map file for your new oscillating group, you just have to determine what names are used in the MD software.

One major thing is implied here: your map requires 6 atoms. A good thing to ask: what does 'require' really mean? All atoms specified here must be present for this map to be applied. In later steps, the properties of these 6 atoms can be easily retrieved: their position, charge, mass, etc. Furthermore, if your map needs to know the electric potential, field and/or gradient at a certain atom's position, that atom has to be included here. However, these electric properties are not necessarily calculated for all 6; see section 5.2.3 for more info. In general, this list of 6 contains all atoms that mean a group is of this type.

In case you truly only need fewer (the molecule maybe only contains 3 atoms total, like water), make sure to have duplicates. Just repeat the last one until there are 6 entries. In the case of water: `WAT O H1 H2 H2 H2 H2`.

It might happen that there is multiple options for a given name. Let's go with the simplest example, the ASN mentioned above. Lets imagine that there is some MD package, which doesn't use the name OD1, but OD instead. You could make a new line like this: `ASN CG OD CB ND2 HD21 HD22`, and add it, to get a total of three lines in the file. There is, however, a nicer way. You can add a second option for a given position like this: `ASN CG OD1,OD CB ND2 HD21 HD22`. You can see they are separated by a comma, without a space. In fact, you can specify as many options as you want with comma's; not just two. Also, this is possible for all 7 positions: yes, also the residue name. If multiple items on a single line have multiple options, all combinations are valid, too. Let's take our water example, but expand it: `WAT,SOL O,OW H1 H2 H2 H2 H2`. Now, there are actually 4 lines here. All 4 have the same 5 final items, so let's ignore those. When each would get their own line, the 4 in this example would start with this: `WAT O`, `WAT OW`, `SOL O`, `SOL OW`. If you don't want to allow certain combinations, create multiple lines.

There are multiple reasons why you might want to specify multiple options within a line, or just multiple lines. Firstly, as with the protein sidechain amides, there might just be multiple situations which are all valid. Alternatively, different MD packages might use different names, which all have to be mentioned.

One word of caution, however: be wary of duplicates! When reading in your map, AIM converts the lines with comma-separated options into separate lines, resulting in possibly many lines of 7 single items. Every combination of atoms that satisfies one of these lines is taken as a valid candidate. Maybe, the best way of explaining is to just show the (pseudo) code of the segment of AIM that finds all oscillating groups in the MD system. This segment of code can be found in the functions osc_finder_extra() and find_osc_atoms() of the Universe() class in the UniverseMaster.py script:

```
def osc_finder_extra():
    create container for all oscillating groups to treat during this calculation
    for every residue in the system:
        for every map chosen by the user:
            if the name of the residue is listed in this map's configurations:
                do find_osc_atoms(residue, map)
                    if any sets were found here:
                    save all the sets to the container

def find_osc_atoms(residue, map):
    create container to return
    for each configuration in the map:
        for each atom in the residue:
            if the atom is one of the 6 of this configuration:
                save the atom to a temporary structure
        after looking through all atoms:
            if all 6 atoms of this configuration have been found:
               save them to the return container
            else:
                ignore this residue
    after looking at all allowed configurations:
        return the container to osc_finder_extra()
```

No futher checks are done to detect duplicates. Every time any group of atoms matches a configuration of a requested .map method, it is treated. For good reason: a single residue may contain multiple instances of a group (consider a small molecule with two terminal amides, one on each end, for example), all of which you want to treat. Similarly, maybe, a given group has two vibrational modes in the frequency range you're interested in. For every mode, you include a map, and these maps should find the same groups - every group of atoms has to be considered again for every possible vibrational mode.

If all this sounds too confusing, just use the output of AIM. After running a (short!) test calculation, it prints how many groups of a certain type it found. Just check whether those numbers are what you expected them to be!

5.1.3 Emap data and Dmap data

These are actually two distinct sections. However, as they are treated identically, both can be explained simultaneously. In these sections, you can add the values your maps should use. Yes, you could just type

them directly into the python code, but adding them here should be more convenient, and clearer. If this sounds vague, that's because there are many options, including specifying nothing at all!

You should imaging the data specified here to be rectangular 2-dimensional arrays. One or both of the dimensions may have a size of one. That's a complicated way of saying that a single number is allowed to - that's basically a grid of 1 item high, and 1 item across. All items must be specified, but may be zero. The reason is that each grid will be interpreted as a numpy array. Every row in such an array must have equal items for it to work. That is why you can have multiple arrays. Most maps will have a default value (called omega), to which corrections are added based on the properties of the field. The Emaps used for amides could, for example, be encoded into a 1*1 array for omega, and another 6*10 (6 high, 10 across) array for the influence of the electric properties. Most of the 60 values in that array would be 0's.

Every array (or grid, if you like) has to have a name, to refer to it later. How to refer to it, you will find in section 5.2.7. This name is specified using the text 'defname', followed by a space, then a name chosen by you. All rows of data following a 'defname' line are assumed part of that map, until the next 'defname' is found. All lines between two consecutive 'defname' lines are assumed to have the same amount of values on them, so they can form the aforementioned grid. Do note, that you can use '#' at the end of a row to indicate what the row codes for, or on a blank line in between to add the meaning of each column. Or one to denote the units!

Please do note that the arrays can only contain numbers. Either whole numbers (integers) or ones with a decimal point (floats), but numbers. You can't specify an array containing names here.

5.1.4 References

AIM prints the references to papers that correspond to the maps used. This way, it is easy for users to know which literature to cite for that calculation. This is also possible with .map files. All information necessary for referencing your sources should go in this section, along with some information for AIM to know what paper was used for what purpose.

Adding a reference to a used maps takes a few steps. For more details on any of these steps, you can always check the corresponding sections.

- Make the 'References' section in your .map file by adding the corresponding header.
- Add the basic bib-style entries to this section, and check/change the entries where necessary (see section 4.5)
- Double check you added the 'AIMnotes' entry to every reference. Within the curly brackets, add what it's used for: we suggest using 'frequency' for references belonging to the frequency calculation, 'dipole' for references belonging to the dipole calculation, and 'Raman' for those belonging to the calculation of the Raman tensor.
- Complete the set_references function in the code section. This is necessary, as maps can sometimes also need references from AIM itself, or only use certain references based on settings in AIM. See section 5.2.1 for more details.
- Last but not least, don't forget to check if the program now displays the intended behaviour!

5.2 code section of .map files

This is where we're going to go full on technical. As you have to know some python to write the python code here, we will assume the same here. But please, don't let this scare you too much: with the example files to help, you probably should be able to write these instructions if you know another programming language than python.

Each of the following functions must be present in order for the code to work. However, not all of them have to be filled: the functions pre_calc() and pre_frame() may only contain the text 'pass'.

Testing during development of the external map for amides in protein sidechains shows that the efficiency of the code in these .map files can heavily influence the efficiency of AIM in general, especially if the code is quite inefficient. More details will follow in the corresponding sections, but know that most data in AIM is stored in numpy arrays. When code is kept numpy-only (using functions like np.dot), it can be very fast, as numpy is optimized very well. However, looping over a numpy array the python way (for item in nparray:) is slower than looping over an identically-sized python list. Some hints on speeding up your code can be found at the end of this section (section 5.2.10).

5.2.1 set_references

This function takes three arguments: self (the way to get to the contents of the map itself), an instance of the filelocations class (FILES), and an instance of the RunParameters class (RunPar), both as defined in FileHandler.py. self contains all information about the map itself (all information added into this file), FILES contains all information regarding input and output files, and RunPar contains all parameters as the user chose them with the input parameter file. The function should return a list of lists. Each of the sublists should have a reference object as the first entry, and a list of strings as the second entry.

The output might sound unnecessarily complex. Please realize that you are allowed to have your map return not only references from the map's 'References' section, but also any other reference available to AIM. To avoid clashes and inconsistencies, you should not edit the AIMnotes attribute of references (especially those from within the core of AIM), and with this format, you don't have to. Whatever you'd put in the AIMnotes attribute, you now enter as the second entry of a sublist.

This is also the reason for having both a python function and a section dedicated to references. The section allows you to add your own, while the python function allows to pull extra ones from AIM, or only actually use some of the references in the 'References' section.

Any reference in the returned list is considered by AIM. AIM checks what the .map file was actually used for (due to the `output_type` parameter, maybe, only dipoles were calculated, but not the hamiltonian). If the map was used for calculating the frequency, any references that have the 'frequency' keyword in the sublist will be printed at the end of the calculation. Similarly, if the map was used for calculating dipoles, any references with the 'dipoles' keyword are printed, just as those with the 'Raman' keyword when using the raman calculation stored inside.

The amide_sidechain.map file contains a good example, as it uses both types of references. The references from the 'References' section in the .map file itself are stored as self.rawreferences. This is a list of reference objects (see section 4.5). The function loops over all references, and adds them to the list that should be returned by the function later. The second object in each sublist is directly taken to be the reference.AIMnotes attribute. Please note that this only works if you use the suggested keywords in section 5.1.4.

After considering the references from the 'References' section, it adds the references that are stored within AIM. The frequency maps are stored with amide-backbone maps. So, it retrieves the map used for the frequency calculation and takes its .references attribute. Then, it looks through this list of references for any that have the keyword 'EmapSC' (denoting the reference is used for the side-chains map). Each of those who do is added to the returned list of references. Note that instead of passing on the default AIMnotes attribute, now, 'frequency' is added to the sublist.

The same is done for the dipoles. The Torii map does not have its own file, so its reference is found separately. For any other dipole map, the method is identical to that of the frequency as discussed above.

Please note that the amide_sidechain.map file only displays one way of having a RunPar-dependent reference selection. Many others are possible, depending on your needs. And of course, if your references don't change, you can treat all of them like the example treated the Raman reference.

5.2.2 set_use_G

This function has to take two arguments: an instance of the filelocations class, and an instance of the RunParameters class as defined in FileHandler.py. Internally, these instances are usually called FILES and RunPar. FILES contains all information regarding input and output files, RunPar contains all parameters as the user chose them with the input parameter file. The function should return a python boolean: True or False.

This function basically defines whether the electric gradient has to be computed in order to apply the map specified in this .map file. If the gradient is required, it is calculated in all six directions, on all required atoms. It's either all, or nothing.

Generally, you know what kind of values your map uses, so you know whether you need to know the gradients on the atoms or not. In those cases, this function contains only a single line of code: 'return True' if you do need the gradients, 'return False' otherwise. You don't need the RunPar object. Still it is included for cases like the amide sidechains - depending on the map the user chose using the map_choice parameter, they are or aren't necessary. If, for some reason, your map also depends on some user-chosen parameter, this is the way to access that choice.

Calculating the gradients is quite expensive. Expensive enough to not just do it for every atom, but give the makers of a map the choice on whether they actually need the gradients, or not. However, they can be a valuable source of extra accuracy for a map!

5.2.3 set_relevant_j

Similar to set_use_G, this function, too, takes two arguments, FILES and RunPar. Unlike it, however, this function returns a list of integers. These integers can take values 0-5, 5 included.

If 0 is included in the list, that means that for the 0th atom (say: zeroeth, or, in normal english, first), the electric properties need to be calculated. Similarly, a 1 means that for the 1th (say: oneth, or, in normal english, second - all hail computers starting counting at 0) atom, the electric properties are desired. Here, the zeroeth atom is the first atom specified in the Resname + Atnames section. Take our water example again. The water was defined as `WAT O H1 H2 H2 H2 H2`. This means that the oxygen atom is the zeroeth atom, the hydrogen labelled H1 the oneth, etc. Say we need the electric properties on all three atoms, this function should contain the following code: `return [0, 1, 2]`. This immediately shows why a function is needed to define this: as 4 of the atom names are identical, there is no use calculating for that atom more than once.

Why not just infer from the list, you might ask? One example for this could be the amide sidechain. While certain maps (like the TCC coupling method) require to know the position of the third atom, none require the electric properties at that position. Then, there is a unique atom in the list, but we still don't need to calculate the field for it. As calculating the field properties is quite expensive, we'd like to avoid doing it whenever we can. Therefore, you specify for which atoms you actually need it!

5.2.4 local_finder

This function takes a single argument: a list of the indices of the six atoms that make up this group. It should return a sorted list of all atoms that should be considered local.

So, what makes an atom local? To calculate the electric properties (say potential) on a given position, AIM looks at the surrounding atoms. If an atom is close enough (the distance is supplied by the SphereSize parameter), it's influence on that given position is calculated, and added for the total. After considering all atoms, that total is the actual potential. You can imagine running into problems, however: the atom itself, is also an atom within range, as is every other atom that is part of this group. But usually, it is not meaningful to take these atoms into account. Which are or aren't meaningful depends on how your map is created.

Usually, a map is created using DFT calculations. A model system is put in a given environment, and a frequency is found. Then, the frequencies are compared to the environments to create a map. Probably, atoms that are part of the model system (or at least close to / part of the oscillating group) are not taken into account when constructing this map. Conversely, these same atoms should not be taken into account when AIM computes the electric properties - in other words, they should be considered local.

Most MD packages save the atom positions as a big list. Imagine your file containing lines, each having three numbers - an x, y and z coordinate. As these files contain many frames, it is a waste of space to include much information about the atom every frame. In order to still link these positions to a certain atom, the position in the file is used. The topology file contains atom names, types, charges, positions, etc. The first line has all info on the first atom, the second on the second, etc. Then, the positions on the first line of a frame correspond to the first atom.

In order to access all these atom properties, each atom is assigned an index, in the same order as they appeared in the MD output files. Using these indices, all properties of an atom can be retrieved. That is why AIM saves all oscillating groups as atom indices. If you need the position of your first atom, you look up the position of the n'th item, where n is the index saved into the first slot. Basically, this is the cheapest and most convenient way to store and access any information.

The MD package might give the atoms of your oscillating group in a different order than you specified them in. This is no problem at all, it just means that the indices that make up your group are not necessarily listed smallest to largest. However, it would give big advantages later on if this list is actually sorted. Therefore, what you do in this function, is return the indices of the atoms you actually need, sorted in ascending order. One example of this can be found in the amide sidechain .map file.

5.2.5 pre_calc

The previous three functions are part of the most complicated part of AIM: to actually process all the inputs. What is the system we have? What parts of it are we interested in? What are we actually going to calculate? Part of this is done by reading the input parameter and default parameter files, but also by investigating the MD trajectories. The result is a variety of created datastructures: the FILES and RunPar objects, for example. But also the creation of extramap objects, in which all info in the .map files is saved. Another example are the static properties of the system: atom names, residue numbers they belong to, atom masses, charges, etc. The array in which all atom indices are saved for the oscillating groups, or the array that stores which coupling method to use for any given pair of oscillators.

It is possible that your map requires similar preparation. Instead of doing a certain operation for every group, every frame, you can opt to only do it once. This function is called at the very last part of the initialization stage: you can make some slight alterations, or do some preparations. For example, AIM works in angstroms, but your map has units of bohr. If desired, using this function, you can do a unit conversion (but maybe it's better to just define the map in the correct units immediately?). But you could also gather other information, and store it for later use. Like a list of all carbon atoms in the same residue as the group is in. In case you need that.

To be able to do this kind of preparation, you are supplied with all info you could possibly need in the form of four arguments. In the order they're supplied, they're called FILES, RunPar, WS, Map. FILES is a class containing all information on the files used for the calculation, especially their names and locations. RunPar contains all choices for the parameters for this run. WS (whole system) contains all information on the MD system. For example, WS.positions is where the positions of all atoms are saved. Finally, Map is the object of the map that this .map file is defining, and probably, this is the one you should get most acquainted with.

The name and ID defined in the Identifier section can be accessed as Map.name and Map.ID. All those valid combinations of atom names as specified in Resname + Atnames? Those are saved in Map.makeup. The Emap data arrays are saved in Map.Emap, Dmap arrays in Map.Dmap. And of course, all the functions we're now talking about are stored in Map.functions. If you, for example, would like to call the local_finder function, you can access it through Map.functions["local_finder"]. The main reason for listing all this here, is that anything you create and save, please save it as part of this Map instance. It will be available in future steps.

For example, lets say you would like to save the text 'test' to this Map object. By using the python setattr, you can do that! The line `setatrr(Map, "text", "test")` saves it, and performing `print(Map.text)` will print 'test'.

5.2.6 pre_frame

This function is very similar to the pre_calc function explained above. It takes the same for arguments, and allows the user to prepare any necessary information for this frame. Again, make it a habit to save anything as part of this Map instance, and only read from the other objects.

The difference? pre_calc is only called once after initialization, while pre_frame is called at the beginning of every frame.

5.2.7 calc_freq

This is where the heart of the map lies. This function returns the calculated frequency for a given group. It takes 8 arguments: P, E, G, OscGroup, FILES, RunPar, WS, Map. Better get used to the final 4, they're the same as pre_calc and pre_frame used. You'll see them more, the explanation is given in section 5.2.5. As for the others, P, E, and G contain the calculated electric potential (P), field (E), and gradient (G). They all have 6 rows, one for each atom in the group. P has 1 column, E has 3 (x, y and z directions), and G has 6 (xx, yy, zz, xy, xz and yz directions). These are in units of angstrom and unit charges.

OscGroup is slightly more complicated. The indices of the six atoms that make up an oscillating group are saved in WS.AllOscGroups. That is an n * 18 array (18 = 3*6, this is needed for the amides), with n being the size of the final hamiltonian. For each frame, AIM loops over these n groups. The index in this array of a given group is called OscGroup. Conversely, if, in this function, you take the slice `WS.AllOscGroups[OscGroup, 6:12]`, you get the 6 indices of the atoms that make up this group.

It is very important to realize that most maps are defined in terms of local coordinates/directions. For example, the bond between two atoms might be considered the x direction. The definition of these

directions will differ between maps, however, so it is impossible to have AIM prepare them. Instead, you will have to do the rotation. This is not as hard as it sounds, however: if you provide the correct rotation matrix, there are functions in AIM you can call to perform the rotation. The only thing left to do is to specify this rotation matrix.

A nice example of the rotation matrix can be found in the amide .map file. Each of the rows is a unit vector in the desired direction. Often, the first direction is specified along a certain atomic bond. The second is then along another, but only taking the component perpendicular to the direction. The third is perpendicular to both others, usually found using a cross product.

A bond vector can most easily be found by subtracting the position of one atom from the other. However, for most MD systems, the periodic boundary conditions (hereafter referred to as pbc) have to be taken into account. Therefore, to find the correct (usually shortest) distance vector, a special difference has to be taken between these two points. Luckily, a function for exactly that can be found in the MathFunctions.py script, accessed by calling AIM_MF.PBC_diff(). Any other function you need for constructing the rotation matrix as described above can also be found as part of AIM_MF.

The function is expected to return both the calculated frequency, and the rotation matrix used.

5.2.8 calc_dipole

Similar to calc_freq, this function is called by AIM to calculate the dipole moment of a specific group for a specific frame. Just like calc_freq, it takes the P, E, G and OscGroup arguments. The fifth argument of this function is the same rotation matrix as was returned by calc_freq. The final four arguments are again FILES, RunPar, WS and Map.

Important to note is that if calc_freq modifies the P, E or G, these modifications are kept in the arrays fed into here, into calc_dipole. That means that if the dipole and frequency maps assume the same units and rotation, these do not have to be applied again.

The calculated dipole can, after just applying the map, be in 'local' coordinates of this group. Before returning the dipole moment, it should be converted back into the box coordinates. This is made much easier as the rotation matrix used before is supplied to this function.

Not only in-box-coordinates dipole is expected as an output: a second output is expected. This second output is the location of the dipole. Most likely, the dipole moment lives on one of the used atoms, or at a fixed position in between them. The reason both of these are required is that these two vectors are needed for calculating the standard dipole-dipole coupling between this, and another group later. Therefore, it is extra important that the units are correct.

5.2.9 calc_raman

Similar to calc_freq, this function is called by AIM to calculate the Raman tensor of a specific group for a specific frame. Just like calc_freq, it takes the P, E, G and OscGroup arguments. The fifth argument of this function is the same rotation matrix as was returned by calc_freq. The final four arguments are again FILES, RunPar, WS and Map.

Important to note is that if calc_freq or calc_dipole modify the P, E or G, these modifications are kept in the arrays fed into here, into calc_raman. That means that if the Raman, dipole and frequency maps assume the same units and rotation, these do not have to be applied again.

The calculated Raman tensor can, after just applying the map, be in 'local' coordinates of this group. Before returning the Raman tensor, it should be converted back into the box coordinates. This is made much easier as the rotation matrix used before is supplied to this function.

The function is expected to return a numpy array of 6 floats. The first of these is the xx-component of the tensor, followed by the xy-, xz-, yy-, yz- and zz-components.

5.2.10 speeding up your code - the easy way

As mentioned, tests have shown that a single inefficient function can slow down the entire program. If you are not concerned about how long the calculation takes to run, you can skip this section. However, it is most likely worth a read.

If you haven't worked with numpy before, here is a very short intro: numpy arrays are somewhat similar to lists. A 1D array can be seen as a list, a 2D as a list of lists, etc. They are accessed similarly: in a list of lists, you would access an item like this `lst[1][2]`, the same item in a numpy array would be accessed using `arr[1, 2]`. There are, however, a few key differences: python lists don't require all items to be the same. Therefore, if you have a list of lists, all lists in the 'main' list can contain anything,

and have any length. In a numpy array, all of them have to be of the same length (or, in numpy terms, shape), giving something like a filled rectangular grid. The other difference is that all items have to have the same datatype. While numpy arrays technically can be composed of strings, this is not usual. In general, a numpy array is composed of just integers, or just floats, etc.

So, what's the advantage of numpy arrays? While looping over one (as mentioned in the intro of this chapter) is slower than looping over python lists, numpy has a huge amount of supported functions optimized for these arrays. Being based in C, many of these operations are much, much faster than any python implementation could ever be. This means that when used correctly, numpy can speed up the python code. Furthermore, numpy also has a built-in functionality that converts the numpy arrays into c-friendly arrays, allowing AIM to do part of the calculation in C++.

So, how do we do our calculations quickly? Take the equation in section 6.3.1. When applying the map on the electric fields, a 'classic' python implementation would look as follows:

```
freq = omega
for j in range(6):
    for direction in range(3):
        freq += Emap[j, direction] * E[j, direction]
```

This works perfectly fine, exactly as expected. However, a faster implementation would use numpy functions:

```
import numpy as np
freq = omega
freq += np.sum(np.multiply(Emap, E))
```

Both code snippets give the same result, but the second runs faster. The result of np.multiply is the array obtained by element-wise multiplication of E and Emap. np.sum then adds up all the entries in that array.

P, E, and G always have fixed dimensions. P is a 1D array of 6 items. E and G are both 2D arrays. E has shape 6 by 3, G has 6 by 6. Most often, however, you won't need the field properties on all 6 atoms. In those cases, you can slice a part of these arrays and use them. The function AIM_MF.apply_map automatically performs the equation given in section 6.3.1, only on the requested atoms. These atom indices are stored in the correct format for this function in Map.relevant_j_arr.

You might have noticed in the example files that any newly created array is assigned a data type. When filling an array, numpy can easily detect whether it should contain integers or floats, but this is not enough information. When converting to c-compatible arrays, the size of a single entry must also be set. In the case of AIM, all arrays have 32-bit slots. So, 32-bit floats and ints. Numpy does have a default it can use, but this default differs between operating systems. So, to get consistent results, the size of a single entry is also defined.

Why are we mentioning this here? Well, these data types serve a different purpose, too. As mentioned in the very beginning of this manual, AIM also uses the numba library. While the most expensive functions have been translated to c, this is not feasible for literally all of them. Therefore, some more somewhat expensive ones are converted to numba functions. For those unfamiliar with numba: You write some python code, and numba converts it into c for you. This means that there are some restrictions to numba functions: they cannot take complex objects. Numpy arrays are supported (as are most of the common numpy functions), but the 'typing' must be consistent. Hence, even arrays that are not converted to c have a manually assigned data type.

Basically every function in AIM_MF is a numba function. Therefore, the typing of all arrays must be correct in order to use them. In general, arrays in AIM have the dtype 'float32' (use the quotation marks). When creating an array, that looks like this: `np.array (listobj, dtype='float32')`. The only exceptions are the MDA-generated arrays with names (WS.atnames, WS.resnames, WS.types), as well as all arrays that deal with indices. Examples of these are WS.atnums, WS.resnums, Map.relevant_j, WS.AllOscGroups. Each of these can be used as an index in another array to obtain a certain piece of information. Any slicing of numpy arrays must be done using integers, hence why they are stored as such. For these, use `dtype='int32'`

Using numba and numpy together, code is much faster than when using regular python. If you are avoiding looping over numpy arrays and making use of numpy and AIM functions, but your code is still slow, consider the following questions. Are you only computing the things you actually need? Many things are already computed by AIM, and all can be accessed. Furthermore, not all info is actually needed. For example, AIM wants you to define for which atoms you actually need the electric properties,

and whether you need the electric gradients. The calculation for those simply is expensive, so it is more useful to not calculate things when not used. The only other option is to convert functions you really need into c code, but that is too complex to cover here.

5.3 non-code sections of .cmap files

A brief warning: this is written for version 9.10 of AIM. The testing of .cmap files has not yet been as extensive as for the .map files, so their structure is more likely to change. The included example .cmap file will be of the correct format, and can thus be used for reference.

The structure of .cmap files is very similar to that of .map files. The 'Identifiers' and 'Cmap data' sections are most likely what you expect them to be, the 'Coupled OscIDs' section is the .cmap analogue to the .map 'Resname + Atnames' section. In general, all these sections will only outline the difference compared to .map files.

5.3.1 Identifiers

Opposed to the identifiers of .map files, there are only very few here: a single ID number. In general, see the remarks on these ID's in the .map file version of this. These things are different/worth noting:

- There is no name here, only an ID, as users cannot request a specific coupling map. A coupling map describes what kind of oscillators it can couple (in the 'Coupled OscIDs' section), and it will couple all of those, and only those. The IDs of .map files are separate from the IDs of .cmap files, so don't worry about a conflict between the two.
- The lowest available ID is 110. (but just as with .map files, don't worry about getting the lowest possible ID)

5.3.2 Coupled OscIDs

Being the .cmap analogue to 'Resname + Atnames' of .map files, the same syntax is used. Here, however, there are only two items, both integer numbers. They are separated by spaces, and multiple options for one can be supplied using a space-less comma (e.g. `4,5 8,10`). So, what do these numbers represent?

A coupling map is usually designed with a certain type of oscillator(s) in mind. This is where you specify which. The numbers used are the IDs of the .map files of these groups. The above example specifies this map can couple oscillators of type 4 and 8, as well as those of 4 and 10, 5 and 8, and 8 and 10. The groups 4 and 5 are NOT coupled using this map.

Please note that if the IDs of any .map file are changed, they should also be changed here. While combinations using 'reserved' OscIDs (0-9) are possible, be careful with those. If it's between a reserved and a non-reserved one, there should be no problem whatsoever. While combinations between reserved OscIDs (say `0,1  0,1`) are in principle possible (the code allows them), they are treated different from all other coupling methods. Therefore, this kind of map might behave differently from what you intent. Luckily, you shouldn't need to define a new map for these couplings: AIM already has 5 options built-in for you to choose from. You could (if so desired) even alter the values the maps use by changing the values in their respective map files, although only advanced users should do this. And of course, the validity of the obtained results can not be guaranteed if the user changes values themselves.

5.3.3 Cmap data

The information within this section is processed literally identical to the Emap and Dmap data sections in .map files, so please, see that section for reference. The very only difference: the Cmap data is accessed using coupmap.Cmap instead of Map.Emap or Map.Dmap, as is the case for the .map files.

5.3.4 References

This section is very similar to the 'References' section of the .map files. All entries should get the same treatment as over there, but with one difference: Each reference in this section will also be printed by AIM if the map is used at all. That means there is no set_references function like .map files have. Also, there are no keywords at the AIMnotes entry that AIM looks at / uses, so that section is ignored. If this behaviour poses a problem for you, please let us know what behaviour you would like / need, and why!

5.4 code section of .cmap files

There are many similarities between the code section of .map and .cmap files. For formatting tips, see the info on the code of .map files.

Also, many of the arguments are the same as for .map files. The meaning of OscGroup, FILES, RunPar, and WS were already explained in the .map files section. The argument coupmap is the object that stores the contents of the .cmap file, just like Map stores the .map file.

5.4.1 pre_calc

This function is exactly the same as the .map version of it. It is meant to do any preparation, if needed. It is wise to, if you calculate anything, save this as part of the coupmap object.

5.4.2 prep_coupling

This function is exactly the same as the .map function pre_frame. The name is changed for it to match AIM's internal prep methods: almost every coupling method can be prepared. The reason for this is the structure of most maps. In general, if you have, say, 100 groups of a single type, which you all want to couple. Every group is coupled with 99 others. If you need any information for that group, it is much more efficient to look it up once (and store it) than to look it up/calculate it 99 separate times.

Good to know: WS.coup_used is a dictionary. The coupling ID's are the keys, and the values are the OscGroup numbers of all groups that this coupling is applied to at least once. It might not make sense to apply the map preparation to groups of the wrong kind.

5.4.3 calc_coupling

This is the function that actually returns the coupling between a given pair of oscillating groups. It takes six arguments: OscGroupi, OscGroupj, FILES, RunPar, WS, coupmap. The final four are the same as for most other functions. OscGroupi and j are the OscGroup indices of the two groups to couple. The return value of this function is the coupling in units of cm^{-1}.

Chapter 6

Theory and execution of the computations

Most sections of this manual focus on a variety of aspects of running/using the program. While the choice of parameters does give some insight into the core of the program, most of it remains a black box that spits out numbers. This section intents to shed some light on the inner workings of the program, and illustrate some equations. It is not, however, an in-depth description of all the functions. If you want to edit the code and know what everything does exactly, please have a look at the script itself, instead.

6.1 Context of the program

The AmideImaps program holds an important position in the chain of operations to generating part of the 2D-IR spectrum of a protein. The process starts with the protein itself. Using molecular dynamics (MD) software, a static image of a protein is brought to life: a trajectory is generated, consisting of many frames between which the protein changes ever-so-slightly. The result is like a movie - the trajectory contains the motion of the protein.

It is such a trajectory that is fed into the AmideImaps program. For each frame, for each amide bond in the protein, the expected vibrational frequency is calculated, along with its coupling to other amide groups, and its dipole moment.

These frequencies and couplings (together forming the Hamiltonian), along with the dipole moments, are fed to the NISE program (Numerical Integration of the Schrodinger Equation). This program considers how the Hamiltonian and dipole moments evolve over time, and uses that information to simulate the 2D-IR spectrum.

The part of the 2D-IR spectrum that is simulated is chosen carefully: it is the Amide-I stretch. This vibration is of particular interest as it is gives a strong signal, and is very sensitive to its direct environment.

6.2 How it is done

There are multiple ways to finding the vibrational frequency of an amide group. One option is to perform a DFT calculation. While expensive, the results are usually very accurate. However, analyzing an entire protein at once through DFT is not possible. It would therefore require many simulations of sub-sections, just for finding the frequencies of a single frame.

Another option would be to perform a very accurate MD simulation, saving a frame often enough that there are sufficient frames per oscillation, allowing to find the frequencies by taking the Fourier transform of the positions through time.[6,7] This method, however, requires large amounts of data (a frame is required at least every 2 fs, maybe even more often), making it cumbersome. Furthermore, the results obtained using MD only are not as accurate as those obtained through DFT.

The AmideImaps program attempts to take the best of both worlds. The basis lies in DFT simulations of model systems. These model systems are short oligopeptides, only a handful of amino acids long. The vibrational frequency of these peptides is calculated for different environments, which are created through placing point charges around them. A so-called map is made, which characterizes the environment through some simple parameters, and links these parameters to the calculated vibrational frequencies. A simple

example is the Tokmakoff map: It only looks at the electric field on the oxygen atom in the C=O bond direction, and links the calculated freqency to it. More complicated (and thus more expensive) maps probe the field at more positions, in more directions, or also take the gradient or electric potential into account.

It is because of these maps that AmideImaps can determine a vibrational frequency from a single, stationary frame: for each frame, for each amide group, it characterizes the electric environment, and uses the chosen map to link this to a vibrational frequency. This method allows for accuracies close to those of DFT, at a fraction of the cost. The need for only a single frame has another advantage: it allows to analyze the change in frequency on shorter time-scales. Some of these changes are oscillations with a period of roughly 5 Amide vibrations.

AmideImaps does not only use this mapping technique for finding frequencies, but also for finding the coupling between groups, and even for some dipole moment methods. The result is that there are quite a few maps, most of which were not generated for this program specifically, but were included because of their scientific value. Their respective sections will refer to publications of the original authors. This means that not all combinations of maps are equally valid, and the user should exercise caution as to what combinations are viable.

6.3 calculation steps

The calculation starts by characterizing the system. To do this, it answers the following questions: Is there a protein? Which atoms/residues are part of the protein? What is the structure of this protein? How many chains does it have? Are these chains cyclic (start and end residue are also bound, leaving no termini), or linear? How many residues do each of the chains contain? Are there amide groups in the side-chains? Which amide groups are neighbours? Are there other oscillators than amides? Which atoms are important for which oscillators?

All those questions will yield the same answer for every single frame. Then, for each frame, the vibrational frequencies, couplings, dipole moments and raman tensors are calculated. To do this, for each frame, the centres of mass of all residues are calculated, and for each oscillator, it is determined which residues are within the given `spheresize` distance. If a residue falls within the sphere around an oscillator, its influence on that group is calculated. This is done by considering each of the atoms of the residue. In total, the electric properties on a single atom of an oscillator are found by summing over all atoms j (J in total, this can easily be of the order of a 1000 atoms) that belong to a residue which is close enough to the oscillators:

$$P = \sum_j \frac{q_j}{|\mathbf{r}_j|}$$

$$E_x = \sum_j \frac{q_j}{|\mathbf{r}_j|^3}(\mathbf{r}_j \cdot \hat{\mathbf{x}})$$

$$G_{xx} = \sum_j \frac{q_j}{|\mathbf{r}_j|^3} + \left(\frac{3q_j}{|\mathbf{r}_j|^5}(\mathbf{r}_j \cdot \hat{\mathbf{x}})^2\right)$$

$$G_{xy} = \sum_j -\frac{3q_j}{|\mathbf{r}_j|^5}(\mathbf{r}_j \cdot \hat{\mathbf{x}})(\mathbf{r}_j \cdot \hat{\mathbf{y}})$$

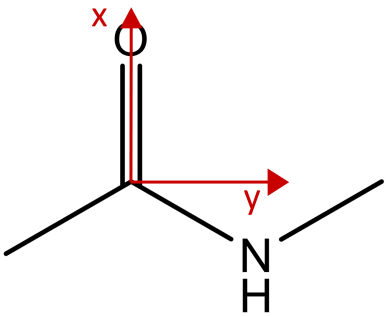

Figure 6.1: The definition of the local $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ vectors for the case of an amide bond. $\hat{\mathbf{z}}$ is perpendicular to both.

Here, P is the electric potential, E_x is the electric field in the x direction (E_y and E_z are computed similarly), and G is the electric field gradient. G_{yy} and G_{zz} are computed just like G_{xx}, G_{xz} and G_{yz} are computed just like G_{xy}. $\mathbf{r}_j = \mathbf{r}_{atom} - \mathbf{r}_j$ is the difference vector between the atom of the oscillator, and one of the many atoms surrounding it. It takes into account the periodic boundary conditions of the simulation box. $|\mathbf{r}_j|$ is the length of $\mathbf{r}_j$ and q_j is the charge of the atom j. $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$ and $\hat{\mathbf{z}}$ are unit vectors in the x, y and z directions. These are local directions, not global. In the case of amide groups, $\hat{\mathbf{x}}$ points from the carbon atom to the oxygen atom (both of the C=O bond), $\hat{\mathbf{y}}$ lies perpendicular to $\hat{\mathbf{x}}$, pointing to the N atom of the amide group. Finally, $\hat{\mathbf{z}} = \hat{\mathbf{x}} \times \hat{\mathbf{y}}$. For other types of oscillators, the definition of these vectors can differ.

In total, there are 10 parameters characterizing the electric environment of an atom. In general, up to 6 atoms are relevant, while for the amide bond, up to 4 atoms (called i) are relevant (the C and O of the C=O bond, N, and H (or D) bonded to N). In total, therefore, there can be up to 60 (40 for an amide bond) parameters for characterizing the electric environment of an oscillator, each of which requires summing over a large amount of atoms.

6.3.1 Frequency

In general, the frequency (a diagonal element of the Hamiltonian) of an oscillator is calculated using the properties of the electric field (explained previously), and a map. Generally, a map contains a gas-phase frequency, and multipliers for (some of) the electric field properties. For example, the frequency could be found by taking the gas-phase frequency, and adding to this $0.1 *$ the potential on the first atom of the group. This factor of 0.1, together with the gas-phase frequency, make up this hypothetical map.

Users can add their own maps for any type of group (see section 5). The version of AIM that came with this manual, only has maps for amide-I oscillations, so only those will be discussed here in more detail.

The frequency (a diagonal element of the Hamiltonian) of the amide group is calculated as follows:

$$\omega = \omega_{map} + \sum_i P_{i,map} P_i + (\mathbf{E}_{i,map} \cdot \mathbf{E}_i) + (\mathbf{G}_{i,map} \cdot \mathbf{G}_i)$$

P_i, $\mathbf{E}_i$ and $\mathbf{G}_i$ are the values calculated through the previous equations. ω_{map}, $P_{i,map}$, $\mathbf{E}_{i,map}$ and $\mathbf{G}_{i,map}$ are pre-defined values, encoded in the map. Depending on the map, one or more of these values might be zero.

- In the Tokmakoff map[8] (`map_choice Tokmakoff`), all values for the C, N and H atoms are 0, so only the environment of the oxygen atom is taken into account. To be even more precise - it only considers $\mathbf{E}_{xx}$.
- In the Skinner map[9] (`map_choice Skinner`), the environment of the C and N atoms is considered. Still, it only considers $\mathbf{E}_{xx}$. Amide groups with a N belonging to a proline residue are mapped using a proline-specific map, which is structured like the Jansen maps.
- In the Cho[10] (`map_choice Cho`) and Hirst4[11] (`map_choice Hirst`) maps, the environment of all four (C, O, N, and H) atoms is considered. For each atom, only the electric potential is used.
- In the Jansen map[12,13] (`map_choice Jansen`), the environment of all four (C, O, N and H) atoms is considered. For each atom, $\mathbf{E}_{xx}$, $\mathbf{E}_{yy}$, $\mathbf{G}_{zz}$ and $\mathbf{G}_{xy}$ are used.

In case nearest neighbours are considered[14] (`TreatNN True`), a shift δ is added to the frequency for each neighbour, if it is there:

$$\delta = (1-u)(1-t)map\left(30\left\lfloor\frac{\phi}{30}\right\rfloor,\ 30\left\lfloor\frac{\psi}{30}\right\rfloor\right) \ +\ (1-u)t\cdot map\left(30\left\lfloor\frac{\phi}{30}\right\rfloor,\ 30\left\lceil\frac{\psi}{30}\right\rceil\right)$$
$$+\ u(1-t)map\left(30\left\lceil\frac{\phi}{30}\right\rceil,\ 30\left\lfloor\frac{\psi}{30}\right\rfloor\right) \ +\ u\cdot t\cdot map\left(30\left\lceil\frac{\phi}{30}\right\rceil,\ 30\left\lceil\frac{\psi}{30}\right\rceil\right)$$

Here, $u = \frac{\phi \ mod\ 30}{30}$, $t = \frac{\psi \ mod\ 30}{30}$, $0 \le \phi \le 360$ and $0 \le \psi \le 360$. ϕ and ψ are the Ramachandran angles between the amide group and its neighbour. $\lfloor\frac{\phi}{30}\rfloor$ is the floor function of $\phi/30$, or, in other words, the largest multiple of 30 which is smaller than ϕ. Similarly, $\lceil\frac{\phi}{30}\rceil$ is the ceiling function of $\phi/30$, or the smallest multiple of 30 which is larger than ϕ. $map(30\lfloor\frac{\phi}{30}\rfloor,\ 30\lfloor\frac{\psi}{30}\rfloor)$ is the value in the map for this pair of angles. It is a single number.

In all, this equation is basically a bilinear interpolation between two pairs of values. These four values are retrieved from a map, and depend on the angles. The map has a value for any combination of ϕ and ψ where ϕ and ψ are multiples of 30. For example, if $\phi = 15$ degrees and $\psi = 67$ degrees, the four map values where $\phi = 0$ degrees or $\phi = 30$ degrees and $\psi = 60$ degrees or $\psi = 90$ degrees are used.

There are multiple maps available for computing this delta. Which map is used, depends on the nature of the two amide groups considered: they can either be a proline residue, or another. If there is a proline present, it can be in either D or L configuration, and the non-proline residue can be bound in either cis or trans configuration. A list of all available maps is given in section 4.10.

6.3.2 Coupling

In general, the coupling (an off-diagonal element of the Hamiltonian) between two oscillators can be calculated in very different ways. Some methods involve approximating each oscillator with a dipole, and then use the dipole-dipole coupling as an approximation. Others might take into account the interactions between all pairs of atoms of both oscillators (closer, but not equal, to quadrupole approximation), or some other property relevant for that kind of oscillator.

Users can add their own coupling maps. This is mainly intended to define the coupling to/from a custom group they also added a frequency map of (see section 5), as mentioned in section 6.3.1. The version of AIM that came with this manual has one generic coupling map, and several for the coupling between two amide groups.

The generic coupling map can be applied when there is no specific map defined. Whether the map will be applied is governed by the `apply_dd_coupling` parameter. This map calculates the interaction between any two dipole moments (labelled i and j) to be the coupling J:

$$J = 5034\left(\frac{\hat{\mathbf{m}}_i \cdot \hat{\mathbf{m}}_j}{|\mathbf{r}_{ij}|^3} - \frac{3(\hat{\mathbf{m}}_i \cdot \mathbf{r}_{ij})(\hat{\mathbf{m}}_j \cdot \mathbf{r}_{ij})}{|\mathbf{r}_{ij}|^5}\right)$$

Here $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$. $\mathbf{m}_i$ is the dipole moment of oscillator i (in debye), $\mathbf{r}_i$ is the position of this dipole moment. The dipole moment and its position must be defined of any group of which a frequency map is added; the calculated dipole is used both for the dipole output file, and for this coupling method.

The program has 5 different methods built-in to calculate the coupling between amide groups. Three of these (TDCKrimm, TDCTasumi and TCC) are usable for any combination of groups (including side chain groups). The other two (Tasumi and GLDP) are only applicable when the two groups are neighbours.

Krimm's transition dipole coupling[15] (`coupling_choice TDCKrimm` and `NN_coupling_choice TDCKrimm`) assumes every group to have a dipole moment placed at 20 degrees from the C=O bond, in the O=C-N plane. Then, the interaction between any two dipole moments (labelled i and j) is calculated to be the coupling J:

$$J = 580\left(\frac{\hat{\mathbf{m}}_i \cdot \hat{\mathbf{m}}_j}{|\mathbf{r}_{ij}|^3} - \frac{3(\hat{\mathbf{m}}_i \cdot \mathbf{r}_{ij})(\hat{\mathbf{m}}_j \cdot \mathbf{r}_{ij})}{|\mathbf{r}_{ij}|^5}\right)$$

with $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$, where $\mathbf{r} = \frac{0.868}{|\mathbf{CO}|}\mathbf{CO} + \mathbf{C}$. Here, $\mathbf{CO}$ is the vector pointing from the C to the O atom, and $\mathbf{C}$ is the position of the C atom. $\mathbf{m} = \hat{\mathbf{y}} \tan 20 - \hat{\mathbf{CO}}$, where $\mathbf{y} = \mathbf{CN} - \frac{\mathbf{CO}\cdot\mathbf{CN}}{\mathbf{CO}\cdot\mathbf{CO}}\mathbf{CO}$, in which $\mathbf{CN}$ is the vector pointing from the C to the N atom. In all these equations, for any vector, $\hat{\mathbf{v}} = \frac{\mathbf{v}}{|\mathbf{v}|}$.

Tasumi's transition dipole coupling[16] (`coupling_choice TDCTasumi` and `NN_coupling_choice TDCTasumi`) is somewhat similar to Krimm's: J depends on the interaction between the dipoles of the amide groups i and j. However, mind the subtle differences:

$$J = 51.43\left(\frac{\hat{\mathbf{m}}_i \cdot \hat{\mathbf{m}}_j}{|\mathbf{r}_{ij}|^3} - \frac{3(\hat{\mathbf{m}}_i \cdot \mathbf{r}_{ij})(\hat{\mathbf{m}}_j \cdot \mathbf{r}_{ij})}{|\mathbf{r}_{ij}|^5}\right)$$

This time, $\mathbf{m}$ is the Torii dipole μ as calculated in section 6.3.3. $\mathbf{r} = \mathbf{s} + \mathbf{C}$, with $\mathbf{s} = 0.665\mathbf{CO} + 0.258\mathbf{CN}$.

Please note, that in AIM the prefactor for the Torii dipoles has been changed to ensure the same units being used everywhere in the program. To compensate for the change of units at the Torii dipoles, the prefactor in AIM is 5034 instead of 51.43.

The TCC method[14] (`coupling_choice TCC` and `NN_coupling_choice TCC`) does not use the dipoles of the two amide groups. Instead, it looks at the interaction between all the individual atoms making up each amide group (The C, O, N and H atoms, but also the Cα atoms of both residues):

$$J = \frac{1}{4\pi\epsilon}\sum_{a,b}\left(\frac{dq_a dq_b}{|\mathbf{r}_{ab}|} - \frac{3q_a q_b(\mathbf{v}_a \cdot \mathbf{r}_{ab})(\mathbf{v}_b \cdot \mathbf{r}_{ab})}{|\mathbf{r}_{ab}|^5} - \frac{dq_a q_b(\mathbf{v}_b \cdot \mathbf{r}_{ab}) - q_a dq_b(\mathbf{v}_a \cdot \mathbf{r}_{ab}) - q_a q_b(\mathbf{v}_a \cdot \mathbf{v}_b)}{|\mathbf{r}_{ab}|^3}\right)$$

Here, a and b number the atoms belonging to the two amide groups. Each atom has a charge q, a transition charge dq and a normal mode coordinate $\mathbf{v}$. The values of these parameters (and the value for $\frac{1}{4\pi\epsilon}$) are encoded in the TCC parameter file. $\mathbf{r}_{ab} = \mathbf{r}_a - \mathbf{r}_b$.

The Tasumi method[16] (`NN_coupling_choice Tasumi`) and GLDP method[14]] (`NN_coupling_choice GLDP`) are only applicable to amide groups that are neighbours. Both are very similar in method to calculating the factor δ that goes into the frequency (section ...):

$$J = (1-u)(1-t)map\left(\left\lfloor\frac{\phi}{30}\right\rfloor\left\lfloor\frac{\psi}{30}\right\rfloor\right) + (1-u)t\cdot map\left(\left\lfloor\frac{\phi}{30}\right\rfloor\left\lceil\frac{\psi}{30}\right\rceil\right)$$
$$+ u(1-t)map\left(\left\lceil\frac{\phi}{30}\right\rceil\left\lfloor\frac{\psi}{30}\right\rfloor\right) + u\cdot t\cdot map\left(\left\lceil\frac{\phi}{30}\right\rceil\left\lceil\frac{\psi}{30}\right\rceil\right)$$

Just as for calculating δ, $u = \frac{\phi \ mod\ 30}{30}$, $t = \frac{\psi \ mod\ 30}{30}$, $0 \leq \phi \leq 360$ and $0 \leq \psi \leq 360$. ϕ and ψ are the Ramachandran angles between the amide groups. For every combination of angles, there is a value in the map. The Tasumi map is a single map (see section 6.3.1), while the GLDP map (just like the nearest-neighbour effect on frequency) consists of seven different maps, each of which is designed for different situations.

6.3.3 Dipoles

The dipole moment of an oscillator can be calculated in many ways. A map can be used (relating the dipole moment to the properties of the local electric field), but it can, for example, also be based on the positions of atoms. In general, the dipole moment calculated by AIM is in units of Debye.

Users can add their own methods for any type of group (see section 5). The version of AIM that came with this manual only has maps/methods for the dipole moment of amide-I oscillations, so only those will be discussed here in more detail.

The two methods of calculating dipoles actually differ considerably. The calculation of Jansen dipoles[12] (`Dipole_choice Jansen`) is very similar to the frequency calculation:

$$\mu_x = \mu_{map_x} + \sum_i P_{i,map_x} P_i + (\mathbf{E}_{i,map_x}\cdot\mathbf{E}_i) + (\mathbf{G}_{i,map_x}\cdot\mathbf{G}_i)$$

The y and z components of μ are computed analogously.

Calculation of Torii dipoles[16] (`Dipole_choice Torii`), on the other hand, is based on the relative position of the C, O and N atoms:

$$\mu = 2.73\left(\mathbf{s} - \left(\left(\mathbf{CO}\cdot\mathbf{s}\right) + \frac{\sqrt{|\mathbf{s}|^2 - (\mathbf{CO}\cdot\mathbf{s})^2}}{\tan 10}\right)\mathbf{CO}\right)$$

Here, $\mathbf{s} = 0.665\mathbf{CO} + 0.258\mathbf{CN}$, $\mathbf{CO}$ is the vector pointing from C to O, and $\mathbf{CN}$ is the vector pointing from C to N.

Please note that the formula is implemented in AIM slightly differently. To make sure that all different methods return dipoles in the same units (Debye), the prefactor of 2.73 has been changed to 0.276.

6.3.4 Raman tensor

The Raman tensor of each amide group is calculated as follows[17]:

$$\alpha = 20\cdot\hat{\mathbf{x}}_r\otimes\hat{\mathbf{x}}_r + 4\cdot\hat{\mathbf{y}}_r\otimes\hat{\mathbf{y}}_r + \hat{\mathbf{z}}_m\otimes\hat{\mathbf{z}}_m$$

With

$$\hat{\mathbf{x}}_r = \cos(34^\circ)\cdot\hat{\mathbf{x}}_m - \sin(34^\circ)\cdot\hat{\mathbf{y}}_m$$

$$\hat{\mathbf{y}}_r = \sin(34^\circ)\cdot\hat{\mathbf{x}}_m + \cos(34^\circ)\cdot\hat{\mathbf{y}}_m$$

Here, for any vector, $\hat{\mathbf{v}} = \frac{\mathbf{v}}{|\mathbf{v}|}$. $\hat{\mathbf{x}}_m$, $\hat{\mathbf{y}}_m$ and $\hat{\mathbf{z}}_m$ are the same x, y, and z directions as defiined at the beginning of section 6.3.

Chapter 7

FAQ/troubleshooting [WIP]

7.1 FAQ

General questions

How can I contact the developers?

You can get in touch with the developers through the github page (https://github.com/Kimvana/AIM-public). If you found a bug in the code you wish to report, click the 'issues' button at the top. Check if someone already reported the same issue. If not, you can notify us by clicking 'open new issue'. Follow the steps there, and we will be notified about the issue. Make sure to give the version number of your installation of AIM (printed every time it runs), a copy of the error message, and the created log file.

If you have a question, remark, request for future updates/additions, etc., please don't use the 'issues' section, but use the 'discussions' section instead. Again, you can see if someone had the same question/remark/etc. if not, click 'new discussion', and ask away!

Please note that both of these are publicly visible. The major advantage of this is that, if another user had the same problem/question/remark, you can immediately see the answer/response there, without having to wait for us!

I've used AIM for my own research. How do I cite this work?

When using AIM for your research, you are using both the AIM program, and the maps applied by it. Please cite the work corresponding to development of the AIM software, as well as all papers corresponding to the maps you've used.

AIM helps with this by keeping track of the methods used, and which references correspond to that. Make sure to have the `Verbose` or the `Verbose_log` parameters set to 1 or above, then AIM will print the references used for that calculation, along with what they did for you!

The reference corresponding to the development of the AIM program is J. Chem. Theory Comput. 2022, 18, 5, 3089–3098[18]

Questions regarding using/changing AIM

What do the different verbose settings do?

It might be useful to know in advance what the different verbose settings print to the command line/log file. Here, per value, follows an overview:

verbose 0
This is the most basic level. It only prints the following things:

- warning function messages → only get triggered when something isn't right.
- message for early stop of frame loop (bc of maxtime)
- message for finished calculation, data dump → nchains

verbose 1
prints everything that level 0 prints, plus:

- Messages over the usage of the c-library, but only if the user did not specify whether to use it. If they did, it is treated by the warning function.
- Program header + version number
- COfinder → amount of detected chains, their nature, and amount of residues present in each.
- end-of-calc datadump: total AmGroups considered, frames considered+frames total, calculation time.

verbose 2
prints everything that level 1 prints, plus:

- if profiler running → warning for slowdown.
- starting iterating over frames
- start on framenum (some frames - interesting to users who want to know when the calculation will finish!)
- end-of-calc datadump: per chain number and per chain count of considered AmGroups + total AmGroups present in system

verbose 3
prints everything that level 2 prints, plus:

- choice of electrostatic map
- choice of dipole method
- perpendicular box vectors
- end-of-calc datadump: per chain present AmGroups

verbose 4
prints everything that level 3 prints, plus:

- for the following functions, message they completed successfully: COfinder, ChooseMapReader, per-frame initialization, per-frame calculations
- start on framenum (every frame)

How do I code a new parameter into the program?

You might have the desire to add a new parameter (like `outfilename` or `SphereSize`) to the program. Of course, you must use the parameter somewhere in the code for it to serve any purpose, but that's not all: the program must be able to read the users choice for this parameter from an input file. The first part, using it, will not be covered here, as anyone who attempts this is expected to have adequate programming skills. However, the second part deserves some attention. The new parameter might have nothing to do with file initialization, so this will save quite some figuring out!

Most of adding the parameter boils down to adding it into some hard-coded lists. Due to the ever-changing nature of the code, line numbers will not be given here. Instead, look for the following functions/files:

default_input.txt
In order for the program to interpret your new parameter, it must be added to this file. Use exactly the same name here, and for the variable in the code. This file can be found in AIM/sourcefiles/. For more info on this file, see 4.2

get_settings_names()
This function can be found as a standalone function in AIM/sourcecode/FileHandler.py. It's main job is to store all the different parameter names the program can encounter in an input file, in a dictionary. After creating the empty dictionary, multiple key-value pairs are created for it. There is a key for every datatype that a parameter setting could have. Any parameter using that datatype goes into the list that

makes up the corresponding value. For example, the parameter `topfile` expects a string, `SphereSize` expects a float, `Verbose` an integer, etc. Depending on what datatype your parameter should accept, add it to the correct list.

get_settings_allowed_values()
This function can be found as a standalone function in AIM/sourcecode/FileHandler.py. It's main job is to store all the allowed choices for a given parameter. For example, the `Verbose` parameter can only take the integers 0 through 4. Adding the new parameter to this section does not always make sense. Booleans, for example, can only take 'true', or false. For parameters like `SphereSize`, there is no use defining which values are/aren't allowed. Same for filenames (their validity is checked elsewhere). Only add your parameter if necessary.

WriteParToLog()
This function can be found as a function belonging to the FileLocations class described in AIM/sourcecode/FileHandler.py. If you want the log file to contain the users choice for this parameter, it should be added here. Follow the examples already present: you're typing exactly what gets printed to the log file.

WriteParToOutPar()
This function can be found as a function belonging to the FileLocations class described in AIM/sourcecode/FileHandler.py. The parameter should be added here, as this function prepares the output parameter file - the one you can use as an input file for a later run. If your new parameter belongs nicely in a existing category, you can add it to the corresponding list. Otherwise, a new block of code is required. Verify that the function prints the parameter the same way as you added it to the input file.

ParameterCheck()
This function can be found as a function belonging to the ParameterFinder class described in AIM/sourcecode/FileHandler.py. Again, not all parameters are stored here. In general, the parameters here do not store a file/folder name/location, and are not special in any other way. While the program can run without any of the parameters here being specified, there are a few that it probably shouldn't. Therefore, for the items added to the second group (also read the comments in the code), the program will print a warning to the user when they're missing.

InitRunSet()
This function can be found as a function belonging to the RunParameter class described in AIM/sourcecode/FileHandler.py. Any parameter you expect to be available as part of the RunPar class later in the code should be added here.

FileFinder()
This function can be found as a function belonging to the FileLocations class described in AIM/sourcecode/FileHandler.py. It creates the requested output files, and tests for the specified data files (like those storing maps). If your new parameter stores a file location, you can consider adding it here, if that suits your needs.

Others?
As the purpose of a new parameter could be very different from anything I can imagine, it is impossible to create a definite list. Therefore, do not just assume this list to be complete - it's not even complete for some of the parameters I included. However, the list should be reasonably complete for parameters with a more 'simple' purpose. In any situation, it is wise to find a parameter which is already included and should be treated most like the one you're adding. Search the code files (mainly the FileHandler.py script, as it's job is to deal with files, including the input file you're trying to read) for where that parameter is being used, and see if yours shoud be added there too. But I probably don't have to tell you this, as adding a parameter means you already know how to code!

7.2 Troubleshooting

During the initial testing, some errors popped up more than others. Below you can find our most common issues, and their solutions. If yours is not in the list (or cannot be solved using the explained method), please do not hesitate to contact us! When contacting us, please describe in detail what you did to trigger the issue, and also give the following: a copy of the error message in the terminal (not all are printed to the log file), the log file generated, and your input parameter file.

The program crashes, but it did not generate the log file to see the problem

The problem is that the program encountered a problem before finding out what name it should use for the log file. In these cases, a special 'crash' log file is created in the default 'log' folder where the program is installed.

Replicate_orig _AIM gives different frequencies when using the Skinner map

The Skinner map is different from the others in one aspect: when deciding whether an atom is close enough to consider its influence, Skinner uses charge groups instead of residues. This version of the program does not support charge groups.

Some residue name(s) where not recognised, so the program won't run. How do I add them?

Before adding these residues, ask yourself the following. What kind of residue is it? There are two options: it is either part of the protein chain itself, or it is part of it's environment (solvent/membrane/cofactor). If it's part of the environment, it can simply be added to the resnames file (see section 4.6). If it is part of the protein, you must be more careful:

Being part of the protein, there are two options. It could be something very special, and different from any 'normal' amino acid. In this case, the code has to be adapted. This either requires some python skill, or asking us nicely. In most cases, however, it is much easier. If the special amino acid has a special name, but still contains the basics (A C=O bond, N-H bond and alpha carbon in the backbone, and a beta-carbon bound to the alpha-carbon), it can be added to the 'Protein_ext' line of the resnames file (see section 4.6).

I'm getting errors that some index is out of bounds for some axis of length x

This error could be caused by many things, most of which shouldn't occur. Try this before contacting us: select the correct atnames file using the atnamesfile keyword. The default settings upon download are for Gromacs, and your files might be generated with different MD software!

These atnames files are stored within the sourcefiles folder. Be sure to select the one named after the used MD package (NAMD trajectories can be analyzed using the CHARMM one). If no such file exists, you can either contact us, or try to make your own. If you do make your own, please realize that there are more differences between packages than just the names, so make sure that no other (sometimes invisible) problems arise.

Error while identifying protein chains - there were no, or more than 1, C-type atoms found.

In any case, it is worth checking that the correct atnames file is used. If no atnamesfile was supplied with AIM for your specific MD package, either contact us, or make your own. Be sure that the name of the backbone C atom that doubly binds to the oxygen matches the name behind 'C' in that file!

If the atnames file is correct, and you have a special residue that terminates a chain, this has to be programmed in. The residue 'FOR' is an example of such a residue (searching for it should simplify implementing this residue a lot!).

If it is no special residue and the atnames file is correct, please contact us!

Error while identifying protein chains - there were more than 1 N-term neighbours

This error occurs when the atom which AIM identified as the backbone C=O carbon is bounded to more than one nitrogen atom. This is unexpected behaviour. If this is truly the case for your system, the code must be adapted to fit this kind of system. If not, please contact us!

7.3 Acknowledgements

This publication is part of the project "Nanoscale Regulators of Photosynthesis" (with project number OCENW.GROOT.2019.086) of the research programme NWO GROOT which is partly financed by the Dutch Research Coucil (NWO).

We are grateful to Markus Meuwly and Maryam Salehi for their support with testing the code for CHARMM trajectories and to Christopher M. Cheatum and Lauren Lambach for their help with testing the code for AMBER trajectories.

We are grateful to Tsjerk Wassenaar, Siewert-Jan Marrink and Ana Cunha for the helpful discussions.

We thank the Center for Information Technology of the University of Groningen for their support and for providing access to the Peregrine high performance computing cluster.

7.4 Main contributors

Main developer (2020 - present) Kim E. van Adrichem
Creation, organization and maintenance of core code. Ideas for and implementation of optimizations (like the c library) and generalizations (like allowing non-amide extramaps). Implementation of amide-I oscillators. Creation and maintenance of manual.

Other developers
Raman tensors (code and manual) - Carleen D. N. van Hengel
Main ideas (and many smaller ideas), testing of code, review - Thomas L. C. Jansen

7.5 References

[1] T. l. C. Jansen and J. Knoester. "Nonadiabatic Effects in the Two-Dimensional Infrared Spectra of Peptides: Application to Alanine Dipeptide". In: *J. Phys. Chem. B* 110.45 (Nov. 2006), pp. 22910–22916.

[2] N. Michaud-Agrawal, E. J. Denning, T. B. Woolf, and O. Beckstein. "MDAnalysis: A Toolkit for the Analysis of Molecular Dynamics Simulations". In: *J. Comput. Chem.* 32.10 (July 2011), pp. 2319–2327.

[3] R. Gowers, M. Linke, J. Barnoud, T. Reddy, M. Melo, et al. "MDAnalysis: A Python Package for the Rapid Analysis of Molecular Dynamics Simulations". In: *Python in Science Conference*. Austin, Texas, 2016, pp. 98–105.

[4] S. K. Lam, A. Pitrou, and S. Seibert. "Numba: A LLVM-based Python JIT Compiler". In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM '15*. Austin, Texas: ACM Press, 2015, pp. 1–6. ISBN: 978-1-4503-4005-2.

[5] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, et al. "Array Programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362.

[6] T. Hasegawa and Y. Tanimura. "A Polarizable Water Model for Intramolecular and Intermolecular Vibrational Spectroscopies". In: *J. Phys. Chem. B* 115.18 (May 2011), pp. 5545–5553.

[7] E. Mosconi, C. Quarti, T. Ivanovska, G. Ruani, and F. De Angelis. "Structural and Electronic Properties of Organo-Halide Lead Perovskites: A Combined IR-spectroscopy and Ab Initio Molecular Dynamics Investigation". In: *Phys. Chem. Chem. Phys.* 16.30 (2014), pp. 16137–16144.

[8] M. Reppert and A. Tokmakoff. "Electrostatic Frequency Shifts in Amide I Vibrational Spectra: Direct Parameterization against Experiment". In: *Chem. Phys.* 138.13 (Apr. 2013), p. 134116.

[9] L. Wang, C. T. Middleton, M. T. Zanni, and J. L. Skinner. "Development and Validation of Transferable Amide I Vibrational Frequency Maps for Peptides". In: *J. Phys. Chem. B* 115.13 (Apr. 2011), pp. 3713–3724.

[10] S. Ham, J.-H. Kim, H. Lee, and M. Cho. "Correlation between Electronic and Molecular Structure Distortions and Vibrational Properties. II. Amide I Modes of NMA–nD2O Complexes". In: *Chem. Phys.* 118.8 (Feb. 2003), pp. 3491–3498.

[11] T. M. Watson and J. D. Hirst. "Influence of Electrostatic Environment on the Vibrational Frequencies of Proteins". In: *J. Phys. Chem. A* 107.35 (Sept. 2003), pp. 6843–6849.

[12] T. l. C. Jansen and J. Knoester. "A Transferable Electrostatic Map for Solvation Effects on Amide I Vibrations and Its Application to Linear and Two-Dimensional Spectroscopy". In: *Chem. Phys.* 124.4 (Jan. 2006), p. 044502.

[13] S. Roy, J. Lessing, G. Meisl, Z. Ganim, A. Tokmakoff, et al. "Solvent and Conformation Dependence of Amide I Vibrations in Peptides and Proteins Containing Proline". In: *Chem. Phys.* 135.23 (Dec. 2011), p. 234507.

[14] T. l. C. Jansen, A. G. Dijkstra, T. M. Watson, J. D. Hirst, and J. Knoester. "Modeling the Amide I Bands of Small Peptides". In: *Chem. Phys.* 125.4 (July 2006), p. 044312.

[15] H. Torii and M. Tasumi. "Model Calculations on the amide-I Infrared Bands of Globular Proteins". In: *Chem. Phys.* 96.5 (Mar. 1992), pp. 3379–3387.

[16] H. Torii and M. Tasumi. "Ab Initio Molecular Orbital Study of the Amide I Vibrational Interactions between the Peptide Groups in Di- and Tripeptides and Considerations on the Conformation of the Extended Helix". In: *J. Raman. Spectrosc.* 29 (1998), p. 6.

[17] M. Tsuboi and G. J. Thomas Jr. "Raman Scattering Tensors in Biological Molecules and Their Assemblies". In: *Appl. Spectrosc. Rev.* 32.3 (Aug. 1997), pp. 263–299.

[18] K. E. van Adrichem and T. L. C. Jansen. "AIM: A Mapping Program for Infrared Spectroscopy of Proteins". In: *J. Chem. Theory Comput.* 18.5 (May 2022), pp. 3089–3098.