Technical tips for INFOMDV

General notes

Report structure

There are several aspects to consider here. They all stem from the fact that your final report should be resemble (in structure, style, and level-of-detail) to any white paper or scientific article. Your report should be self-contained and structured so that any interested reader (having the required technical background) can understand and replicate your work just by reading the report. Several aspects flow from this:

Structure: Do not structure the report as a per-week progress. Rather, structure it so that every section covers a step of the assignment. List explicitly the *aims/goals* of each step at the start of the respective section. This way, the reader knows exactly what that section will cover. Split large sections (roughly: taking more than 2 pages) in subsections, to ease reading.

Sections and subsections: Apart the above per-step sections, you should add a general introduction, describing the overall aims and scope of the document and a short overview of what comes in the next sections, as in any technical report), a section with notation used throughout the document and, a conclusion section. For every step, you can, if you find it appropriate, add some other subsections such as Algorithm, Implementation Choices, Experiment, etc.

Number: figures, tables, equations, sections, and pages. This makes it easy for you (and also the reader) to refer to any such element by its number. Any figure or table should have a simple and self-contained explanation in its caption (text below the figure or table) that tells what the respective element shows. This allows one to quickly skim through the report and understand what all such visuals are about, before diving into the detailed explanations in the main text.

References: Add references (listed in a bibliography at the end, not in footnotes) to all elements which are not your own creation. These include algorithms, methods, software toolkits, libraries, and frameworks. For software artefacts, specify explicitly the version number and URL you got them from. This helps the reader in precisely seeing what you have used in your work and thus makes your work replicable. Preferred references are published papers or technical reports. However, URLs that point to web pages are also allowed as references.

Source code/pseudocode: You do *not* need to list the source code in the report, since you will provide this code separately. Listing extensive code fragments makes the report unnecessarily long without bringing in additional clarity. Additionally, if you use a package to perform an operation, it is highly preferred that you describe that operation by using *math* notation (*e.g.*, computing an

average or performing an iterative update of a point's position) instead of listing the API call. The reader will then immediately see what you are doing in the math without needing to know the respective API. The only case where using API calls is preferred is when you (a) refer to operations which are too complex to express by simple math; and (b) you use specific parameter settings of the respective API.

If you find yourself describing what your code does, consider writing a high-level pseudocode instead. For writing pseudocode we recommend the package algorithm2e. However, feel free to write pseudocode (if needed) in any suitable notation you find easy to use.

Finally, you do not need to provide complete source code with each progress iteration. Source code will change a lot over the iterations. Having the lecturer assess it continuously is neither practical nor useful. The source code will only be assessed at the end when you provide its final version.

Language: Do not describe your report in a chronological manner, for instance "We did A and then we did B". Instead, describe the *logical* flow of your work – as driven by both the assignment steps and the implementation decisions you took. For intermediate report versions: If parts are not ready, simply mark them *e.g.* with a tag "work in progress".

Always use present tense in your report. Do not make paragraphs in the text overly long. Each paragraph should have a *single* message; all messages (taken from all paragraphs) together need to make a logical story. If you find yourself writing a paragraph containing 3 messages, consider splitting it in three paragraphs. However, paragraphs of one sentence also do not look good. Avoid using passive voice as well – say "we do A" and not "A is done".

Text formatting: Whenever you define a term for the first time, make it italic, using the `\emph` command. Use math mode for all mathematical notations (variables, metrics, formulas, equations). Use `\texttt` to mark all elements which suggest code fragments, *e.g.*, `MyAlgorithm`. To avoid retyping same lengthy command multiple times consider defining a command, for instance in the beginning of the document type `\newcommand{\R}{\mathbb{R}}` and then use `\R` throughout your document. In case you want to write text in math environment, do that using `\texrm` command, to obtain $y = \text{myFunction}(x)$ instead of $y = myFunction(x)$.

Level of reporting

It is crucial that you understand what is the right *level of detail* for reporting. A good report tells (1) *what* was done, and *how*; and (2) shows *proof* that the results are correct and complete. That means:

(1) What was done: For every step, explain in detail what that step aims to accomplish. To do this, you have to introduce notations in the report early on. Easiest: Introduce as many notations right in or before step 1 (*e.g.*, graph, nodes, edges, graph characteristics, quality metrics, etc). Consider creating an introductory 'notations' section which lists as many of these as possible. This is quite similar to listing *declarations*, when coding, before actually writing the *functions*. This will also force you to think well about all needed terms and how to define them. If you do this properly, writing the rest of the report will be *much* easier!

Once introduced, any notation should be used consistently throughout the whole report. Same as in coding: Once you define a type `A`, you call it `A` all over the place; and you don't call something else `A`.

(2) Proof of completeness and correctness: For every computational step of your pipeline, you have to give proof (evidence) that that step was designed and implemented correctly. How you do this, depends on the actual step. For example:

- Computing some simple quantities (*e.g.*, the length of a sampled curve): Just provide the math formula describing what you did; this is enough;
- Computing more complex results of an algorithm which you cannot reduce to a simple formula. To support your claims, consider (a mix of) the following:
  - **Visual evidence:** Show a few results generated by your algorithm for different inputs. Think it this way: You have an algorithm $A : B \to C$, that is, which reads some data of type B and generates results of type C. How to prove it works right? Show as many pairs $(\mathbf{x} \in B, A(B) \in C)$ as practically possible! This *statistically* shows that your A does the right job. If A depends on parameters p (besides its input A), then show additional images of how A works for different p values.
  - **Numerical evidence:** For some visualization algorithms, you can gauge their quality by a few quality metrics, including number of crossings, stress, crossing angles, etc. Simply put: You evaluate the quality of a force directed layout F algorithm, by applying F on few graphs $G_i, i = 1 \ldots, n$, thus constructing the layouts $F(G_i), i = 1 \ldots, n$ and by measuring metric M of these layouts, thus computing the values $M(F(G_i)), i = 1 \ldots, n$. If these values of the metric look good (for instance are smaller after applying the algorithm), then likely the layouts are good and therefore the algorithm F is good as well. For instance, for a force-directed algorithm, you can compute the stress and the number of crossings of the layout of a graph. In case of small scale experiments, e.g. n is very small, just present the numbers for each graph individually. In case of large values of n, present the values $M(F(G_i)), i = 1 \ldots, n$ in perspective: describe how many graphs (number n) you used to compute them; how you selected these graphs; and how the metrics values spread, *e.g.*, using statistics such as averages, standard deviations, histograms.

Generic tools

As you execute the assignment, you will find that a number of tools will be useful in many different steps. Hence, it is good to think upfront about implementing them in a generic way. Once you do that, you can then reuse them very easily in subsequent steps. Examples include: graph data structures, graph data structure traversal-algorithms, computation of quality metrics, visualizers for a graph, etc.

As always in software engineering:

- Don't jump in randomly coding stuff; think about what you *will* need to code;
- Think 1..2 steps ahead when coding something; find general patterns; then, design a single code for those steps;
- Do not invest time in designing complex *user interfaces*. We do not grade, nor aim to teach how to design, interactive systems.