

From: http:// ... TKP4106
(Automatic HTML etc. to PDF Conversion)

Creator: Tore Haug-Warberg
Department of Chemical Engineering
NTNU (Norway)

Created: Thu Jan 22 15:05:03 +0100 2015

PDF name: 2015_01_22_15_05_03.pdf

Contents

1	Homepage	4
2	Tore Haug-Warberg (Programming)	6
2.1	Real Programmers use FORTRAN	10
3	Heinz A. Preisig (Modelling)	18
4	Frequently Asked Questions (FAQ)	20
5	Syllabus	23
5.1	Introduction to Python	24
5.1.1	Seven Topics in Python	28
5.1.2	Verbatim: “script”	56
5.1.3	Emacs quick reference	57
5.1.4	Vim quick reference	59
5.1.5	TextPad quick reference	61
5.1.6	LaTeX (Cambridge University)	63
5.1.7	High-quality portable PDF (Schatz)	69
5.1.8	Regex (Stephen Ramsay)	71
5.1.9	Regex quick reference	74
5.1.10	BNF and EBNF (L. M. Garshol)	75

5.1.11	Windows shortcuts (OIT)	86
5.1.12	Linux programming (digilife)	87
5.1.13	Mac shortcuts (macmost)	89
5.1.14	Commenting Python code (MIT)	90
5.1.15	Programming paradigms (Kurt Normark)	92
5.1.16	Real Programmers (Ed Post), see also Sec. 2.1	98
5.1.17	Seven Topics in Python, see also Sec. 5.1.1	99
5.1.18	Seven Topics in Python (Haug-Warberg), see also Sec. 5.1.1	100
5.1.19	Epydoc (sourceforge)	101
5.1.20	Epytext markup (sourceforge)	102
5.1.21	Python Docstrings (Sourceforge)	112
5.1.22	Scientific Python (numpy.org)	114
5.2	Exercise 1	116
5.3	Regular expressions	123
5.3.1	A Smalltalk about Modelling	127
5.3.2	Regular Expressions, see also Sec. 5.1.8	132
5.4	Exercise 2	133
5.5	Documenting your code	139
5.5.1	The real programmer, see also Sec. 2.1	143
5.5.2	epydoc, see also Sec. 5.1.19	144
5.5.3	Verbatim: “atoms.py”	145
5.5.4	epytext, see also Sec. 5.1.20	147
5.5.5	docstring, see also Sec. 5.1.21	148
5.5.6	Epydoc output file	149
5.6	Exercise 3	150
5.7	Molecular formula parser	163
5.7.1	Verbatim: “atoms.py”	166
5.7.2	Backus-Naur Formalism, see also Sec. 5.1.10	168
5.8	Exercise 4	169
5.9	The atom matrix	181
5.9.1	Verbatim: “atom_matrix.py”	185
5.9.2	Verbatim: “molecular_weight.py”	186
5.10	Exercise 5	188
5.11	Independent reactions	201
5.11.1	Verbatim: “rref.py”	205
5.11.2	Verbatim: “null.py”	207
5.11.3	The mass balance	208
5.12	Exercise 6	214
5.13	Root solvers	240
5.13.1	Verbatim: “sqrt.py”	242
5.13.2	Verbatim: “pv.py”	243
5.13.3	The energy balance	245
5.14	Exercise 7	254
5.15	A thermodynamic equation solver	260
5.15.1	Verbatim: “solve.py”	261
5.15.2	Verbatim: “hpn.py”	262
5.15.3	Verbatim: “mprod.py”	265
5.15.4	The energy balance	266
5.16	Exercise 8	275
5.17	The reactor model	289
5.17.1	Verbatim: “srk_ammonia.py”	291
5.17.2	Verbatim: “flowsheet.py”	294

5.17.3 Verbatim: “ammonia_reactor.py”	300
5.17.4 Verbatim: “tkp4106.py”	303
5.17.5 ammonia_reactor.py, see also Sec. 5.19.2	304
5.17.6 srk_ammonia.py, see also Sec. 5.17.1	305
5.17.7 Modelling issues	306
5.18 Exercise 9	319
5.19 Integration	327
5.19.1 Verbatim: “flowsheet.py”	329
5.19.2 Verbatim: “ammonia_reactor.py”	335
5.19.3 flowsheet.py, see also Sec. 5.19.1	338
5.19.4 ammonia_reactor.py, see also Sec. 5.19.2	339
5.19.5 Modelling issues	340
5.20 Exercise 10	353
5.21 Unit testing	379
5.22 Exercise 11	380
5.23 Putting the model to work	397
5.23.1 Verbatim: “graph.gp”	399
5.23.2 Verbatim: “graph.dat”	400
5.23.3 graph.pdf	401
5.23.4 ammonia_reactor.py, see also Sec. 5.19.2	402
5.23.5 graph.gp, see also Sec. 5.23.1	403
5.23.6 Modelling perspectives (Norwegian)	404
5.24 Exercise 12	405

TKP4106 Process Modelling

This is the joint homepage for TKP4106 Process Modelling. The course is composed of two parallel sessions - Modelling theory (HAP) and Programming (THW) - each of them having a dedicated webpage. The syllabus and the FAQ list are in common.

ALL THE COURSE MATERIAL NEEDED TO COMPLETE THE COURSE IS ON THE WEB.

THAT INCLUDES WHITE PAPERS, SOURCE CODE AND EXTERNAL REFERENCES.

WHICH MEANS YOU'LL ACTUALLY HAVE TO BE READ THE PAGES TOP DOWN - NOT SIMPLY BROWSE THEM.

We also expect you to **visit** all the external links in order to get an overview of the entire course.

Lecturer's home page:

1. [Tore Haug-Warberg \(Programming\)](#)
2. [Heinz A. Preisig \(Modelling\)](#)

Common parts:

1. [Python manual \(very good\)](#)
2. [Frequently Asked Questions \(FAQ\)](#)
3. [Syllabus](#)

Process modelling builds on the basic conservation principles, transport phenomena, thermodynamics and mathematical physics. We teach on how these models are being built systematically so that we have precisely the knowledge required - neither more nor less. Models we establish formulate many different mathematical problems that need to be solved simultaneously in order to get an over-all solution. We learn on how to approach, and to solve, these problems effectively using mathematical and computer-based numerical tools. Programming is seen as a core activity for achieving this latter goal. Examples taken from the different corners of our discipline are the subjects of our discussions.

Learning outcome:

1. Get a birdsview of the modelling process.
2. Establish an integration of the different involved subjects.

3. Programming as part of solving technical problems.
4. Abstraction of the plant.
5. Formulation of complete process models.
6. Solving simple mathematical and numerical problems using computers.
7. Programming methods and a programming language.
8. Have a systematic approach to problem solving.
9. Know how to generate models.

TKP4106 Programming Activities



"The easiest way to tell a Real Programmer from the crowd is by the programming language he (or she) uses. Real Programmers use Fortran. Quiche Eaters use Pascal. Nicklaus Wirth, the designer of Pascal, gave a talk once at which he was asked, "How do you pronounce your name?". He replied, "You can either call me by name, pronouncing it 'Veert', or call me by value, 'Worth'." One can tell immediately by this comment that Nicklaus Wirth is a Quiche Eater. The **only** parameter passing mechanism endorsed by Real Programmers is call-by-value-return, as implemented in the IBM/370 Fortran G and H compilers. Real Programmers don't need all these abstract concepts to get their jobs done-- they are perfectly happy with a keypunch, a Fortran IV compiler, and a beer."

[Real Programmers use FORTRAN](#)

This page gives a brief introduction to the programming activities in Process Modelling TKP4106. For easy off-line browsing you can download the contents of the entire course as a 6.2 MB PDF-file [here](#). There is also a [FAQ](#) list and a [syllabus](#) available. See also the other links at the top of the page. The Goals section below gives an overview of where we are heading. We will be using Python for the programming, but we shall try to stay away from external libraries and rather work out the software as needed. This brings in topics like: formula parsing, atom matrix and matrix product calculation, row-reduced-echelon-form, nullspace, linear and non-linear equation solving, Euler and Runge-Kutta integration, a thermodynamic equation of state and an object-oriented flowsheet module with stream and reactor objects. To increase the learning effect you are **not** given the programs out of the box. Instead you are asked to change these [stub programs](#) into workable code as a compulsory part of the course.

The work flow of a typical modelling job has 5 dedicated tasks:

- Algebra		Theory part of TKP4106
- Analysis		- " -
.....	
- Algorithm		Computer lab activity in TKP4106
- Programming		- " -
- Simulation		- " -

but most students find it difficult to work in a top-bottom style and have a tendency of doing things backwards. In particular there is a lack of understanding (and appreciation) for the necessity of doing the algebra and the

analysis **before** one starts doing programming. The short-cut seldomly works and a lot of time and frustration is spent on a task that would otherwise be affordable. The Golden Rule is therefore: Always do your paper work before you attempt any serious programming!

There are many ways to Rome, and even more ways to learn how to do computer programming, but one way to learn is to travel the long and winding road of a complete modelling task. The problem we are going to study here is that of a steady-state Plug Flow Reactor. This will naturally bring in the algebra and analysis needed for understanding things like: Algorithmic parsing of chemical formulas, matrix theory, thermodynamic Jacobian transformations and integration of ordinary differential equations, before we finally end up doing a chemical reactor simulation. Our scientific "value chain" looks something like this:

$$\begin{aligned}
 & ['H_2', 'N_2', 'NH_3'] \\
 & \Rightarrow \\
 A = & \left| \begin{array}{ccc} 2 & 0 & 3 \\ 0 & 2 & 1 \end{array} \right| \\
 & \Rightarrow \\
 N = & \left| \begin{array}{c} 3/2 \\ 1/2 \\ -1 \end{array} \right| \\
 & \Rightarrow \\
 & \left| \begin{array}{ccc|ccc} dh/dT & dh/dv & dh/dc & | & grad(T) & | & 0 \\ dT/dp & dp/dv & dp/dc & * & grad(v) & | & 0 \\ 0 & 0 & I & | & grad(c) & | & N \cdot r \end{array} \right|
 \end{aligned}$$

Here, A is the so-called atom or formula matrix, $N = \text{null}(A)$ is the nullspace of A , $h(T, v, c)$ is a thermodynamic function called enthalpy and $r(T, v, c, x, t)$ is the rate of reaction (chemical kinetics). It will be our pride to learn how this grand picture evolves from basic physical principles and a few pages of computer code. But, we should ask:

Q: Why?

A: The understanding and use of physically based models is becoming increasingly important in industry, teaching and academia.

Q: What?

A: Algorithmic description of dynamics, events and static processes. Conservation of mass and energy (not so much momentum in our case). The models can be simple yet complex (networks).

Q: How?

A: Linear algebra (ODE and DAE), root solvers (NR), syntax (regex and BNF)

parsers), code structure (OOP and FP), containers (tuple, list, hash, struct and array), code design (epydoc, patterns and exceptions).

So, our goals are obviously quite widespread and it is worth while reflecting a little over what we actually need to understand of mathematics, physics and programming in solving the chemical reactor problem:

Goals (programming):	Goals (paradigms):	Goals (modelling):
<ol style="list-style-type: none"> 1. Formula parser <code>dict = atoms(str)</code> 2. Algebra <code>mw = molecular_weight(str)</code> 3. Formula matrix <code>A = amat([str1, str2, ...])</code> 4. Row-reduced-echelon-form <code>B = rref(A)</code> 5. Nullspace <code>N = null(A)</code> 6. Linear equations <code>X = solve(A, B)</code> 7. Matrix product <code>C = mprod(A, B)</code> 8. Integration <code>hpn_vs_tvn('Euler', r, 0, 1, n)</code> 	<ol style="list-style-type: none"> 1. Backus-Naur formalism 2. Regular expressions 3. Strings 4. Lists (arrays) 5. Tuples 6. Dictionaries (hashes) 7. Lambda functions 8. Modules 9. Classes 10. Objects 11. Exceptions 	<ol style="list-style-type: none"> 1. Applying energy, momentum and mass conservation 2. Chemical reactions and nullspace 3. Linear and non-linear system descriptions 4. Linearization of models 5. Solving linear equations 6. Newton-Raphson iteration 7. Systems of ordinary differential equations 8. Dynamic versus steady state approximation 9. Numerical integration using Euler's method 10. The needs for an equation of state 11. Thermodynamic Jacobian transformations 12. Hand calculations of (1 x 1) up to (3 x 6) matrices

Occasionally, there are matter-of-programming-fact discussions going on in the corridor and my colleagues wonder whether the choice of a computer language really **matters** (which of course it does because there are more than [2000 languages](#) around), why a switch-case test is **better** than if-else-if-else (a compelling thought indeed), why Object Oriented Programming (OOP) is **better** than Imperative Programming (IP) (which is not always the case), why Python is **better** than Matlab (which is maybe true), and so on. My personal attitude to a few of these questions is collected in a list of inFrequently Asked Questions below. However, in the name of science I should be more objective, really. So, to give a better understanding of what programming is I have collected some data showing the chronology from the late 1950s to present. I believe that this knowledge is important for understanding how, and along which lines, the computer languages have evolved.

Procedural	Structured	Object-oriented	Functional
<ol style="list-style-type: none"> 1. FORTRAN (1957) 2. ALGOL (1958) 3. COBOL (1959) 	<ol style="list-style-type: none"> 1. Pascal (1972) 2. C (1972) 3. Ada (1983) 	<ol style="list-style-type: none"> 1. Simula (1967) 2. Smalltalk (1972) 3. C++ (1985) 	<ol style="list-style-type: none"> 1. Lisp (1958) 2. APL (1962) 3. ML (1974)

4. BASIC (1964)		4. Perl (1987) 5. Python (1990) 6. Java (1994) 7. Ruby (1995) 8. C# (2002)	4. Scheme (1975) 5. Miranda (1985) 6. Haskell (1987)
-----------------	--	--	--

(in)Frequently Asked Questions (iFAQ):

- Which language?
Use the language that is ideal for you **and** your task. Always. Switch to another language if you feel constrained.
- Why do I need an editor?
The editor and the keyboard are your textual links to the computer. Forget about the mouse and fancy GUIs. Such things are only useful for graphics work and hyperlinks. Learn about the **shortkeys** of your computer, learn to master one editor efficiently, learn to manipulate several files at once and learn to run scripts from the terminal (command) window. Use these tools for all your stuff afterwards. This is not about religion but about productivity and self-consciousness.
- Matlab or Python?
Matlab stands for Matrix Laboratory while Python is a generic **programming** language. Matlab is good at doing numbers while it sucks on doing strings. Python is good at handling strings and have good numerics too. More important, however, Matlab is proprietary while Python is open source. NTNU should not promote proprietary languages... Python has also a much bigger community than has Matlab (about 10 times higher activity according to [The Transparent Language Popularity Index](#)). Actually, we should rather been using Ruby because it has a nice, rich and beautiful syntax!
- OOP, IP or FP?
Object oriented programming (OOP) is valuable for administrating calculations at a high level using the concept of a **class**. Imperative programming (IP) is, quite inevitably, what is used in the inner loops of calculation intensive algorithms like e.g. matrix calculations. Functional programming (FP) offers a beatiful way of doing recursive calculations on infinite lists and so-called **higher order** programming working with functors (akin to functionals in mathematics). In most program systems of reasonable size all three paradigms will be used.
- IF-ELSEIF-ELSE or CASE?
The answer is almost religious: Never use **if-elseif-else** only **if-else** and **switch-case** or **case-when**. The reason is that an **if-elseif** has to be evaluated one test at a time (you can be comparing strings in one test and numbers in the next) while the **switch-case** is precompiled (you compare one single object to a set of predefined matches). The **if-elseif** clutters the code because you have to read every single statement in order to understand what is being tested. The scope of the **switch-case** is, on the other hand, determined by one single line of code and it consequently looks more clean and coherent to the human eye.
- TDT41100 vs TKP4106?
Why are we going to have yet-another introduction course in programming? Why is not TDT41100 sufficient? The answer is simple: TDT41100 offers you an introduction to information technology while TKP4106 focuses at writing beautiful code that stands the test of documentation standards, unittesting and reusability.

Real Programmers Don't Use Pascal

[A letter to the editor of *Datamation*, volume 29 number 7, July 1983. I've long ago lost my dog-eared photocopy, but I believe this was written (and is copyright) by Ed Post, Tektronix, Wilsonville OR USA.

The [story of Mel](#) is a related article.]

Back in the good old days-- the "Golden Era" of computers-- it was easy to separate the men from the boys (sometimes called "Real Men" and "Quiche Eaters" in the literature). During this period, the Real Men were the ones who understood computer programming, and the Quiche Eaters were the ones who didn't. A real computer programmer said things like "DO 10 I=1,10" and "ABEND" (they actually talked in capital letters, you understand), and the rest of the world said things like "computers are too complicated for me" and "I can't relate to computers-- they're so impersonal". (A previous work [1] points out that Real Men don't "relate" to anything, and aren't afraid of being impersonal.)

But, as usual, times change. We are faced today with a world in which little old ladies can get computers in their microwave ovens, 12 year old kids can blow Real Men out of the water playing Asteroids and Pac-Man, and anyone can buy and even understand their very own personal Computer. The Real Programmer is in danger of becoming extinct, of being replaced by high school students with TRASH-80s.

There is a clear need to point out the differences between the typical high school junior Pac-Man player and a Real Programmer. If this difference is made clear, it will give these kids something to aspire to-- a role model, a Father Figure. It will also help explain to the employers of Real Programmers why it would be a mistake to replace the Real Programmers on their staff with 12 year old Pac-Man players (at a considerable salary savings).

The easiest way to tell a Real Programmer from the crowd is by the programming language he (or she) uses. Real Programmers use Fortran. Quiche Eaters use Pascal. Nicklaus Wirth, the designer of Pascal, gave a talk once at which he was asked, "How do you pronounce your name?". He replied, "You can either call me by name, pronouncing it 'Veert', or call me by value, 'Worth'." One can tell immediately by this comment that Nicklaus Wirth is a Quiche Eater. The only parameter passing mechanism endorsed by Real Programmers is call-by-value-return, as implemented in the IBM/370 Fortran G and H compilers. Real Programmers don't need all these abstract concepts to get their jobs done-- they are perfectly happy with a keypunch, a Fortran IV compiler, and a beer.

- Real Programmers do List Processing in Fortran.
- Real Programmers do String Manipulation in Fortran.
- Real Programmers do Accounting (if they do it at all) in Fortran.
- Real Programmers do Artificial Intelligence programs in Fortran.

If you can't do it in Fortran, do it in assembly language. If you can't do it in assembly

language, it isn't worth doing.

The academics in computer science have gotten into the "structured programming" rut over the past several years. They claim that programs are more easily understood if the programmer uses some special language constructs and techniques. They don't all agree on exactly which constructs, of course, and the example they use to show their particular point of view invariably fit on a single page of some obscure journal or another-- clearly not enough of an example to convince anyone. When I got out of school, I thought I was the best programmer in the world. I could write an unbeatable tic-tac-toe program, use five different computer languages, and create 1000 line programs that WORKED (Really!). Then I got out into the Real World. My first task in the Real World was to read and understand a 200,000 line Fortran program, then speed it up by a factor of two. Any Real Programmer will tell you that all the Structured Coding in the world won't help you solve a problem like that-- it takes actual talent. Some quick observations on Real Programmers and Structured Programming:

- Real Programmers aren't afraid to use GOTOs.
- Real Programmers can write five page long DO loops without getting confused.
- Real Programmers like Arithmetic IF statements-- they make the code more interesting.
- Real Programmers write self-modifying code, especially if they can save 20 nanoseconds in the middle of a tight loop.
- Real Programmers don't need comments-- the code is obvious.
- Since Fortran doesn't have a structured IF, REPEAT ... UNTIL, or CASE statement, Real Programmers don't have to worry about not using them. Besides, they can be simulated when necessary using assigned GOTOs.

Data structures have also gotten a lot of press lately. Abstract Data Types, Structures, Pointers, Lists, and Strings have become popular in certain circles. Wirth (the above mentioned Quiche Eater) actually wrote an entire book [2] contending that you could write a program based on data structures, instead of the other way around. As all Real Programmers know, the only useful data structure is the Array. Strings, Lists, Structures, Sets-- these are all special cases of arrays and can be treated that way just as easily without messing up your programming language with all sorts of complications. The worst thing about fancy data types is that you have to declare them, and Real Programming Languages, as we all know, have implicit typing based on the first letter of the (six character) variable name.

What kind of operating system is used by a Real Programmer? CP/M? God forbid-- CP/M, after all, is basically a toy operating system. Even little old ladies and grade school students can understand and use CP/M.

Unix is a lot more complicated of course-- the typical Unix hacker never can remember what the PRINT command is called this week-- but when it gets right down to it, Unix is a glorified video game. People don't do Serious Work on Unix systems: they send jokes around the world on UUCP-net and write Adventure games and research papers.

No, your Real Programmer uses OS/370. A good programmer can find and understand the description of the IJK305I error he just got in his JCL manual. A great programmer can write JCL without referring to the manual at all. A truly outstanding programmer can find bugs buried in a 6 megabyte core dump without using a hex calculator. (I have actually seen this done.)

OS is a truly remarkable operating system. It's possible to destroy days of work with a single misplaced space, so alertness in the programming staff is encouraged. The best way to approach the system is through a keypunch. Some people claim there is a Time Sharing system that runs on OS/370, but after careful study I have come to the conclusion that they were mistaken.

What kind of tools does a Real Programmer use? In theory, a Real Programmer could run his programs by keying them into the front panel of the computer. Back in the days when computers had front panels, this was actually done occasionally. Your typical Real Programmer knew the entire bootstrap loader by memory in hex, and toggled it in whenever it got destroyed by his program. (Back then, memory was memory-- it didn't go away when the power went off. Today, memory either forgets things when you don't want it to, or remembers things long after they're better forgotten.) Legend has it that Seymour Cray, inventor of the Cray I supercomputer and most of Control Data's computers, actually toggled the first operating system for the CDC7600 in on the front panel from memory when it was first powered on. Seymour, needless to say, is a Real Programmer.

One of my favorite Real Programmers was a systems programmer for Texas Instruments. One day, he got a long distance call from a user whose system had crashed in the middle of saving some important work. Jim was able to repair the damage over the phone, getting the user to toggle in disk I/O instructions at the front panel, repairing system tables in hex, reading register contents back over the phone. The moral of this story: while a Real Programmer usually includes a keypunch and line printer in his toolkit, he can get along with just a front panel and a telephone in emergencies.

In some companies, text editing no longer consists of ten engineers standing in line to use an 029 keypunch. In fact, the building I work in doesn't contain a single keypunch. The Real Programmer in this situation has to do his work with a "text editor" program. Most systems supply several text editors to select from, and the Real Programmer must be careful to pick one that reflects his personal style. Many people believe that the best text editors in the world were written at Xerox Palo Alto Research Center for use on their Alto and Dorado computers[3]. Unfortunately, no Real Programmer would ever use a computer whose operating system is called SmallTalk, and would certainly not talk to the computer with a mouse.

Some of the concepts in these Xerox editors have been incorporated into editors running on more reasonably named operating systems-- EMACS and VI being two. The problem with these editors is that Real Programmers consider "what you see is what you get" to be just as bad a concept in Text Editors as it is in Women. No, the Real Programmer wants a "you asked for it, you got it" text editor-- complicated, cryptic, powerful, unforgiving, dangerous. TECO, to be precise.

It has been observed that a TECO command sequence more closely resembles transmission line noise than readable text[4]. One of the more entertaining games to play

with TECO is to type your name in as a command line and try to guess what it does. Just about any possible typing error while talking with TECO will probably destroy your program, or even worse-- introduce subtle and mysterious bugs in a once working subroutine.

For this reason, Real Programmers are reluctant to actually edit a program that is close to working. They find it much easier to just patch the binary object code directly, using a wonderful program called SUPERZAP (or its equivalent on non-IBM machines). This works so well that many working programs on IBM systems bear no relation to the original Fortran code. In many cases, the original source code is no longer available. When it comes time to fix a program like this, no manager would even think of sending anything less than a Real Programmer to do the job-- no Quiche Eating structured programmer would even know where to start. This is called "job security".

Some programming tools NOT used by Real Programmers:

- Fortran preprocessors like MORTRAN and RATFOR. The Cuisinarts of programming-- great for making Quiche. See comments above on structured programming.
- Source language debuggers. Real Programmers can read core dumps.
- Compilers with array bounds checking. They stifle creativity, destroy most of the interesting uses for EQUIVALENCE, and make it impossible to modify the operating system code with negative subscripts. Worst of all, bounds checking is inefficient.
- Source code maintenance systems. A Real Programmer keeps his code locked up in a card file, because it implies that its owner cannot leave his important programs unguarded [5].

Where does the typical Real Programmer work? What kind of programs are worthy of the efforts of so talented an individual? You can be sure that no Real Programmer would be caught dead writing accounts-receivable programs in COBOL, or sorting mailing lists for People magazine. A Real Programmer wants tasks of earth-shaking importance (literally!).

- Real Programmers work for Los Alamos National Laboratory, writing atomic bomb simulations to run on Cray I supercomputers.
- Real Programmers work for the National Security Agency, decoding Russian transmissions.
- It was largely due to the efforts of thousands of Real Programmers working for NASA that our boys got to the moon and back before the Russkies.
- The computers in the Space Shuttle were programmed by Real Programmers.
- Real Programmers are at work for Boeing designing the operation systems for cruise missiles.

Some of the most awesome Real Programmers of all work at the Jet Propulsion Laboratory in California. Many of them know the entire operating system of the Pioneer and Voyager spacecraft by heart. With a combination of large ground-based Fortran programs and small spacecraft-based assembly language programs, they are able to do incredible feats of navigation and improvisation-- hitting ten-kilometer wide windows at Saturn after six years in space, repairing or bypassing damaged sensor platforms, radios, and batteries. Allegedly, one Real Programmer managed to tuck a pattern matching program into a few hundred bytes of unused memory in a Voyager spacecraft that searched for, located, and photographed a new moon of Jupiter.

The current plan for the Galileo spacecraft is to use a gravity assist trajectory past Mars on the way to Jupiter. This trajectory passes within 80 +/- 3 kilometers of the surface of Mars. Nobody is going to trust a Pascal program (or Pascal programmer) for navigation to these tolerances.

As you can tell, many of the world's Real Programmers work for the U.S. Government-- mainly the Defense Department. This is as it should be. Recently, however, a black cloud has formed on the Real Programmer horizon. It seems that some highly placed Quiche Eaters at the Defense Department decided that all Defense programs should be written in some grand unified language called "ADA" ((C), DoD). For a while, it seemed that ADA was destined to become a language that went against all the precepts of Real Programming-- a language with structure, a language with data types, strong typing, and semicolons. In short, a language designed to cripple the creativity of the typical Real Programmer. Fortunately, the language adopted by DoD had enough interesting features to make it approachable-- it's incredibly complex, includes methods for messing with the operating system and rearranging memory, and Edsger Dijkstra doesn't like it [6]. (Dijkstra, as I'm sure you know, was the author of "GOTOs Considered Harmful"-- a landmark work in programming methodology, applauded by Pascal Programmers and Quiche Eaters alike.) Besides, the determined Real Programmer can write Fortran programs in any language.

The Real Programmer might compromise his principles and work on something slightly more trivial than the destruction of life as we know it. Providing there's enough money in it. There are several Real Programmers building video games at Atari, for example. (But not playing them-- a Real Programmer knows how to beat the machine every time: no challenge in that.) Everyone working at LucasFilm is a Real Programmer. (It would be crazy to turn down the money of fifty million Star Trek fans.) The proportion of Real Programmers in Computer Graphics is somewhat lower than the norm, mostly because nobody has found a use for Computer Graphics yet. On the other hand, all Computer Graphics is done in Fortran, so there are a fair number of people doing Graphics in order to avoid having to write COBOL programs.

Generally, the Real Programmer plays the same way he works-- with computers. He is constantly amazed that his employer actually pays him to do what he would be doing for fun anyway (although he is careful not to express this opinion out loud). Occasionally, the Real Programmer does step out of the office for a breath of fresh air and a beer or two. Some tips on recognizing Real Programmers away from the computer room:

- At a party, the Real Programmers are the ones in the corner talking about operating system security and how to get around it.

- At a football game, the Real Programmer is the one comparing the plays against his simulations printed on 11 by 14 fanfold paper.
- At the beach, the Real Programmer is the one drawing flowcharts in the sand.
- At a funeral, the Real Programmer is the one saying "Poor George. And he almost had the sort routine working before the coronary."
- In a grocery store, the Real Programmer is the one who insists on running the cans past the laser checkout scanner himself, because he never could trust keypunch operators to get it right the first time.

What sort of environment does the Real Programmer function best in? This is an important question for the managers of Real Programmers. Considering the amount of money it costs to keep one on the staff, it's best to put him (or her) in an environment where he can get his work done.

The typical Real Programmer lives in front of a computer terminal. Surrounding this terminal are:

- Listings of all programs the Real Programmer has ever worked on, piled in roughly chronological order on every flat surface in the office.
- Some half-dozen or so partly filled cups of cold coffee. Occasionally, there will be cigarette butts floating in the coffee. In some cases, the cups will contain Orange Crush.
- Unless he is very good, there will be copies of the OS JCL manual and the Principles of Operation open to some particularly interesting pages.
- Taped to the wall is a [line-printer Snoopy calendar for the year 1969.](#)
- Strewn about the floor are several wrappers for peanut butter filled cheese bars-- the type that are made pre-stale at the bakery so they can't get any worse while waiting in the vending machine.
- Hiding in the top left-hand drawer of the desk is a stash of double-stuff Oreos for special occasions.
- Underneath the Oreos is a flow-charting template, left there by the previous occupant of the office. (Real Programmers write programs, not documentation. Leave that to the maintenance people.)

The Real Programmer is capable of working 30, 40, even 50 hours at a stretch, under intense pressure. In fact, he prefers it that way. Bad response time doesn't bother the Real Programmer-- it gives him a chance to catch a little sleep between compiles. If there is not enough schedule pressure on the Real Programmer, he tends to make things more challenging by working on some small but interesting part of the problem for the first nine weeks, then finishing the rest in the last week, in two or three 50-hour marathons. This not only impresses the hell out of his manager, who was despairing of ever getting the project done on time, but creates a convenient excuse for not doing the

documentation. In general:

- No Real Programmer works 9 to 5. (Unless it's the ones at night.)
- Real Programmers don't wear neckties.
- Real Programmers don't wear high heeled shoes.
- Real Programmers arrive at work in time for lunch.
- A Real Programmer might or might not know his wife's name. He does, however, know the entire ASCII (or EBCDIC) code table.
- Real Programmers don't know how to cook. Grocery stores aren't open at three in the morning. Real Programmers survive on Twinkies and coffee.

What of the future? It is a matter of some concern to Real Programmers that the latest generation of computer programmers are not being brought up with the same outlook on life as their elders. Many of them have never seen a computer with a front panel. Hardly anyone graduating from school these days can do hex arithmetic without a calculator. College graduates these days are soft-- protected from the realities of programming by source level debuggers, text editors that count parentheses, and "user friendly" operating systems. Worst of all, some of these alleged "computer scientists" manage to get degrees without ever learning Fortran! Are we destined to become an industry of Unix hackers and Pascal programmers?

From my experience, I can only report that the future is bright for Real Programmers everywhere. Neither OS/370 nor Fortran show any signs of dying out, despite all the efforts of Pas- cal programmers the world over. Even more subtle tricks, like adding structured coding constructs to Fortran, have failed. Oh sure, some computer vendors have come out with Fortran 77 compilers, but every one of them has a way of converting itself back into a Fortran 66 compiler at the drop of an option card-- to compile DO loops like God meant them to be.

Even Unix might not be as bad on Real Programmers as it once was. The latest release of Unix has the potential of an operating system worthy of any Real Programmer-- two different and subtly incompatible user interfaces, an arcane and complicated teletype driver, virtual memory. If you ignore the fact that it's "structured", even 'C' programming can be appreciated by the Real Programmer: after all, there's no type checking, variable names are seven (ten? eight?) characters long, and the added bonus of the Pointer data type is thrown in-- like having the best parts of Fortran and assembly language in one place. (Not to mention some of the more creative uses for #define.)

No, the future isn't all that bad. Why, in the past few years, the popular press has even commented on the bright new crop of computer nerds and hackers ([7] and [8]) leaving places like Stanford and MIT for the Real World. From all evidence, the spirit of Real Programming lives on in these young men and women. As long as there are ill-defined goals, bizarre bugs, and unrealistic schedules, there will be Real Programmers willing to jump in and Solve The Problem, saving the documentation for later. Long live Fortran!

References:

- [1] Feirstein, B., "Real Men don't Eat Quiche", New York, Pocket Books, 1982.
- [2] Wirth, N., "Algorithms + Data Structures = Programs", Prentice Hall, 1976.
- [3] Ilson, R., "Recent Research in Text Processing", IEEE Trans. Prof. Commun., Vol. PC-23, No. 4, Dec. 4, 1980.
- [4] Finseth, C., "Theory and Practice of Text Editors - or - a Cookbook for an EMACS", B.S. Thesis, MIT/LCS/TM-165, Massachusetts Institute of Technology, May 1980.
- [5] Weinberg, G., "The Psychology of Computer Programming", New York, Van Nostrand Reinhold, 1971, p. 110.
- [6] Dijkstra, E., "On the GREEN language submitted to the DoD", Sigplan notices, Vol. 3, No. 10, Oct 1978.
- [7] Rose, Frank, "Joy of Hacking", Science 82, Vol. 3, No. 9, Nov 82, pp. 58-66.
- [8] "The Hacker Papers", Psychology Today, August 1980.

ACKNOWLEDGEMENT

I would like to thank Jan E., Dave S., Rich G., Rich E. for their help in characterizing the Real Programmer, Heather B. for the illustration, Kathy E. for putting up with it, and atd!avsdS:mark for the initial inspiration.

Webbed by [Greg Lindahl](#) (lindahl@pbm.com)

HEINZ A PREISIG

Process Systems Engineering, NTNU

Menu

Browse: Home / Courses / TKP4106 Script and
Exercises

TKP4106 SCRIPT AND EXERCISES

LINKS >>>

[Script_v2](#)

[Slides](#)

[Exercise sets](#)

Script changes:

- complete revision
- active nomenclature

Script TO DO:

- Chapter 7: to be moved to after Chapter 10 and add pressure distribution network (event-dynamic). Alternatively do not move but add a event-dynamic mass system and add the

pressure distribution discussion to Chapter 10.

- Chapter 12: extend into finite volume because it matches Chapter 4
- Chapter 13: Something to think about. Should probably be properly extended. Needs some more thinking.
- Extensions beyond state space: More on linearisation, properties like observability and controllability, stability. Model reduction due to assumptions an extension of what is woven into the text until now.

Note: This course is under development, consequently the linked documents may change during the session. There is a slight miss-match with titles on the exercises due to mix of old and new versions.

Computing exercises: compulsory – required because they are necessary ingredients for later assignments.

Theory exercises: Some compulsory exercises – will be defined on distribution. 80 % of the remaining exercises.

Pensum: Script

Solutions: Will be made available

Frequently asked questions (FAQ)

Nuts and bolts:

1. How to create a homepage on the stud server: [change permissions here](#).
2. Publishing .py files: By default the server **folk.ntnu.no** treats files ending in .py as binary files. So, if you click a link to a .py file, the file will be downloaded. This is OK if you actually want to edit or play around with the file, but not OK if you want to have a quick look and then leave the file. However, **folk.ntnu.no** runs apache web server and it can be configured (recursively on a folder by folder basis) using a special hidden file called **.htaccess**. The trick is to configure the server such that .py files get served as text/plain mime type. Because different versions of Windows can make it confusing working with hidden files, and because the task is very simple to solve in LINUX which is available to all students at **logon.stud.ntnu.no**, open Putty (it is on most NTNU computers, or you can install it on your own) and enter **logon.stud.ntnu.no**. Then logon using your normal username and password and copy the following into the terminal **echo 'AddType text/plain .py' >> ~/public_html/.htaccess**.
- 3.
- 4.

Python:

1. Use **quit()** or **ctrl + Z** to exit Python in the command window.
2. Comparison operators in Python are the same as in C/C++ that are **==**, **!=**, **<=** and **>=**.
3. The indexing of lists, vectors, etc. starts at 0 - not at 1 as in FORTRAN and Matlab.
4. Use colon (**:**) to terminate **if**, **else**, **while** and **for** conditionals.
5. The **elif** in Python corresponds to **else if** in C/C++.
6. To start writing Python you must be familiar with the most basic programming concepts:
 - recursion
 - loops (for, while)
 - regular expressions
 - functions
7. You must know how to work with basic objects and containers:
 - string
 - number
 - float
 - int
 - list

- dictionary

8. Finally, you must know the meaning of a few reserved words:

- for, while
- if, else, elif
- def
- len
- return
- int
- help
- import
- help
- dir

9. Importing mathematics package:

- import math

10. Importing regular expression package:

- import re
- re.match('looking for re', 'in string')
- re.group()

11. Working with dictionaries:

- dict.get()
- dict.pop()
- dict.iteritems()

12.

13.

Unix/Linux/Cygwin:

1. Find all files of kind TeX or LaTeX in your Document catalogue: **find ~/Documents/ -iname *.tex**
2. Find all occurrences of PYTHON, Python, python etc. in those files: **grep -E -i --color 'python' `find ~/Documents/ -iname *.tex`**
3. Collect all .py files in every sub-directory into a new file called tmp: **for file in **/*.py; do cat \$file; done > tmp**
4. Calculate the cumulative number of words in the entire directory tree: **ls -R ./**/* | wc -w**
5. Remove comment lines from Python script: **grep -Ev '^\\s*(#.*)?\$\$' foo.py**
- 6.
- 7.

Windows:

1. Use **quit()** or **ctrl + Z** to exit Python in the command window.
2. How to [use epydoc](#) in the command window: Open the cmd window.

Change directory (cd) to the folder where epydoc.py was saved (C:\Python27\Scripts). Enter the command epydoc.py and then the path to the file you want to run epydoc on (e.g. epydoc C:\Python27\myfiles\atoms.py). Command line options can be (e.g. -o myfiles\html will send output to an html folder in myfiles).

3. How to set the Python path in windows 7: My computer -> system properties -> advanced settings -> environment variables -> scroll down to path in the window below -> edit -> add ;C:\Python27 at the end of list. Press OK. Now you can open python.exe in the cmd window independent of where in the path you are at the present.
4. How to change text colour in command window: Right click on the command line. Choose properties -> colors -> windows text -> choose the pale green color.
- 5.
- 6.

TextPad:

1. How to find syntax for regular expressions: help -> help topics -> how to... -> find and replace text -> use regular expressions. Will then get a list of all legal search expressions.
2. How to get line numbers: Configure -> preferences -> view -> line numbers (tick off).
3. How to get default file ending of .py: Configure -> preferences -> file -> default extension: py.
4. Downloading of syntax highlighting: Choose the one of python(8) -> download to the Samples folder where TextPad has been installed. Go to TextPad, close all open documents. Choose configure-> new document class -> follow the instructions for installation. Remember to tick off the Enable syntax highlighting box. In the drop-down window: Syntax definition file -> choose the file you just have downloaded.
5. How to change background colours: Close all open documents. Configure -> preferences -> document classes -> python... -> colors -> choose more colors -> choose the yellow color close to the centre of the circle.
6. Use **ctrl + tab** to switch between open documents.
- 7.
- 8.

Syllabus (TKP4106)

Week	Programming topics (THW)	Modelling (HAP)
Tore	Introduction to Python Running Python from the command line using text files.	...
Heinz	...	Exercise 1 Info about this week
Dove	Regular expressions Editors and regular expression search-and-replace, and the handling of multiple files.	Exercise 2 Info about this week
Chicken	Documenting your code Embedded documentation (epytext), and automatic documentation (epydoc).	Exercise 3 Info about this week
Lion	Molecular formula parser Backus-Naur formalism, regular expressions, and string parsing.	Exercise 4 Info about this week
Penguin	The atom matrix Dictionaries (hash tables) and iterators.	Exercise 5 Info about this week
Fish	Independent reactions Matrix algebra, null space, and the mass balance of chemically reacting systems.	Exercise 6 Info about this week
Elephant	Root solvers Solving non-linear problems in one variable. Safe-guarding the iteration.	Exercise 7 Info about this week
Beaver	A thermodynamic equation solver Solving a specification in $H, p, N1, N2, \dots$ with respect to $T, V, N1, N2, \dots$	Exercise 8 Info about this week
Kangaroo	The reactor model Making a generic simulation model for plug-flow reactors.	Exercise 9 Info about this week
Giraffe	Integration Solving ODEs using explicit and implicit Euler integration.	Exercise 10 Info about this week
Cow	Unit testing Verification and validation of computer code, and exception handling.	Exercise 11 Info about this week
Monkey	Putting the model to work Unit testing the model, and producing plots.	Exercise 12 Info about this week

A First Introduction to Python (TKP4106)



[/about/gettingstarted](#)

"Talking, you can only hope that somebody is listening. Writing, you can only hope that someone will be reading. When doing programming, however, you can tell the computer **what** to do, **how** to do it and **when** it should be done. That makes a heck of a difference to the scientist."

Corollary: In speech and writing it does not matter how wrong you are if you are a little right. In programming it does not matter how right you are if you are a little wrong."

Introductory words to TKP4106, Tore Haug-Warberg (2011)

About Python as a language I am not religious. Not at all since I have coded only a few projects in Python. The syntax is admittedly not very juicy but the language offers a good compromise between stringency and sloppiness, and it got tons of useful libraries. It also enforces strict indentation rules which is definitely a **Good Thing** for newbies. For this reason alone Python stands out as a good learning platform, besides being one of the more popular scripting languages available today (far more so than Matlab for instance). The impatient reader may already find the [Seven Topics in Python](#) useful. The newbie should attempt a peek into this Python [script](#), stored [here](#), for «understanding» how a complete modelling project might look.

To do computer programming you'll need an editor. It will soon turn out that this will be your most valuable asset. Forget about fancy GUI's and IDE's used for large scale programming. There are several good editors on the market being free, stable, powerful and omnipresent. So, dispose the mouse, learn shortkeys and teach yourself TextPad, Vim, Emacs or ... That's it.

And yes, while programming you shall document your code. Always. Coding is about **syntax** — documentation is about **semantics**. Remember that. The idea is to document as much as possible in the code itself using plain 7-bit ASCII characters. The chances are that this documentation will survive all future changes made to both computers and software.

When it comes to technical details involving complicated mathematics, tables, pictures and other graphical elements we have to employ more advanced principles, but I still recommend to write everything as readable text and avoid using Microsoft Word and other text editors having proprietary data formats. For

scientific uses (La)TeX is still a good candidate.

You must also test your code. Always. What is now known as **unittesting** is a Good Thing.

Finally, a little humor helps a lot when you are hit hard by a dead-line and are forced to work late hours. Programmers are infamous for their dry and nerdish humour.

The table below presents what I think is a set of good links, but please feel free to suggest other links that are better. This homepage is under continuous development!

Editors:	Text processing:	Programming en masse:	Mostly fun:
<ol style="list-style-type: none">1. Emacs (all platforms)2. Emacs quick reference3. Vim (UNIX)4. Vim quick reference5. TextPad (Windows)6. TextPad quick reference	<ol style="list-style-type: none">1. LaTeX (Cambridge University)2. LaTeX in Norwegian (Hanche-Olsen)3. LaTeX professional math (Voss)4. High-quality portable PDF (Schatz)5. Regex (Stephen Ramsay)6. Regex quick reference7. BNF and EBNF (L. M. Garshol)	<ol style="list-style-type: none">1. Windows shortcuts (OIT)2. Linux programming (digilife)3. Mac shortcuts (macmost)4. The Transparent Language Popularity Index5. Commenting Python code (MIT)6. 99 bottles of beer (1000++ languages)7. Programming paradigms (Kurt Normark)	<ol style="list-style-type: none">1. Real Programmers (Ed Post)2. The story of Mel (Ed Nather.)3. The Tao of programming (Kragen Sitaker)4. Computer languages (E. Levenez)5. Shoot yourself in the foot (WWW)6. Lord of the Rings (D. Pritchard)7. About spell checkers (WWW)8. Foobar etymology (Jargon File)

It is said that Python is an Object Oriented Programming language. But Python and most other languages can also be used for Imperative Programming. So what does OOP mean in contrast to IP? Let me try to explain the difference in terms of how NTNU organizes the exams. Assume for the moment that NTNU is a central Python module and that you (the student) is a data object floating around in cyberspace. In Python jargon we can then state the following:

```
...
...
# A list of all courses at NTNU.
courses = [..., TKP4106, ...]
...
...
# It's time for arranging exams.
```

```

for course in courses:
    arrange_exam(course)
...
...
# Make sure all students do their exams.
def arrange_exam(course):
    for student in course.students():
        answer = student.do_exam(course)
        if answer == None:
            mark = 'Failed'
        else:
            mark = evaluate_exam(course, answer)
    end
    print(student, course, mark)
...
...

```

The big difference is how the methods `arrange_exam()` and `do_exam()` are implemented. NTNU is the official authority and knows exactly **why**, **what**, **who**, **when** and **where** to examine. NTNU's function `arrange_exam()` is therefore implemented as a global function which is part of an imperative schedule called a study program. I.e. NTNU tells you what to do at each level of your study. But, whenever NTNU alarms you to conduct an exam it invokes `do_exam()` which is an object method installed on you (and on all other student objects). It is in fact a **singleton** since it is installed on a one-to-one basis and will be different for each student. For that reason NTNU cannot rely fully on your scientific integrity and it therefore invokes another global function called `evaluate_exam()` which marks your answer. The rest of the story you all know... I hope this little allegory helps you understand the difference of OOP and IP.

To help you getting started with Python I have collected a set of links (below) which take you all the way from a novice's trial-and-error efforts to advanced programming issues using Matlab-style linear algebra packages. For those who would prefer an introduction that is more tightly bound to the contents of TKP4106 I can offer you this [Seven Topics in Python](#) written specifically to meet the needs of students coming in from the sideline without having any prior experiences with computer programming.

Getting started:	Going a little further:	The full story:
<ol style="list-style-type: none"> 1. Seven Topics in Python (Haug-Warberg) 2. A Beginner's Python Tutorial (Steven Thurlow) 3. Epydoc (sourceforge) 4. Epytext markup (sourceforge) 	<ol style="list-style-type: none"> 1. Python Docstrings (Sourceforge) 2. Regex in Python (python.org) 3. Unittesting in Python (python.org) 4. Python best practise (Well House) 	<ol style="list-style-type: none"> 1. Numerical Python (numpy.org) 2. Plotting with Python (matplotlib) 3. Scientific Python (numpy.org) 4. Symbolic Python (sympy.org)

Let's conclude this introduction by writing our first Python script:

```
print "Hello world"
```

As you can understand it simply prints a string saying hello world to the screen (standard output). Store the code in a file called TKP4106/Python/hello.py and run it from the terminal (while being positioned inside TKP4106/Python) issuing the command: python hello.py to the operating system. There is nothing magical about TKP4106/Python — it serves only as a decent placeholder for the Python files you are going to write throughout this course. To continue, we can make the script a little more useful for our future needs by re-writing it into:

```
hello = "Hello world"

if __name__ == '__main__':
    print hello
```

The program does still print its greeting to the world, but it first **assigns** the **string** "hello world" to the **variable** hello and then tests that you are actually **invoking** the **script** directly from the **command line** identified as "**__main__**" before it prints the **content** of hello to **standard output**. If this not the case — that is to say the file is being imported by another program — it will **not** print anything. This feature proves to be very handy while testing programs which are under development. More about these technical details later... What is crucial for the current understanding is that you start appreciating the language, or jargon, which is being used when talking about programmatic issues (assignments, variables, strings, command line, etc.).

Seven Topics in Python Programming

Editor: Tore Haug-Warberg

Programmer: Eivind Haug-Warberg

Copyright © 2012 by Tore Haug-Warberg

Department of Chemical Engineering

NTNU (Norway)

Created: August 25, 2014

Contents

Introduction	3
Starting Python	4
Assignments	4
Programming vs calculus	4
Datatypes	5
String formatting	6
Objects	6
Classes	8
Loops	8
Logic operators	9
Functions	9

Methods	11
1 Lists	12
New content	12
Generating arithmetic lists (ranges)	12
Generating arbitrary lists	13
The biggest element in a list	13
The code	13
Run by itself	14
2 Exceptions	15
New content	15
Exception handling	15
3 Regular expressions	17
New content	17
Regular expressions	17
Splitting a string	18
The code	18
Run by itself	20
4 Dictionaries	21
New content	21
The <code>set</code> datatype	21
The <code>dict</code> datatype	21
Default values for dictionaries	21
The code	21
Run by itself	22
5 Lambda functions	23
New content	23
Anonymous function calls	23
The code	23
Run by itself	24
6 Classes	25
New content	25
Creating a new class	25
The code	25
Run by itself	27
7 Unitests	28

Introduction

This little Python guide is designed to fill the needs of **Process Modelling** TKP4106 at IKP (NTNU). Its main purpose is to give our 3rd year students enrolled in the Chemical Engineering program a chance to become acquainted with computer programming. Python was chosen in competition with other scripting languages—not because it excels in any particular way—but because it has an easy and compact syntax and because it imposes *strict* indentation rules on the user. Rules which are consistent with programming in almost any language but which are easy to forget by newcomers in the field. From a practical point of view it is more efficient for Python to enforce the rules than for the teacher to advise the students to follow them.

The entire course is screwed together with two simple thoughts in mind: The examples ought to be simple yet realistic and they should not constitute more than 30 pages of documented code. A small program called `m2py` plays the role of the centerpiece in this matter. It aims at using different parsing techniques for reading Matlab input strings into numerical matrices. The motivation for this choice is that Matlab is in widespread use for technical calculations and that it can be of interest in its own right to parse Matlab input directly from Python. The string parsing requires besides non-trivial programming with a minimum knowledge of math, physics, numerics or whatever. Still, the programs are not dummies but real-working programs that can be used afterwards by the students.

But why do we take the troubles to design a course ourselves? Why don't we simply drop into the internet and link up to some other resources? Well, we think that to teach students programming, or any other kind of technical subject, they must feel that the examples are relevant and you must know the problem yourself—by heart. This does not mean that we can simply ignore the rest of the world and the interested reader is therefore given a list of similar projects below. These projects are from other universities with a profile similar to NTNU. The google terms we used to locate the courses were: <univ.name> python modelling and <univ.name> python physics:

02450 Introduction to Machine Learning and Data Modeling. ECTS: 5.
texttthttt://www2.imm.dtu.dk/courses/02450/

02820 Python Programming. ECTS: 5.
texttthttt://www2.imm.dtu.dk/courses/02820/

X442.3 Python Programming. EECS: 1 semester unit.
texttthttt://extension.berkeley.edu/catalog/course522.html

CS9H Python. Self paced study.
texttthttt://inst.eecs.berkeley.edu/ selfspace/class/cs9h/

6.189 A Gentle Introduction to Programming Using Python. 6 units.
texttthttt://ocw.mit.edu/courses/electrical-engineering-and-computer-science/

CS 11 Python track. 3 units.
texttthttt://courses.cms.caltech.edu/cs11/material/python/index.html

Physics 20 Introduction to Tools of Scientific Computing. 6 units.
texttthttt://www.pma.caltech.edu/ physlab/physics_20_fall12.html

IN4186TU Datastructures, Algorithms and Databases. ECTS: 6.
texttthttt://studiegids.tudelft.nl/a101_displayCourse.do?course_id=16896

AE1205 Programming & Scientific Computing in Python for AE1. ECTS: 2.
texttthttt://www.studiegids.tudelft.nl/a101_displayCourse.do?course_id=27066

Note: The URLs are not hyperlinked because they are probably broken within a couple of years or so. It is at least reasons to think so because computer courses are among the most dynamic courses at the University level. The two courses given at Caltech are of special interest to TKP4106, however, and should be followed up more closely in the future.

Starting Python

There are two ways to use Python. Either by running it interactively from the console, that is the “Terminal” on Mac, the “command line” in Linux, or the “command prompt” in MS-Windows, or by asking it to compile and run an input-file written in Python syntax. Typing `python` by itself starts the Python interpreter:

```
1 | $ python
2 | >>> print 'Hello world'
3 | 'Hello world'
```

Everything you enter afterwards is interpreted as Python code. For example `print()` is a function that displays its arguments in your console window, in this case it is the content of a string. Typing `python <filename>.py` executes the contents of the file `<filename>.py`:

```
1 | $ python hello.py
2 | 'Hello world'
3 | $
```

In this case the input file `hello.py` is supposedly containing a single line of code that says `print 'Hello world'`. The output is therefore the same in both cases.

Assignments

Variables are used to keep track of the information flow. In Python a variable does not store the actual value but rather stores the memory address there the value is being located. To assign an address to a variable you write `<name> = <value>`. This lets you type `<name>` instead of `<value>` later.

```
1 | >>> a = 1
2 | >>> a + 2
3 | 3
```

To assign the same value to several variables at a time you type: `a = b = 1`. If you want several variables to have all different values you can do this: `a, b = 1, 2`.

Programming vs calculus

Arithmetic operators (`+`, `-`, `*` and `/`) in Python are similar to what you would expect while the syntax for “power” is `**`. Typing `a+=n`, `a-=n`, `a/=n`, `a*=n` etc. will do the same as `a=a+n`, `a=a-n`, `a=a/n`, `a=a*n`; it will increase, decrease, divide or multiply itself with `n`. Python is a list oriented language and is not made for doing calculus, really, and the `math` module is of great help when doing more advanced mathematics. E.g. importing `sqrt()` from the `math` module makes these calculations possible:

$$\begin{aligned}a &= 1, b = 4, c = -2 \\c \cdot (a + \sqrt{b})^3 &= -54\end{aligned}$$

Here is how it would look in Python:

```

1 | >>> from math import sqrt
2 | >>> a, b, c = 1, 4, -2
3 | >>> c * (a + sqrt(b))**3
4 | -54.0

```

The code segment `from math import sqrt` means that you import the function `sqrt()` from a file called `math`. On its own, `import math` imports the entire file and `sqrt()` must then be accessed by `math.sqrt()`. If you want to rename one of the functions in a module, or the module itself, you can use the `as` keyword: `from math import sqrt as squareroot`. Hereafter, `squareroot(x)` calculates the square root of `x`.

Python is also useful for more advanced operations and is especially well designed for handling lists and sequences efficiently. Take this sum as an example:

$$\sum_{n=1}^{50} n = 1275$$

To encode it in Python you can use the `range()` function which unlike `sqrt()` is a built-in function. It returns a list of numbers extending from the first argument up to (but not including) the second argument (see Section 1.2). The `for` loop (explained on page 9) is used to iterate through the list returned from `range()` and sum up the values of each of the elements. Here is how it would look in Python:

```

1 | >>> sum = 0
2 | >>> for i in range(50):
3 | ...     sum += i + 1
4 | ...
5 | >>> print sum
6 | 1275

```

Note that there is no explicit ending of the `for` loop. The loop is instead controlled by a special Python-feature: Code that belongs to the `for` loop will have to be indented! This also includes conditional tests, functions and classes which are explained later. All these code blocks are ended by simply stopping the indentation.

Another example: π is a number used frequently in calculus which you typically access by simply pressing a “pi-button” on the calculator. In Python programming you can feel the satisfaction of calculating it yourself:

```

1 | >>> pi = 0.0
2 | >>> for i in range(int(1e6)):
3 | ...     pi += (-1)**i * (1.0 / (i * 2 + 1))
4 | ...
5 | >>> print pi * 4
6 | 3.14159265359

```

Datatypes

There are eight basic datatypes in Python which can hold different types of data. Assigning the objects that are listed in the “Example” column to a variable name will make that variable an object of the corresponding “Datatype”.

Datatype	Name	Example	Description
Integer	int	1, 2, -1, -2, 0	integer number
Float	float	-1.0, 0.0, 1.2e2	decimal number (also scientific notation)
String	str	'a', 'b c'	alphanumeric characters inside quotes
Boolean	bool	True, False	either True or False
List	list	[2.0, 'Foo', [1, None]]	a list of any data objects
Dictionary	dict	{'a': 1, 'b': 'a'}	unordered list, often indexed by a string
Tuple	tuple	(1, 2, 3)	a list that cannot be changed (immutable)
Exception	exception	SyntaxError, ValueError	errors that might occur during execution
None	NoneType	None	nothing, really

These concepts are not needed in the beginning but you should nevertheless be aware that there are many different data types. Note also that tuples are *immutable*. Integers, floats, booleans and strings cannot be changed too. True, you can change the value of a variable that holds an integer but you cannot change the integer itself, say from 2 to 3. This issue will be explained more carefully (further down on the current page) talking about objects.

String formatting

Strings, lists and dicts are among the most common Python objects. As stated above the strings are immutable which means they cannot be modified per se. To modify a string we have to make a brand new copy. One way to edit strings dynamically is from using string formatting. In the example below we have defined a function that takes a person's name and his or her age as input and generates a greeting line out of it:

```

1 | >>> def intro(name, age):
2 | ...     return 'My name is %s, I am %d years old.' % (name, age)
3 | ...
4 | >>> intro('Bob', 20)
5 | My name is Bob, I am 20 years old.

```

Typing `%s` inside a string lets you replace `%s` with any kind of text later, whereas `%d` can be replaced with integers only and `%f` lets you insert floats. If you want a string of fixed length in the output you can specify its length (e.g. 6) as a format modifier `%6s`:

```

1 | >>> def printrow(pattern, *columns):
2 | ...     return pattern * len(columns) % columns
3 | ...
4 | >>> print printrow('%6s', 123, 456, 789)
5 |     123    456    789
6 | >>> print printrow('%-5s', 123, 456, 789)
7 | 123  456  789

```

As you can see the extra spaces are placed in front of the string. To make the string left justified use a negative number like in `%-6s`.

Objects

All variables in Python are objects. An object has a unique identifier, a type and some content. The identifier is a reference to the computer memory and will not change run-time. The type of an object is what we previously have called the *datatype*. This cannot be changed either. The example below shows how references work:

```

1 | >>> a = 1
2 | >>> b = a
3 | >>> id(a); id(b); id(1)
4 | 140645150830792

```

```

5 | 140645150830792
6 | 140645150830792
7 | >>> b = 2
8 | >>> id(a); id(b); id(2)
9 | 140645150830792
10 | 140645150830768
11 | 140645150830768

```

Here, `a` is the name of an object of type `int`. It has the value 1. `b` is another name for the same object. An assignment in Python evaluates the right hand side first and assigns the value to the name appearing on the left side. When two right hand sides evaluate to the same object we get two different names for the same thing—in this case `a` and `b`. In other words: `a` and `b` keep references to the same memory address which happens to be the reference to 1. When assigning `b = 2` a new object is created which has a new identifier referring to 2. Even though we assigned `b = a` the changing of `b` will not affect `a`. Python is not a language for doing symbolic math like you do on a piece of paper. The order of assignment does therefore matter.

The same does not happen when using lists, dictionaries and tuples. Here, assigning `a = [1]` not only creates a named object but it also makes a new memory address where the list elements are stored. Then, `b = a` makes sure `b` refers to the same address as `a`, just like in the previous example.

```

1 | >>> a = [1]
2 | >>> b = a
3 | >>> id(a); id(b)
4 | 4364804752
5 | 4364804752

```

The difference is that when you change an element inside `b` a new object is not made. Rather, the element stored in that memory location is being changed and since `b` and `a` still refer to the same address both objects are affected:

```

1 | >>> b[0] = 2
2 | >>> id(a); id(b)
3 | 4364804752
4 | 4364804752
5 | >>> a
6 | [2]

```

Whenever `a` is assigned to a new list then a new object with a new memory location is returned. In this case `a` and `b` have different references and they are therefore independent:

```

1 | >>> a = [1]
2 | >>> b = a
3 | >>> a; b
4 | [1]
5 | [1]
6 | >>> a = [2]
7 | >>> a; b
8 | [2]
9 | [1]

```

In Python, multiplying a list by `n` will repeat the list `n` times. This means that `n` objects having exactly the same reference will be returned. The feature does not cause any problems for numbers and strings but for other objects it might cause trouble:

```

1 | >>> a = [[0]] * 3
2 | >>> a
3 | [[0], [0], [0]]
4 | >>> a[0][0] = 1
5 | >>> a
6 | [[1], [1], [1]]

```

Here, `a` is assigned to a list-of-lists. The inner list is repeated three times. This means that we have a list of three lists sharing the same reference. Changing the element of say the first of the inner lists affects all the other lists as well because they all share the same memory address.

Classes

The last two sections were about objects and datatypes. Now the scope is broadened with classes. All these concepts are quite abstract and hard to grasp. In loose terms, however, you can think of an *object* as an *instance* of a named *class* which is used to represent the *data type*. An example explains the concept quite easily. Think about a rational number. It consists of two integers. How can we represent this number as a new datatype in Python? The code below is doing the job with a minimum of fuzz:

```
1  class Rational:
2      def __init__(self, m, n):
3          self.m = int(m)      # the numerator
4          self.n = int(n)      # the denominator
5
6      # Rational numbers should be stored on a reduced form. Else m and n will blow
7      # up in algebraic operations like a+b and a*b. Try to implement the algorithm
8      # yourself using e.g. the Greatest Common Divisor scheme.
9
10     def __add__(self, arg):
11         return Rational(self.m*arg.n+self.n*arg.m, self.n*arg.n)
12
13     def __mul__(self, arg):
14         return Rational(self.m*arg.m, self.n*arg.n)
15
16     def __str__(self):
17         return '%s/%s' % (self.m, self.n)
18
19     def __getattr__(self, arg):
20         errmess = "no function %s in object %s" % (arg, type(self))
21         raise AttributeError(errmess)
22         return
```

The Rational class is admittedly quite immature but it can be used to illustrate a couple of non-trivial class aspects:

```
1  >>> from rational import Rational as rational
2  >>> print rational(2, 3) + rational(3, 4)
3  17/12
4  >>> print rational(2, 3) * rational(3, 4)
5  6/12
```

From implementing a bare-bone class with 5 functions we are able to make two instances of the rational datatype and both add and multiply the numbers (without reduction). We are also able to pretty-print the answer. Not bad. However, there is even more to it:

```
1  >>> print rational(2, 3).float()
2  ...
3  AttributeError: no function float in object <class 'instance'>
```

As you can see, trying to convert the rational number to float fails because function `float()` is not known to instances of the Rational class. Rational is not a built-in class and Python gives all instances of any user-defined classes the generic name `instance` unless we explicitly tell it to do something else. The full story is lengthy and we therefore drop the theme and pick it up later in Section 6.

Loops

Running code that does the same stuff over and over again requires a *loop*. There are two kinds of loops in Python: The `for` loop and the `while` loop. The `for` loop is used to execute the same code for all the elements in a iterable object, like lists, dicts etc. The `while` loop is used to repeat the code as long as a given statement is true.

Do for iteration objects

As already said the `for` loop is used to iterate a list, dict or any other iterable object, and to execute a block of code in each iteration:

```
1 | >>> for i in [1, 2, 3]:  
2 | ...     print i  
3 | ...  
4 | 1  
5 | 2  
6 | 3
```

The list that is traversed in this case is `[1, 2, 3]`. The local variable that is assigned to each of the elements in turn is `i`. The code that is executed in each step is `print i`.

Do while a statement is True

The `while()` function is used to execute a block of code for as long as its conditional is true:

```
1 | >>> i = 0  
2 | >>> while i < 3:  
3 | ...     i += 0.9  
4 | ...     print i  
5 | ...  
6 | 0.9  
7 | 1.8  
8 | 2.7  
9 | 3.6
```

The condition is `i < 3` and the code that is executed is `i += 0.9`. The looping continues till `i` is bigger than 3.

Logic operators

The `if` statement is used to differ between something that is true or not true:

```
1 | >>> if 1 == 2:  
2 | ...     print '1 equals 2'  
3 | ... elif 1 == 3:  
4 | ...     print '1 equals 3'  
5 | ... else:  
6 | ...     print '1 not equal 2 or 3'  
7 | ...  
8 | '1 not equal 2 or 3.'
```

When extending to list comprehension in Section 1.3 and to lambda functions in Section 5.2 it can be meaningful to keep everything on one line:

```
1 | >>> '1 equals 2' if 1 == 2 else '1 equals 3' if 1 == 3 else '1 not equal 2 or 3'  
2 | '1 not equal 2 or 3'
```

Functions

Functions are used to save pieces of code. Typically this means code that you want to reuse later. Calling a function means that some dedicated code is being executed and that selected results are conveyed back to the user. Functions you meet in programming are quite similar to functions you know from math. They take zero or more arguments and return either `None` or any data object(s) of your own request. This example shows how to define and call a function:

```

1 | >>> def f(x):
2 | ...     return 3 * x + 4
3 | ...
4 | >>> f(4)
5 | 16
6 | >>> f(10)
7 | 34

```

Note the parantheses in `f(4)` and `f(10)`. Skipping these makes Python believe that `f` is a variable. Note also that `def` is a reserved word for defining a new function. You cannot use `def` as a local variable in your programs.

A function has its own *namespace* which means all the variables that are created inside it are accessible (only) until the function returns. This means you can have two different variables sharing the same name as long they are in different namespaces. Finally, we should mention that functions can take several arguments which do not need to be integers.

More about functions

You can also define default values for the variables in a function. If a variable is not specified run-time it will be set to the default value automatically:

```

1 | >>> def func(a = 'a', b = 1.0):
2 | ...     print a, b
3 | ...
4 | >>> func()
5 | a 1.0
6 | >>> func(a = 'foo')
7 | foo 1.0
8 | >>> func(a = 'foo', b = 'bar')
9 | foo bar
10 | >>> func('foo', 'bar')
11 | foo bar

```

Functions taking any number of arguments

By using default values the user can freely enter from 0 to n arguments, assuming that n is the total number of local variables in the function header. But the number of arguments is still limited to some fixed value n . We can, however, define functions without such restrictions. Adding an asterics `*` in front of the last variable collects all the additional arguments that are passed to the function into that variable. This special variable belongs to a datatype called `tuple` which is an immutable object. A tuple can be made but it cannot be changed. It must appear as the last variable in the function header:

```

1 | >>> def letter(recipient, sender, *lines):
2 | ...     print 'Dear', recipient, ',\n'
3 | ...     for line in lines:
4 | ...         print line
5 | ...     print '\nFrom', sender
6 | ...
7 | >>> letter('Foo', 'Bar', 'How are you?', 'I am fine.')
8 | Dear Foo,
9 |
10 | How are you?
11 | I am fine.
12 |
13 | From Bar

```

If you want to index the variable `lines` using strings like in a dictionary you can use `**lines` instead of `*lines`:

```

1 | >>> def letter(recipient, sender, **lines):
2 | ...     print 'Dear', recipient, ',\n'

```

```

3 ...     for line in lines:
4 ...         print line, ':', lines[line]
5 ...     print '\nFrom', sender
6 ...
7 >>> letter('Foo', 'Bar', Line1 = 'Hey', Line2 = 'Yo')
8 Dear Foo ,
9
10 Line2 : Yo
11 Line1 : Hey
12
13 From Bar

```

Note: When iterating on dicts the keywords may come in a seemingly random order! This stems from the fact that dicts are using a kind of random storage which may change run-time.

Methods

A function can take any number of arguments, calculate its results, and return something. Only the namespace of the function is affected when it is called. For objects you also want to write functions that modify the *state* of the object. These functions affect the namespace of the object per se and should therefore be called something else. They are often referred to as *methods*. To show the difference we can wrap the function `f()` inside a minimal class `obj` and refer to it as a method registered to the instance of that class:

```

1 >>> class obj:
2 ...     def __init__(self):
3 ...         self.answer = None
4 ...     def f(self, x):
5 ...         self.answer = 3 * x + 4
6 ...     return self
7 ...
8 >>> print obj().answer
9 None
10 >>> print obj().f(4).answer
11 16

```

The borderline between a *namespace* and an *object* is not very clear in Python and the distinction between `f()` and `obj().f()` suffers from this fact. The issue is straightened out more clearly in other languages like e.g. C++ and Ruby, but in Python it remains subtle and we shall therefore stick to the word *function* to avoid any further confusion.

1 Lists (m2py_list.py)

1.1 New content

Lists are in focus in this section. The task is to make a program that takes any list-of-lists and generates a regular table. By regular we mean a table that is full and rectangular. Elements that are missing are filled in using a *padding* value which is supplied by the user. As a part of the programming you will learn to use `range()`, `append()` and `max()`. These are functions built into the list type. You will also learn to use list comprehension which is a Python'ish way to generate lists. Finally, there is an important issue about assignments.

1.2 Generating arithmetic lists (ranges)

`range()` returns a list of integers. It takes up to 3 integer arguments. Given one argument it returns a list of numbers from zero and up to, but not including, the given argument:

```
1 |>>> range(4)
2 |[0, 1, 2, 3]
```

Given two arguments it returns a list of all integers from the first argument and up to, but not including, the second argument:

```
1 |>>> range(4, 8)
2 |[4, 5, 6, 7]
```

Given three arguments it returns a similar list using the third argument as the increment:

```
1 |>>> range(4, 11, 2)
2 |[4, 6, 8, 10]
```

The list function `append()` adds a single value to the end of the list:

```
1 |>>> a = [1, 2, 3]
2 |>>> a.append(4)
3 |>>> a
4 |[1, 2, 3, 4]
```

Note that `a.append(n)` is the same as typing `a += [n]`, but it is not the same as `a = a + [n]`. The last call is an assignment, and an assignment results in a new object with a new reference to a new memory address. This means that `a` will be *copied by value*:

```
1 |>>> a = [1, 2]
2 |>>> b = a
3 |>>> a.append(3)
4 |>>> b
5 |[1, 2, 3]
6 |>>> a += [4]
7 |>>> b
8 |[1, 2, 3, 4]
9 |>>> a = a+[5]
10 |>>> b
11 |[1, 2, 3, 4]
12 |>>> a
13 |[1, 2, 3, 4, 5]
```

Line 1 to 5 show that `a` and `b` refer to the same object.

Line 6 to 8 the same happen here.

Line 9 to 13 object `a` has gotten a new reference and is therefore independent of `b`.

1.3 Generating arbitrary lists

List comprehension is a nifty way of creating lists. It uses a shorthand `for` loop to generate each element in the list. The value of the element is calculated by a one-liner (code written entirely in one line). The syntax is: [`<code> for <element> in <list>`]. The `<code>` is doing the calculations and `<element>` is a local variable accessing the elements of `<list>`. The square brackets indicate that you are working on a list. Here are some examples:

```
1 | >>> [i for i in range(1, 11)]
2 | [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
3 | >>> [i*2 for i in range(1, 11)]
4 | [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
5 | >>> [[i, j**2] for i in range(1, 4) for j in range(1, 3)]
6 | [[1, 1], [1, 4], [2, 1], [2, 4], [3, 1], [3, 4]]
```

The same syntax is used for making dictionaries in Section 4. The brackets are then replaced by curly braces:

```
1 | >>> {c: c for c in ['a', 'b', 'c']}
2 | {'a': 'a', 'c': 'c', 'b': 'b'}
```

1.4 The biggest element in a list

`max()` returns the list element that is considered to have the biggest value. For numbers, the largest value is the biggest one. If the list is nested then the maximum value is calculated according to the first element in the lists, then to the second element and so on. For dictionaries, the tuple having the biggest keyword will be returned—the values are decisive only if the keywords are equal. Strings are sorted in lexicographic order:

```
1 | >>> max([1, 5, 2])
2 | 5
3 | >>> max([[0, 1], [2]])
4 | [2]
5 | >>> max(['a', 'b', 'ca', 'd'])
6 | 'd'
7 | >>> max([{‘b’: ‘d’}, {‘b’: ‘eb’}])
8 | {‘b’: ‘eb’}
```

When the list contains several different datatypes strings are considered the highest, next up are lists, then dicts, and at last numbers.

1.5 The code

The code below converts a non-regular table into a regular matrix. The empty cells in the matrix are filled with a “padding” value. Matrices are interpreted as lists-of-lists in this context:

```
1 | def m2py(rows, padding=0, debug=False):
2 |     maxlen = max([len(row) for row in rows])
3 |
4 |     if debug:
5 |         print 'm2py_list: Max length: %s' % maxlen
6 |
7 |     return [row + [padding] * (maxlen-len(row)) for row in rows]
```

Line 1 defines a function called `m2py()`. It takes 1, 2 or 3 arguments: `rows`, `padding` and `debug`. If `padding` is not given it will be assigned the value 0 by default. Likewise, if `debug` is missing it is assigned the value `False`.

Line 2 traverses each row in the input table called `rows`, using the `len()` function to get hold of the length of that row. Then it uses `max()` to calculate the maximum length of all the rows.

Line 7 traverses all the rows and adds one `padding` value for each of the missing elements.

1.6 Run by itself

The last part of file `m2py_list.py` is executed only when Python runs the code by itself. It means you enter `python <file>.py` directly in the terminal. The code is skipped when the file is loaded with `import <file>` inside another program. This is the Python-way of writing stand-alone unittests:

```
9  | if __name__ == '__main__':
10 |     print m2py([], [', ', ', '], None)
11 |     print m2py([[1, 2], [4, 5, 6], []], 3)
```

The code runs `m2py()` and displays the output. The padding value is `None` and `3` respectively:

```
1  | [[None, None], [', ', ',']]
2  | [[1, 2, 3], [4, 5, 6], [3, 3, 3]]
```

This feature makes it a snap to write small tests showing how the program works.

2 Exceptions

2.1 New content

Exceptions allow the program to exit in a controlled manner when errors occur. Rather than printing local messages from deep inside the code, `python` takes care of the entire operation and lets you simply `raise` an error datatype with a message attached. Most modern languages have this feature, but it is still common to see local error messages printed from all over the place. This should now be considered bad practise.

2.2 Exception handling

The keyword `try` executes a piece of code that can produce an error without crashing the program. The code that comes after `try` is used to describe what you intend to do, the code that comes after `except` is what should be done if an error occurs:

```
1 | >>> a = []
2 | >>> for i in range(-10,10):
3 | ...     a.append(i*abs(i)/i)
4 | ...
5 | Traceback (most recent call last):
6 |   File <stdin>", line 2, in <module>
7 | ZeroDivisionError: integer division or modulo by zero
```

In this case the error is caused by a division-by-zero inside the loop. The error is captured like this:

```
1 | >>> a = []
2 | >>> for i in range(-10, 10):
3 | ...     try:
4 | ...         a.append(i*abs(i)/i)
5 | ...     except ZeroDivisionError:
6 | ...         a.append(i)
7 | ...
8 | >>> a
9 | [10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

You cannot use the `try` keyword alone—it must be followed by something. But, the keyword `except` may affect the error traceback. Use `finally` to prevent this. The code after `finally` is what you will do in the end, no matter if a bug has occured or not:

```
1 | >>> def tryfloat(var):
2 | ...     try:
3 | ...         var = float(var)
4 | ...     finally:
5 | ...         return var
6 | ...
7 | >>> tryfloat('1e3')
8 | 1000.0
9 | >>> tryfloat('1e3e')
10 | '1e3e'
```

The last of the keywords is `else`. The `else` block is executed only if the `try` block did not fail. This can be helpfull when two actions give the same type of error:

```
1 | >>> from math import sqrt
2 | >>> def trysqrt(var):
3 | ...     try:
4 | ...         var = float(var)
5 | ...     except ValueError:
6 | ...         var = float('nan')
7 | ...     else:
8 | ...         try:
9 | ...             var = sqrt(var)
10 | ...         except ValueError:
```

```
11 ...         var = str(sqrt(abs(var))) + 'i'
12 ...     finally:
13 ...         return var
14 ...
15 >>> trysqrt('4')
16 2.0
17 >>> trysqrt('-4')
18 '2.0i'
19 >>> trysqrt('-4d')
20 nan
```

There are no limits on how to break the program. You can even raise an exception by yourself. This is useful for giving intelligent error messages connected to semantic faults in your own code. The feature blends nicely with the handling of syntax errors, divide-by-zero traps and other run-time errors:

```
1 >>> raise SyntaxError('Custom error message')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 SyntaxError: Custom error message
```

You can choose any error type you want but it is advisable to pick one that describes the error situation adequately.

3 Regular Expressions (`m2py_regex.py`)

3.1 New content

Strings and regular expressions are in focus in this section. Basically, we shall redo the programming in Section 1 but with a more flexible understanding of what a list element is. You will need to learn how to use regular expressions, how to split strings using other strings, and even to split strings using regular expressions. You will also learn more about functions.

3.2 Regular expressions

A regular expression, or simply a *regex*, is used to recognize string patterns. Let's say you are writing a text with a lot of numbers. You want to make sure that all these numbers have a comma after each third digit (e.g. 1000000 should be 1,000,000). In this case we can use a regex to check the spelling and even to correct the written text automagically (not covered here):

```
1 |  >>> import re
2 |  >>> def spellchecker(text):
3 |  ...     badspelling = re.match('.*(\d{4,}).*', text)
4 |  ...     if badspelling:
5 |  ...         return '%s is not formatted correctly.' % badspelling.group(1)
6 |  ...     else:
7 |  ...         return 'OK'
8 |
9 |  >>> spellchecker('One "grand" equals 100 dollars.')
10 | 'OK'
11 | >>> spellchecker('No, one grand is 1000 dollars.')
12 | '1000 is not formatted correctly.'
13 | >>> spellchecker('All right, it is 1,000 dollars.')
14 | 'OK'
```

So, what happened? Obviously, the `spellchecker()` function picks up the given text as its first and only argument. Inside that function `badspelling` is assigned to whatever value is returned from the `re.match()` function. This will be a *match object* if the string matches the pattern (explained below), and `None` if it does not match. The rest of the program is straightforward.

Regular expressions are among the simplest of all computer languages. They are known as *context free* languages using character patterns where each symbol has a definite meaning. The most basic pattern symbols are the ones explained below:

Pattern	Meaning
<code>^</code>	start of string or negated range operator <code>[^]</code>
<code>\$</code>	end of string or end of line in multiline mode
<code>.</code>	any character except newline
<code>\d</code>	any digit in the range [0-9]
<code>\s</code>	any white-space character
<code>\w</code>	any character in the range [a-zA-Z_]
<code>[a-w]</code>	any character in the range <code>a</code> to <code>w</code>
<code>[^a-w]</code>	any character not in the range <code>a</code> to <code>w</code>
<code>*</code>	0 or more of the previous token, range or group
<code>+</code>	1 or more of the previous token, range or group
<code>?</code>	0 or 1 of the previous token, range or group
<code>{n,m}</code>	<code>n</code> to <code>m</code> of the previous token, range or group
<code>(a b)</code>	either <code>a</code> or <code>b</code> stored in a group
<code>(ab)</code>	<code>a</code> followed by <code>b</code> stored in a group
<code>(?:ab)</code>	same as above but no group is formed

In the table there is a term called *group* which needs explanation. Take the group `(ab)` as an example. That group is made available to a function called `<matchobject>.group(1)` if and only if the regex matches the string. If the string is not matched then the return value of `<matchobject>.group(1)` is `None`. In this way we dissect the string *after* it has been created. Such operations are called *introspection* in the computer science lingo. `(a|b)` is doing the same thing for a string which is either `a` or `b`. `(?:ab)` matches again the token `ab` but without making any group. If there are no parentheses then the match object is empty but still not `None`.

The match object have two useful functions: `group()` and `groups()`. `group(1)` returns whatever matched the first `()` in the pattern, `group(2)` returns the second `()`, etc. `group(0)` returns everything that matched the pattern even when it is not inside parentheses, while `groups()` returns a tuple of all the matched groups:

```

1 |>>> object = re.match(r'([N-Z]|[a-m])\d(?:[A-M]|[n-z])(\d)', 'R2D3 is awesome')
2 |>>> object.groups()
3 |('R', '3')
4 |>>> object.group(0)
5 |'R2D3'
6 |>>> object.group(1)
7 |'R'
8 |>>> object.group(2)
9 |'3'
```

This pattern matches any character from `N` to `Z` or from `a` to `m` putting the result into `group(1)`. Then it continues matching any digit `\d`. Then comes any character, this time from `A`-`M` or from `n`-`z`, but without making a new group. Last comes any digit `\d` which goes into `group(2)`.

3.3 Splitting a string

The function `split()` is built-in for strings. It takes a single argument and splits the string into a list of smaller strings everywhere the argument is matching the string. The matching part of the string is consumed in the process. The return value is a list with one element (the string itself) if the argument matches the string nowhere:

```

1 |>>> 'R2D2 is awesome'.split('2')
2 |['R', 'D', ' is awesome']
3 |>>> 'abc'.split('r')
4 |['abc']
```

The `re` module does also provide a similar `split()` function. One major difference between the two functions is that `re.split()` takes the string as input. In addition `re.split()` does of course support regular expressions:

```

1 |>>> re.split(r'\s*[,;]? \s*', 'a, b ; c d')
2 |['a', 'b', 'c', 'd']
3 |>>>
```

3.4 The code

The program takes a string that is formatted like a Matlab matrix `'[1 2 3; 4 5 6]'` and parses it into a Python list-of-lists which we can think of as a matrix: `[[1, 2, 3], [4, 5, 6]]`. All numbers in the Matlab string are explicitly converted to float. The reason for this choice is that Python does integer division whenever possible which sooner or later will cause problems if the matrix is being factored or inverted. The “number” is returned as a string if it cannot possibly be converted to a float. The function `m2py.list.m2py()` is used to fill in missing elements if the input does not conform with a regular matrix:

```

1 |def m2py(matstring, padding = 0.0, debug = False):
2 |    if isinstance(padding, int):
3 |        padding = float(padding)
```

```

4
5     import re
6     import m2py_list
7
8     try:
9         tokens = re.match(r'^\s*\[\s*(.*\s*)\s*\]\s*$', matstring).group(1)
10    except TypeError:
11        raise TypeError('Input must be %s, was %s' % (type(''), type(matstring)))
12    except AttributeError:
13        raise SyntaxError('Input format is: [obj_11, obj_12, ...; obj_21, ...; ...]')
14
15    if tokens == '':
16        if debug:
17            print 'm2py_regex: No tokens given - empty matrix returned.'
18        return []
19
20    output = []

```

Line 2 to 3 Because we convert the values of the input list-of-lists to float, we should do the same with the padding value too. But, if we do that straight ahead without testing first we will deprive the user the possibility of using a (numerical) string as the padding value, because it will be floated automatically.

Line 8 to 13 Matlab allows white-spacing before and after the square brackets. These spaces are filtered out together with the brackets. The regular expression matches any number of whitespaces followed by a left square bracket and any number of whitespaces. Then, it matches everything, followed by any number of whitespaces, a right square bracket and any whitespaces again. The \$ means end-of-line. This tells the string must be ending after the whitespaces. Note that “everything” is the only part of the pattern that is grouped in the match object.

There are two things that can go wrong on line 7. A `TypeError` is raised if `matstring` is not of type `str`. The other issue is that we call `group()` on the returned match object. If `matstring` does not match the input pattern then `None` is being returned. But, `None` has no `group()` function! So, calling this function blindly will cause an `AttributeError`.

Line 15 to 18 If there’s nothing inside the brackets of the input string then `tokens` will be an empty string. Later, when we split `tokens` using row and column separators it returns a list-of-lists containing an empty string. Realising that this string is empty it is replaced by the padding value. However, what we expect from an empty input is an empty output matrix. To rectify this situation we test if `tokens` is empty and returns an empty list-of-lists if that is the case.

```

22     for row in re.split(r'\s*;\s*', tokens):
23         output.append([])
24
25     for col in re.split(r'\s*,?\s*', row):
26         try:
27             col = float(col)
28         except ValueError:
29             if col == '':
30                 if debug:
31                     print(\n
32                         'm2py_regex: [%s][%s]: Cell is empty, inserted %s value.'\
33                         ) % (len(output), len(output[-1]), padding)
34             col = padding
35         else:
36             if debug:
37                 print(\n
38                     'm2py_regex: [%s][%s]: \
39                         "%s" cannot be converted to float, inserted string value.'\
40                         ) % (len(output), len(output[-1]), col)
41             output[-1].append(col)
42
43     # The next line can be the source of much grief. Try indenting one extra level

```

```
44 | # and see what happens!
```

Line 22 to 23 The easiest way to turn the string into a matrix is using the `re.split()` function. We split `tokens` at semicolons, possibly embedded in whitespaces, and iterate on all the sub-strings. For each sub-string we add a new row to the output matrix.

Line 25 to 41 The sub-string is thereafter split at whitespaces, or commas, and iterated on the subsub-string level. But, appending `col` to the last row in the matrix would yield a matrix with strings—not numbers. We therefore make use of `float(col)` which either returns the float value of `col` or it raises an `ValueError`. The error is easily checked using a `try` block where `col` is assigned to the float value of itself. If the conversion fails the string value is retained in the matrix.

Note: Appending `col` to the matrix on line 41 makes a matrix with mixed floats and strings. The problem is that e.g. `'[1, , 2]'` returns `[[1, '', 2]]` although the empty string was supposed to be replaced by padding. This is why `col` must be tested to see if it is empty before it is appended to the matrix. Finally, `col` is appended to the matrix and the iterations are complete. We now have a matrix-like object that may still be irregular. The final call to `m2py_list.m2py()` makes a regular matrix for us.

3.5 Run by itself

The last part of the file is executed only when Python runs the code by itself. It means you enter `python <file>.py` directly in the terminal. The code is skipped when the file is loaded with `import <file>` inside another program. This is the Python-way of writing stand-alone unitests:

```
45 |     return m2py_list.m2py(output, padding, debug)
46 |
47 |     if __name__ == '__main__':
48 |         print m2py('[ ; , ]', None)
49 |         print m2py('[1, 2; 4, 5, 6; ]', 3)
```

The code runs `m2py()` and displays the output. The padding value is `None` and `3` respectively:

```
1 | [[None, None], [None, None]]
2 | [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [3.0, 3.0, 3.0]]
```

The `m2py_regex.m2py()` function has the same scope as `m2py_list.m2py()` in Section 1, but since the parser rules are different there will be semantic differences. The second line in the output is apparently the same as before except that elements are now converted to float. The first line, however, is different. In `m2py_list.m2py()` we could supply empty strings as list elements, but in `m2py_regex.m2py()` all the whitespace is first gobbled up by the parser before it is replaced by the padding value (object `None` in this case).

4 Dictionaries (`m2py_dict.py`)

4.1 New content

So far we have been using lists for storing many values in one place. In this section we will introduce sets and dictionaries. The scope is the same as in Sections 1 and 3, but rather than storing the incoming tokens as either float or strings we shall go one step further and anticipate a certain *structure* of the string input. The anticipated structure is that of a chemical formula. In a way, we are going to train the computer to interpret human readable strings. This process is called *parsing*. The regular expression and the dictionary are ideal companions for this purpose.

4.2 The set datatype

The set is an unordered list. You can make an empty set by typing `set()`. It can hold any number of values just like a list, but two values cannot be equal. New elements are added to the set using `add()`. The argument given to `add()` will be added to the set unless it already exists.

4.3 The dict datatype

While lists can only be indexed by numbers and sets cannot be indexed at all, a dict can be indexed by strings as well as numbers (or any other object that is capable of producing a descent *hash* value). Another characteristic is that the dict is not ordered in any way. It behaves more like a random storage database. You have a key and look up a value. Iterating on a dict returns (key, value) pairs in a seemingly random order, but for small dicts the ordering is what you'd expect:

```
1 |>>> d = {}           # This creates an empty dict.
2 |>>> d['a'] = 3       # Assigns a value for the a-element
3 |>>> d['b'] = [1, 2]  # Assings a value for the b-element
4 |>>> d
5 |{'a': 3, 'b': [1, 2]}
```

4.4 Default values for dictionaries

The `get()` function takes 2 arguments, the first is the keyword that you'd like to fetch the value of, the second is what should be returned if there is no value registered for the given keyword:

```
1 |>>> a = {'a': 1}
2 |>>> a.get('a', 0)
3 |1
4 |>>> a.get('b', 0)
5 |0
```

4.5 The code

The input to the program is a string of chemical formulas separated by semicolons like in '[H2O; NH4]'. The substring inside the brackets is split on semicolon and the occurrences of each atom is counted for each of the formulas. The return value is a list-of-lists where each row is synonymous to a formula. There will be exactly one column for each atom that is found. Each of the elements in the list-of-lists holds the number of atoms in the corresponding formula. This is the celebrated *formula matrix* known from chemical reaction theory:

```
1 |def m2py(matstring, debug=False):
2 |    import re
3 |    from atoms import atoms as atoms_parser
```

```

5     try:
6         tokens = re.match(r'^\s*\[\s*(.*\s*)\s*\]\s*$', matstring).group(1)
7     except TypeError:
8         raise TypeError('Input must be %s, was %s' % (type(''), type(matstring)))
9     except AttributeError:
10        raise SyntaxError('Input format is: [formula_1; formula_2; ...].')
11
12    if tokens == '':
13        if debug:
14            print 'm2py_dict: No tokens given - empty matrix returned.'
15        return []
16
17    dictionaries = [] # list of molecular formula dicts
18    formulas = re.split(r'\s*;\s*', tokens) # list of chemical formulas
19    atoms = set() # set of atomic symbols
20
21    for formula in formulas:
22        try:
23            dictionaries.append(atoms_parser(formula)[0])
24            atoms.update(dictionaries[-1])
25        except:
26            raise SyntaxError('Cannot parse formula "%s".' % formula)
27
28    atoms = list(atoms)
29    atoms.sort()
30
31    return [[dictionary.get(atom, 0) for atom in atoms] for dictionary in dictionaries], \
32           atoms, \
33           formulas

```

Line 5 to 15 are the same as in the previous section. The required information is extracted from the input string and a proper error message is raised if something went wrong.

Line 17 to 29 The `tokens` variable is split into several formulas which are interpreted by the `atoms_parser()` one by one. If successful a list with one single dict is returned. The dict has keywords for each of the atoms in the formula. The dict values are the total number of atoms of each kind. Since `atoms_parser()` returns a list of dictionaries we use `atoms_parser()[0]` to access the first (and only) dict. Function `update()` on line 24 adds the keywords from each dictionary to the `atoms` set variable. Finally, `atoms` is turned into a list object and sorted.

Line 31 It is necessary to loop over all the atoms in each of the dictionaries (i.e. molecular formulas) in order to build the output object. Note that `get(atom, 0)` returns the number of each `atom` in the formula, or zero if the atom is missing from the formula. The list comprehension will eventually return a list-of-lists, which is the output from the program. In addition `atoms` and `formulas` are also returned because they will find use later on.

4.6 Run by itself

The last part of the file is executed only when Python runs the code by itself. It means you enter `python <file>.py` directly in the terminal. The code is skipped when the file is loaded with `import <file>` inside another program. This is the Python-way of writing stand-alone unitests:

```

35  if __name__ == '__main__':
36      print m2py([], True)[0]
37      print m2py(['CH3OH; HNO3'])[0]
38      print m2py(['H2O; CH3(CH2)2NO3'])[0]

```

Note the number `[0]`. It is an index, not a padding value. Zeros are padded automatically. The empty list argument in line 36 causes a warning message:

```

1 | m2py_dict: No tokens given - empty matrix returned.
2 | []
3 | [[1, 4, 0, 1], [0, 1, 1, 3]]
4 | [[0, 2, 0, 1], [3, 7, 1, 3]]

```

5 Lambda Functions (`m2py_lambda.py`)

5.1 New content

All the functions we have seen so far have their own namespaces declared by `def`. In this section we will meet lambda functions that lack separate namespaces. They are anonymous functions (given no readable names) operating in the local namespace. This is hardly worth a notice before we suddenly realise that it opens up a whole new world of programmatic possibilities. The point being that functions in Python are first-class variables. You can assign a function definition to a variable and send it to another function which then becomes a *functor*, something akin to a *functional* in mathematics.

Our scope is the same as in the last section. What we achieve from using lambda functions is that the regular expressions inside `m2py()` go away. This has a tremendous effect on the program design because we can change the parser rules from the outside of the function without changing the function definition itself. Think code maintenance and you'll understand why this is a great idea.

5.2 Anonymous function calls

The lambda function uses the reserved word `lambda` to create a function definition that is assigned to a variable in the *local* namespace. It takes any number of arguments and processes them all in a single line of code called a *one-liner*:

```
1 |>>> f = lambda x, y: 2 * x + y
```

The lambda function `f` defined above does essentially the same as this:

```
1 |>>> def f(x, y):  
2 |...     return 2 * x + y  
3 |...
```

The one-liner syntax is similar to normal Python syntax. It supports e.g. list comprehension, logical operators, and so on:

```
1 |>>> evenrange = lambda x: [i for i in range(x) if i%2 == 0]  
2 |>>> evenrange(10)  
3 |[0, 2, 4, 6, 8]  
4 |>>> import math  
5 |>>> safesqrt = lambda x: math.sqrt(x) if x > 0 else 0 if x == 0 else '%si' % math.sqrt(abs(x))  
6 |>>> safesqrt(100)  
7 |10  
8 |>>> safesqrt(-9)  
9 |'3.0i'
```

The first of the two examples above is but a silly example on how to make a lambda function. It does *not* illustrate how to make ranges. That is more elegantly done this way:

```
1 |>>> evenrange = lambda x: range(0, x, 2)  
2 |>>> evenrange(10)  
3 |[0, 2, 4, 6, 8]
```

5.3 The code

The functions are now passed as arguments instead of being hard-coded into the file. The variable `matstring` is still a string, but `getstr()`, `split()` and `parse()` are lambda functions defined by the user:

```
1 | def m2py(matstring, getstr, split, parse, debug=False):  
2 |     try:  
3 |         tokens = getstr(matstring)
```

```

4     except TypeError:
5         raise TypeError('Input must be %s, was %s' % (type(''), type(matstring)))
6     except AttributeError:
7         raise SyntaxError('Input format is: [formula_1; formula_2; ...].')
8
9     if tokens == '':
10        if debug:
11            print 'm2py_lambda: No tokens given - empty matrix returned.'
12        return []
13
14     dictionaries = []                      # list of molecular formula dicts
15     formulas = split(tokens)               # list of chemical formulas
16     atoms = set()                         # set of atomic symbols
17
18     for formula in formulas:
19         try:
20             dictionaries.append(parse(formula))
21             atoms.update(dictionaries[-1])
22         except:
23             raise SyntaxError('Cannot parse formula "%s".' % formula)
24
25     atoms = list(atoms)
26     atoms.sort()
27
28     return [[dictionary.get(atom, 0) for atom in atoms] for dictionary in dictionaries], \
29             atoms,
30             formulas

```

The code looks almost the same as in the previous section. The few exceptions are in located line 3, 15 and 20, where explicit code has been replaced with lambda functions.

5.4 Run by itself

Running the program will require that you define the lambda functions first. It is a good idea to do this inside the main module to keep everything that is associated with the program at one place. Here, it shows by example how the lambda functions should be defined to make the new `m2py()` function work:

```

32 | if __name__ == "__main__":
33 |     import re
34 |     from atoms import atoms as atoms_parser
35 |
36 |     getstr = lambda x: re.match(r'^\s*\[\s*(.*\s*)\s*\]\s*$', x).group(1)
37 |     parse = lambda x: atoms_parser(x)[0]
38 |     split1 = lambda x: re.split(r'\s*,?\s*', x)
39 |     split2 = lambda x: re.split(r'\s*;?\s*', x)
40 |
41 |     print m2py('[H2O, CH3(CH2)2NO3]', getstr, split1, parse)[0]
42 |     print m2py('[CH3OH; HNO3]', getstr, split2, parse)[0]

```

Line 33 to 34 Regular expression module and molecular formula parser loaded from the outside of the `m2py()` function.

Line 36 to 39 Lambda functions for removing square brackets, for calculating the formula matrix, and for splitting the string on comma or semicolon (notice the small difference between `split1` and `split2`).

The output should look like this:

```

1 | [[1, 4, 0, 1], [0, 1, 1, 3]]
2 | [[0, 2, 0, 1], [3, 7, 1, 3]]

```

6 Classes (`m2py_class.py`)

6.1 New content

Python has eight basic datatypes. Each of these datatypes are uniquely connected to a programmatic entity called a class. Time has now come to learn more about classes and to see how the class concept can be used to make user defined datatypes. You will also learn how to customize functions that reside inside the namespace of the class. The outcome of this section is a class that makes the pretty-printing of a composite table quite easy.

6.2 Creating a new class

Creating a new class is similar to creating a new function. A class has its own namespace including functions that are callable only on the objects of that class. The class definition also has an optional argument known as the parent class. Everything that is known to the parent is known to the sibling but not vice versa: Function definitions that are overridden by the sibling will not be known to the parent. This mechanism is called *inheritance*. The idea is very simple but in the 1960's when Kristen Nygaard and Ole Johan Dahl pioneered the concept under the umbrella of the SIMULA language they were way ahead the rest of the world:

```
1 | >>> class matrix(list):
2 | ...     def __init__(self, arg):
3 | ...         super(matrix, self).__init__(arg)
4 | ...     def tr(self):
5 | ...         return matrix([[self[j][i] \
6 | ...                         for j in range(len(self))]] for i in range(len(self[0]))))
7 | ...
```

To repeat ourselves: The `class matrix(list)` declaration makes a new class `matrix` that inherits from the existing `list` datatype. All attributes known to `list` will automatically be available to `matrix`. You can spot the attributes of the two classes by issuing the commands `dir(list)` and `dir(matrix)`. The function `__init__()` inside the class scope is special. Actually, all functions in Python that start and end with two underscores are special. In particular, the `__init__()` function is called every time a new object is created from its class definition. In this case by issuing the command `matrix()`. This operation is known as a *constructor* call in object oriented programming. The `self` argument enables the `__init__()` function with a reference to the object itself. This must always be the first (mandatory) argument. The other function `tr()` is registered to all objects created from class `matrix`. It also takes `self` as its first argument (you never actually do this—it is supplied automatically by Python) and returns the transposed of itself:

```
1 | >>> m = matrix([[1, 2], [3, 4]])
2 | >>> m.tr()
3 | [[1, 3], [2, 4]]
```

6.3 The code

The program makes use of `m2py_lambda.m2py()` to parse a string of molecular formulas and to calculate the corresponding formula matrix. The goal is to decorate the output matrix with a new row and column vectors showing the atomic symbols and the molecular formulas. We must then modify the `__str__()` function in class `Table` to make it display its objects as formatted tables rather than standard list-of-lists. The starting point is a 4-dimensional list-of-lists-of-lists-of-lists which can be visualized below (the chemistry is the same as in Section 6.4):

```
[ [ [ ['] ],
  [[‘C’, ‘Cl’, ‘H’, ‘N’, ‘Na’, ‘O’]] ],
[ [‘H2O’],
  [‘CH3(CH2)2NO3’],
  [‘NaCl’]],
[[0, 0, 2, 0, 0, 1],
 [3, 0, 7, 1, 0, 3],
 [0, 1, 0, 0, 1, 0]] ]
```

The elements are a mix of strings and integers but they could be of any kind as long as they have a descent string representation. We have now come close to the real problem: By stacking the horizontal lists as rows in the output table, and removing 2 levels of innermost brackets, it is possible to produce something of this kind:

```
[ [      ‘’, ‘C’, ‘Cl’, ‘H’, ‘N’, ‘Na’, ‘O’],
  [      ‘H2O’, 0, 0, 2, 0, 0, 1 ],
  [‘CH3(CH2)2NO3’, 3, 0, 7, 1, 0, 3 ],
  [      ‘NaCl’, 0, 1, 0, 0, 1, 0 ] ]
```

Our task is to encode the operations required into the function `__str__()` of class `Table`. In programming it is often the case that there are more ways to Rome than there are travellers and this problem poses no exception to the rule. The problem presented here is in fact quite open and the code given below must not be taken to be the ultimate answer (the comments do by the way refer to the old example from Section 6.4):

```
1  class Table(list):
2      def __init__(self, *arg):
3          super(Table, self).__init__(*arg)
4
5      def __str__(self):           # e.g. self = [[t11, t12], [t21, t22]]
6          tmp = []
7          for row in self:        # -- [t21, t22]
8              for i in range(len(row[0])): # -- i = 2
9                  tmp.append([])
10                 for j in range(len(row)): # -- j = 1
11                     for rji in row[j][i]: # -- rji = t22[i=2][k=0] = 3
12                         tmp[-1].append(str(rji)) #
13
14      mcw = [max([len(tmp[i][j]) for i in range(len(tmp))]) \
15             for j in range(len(tmp[0]))]
16
17      return "\n".join([
18          ''.join(['%' * s % (mcw[j], tmp[i][j]) for j in range(len(tmp[0]))]) \
19             for i in range(len(tmp))])
```

Line 3 The constructor of class `Table` makes an explicit call to the constructor of the parent class. This means that all the details of the `Table` class will be the same as for the `list` datatype. The only difference between the two class descriptions lies in the `__str__()` method which is explicitly redefined by us.

Line 5 to 12 The 4-dimensional list-of-lists-of-lists-of-lists is reduced to a 2-dimensional list-of-lists. The code is far from perfect. There is no error checking and no clever code tricks.

Line 14 Calculate the (maximum) column width for each column in the table. This information is required in the output to make the elements properly aligned.

Line 17 The column elements are joined in the inner loop and the rows in the outer loop.

6.4 Run by itself

The last part of the file is executed only when Python runs the code by itself. It means you enter `python <file>.py` directly in the terminal. The code is skipped when the file is loaded with `import <file>` inside another program. This is the Python-way of writing stand-alone unitests:

```
21 | if __name__ == "__main__":
22 |
23 |     matstring = '[H2O, CH3(CH2)2NO3, NaCl] '
24 |
25 |     import re
26 |     from atoms import atoms as atoms_parser
27 |     import m2py_lambda
28 |
29 |     getstr = lambda x: re.match(r'^\s*\[\s*(.*\s*)\s*\]\s*\$', x).group(1)
30 |     parse = lambda x: atoms_parser(x)[0]
31 |     split = lambda x: re.split(r'\s*,?\s*', x)
32 |
33 |     atoms_lol, atoms, formulas = m2py_lambda.m2py(matstring, getstr, split, parse)
34 |
35 |     t11 = [[',']]
36 |     t12 = [atoms]
37 |     t21 = [[f] for f in formulas]
38 |     t22 = atoms_lol
39 |
40 |     tab = Table([[t11, t12], [t21, t22]])
41 |
42 |     print type(tab)
43 |     print tab
```

Line 23 to 31 is essentially the same as in Section 5.4.

Line 33 The formula parser returns a list of atomic symbols and a list of chemical formulas in addition to the formula matrix.

Line 35 to 38 Making list-of-lists to populate the 4-dimensional table.

Line 40 Objects of class `Table` can be printed in the same manner as standard Python objects.

The output should look like this:

```
1 | $ python m2py_class.py
2 | <class '__main__.Table'>
3 |          C  Cl  H  N  Na  O
4 |      H2O  0   0   2   0   0   1
5 |  CH3(CH2)2NO3  3   0   7   1   0   3
6 |      NaCl  0   1   0   0   1   0
7 | $
```

Line 2 Objects of class `Table` have their own datatype registered in the `__main__` namespace.

7 Unittests

In the Sections 1.6, 4.6, 5.4 and 6.4 we used the run-by-itself idiom to test the programs we've written up to that point—one by one. This is good for small projects with a couple of modules but it is not adequate for bigger programs having many classes spread over several modules. To handle large projects it has over the years crystallized another idiom called *unittesting*. The current doctrine is to work in small cycles: coding + testing + coding + ⋯ + testing and let the `unittest` module decide whether the code is error-free or not. In this way the programmer can concentrate solely on the coding. A minimal example on how the `m2py_xxx` modules might be tested is shown below:

```
1 import unittest
2
3 class TestAll(unittest.TestCase):
4     '''Local namespace for testing m2py_xxx modules.'''
5
6     def setUp(self):
7         pass
8
9     def test_m2py_class(self):
10        import re
11        from atoms import atoms as atoms_parser
12        import m2py_lambda
13        import m2py_class
14
15        matstring = '[H2O, CH3(CH2)2NO3, NaCl]'
16
17        getstr = lambda x: re.match(r'^\s*\[\s*(.*\s*)\s*\]\s*\$', x).group(1)
18        split = lambda x: re.split(r'\s*,?\s*', x)
19        parse = lambda x: atoms_parser(x)[0]
20
21        atoms_lol, atoms, formulas = m2py_lambda.m2py(matstring, getstr, split, parse)
22
23        t11 = [[',']]
24        t12 = [atoms]
25        t21 = [[f] for f in formulas]
26        t22 = atoms_lol
27
28        tab = m2py_class.Table([[t11, t12], [t21, t22]])
29
30        ans = "          C  Cl  H  N  Na  O\n" \
31              "      H2O  0  0  2  0  0  1\n" \
32              "CH3(CH2)2NO3  3  0  7  1  0  3\n" \
33              "      NaCl  0  1  0  0  1  0"
34
35        self.assertEqual(str(tab), ans)
36
37 if __name__ == '__main__':
38     unittest.main()
```

Function `assertEquals()` on line 35 compares the calculated table with the correct answer and makes sure the user is notified if the comparison fails. Note that equality is by no means the only option—there are several other assert-functions. In this case everything is fine:

```
1 $ python test_m2py.py
2 .
3 -----
4 Ran 1 test in 0.006s
5
6 OK
```

5.1.2 Verbatim: “script”

```
1      """
2      @summary: Integration along the periphery of a circle using explicit
3              Euler integration. The circle is defined by the formula:
4
5               $x^{**2} + y^{**2} = r^{**2}$ 
6
7              where the radius "r" is constant. Differentiation yields:
8
9               $2*x*dx + 2*y*dy = 0$ 
10
11             or:
12
13              $x*dx + y*dy = 0$ 
14
15             that is:
16
17              $[x, y] * \begin{vmatrix} dx \\ dy \end{vmatrix} = 0$ 
18
19
20             which means the differentials dx and dy are in the null
21             space of [x, y]. In this particular case we are able to
22             solve the null space by inspection:
23
24             \begin{vmatrix} dx \\ dy \end{vmatrix} \propto \begin{vmatrix} -y \\ x \end{vmatrix}
25
26
27             Hence, we are going to integrate [x, y] in the direction
28             of [-y, x]. That's all.
29
30
31     @author: Tore Haug-Warberg
32     @since: 23 Aug 2014 (started)
33     """
34
35
36     import sys
37
38     nstep = int(sys.argv[1]);      # picks up 1st argument on command line
39     dt    = float(sys.argv[2]);    # picks up 2nd argument on command line
40     x     = [2.0, 0.0];          # start position
41
42     for i in range(0,nstep):
43         x = [x[0]-x[1]*dt, x[1]+x[0]*dt];
44         print "%5.3f %5.3f" % (x[0], x[1])
```

Emacs Command Reference Page

based on emacs v20.7

Robert Evans, rbevans@akane.jhuapl.edu

7/10/00

1. Nomenclature

- C-key** Hold the 'Control key' down while hitting "key"
- A-key** Hold the "Alt Key" down while hitting "key". *On Windows, you can use the "Alt Key" or "Escape Key" and on Solaris you use the "Escape Key" or the "Meta (black diamond) key". these are used in place of the "Meta" key, from the original emacs on Symbolics computers, hence you will see "M-" when you type "Alt" or "Escape".*
- A-x command** Hold the "Alt" Key down while hitting the "x" key, then type the command shown
- <space>** Spacebar

2. Program Control

- C-x C-f** Open or Create a file
- C-x C-s** Save file
- C-x s** Save all open buffers
- C-x C-w** Save file as...
- C-x C-c** Exit Emacs
- C-g** Cancel current command
- C-l** Redisplay screen and center on cursor
- C-/** Undo action (keystroke or command)
- C-x d** Open Directory

3. Cursor Movement

- C-n** Move cursor down one line
- C-p** Move cursor up one line
- C-f** Move cursor forward one line
- C-b** Move cursor back one line
- C-a** Move cursor to beginning of line
- C-e** Move cursor to end of line
- C-v** Move one page down
- A-v** Move one page up
- A-<** Move to beginning of file
- A->** Move to end of file
- A-x goto-line n** Go to line n

4. Editing

- C-k** Erase(Kill) line. Does not remove newline at end unless line is empty. Content Removed is sent to Kill Ring Buffer
- C-d** Erase next Character
- A-d** Erase Word
- Delete** Delete next character
- C-<space>** Begin Marking of Region at Cursor
- C-w** Delete region from begin of mark to current cursor position. Content is moved to Kill Ring Buffer
- C-y** Insert content(Yank) of Kill Ring Buffer at cursor position
- A-y** If typed after a C-y, it goes through any entries in the Kill Ring and replaces the C-y insert with the next former Kill Ring Entry
- A-w** Copy content of region from marked beginning to current cursor location into the Kill Ring Buffer
- C-t** Transpose characters on either side of cursor
- A-t** Transpose words on either side of cursor
- C-q** Escape the special meaning of the next character, if you want to insert a ^C into the code, you type C-q C-c.
- C-c C-c** Comment out a marked region
- C-u C-c C-c** Un-comment out the marked region

5. Search and Replace

- C-s** Search for string starting at the cursor.. The line at the bottom of the window has you type in the string you search for. The search is done as you type. To look for the next occurrence, hit C-s again, and your last entry is used as the default search string. If you hit the bottom of the buffer, you type C-s again to go back to the top.
- C-r** Reverse search. Same as C-s above, except it searches backward from the cursor

A-x replace-string Works from current cursor location. Begins a two part dialog, at the bottom of the window it first asks for the string you want to search for, and then asks for what you want to replace it with. This operation immediately works on the entire buffer

A-% Query replace string. Works from current cursor location. Begins a two part dialog, at the bottom of the window it asks you what you want to search for and then what you want to replace it with. It then searches for each occurrence of the string and highlights the match. If you want the replacement to occur, hit the spacebar, if you want it to skip this match, hit the “delete” key.

6. Buffers, Regions, & Windows

- C-x 2** Split window into two equal sized buffers, one on top of the other
- C-x 3** Split window into two equal sized buffers, one to the left/right of the other.
- C-x o** Move the cursor to the next buffer in the visible window
- C-x 5 2** Create a new window, with the contents of the current buffer in the new window as well
- C-x 1** If in a two buffer window, have the buffer that the cursor is in take up the entire window.
- C-x 0** If in a two buffer window, have the buffer that the cursor is *not* in take up the entire window
- C-x b** Shift back to the most recent buffer you were visiting prior to your current buffer
- C-x b filename** Shift to the named filename buffer
- C-x C-b** List all of the buffers currently opened. If the cursor is on a line in this buffer window, typing an “e” will open that buffer in the current buffer, typing an “o” will open it in a second buffer in the current window.
- C-x k** Kill the current buffer. If the buffer is not saved you will be prompted to save it

7. Java Development Environment (JDE) Commands

- C-c C-v C-c** Compile the object in the current buffer
- C-c C-v C-r** Run main method of object in current buffer
- C-c C-v C-j** Run the “beanshell” java interpreter. Allows you to run an interactive java shell

8. Printing

- A-x print-buffer** Prints the current buffer
- A-x print-region** Prints the current marked region

9. Command Shell

- A-x shell** Creates a command shell on the native OS. All of emacs commands work within this shell for cutting and pasting.
- A-p** Yank command from prior command history
- A-n** Next command from prior command history
- C-x k** Kill buffer and the shell along with it

10. Additional Notes:

Emacs is fantastically customizable and flexible. Check the GNU website for more information about customizing emacs. If you are using JDE, you can use project files for a package directory, which allows you to customize emacs settings for each package you work in.

VIM QUICK REFERENCE CARD

Basic movement

`h l k j` character left, right, line up, down
`b w` word/token left, right
`ge e` end of word/token left, right
`{ }` beginning of previous, next paragraph
`()` beginning of previous, next sentence
`0 gm` beginning, middle of line
`^ $` first, last character of line
`nG ngg` line n , default the last, first
`n%` percentage n of the file (n must be provided)
`n|` column n of current line
`%` match of next brace, bracket, comment, `#define`
`nH nL` line n from start, bottom of window
`M` middle line of window

Insertion & replace → insert mode

`i a` insert before, after cursor
`I A` insert at beginning, end of line
`gI` insert text in first column
`o O` open a new line below, above the current line
`rc` replace character under cursor with c
`grc` like `r`, but without affecting layout
`R` replace characters starting at the cursor
`gR` like `R`, but without affecting layout
`cm` change text of movement command m
`cc or S` change current line
`C` change to the end of line
`s` change one character and insert
`~` switch case and advance cursor
`g~m` switch case of movement command m
`gum gUm` lowercase, uppercase text of movement m
`<m>m` shift left, right text of movement m
`n<< n>>` shift n lines left, right

Deletion

`x X` delete character under, before cursor
`dm` delete text of movement command m
`dd D` delete current line, to the end of line
`J gJ` join current line with next, without space
`:rd` delete range r lines
`:rdx` delete range r lines into register x

Insert mode

`'Vc `Vn` insert char c literally, decimal value n
`^A` insert previously inserted text
`^@` same as `^A` and stop insert → command mode
`'Rx `R`Rx` insert content of register x , literally
`'N `P` text completion before, after cursor
`~W` delete word before cursor
`U` delete all inserted character in current line
`^D T` shift left, right one shift width
`'Kc1c2 or c1←c2` enter digraph {c₁, c₂}
`'C` execute c in temporary command mode
`X E `X Y` scroll up, down
`(esc) or ^[` abandon edition → command mode

Copying

`"x` use register x for next delete, yank, put
`:reg` show the content of all registers
`:reg x` show the content of registers x
`ym` yank the text of movement command m
`yy or Y` yank current line into register
`p P` put register after, before cursor position
`lp [p` like `p, P` with indent adjusted
`gp gP` like `p, P` leaving cursor after new text

Advanced insertion

`g?m` perform rot13 encoding on movement m
`n^A n^X` + n , - n to number under cursor
`gqm` format lines of movement m to fixed width
`:rce w` center lines in range r to width w
`:rle i` left align lines in range r with indent i
`:rri w` right align lines in range r to width w
`!mc` filter lines of movement m through command c
`n!!c` filter n lines through command c
`:r!c` filter range r lines through command c

Visual mode

`v V `V` start/stop highlighting characters, lines, block
`o` exchange cursor position with start of highlighting
`gv` start highlighting on previous visual area
`aw as ap` select a word, a sentence, a paragraph
`ab aB` select a block (), a block { }

Undoing & repeating commands

`u` undo last command, restore last changed line
`. R` repeat last changes, redo last undo
`n` repeat last changes with count replaced by n
`qc qC` record, append typed characters in register c
`q` stop recording
`@c` execute the content of register c
`@@` repeat previous @ command
`:@c` execute register c as an Ex command
`:rg/p/c` execute Ex command c on range r
[where pattern p matches

Complex movement

- + line up/down on first non-blank character
`B W` space-separated word left, right
`gE E` end of space-separated word left, right
`n` down $n - 1$ line on first non-blank character
`g0` beginning of screen line
`g^ g$` first, last character of screen line
`gk gj` screen line up, down
`fc Fc` next, previous occurrence of character c
`tc Tc` before next, previous occurrence of c
; repeat last `fFT`, in opposite direction
`[[]]` start of section backward, forward
`]] [` end of section backward, forward
`(())` unclosed (,) backward, forward
`{()}` unclosed {, } backward, forward
`[m]m` start, end of backward, forward java method
`[#]#.` unclosed #if, #else, #endif backward, forward
`[*]*` start, end of /* */ backward, forward

Search & substitution

`/s` search forward, backward for s
`/s/o` search fwd, bwd for s with offset o
`n or /` repeat forward last search
`N or ?` repeat backward last search
`# *?` search backward, forward for word under cursor
`g# g*` same, but also find partial matches
`gd gd` local, global definition of symbol under cursor
`:rs/f/t/x` substitute f by t in range r
[: g—all occurrences, c—confirm changes
`:rs x` repeat substitution with new r & x

Special characters in search patterns

. ^ \$ any single character, start, end of line
\< \> start, end of word
[c₁..c₂] a single character in range c₁..c₂
[^c₁..c₂] a single character not in range
\i \I an identifier, excluding digits
\k \K a keyword, excluding digits
\f \F a file name, excluding digits
\p \P a printable character, excluding digits
\s \S a white space, a non-white space
\e \t \r \b <esc>, <tab>, <→>, <←>
\= * \+ match 0..1, 0..∞, 1..∞ of preceding atoms
\| separate two branches (≡ or)
\(\) group patterns into an atom

Offsets in search commands

n_{or} +n n line downward in column 1
-n n line upward in column 1
e+n e-n n characters right, left to end of match
s+n s-n n characters right, left to start of match
;sc execute search command sc next

Marks and motions

mc mark current position with mark c ∈ [a..Z]
'c 'C go to mark c in current, C in any file
'0..9 go to last exit position
'` '' go to position before jump, at last edit
'[']' go to start, end of previously operated text
:marks_c print the active marks list
:jumps_c print the jump list
n^0 go to nth older position in jump list
n^I go to nth newer position in jump list

Key mapping & abbreviations

:map c e_c map c ↪ e in normal & visual mode
:map! c e_c map c ↪ e in insert & cmd-line mode
:unmap c remove mapping c
:mk f_c write current mappings, settings... to file f
:ab c e_c add abbreviation for c ↪ e
:ab c show abbreviations starting with c
:una c remove abbreviation c

Tags

:ta t_c jump to tag t
:nta_c jump to nth newer tag in list
'] T jump to the tag under cursor, return from tag
:ts t_c list matching tags and select one for jump
:tj t_c jump to tag or select one if multiple matches
:tags_c print tag list
:npo_c :n T_c jump back from, to nth older tag
:tl_c jump to last matching tag
'W} :pt t_c preview tag under cursor, tag t
'W] split window and show tag under cursor
'Wz or :pc_c close tag preview window

Scrolling & multi-windowing

'E 'Y scroll line up, down
'D 'U scroll half a page up, down
'F 'B scroll page up, down
zt or z_c set current line at top of window
zz or z. set current line at center of window
zb or z- set current line at bottom of window
zh zl scroll one character to the right, left
zh zL scroll half a screen to the right, left
'Ws or :split_c split window in two
'Wn or :new_c create new empty window
'Wo or :on_c make current window one on screen
'Wj 'Wk move to window below, above
'Ww 'Ww move to window below, above (wrap)

Ex commands (→)

:e f edit file f, unless changes have been made
:e! f edit file f always (by default reload current)
:wn :wN write file and edit next, previous one
:n :N edit next, previous file in list
:rw write range r to current file
:rw f write range r to file f
:rw>>f append range r to file f
:q :q! quit and confirm, quit and discard changes
:wq or :x or ZZ write to current file and exit
(up) (down) recall commands starting with current
:r f insert content of file f below cursor
:r! c insert output of command c below cursor
:all open a window for each file in the argument list
:args display the argument list

Ex ranges

, ; separates two lines numbers, set to first line
n an absolute line number n
. \$ the current line, the last line in file
% * entire file, visual area
't position of mark t
/p/ ?p? the next, previous line where p matches
+n -n +n, -n to the preceding line number

Miscellaneous

:sh_c !c_c start shell, execute command c in shell
K lookup keyword under cursor with man
:make_c start make, read errors and jump to first
:cn_c :cp_c display the next, previous error
:cl_c :cf_c list all errors, read errors from file
'L 'G redraw screen, show filename and position
g'G show cursor column, line, and character position
ga show ASCII value of character under cursor
gf open file which filename is under cursor
:redir>f_c redirect output to file f
^@ 'K ^_ \ unused keys, available for mapping

This card may be freely distributed under the terms of the GNU general public licence — Copyright © 2002 by Laurent Grégoire (laurent.gregoire@icam.fr) — v1.3 — The author assumes no responsibility for any errors on this card. The latest version can be found at <http://tnerual.erriegerg.free.fr/>

TextPad Quick Reference Card

version 0.03 – editor: John Bokma – freelance programmer

Cursor Movement

Cursor left one character
Cursor left one word
Cursor right one character
Cursor right one word
Cursor down one line
Cursor down to the start of the next paragraph
Cursor up one line
Cursor up to the start of the previous paragraph
Move cursor forward to start of word
Move the cursor back to start of word
Move cursor back to end of word
Cursor to start of line, press twice to go to the left margin
Cursor to end of line
Cursor to start of document
Cursor to end of document
Cursor to the first visible line, in the current column, if possible
Cursor to the last visible line, in the current column, if possible
Move cursor to the next tab stop, or indent selected lines
Move cursor to the previous tab stop, or reduce indentation of selected lines
Go to line
Find matching { [(< or >)] }

Deleting

Delete selection, or character before the cursor, (replace it with a space in overtype mode)
Delete back to the last start of word
Delete selection, or character after the cursor
Delete forward to the next start of word
Delete to the end of the line
Delete all lines in the document

Undo and Redo

Undo last edit
Undo all edits
Redo last undo
Redo all undos

Selection and Clipboard

Select all
Cancel any existing selection
Select left one character
Select left one word
Select right one character
Select right one word
Select down one line
Select to the start of the next paragraph
Select up one line

← Select to the start of the previous paragraph
→ Select forward to start of word
Select back to start of word
Select back to end of word
Select to start of line, press twice to select to the left margin
c← Select to end of line
→ Select to start of document
c Select to end of document
↓ Select to matching { [(< or >)] }
a↓ Switch in and out of selection mode
↑ Copy selection to clipboard
a↑ Append selection to clipboard
c-W Cut the selection to the clipboard
c-B Cut and append the selection to the clipboard
c-D Paste text from the clipboard
Home Indent selected lines
End Reduce indentation of selected lines
c-Home Delete selection
c-End Delete selection, or character after the cursor
a-Home Invert case of selection
Convert first character of selection to upper case and the rest to lower case
a-End Check the spelling of the selection

Tab
s-Tab Start a new line
c-G Insert new line after current line
c-M Insert new line before current line
Enter Increase indentation
c-Enter Reduce indentation
c-s-Enter Join selected lines
Backspace Reformat selected lines
c-Backspace Split word-wrapped lines
Delete Center text
c-Delete Right align text
c-s-Delete Insert a page break
a-Delete Display/hide visible spaces, tabs and paragraphs
Display/hide line numbers
Set the right margin at the cursor position
Switch in and out of word-wrap mode

Case Change and Transposing

Convert selection to lower case
Convert selection to upper case
Convert first character of selection to upper case and the rest to lower case
Invert case of selection
Transpose the lines or characters either side of the cursor
Transpose the words either side of the cursor

c-A Escape
s← c-s← → c-s→
c-K c-T c-s-T

Search and Replace

s↓ Invoke the Replace dialog box

a-s↓ Replace next instance of search pattern

↑ Invoke the Find dialog box

a↑ Invoke the Find in Files dialog box
c-W Find next instance of search pattern
c-s-B Find previous instance of search pattern
c-s-D Hypertext jump in Search Results window
s-Home Hypertext jump to next item in Search Results window
s-End Hypertext jump to previous item in Search Results window
c-s-Home Activate the Search Results window

Bookmarks

c-s-End Set or clear a bookmark on the current line
c-Q-S Go to next bookmark
c-C Go to previous bookmark

Edit Modes

c-s-C Switch between insert and overtype mode
c-X Switch in and out of block select mode
c-V Switch between read-only and edit modes
c-W Switch in and out of word-wrap mode

Backspace

Delete c-K
c-s-U F7

Record a new macro
Playback the scratch macro
Invoke the Playback Macro dialog box

Macros

c-s-R Create a new document
c-S Save the active document
c-s-S Save all documents
c-O Save as
c-I Open a document using the Open File dialog box
c-O Open a document by typing its name
c-s-O Insert the contents of a file at the cursor position
c-s-V Delete all lines in the document
a-Delete c-Delete
c-Tab or c-F6

Next window c-Tab or c-F6
Previous window c-F4
Close the active window a-Enter
Display in-context properties dialog box e-F1
Display document statistics on status bar e-F2
Invoke the Manage Files dialog box F3
Invoke Windows File Manager or Explorer a-F3
Print active document c-P
Preview the active document as it will print c-s-P
Check the spelling of the active document F7
Sort F9
Compare c-F9
Invoke the document selector F11

Scrolling and Scroll Bars

Scroll the view up one line, without moving the cursor c↓
Scroll the view down one line, without moving the cursor c↑
Locks cursor position when scrolling with page up/down keys Scroll Lock
Display/hide the horizontal scroll bar c-Q-H
Display/hide the vertical scroll bar c-Q-V
Switch in and out of synchronized scrolling mode c-Q-Y

Command Results

Stop the tool running in the command window
Hypertext jump in Command Results window
Hypertext jump to next item in Command Results window
Hypertext jump to previous item in Command Results window
Activate the Command Results window

c-Break

Enter

F4

s-F4

c-F11

\

"Escapes" the special meaning of the above expressions, so that they can be matched as literal characters.

Any letter.

Any lower case letter.

Any upper case letter.

Any digit or letter.

Any digit.

Any hexadecimal digit (0-9, a-f or A-F).

Space or tab.

Space, tab, vertical tab, return, line feed, form feed.

Control characters (Delete and ASCII codes less than space).

Printable characters, including space.

Printable characters, excluding space.

Anything that is not a control or alphanumeric character.

Letters, hyphens and apostrophes.

Any of the characters defined on the Syntax page for the document class, or in the syntax definition file if syntax highlighting is enabled for the document class.

Views

Activate next view
Activate previous view

F6

s-F6

Help

In-context help
Invoke in-context help cursor

F1

s-F1

Miscellaneous

Activate the Clip Library
Show or hide the Clip Library
Display in-context properties dialog box
Activate the main menu
Popup in the in-context document menu
Popup the insert date/time menu
Display the Preferences dialog box

a-0

c-F3

a-Enter

F10

s-F10 or right mouse

c-F10 or c-right mouse

c-Q-P

Regular Expressions (POSIX)

. Any single character.
[] Any one of the characters in the brackets, or any of a range of characters separated by a hyphen (-), or a character class operator (see below).
[^] Any characters except for those after the caret "^".
^ The start of a line (column 1).
\$ The end of a line (not the line break characters).
< The start of a word.
> The end of a word.
\t The tab character.
\f The page break (form feed) character.
\n A new line character, for matching expressions that span line boundaries. This cannot be followed by operators '*' '.*' or '{}'. Do not use this for constraining matches to the end of a line. It's much more efficient to use "\$". "dd" is the two-digit hexadecimal code for any character.
\ddd Groups a tagged expression to use in replacement expressions. An RE can have up to 9 such expressions.
\| Matches either the expression to its left or its right.
* Matches zero or more preceding characters/expressions.
\? Matches zero or one preceding characters/expressions.
\+ Matches one or more preceding characters/ expressions.
\{count\} Matches the specified number of the preceding characters or expressions.
\{min,\} Matches at least the specified number of the preceding characters or expressions.
\{min,max\} Matches between min and max of the preceding characters or expressions.

\

"Escapes" the special meaning of the above expressions, so that they can be matched as literal characters.

Any letter.

Any lower case letter.

Any upper case letter.

Any digit or letter.

Any digit.

Any hexadecimal digit (0-9, a-f or A-F).

Space or tab.

Space, tab, vertical tab, return, line feed, form feed.

Control characters (Delete and ASCII codes less than space).

Printable characters, including space.

Printable characters, excluding space.

Anything that is not a control or alphanumeric character.

Letters, hyphens and apostrophes.

Any of the characters defined on the Syntax page for the document class, or in the syntax definition file if syntax highlighting is enabled for the document class.

Replacement Expressions

&

Substitute the text matching the entire search pattern.

\0 to \9

Substitute the text matching tagged expression 0 through 9. \0 is equivalent to &.

\f

Substitute a page break (form feed).

\i<no>

Substitute a sequence number.

\n

Substitute a newline.

\p

Substitute the contents of the clipboard.

\t

Substitute a tab.

\xdd

Substitute the character with hex code dd (must be 2 hex digits, excluding 00).

\u

Force the next substituted character to be in upper case.

\l

Force the next substituted character to be in lower case.

\U

Force all subsequent substituted characters to be in upper case.

\L

Force all subsequent substituted characters to be in lower case.

\E or \e

Turns off previous \U or \L.

Tool Parameter Macros

\$File

The fully qualified filename of the current document.

\$DOSFile

Same as \$File, except that DOS aliases are substituted for any long names in the path, and characters are converted to the DOS (OEM) code set.

\$UNIXFile

Same as \$File, except any '\v' characters are changed to '/'.

\$FileName

The simple filename of the current document.

\$BaseName

\$FileName stripped of any extension.

\$DOSBaseName

Same as \$BaseName, except that the DOS alias is substituted for a long file name, and characters are converted to the DOS (OEM) code set.

\$WspBaseName

The workspace filename, stripped of any path and extension.

The drive and directory of the current document.
The drive and directory of the current workspace file.

The directory of the current document, stripped of the drive.

Same as \$FilePath, except any '\v' characters are changed to '/'.

The current working drive and directory.

Same as \$Dir, except any '\v' characters are changed to '/'.

The cursor line within the current document.

The cursor column within the current document.
Prompt for a value to substitute for \$Prompt. If it is followed by a string in brackets, that string will be displayed in the prompt dialog box.

Prompt for a value to substitute for \$Password. The value will not be echoed as it is typed. If it is followed by a string in brackets, that string will be displayed in the prompt dialog box.

Selected text in the active document. This is limited to the first line in a multi-line selection.

The text on the line containing the cursor. This has the side effect of selecting that line.

The word containing the cursor. This has the side effect of selecting that word.

Selected text in the active document, or the whole document if nothing is selected, is copied to the clipboard before running the tool.

The handle of the main application window. This is a decimal number.

The handle of the active document's window. This is a decimal number.

The characters encoding of the active document. This is of the forms: windows-ddd (or cpddd for DOS), UTF-8, UTF16-LE or UTF-16BE, where ddd is a code page number.

Page Header/Footer Macros

The normal font for subsequent text	&n
A bold font for subsequent text	&b
An italic font for subsequent text	&i
A bold italic font for subsequent text	&i
Subsequent text to be left justified	&l
Subsequent text to be centered (this is the default)	&c
Subsequent text to be right justified	&r
The current date in Windows short form	&d
The current date in Windows long form	&D
The current time in Windows format	&t
The filename, excluding its path	&F
The full filename, including its path	&F
The page number	&p
The total number of pages	&P

Based on the TextPad help file. Edited by John Bokma (freelance programmer). For the latest version: <http://johnbokma.com/textpad/>

Department of Engineering IT Services

 [University of Cambridge](#) > [Department of Engineering](#) > [Computing Help](#) > [LaTeX](#)

[introductions](#)

[writing guides](#)

[printable documentation](#)

[bibliographies](#)

[graphics](#)

[maths](#)

[tables](#)

[packages](#)

[fonts](#)

[sources of information](#)

[FAQ](#)

[local search](#)

[distributions](#)

[converters](#)

[editors/front-ends](#)

[Miscellaneous](#)

[local updates \(Jan 2014\)](#)

[example](#)

[exercises](#)

[more exercises](#)

Text Processing using LaTeX

TeX is a powerful text processing language and is the required format for some periodicals now. TeX has many macros to

which you can eventually add your own. LaTeX is a macro package which sits on top of TeX and provides all the structuring facilities to help with writing large documents. Automated chapter and section macros are provided, together with cross referencing and bibliography macros. LaTeX tends to take over the style decisions, but all the benefits of plain TeX are still present when it comes to doing maths. The [Why LaTeX?](#) page discusses LaTeX's strengths/weaknesses.

On CUED's central system you can run latex from the command line using latex or pdflatex. We also have Kile and Lyx



Introductions

- [LaTeX: An introduction](#), [Advanced LaTeX \(full of examples\)](#) and [LaTeX Maths and Graphics](#) contain all you'll need to know for writing most documents - the "how" rather than the "why".
- [LaTeX workshop exercise for beginners](#)
- [The Not So Short Introduction to LaTeX2e](#) is a 141 page introduction to LaTeX2e by Tobias Oetiker et al. Worth a read. There are versions in [german](#) and [french](#), [italian](#) etc.
- [The very short guide to typesetting with LATEX](#) (4 pages)
- [LaTeX and Friends](#) (M.R.C. van Dongen) (250+ pages)
- [LaTeX for Complete Novices](#) (Nicola L. C. Talbot)
- [Introduzione al Mondo di LaTeX](#) is a guide (PDF slides) in Italian
- [online tutorials](#) (Andy Roberts)
- [TeX Resources](#) (A.J. Hildebrand)
- [LaTeX for Word Processor Users](#)
- The Indian TeX Users Group has [tutorials](#) on several subjects.
- [The LaTeX Wikibook](#)
- [Making Friends with Latex](#)
- [LaTeX course](#) (University of Cambridge Computing Service)

Packages

There are numerous "add-ons" for LaTeX. Some [enumerate](#) and [fancyhdr](#) slightly enhance existing features, others provide extensive new functionality. The [TeX](#) and [LaTeX Catalogue](#) describes packages available elsewhere. See the [Configuring LaTeX](#) document if you intend to install many packages.

Bibliographies, Graphics and Maths

Front/Back matter

- See the [bibliographies](#) page.
- [bibliographies with biblatex](#)

- [Natural Science Citations](#) - provides many options. See also the [reference sheet](#)
- CTAN has many bibliography styles in its [bibtex](#) section.
- [Using Makeindex](#). *How to add an index to your document*
- [Simple LaTeX Glossaries and Acronyms using the glossaries package](#)
- The [nomencl](#) package *How to add nomenclature sections*

Graphics

- [Using Imported Graphics in LaTeX and PDFLaTeX](#) (by Keith Reckdahl) explains all there is to know about putting graphics into LaTeX documents. The [Hints about tables and figures in LaTeX](#) and [Hints on adding figures to multicolumn environments](#) documents deal with common problems. See also Klaus Hoeppner's [Strategies for including graphics in LaTeX documents](#)
- [How to influence the position of float environments like figure and table in LaTeX](#) (Frank Mittelbach)
- [Graphics for Inclusion in Electronic Documents](#) (Ian Hutchinson)
- The [xfig](#) graphics editor.
- *Gnuplot displays data graphically. Use its "set term postscript eps color" to produce a postscript file which can be added to your latex document in the usual way. Matlab may be preferable.*
- The [pstricks](#) tutorial show how to use the pstricks package to produce line drawings
- [Matlab graphics with LaTeX](#)

Maths

- The [psfrag](#) handout addresses the common problem of how to add LaTeX maths to a postscript file.
- Part of [Math into LaTeX](#) (by G. Grätzer) is online
- [AMS-LaTeX](#) provides specialist support.
- The [Short Math Guide for LaTeX](#) comes from the American Mathematical Society
- [mathmode](#) (133 pages) by Herbert Voß is useful.
- Matlab has some support for LaTeX production. Type "help latex" inside matlab for details.
- [Effective Scientific Electronic Publishing](#) (by Markus G. Kuhn) and [AcroTeX](#) by D.P.Story cover PDF production.
- [Maths cheat sheet](#) (Martin Jansche)
- [Math Tutorial for mimeTeX](#)
- [A Survey of Free Math Fonts for TeX and LaTeX](#) (Stephen G. Hartke)
- [Detexify - LaTeX symbol classifier](#) lets you draw a symbol and will give you the corresponding LaTeX

Tables

- [Tables in LaTeX: packages and methods](#)

Guides to writing various types of documents

- [Creating Technical Posters With LaTeX](#) (by Nicola Talbot)
- [Reports](#) (the squeezing space in LaTeX notes may also be useful)
- [Using LaTeX to Write a PhD Thesis](#) (Nicola L. C. Talbot)
- [LaTeX IIB project report classes](#) NEW
- [The CUED PhD/MPhil Thesis Style](#)
- [HTML or PDF from LaTeX](#)
- [Creating a PDF document using PDFflatex](#) (by Nicola Talbot)
- [Producing PDF](#)
- [Multi-column output](#)
- For collaborative or multi-draft documents, `latexdiff` might be useful. Doing

```
latexdiff -CCHANGEBAR old.tex new.tex > diff.tex
pdflatex diff.tex
```

should produce a document that compares and contrasts the 2 versions of the file.

CUED users can access the current university identifiers (crests) using

\includegraphics{BWUni3.eps} or \includegraphics{CUni3.eps} on our linux servers. These should only be used in their original sizes.

Other sources of information

General

- You can do a [keyword search](#) of the LaTeX documents on this server.
- [LaTeX Matters](#) (a blog)
- [LaTeX Community](#)
- See the [Frequently Asked Questions](#) (or the Engineering Department's [LaTeX FAQ](#)) for more information.
- The UK archive of TeX-related material, [CTAN](#) contains everything to do with LaTeX. Use the [CTAN search](#) to search your nearest CTAN archive.
- [TeX Live documentation](#)
- [Hypertext Help with LaTeX](#) (an extensive indexed reference)
- The [TeX Users Group](#) (TUG) keeps lists of TeX resources and packages (free and commercial), etc. The [LaTeX project](#) site is useful too.
- [References for TeX and Friends](#) from mixie.org offers material in several formats.
- [LaTeX cheat sheet](#)
- The [comp.text.tex](#) newsgroup covers LaTeX issues.
- [tex.stackexchange.com](#) is a forum for questions and answers
- The [PracTeX Journal](#) includes low-tech articles like `\begin{here} % getting started etc.`
- `texdoctk` is often installed with LaTeX. It's an easy way to access installed documentation

Quit	Search	Settings	Help/About
Fundamentals/general references	General layout	Mathematics/math fonts	
Administration	Floating objects/captions	Special text elements	
Macro programming	Graphics	Verbatim and code printing	
Accessory programs	Tables, arrays and lists	Auxiliary tools	
General fonts	ToC, index and glossary	Non-standard document styles	
Language/national specials	Bibliography	Miscellaneous	

Distributions

Note that the "front-end" (the program with an editor, buttons and menus) and the LaTeX files may well be separately distributed. If you install texmaker, for example, it will assume that you've already downloaded the latex system.

- Distributions for many machine types are available in CTAN's [systems](#) directory.
- For MS Windows 95/98/NT/2000 machines, [proTeXt](#) (based on [MiKTeX](#)) is worth a look. See [LaTeX using MiKTeX and WinEdit](#) for information about using MiKTeX and WinEdit on Windows. [BaKoMa TeX](#) might also be useful.
- [TeX Live](#) has binaries for most flavors of Unix, including GNU/Linux, and also Windows
- [MacTeX](#) for Macs includes support for using Mac fonts.
- The [Macintosh TeX/LaTeX Web Site](#) is very informative.

Converters

- [wvLaTeX](#) is installed (Word to LaTeX).
- [OpenOffice](#) has an option to export Word files as LaTeX
- There's a list of [RTF/Word/WP - LaTeX - converters](#) online.
- [Excel2Latex](#) may be useful to Windows users

Fonts and Characters

- [Using common PostScript fonts with LaTeX](#)
- [The Comprehensive LaTeX Symbol List](#)
- [LaTeX and fonts](#)
- [The Font Installation Guide](#) (Philipp Lehman)
- [character sets](#)

Typesetting

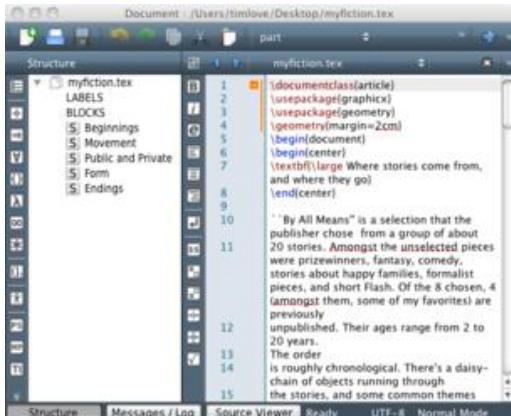
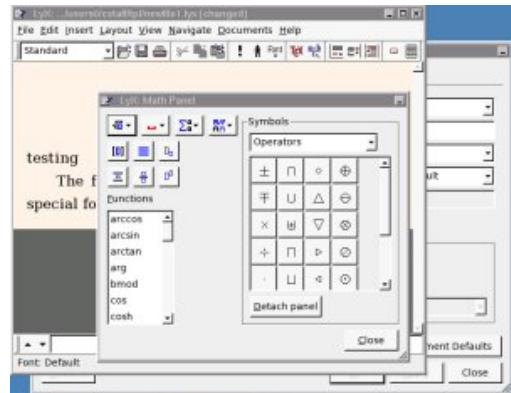
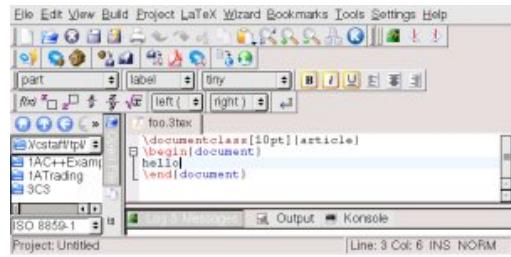
- The [memoir](#) package has very extensive documentation about design.

Editors/Front-ends

- With **Kile** (installed on our local system - type kile in the Terminal window to start it) you still need to type LaTeX code, but Kile has many facilities (templates, wizards, etc) to make it easier.

You should be able to find what you want in the menus (for example, the File->Statistic option gives a word-count, etc). You can print the LaTeX file directly from Kile. To print the output file you need to use another program. For example, if you want to create a PDF file you can produce the DVI file, use the Build->Convert->DVIToPDF option, then the Build->View->ViewPDF option to view the file. The viewer has a Print option.

- lyx** is a WYSIWYG front-end for LaTeX that's getting better all the time. It's installed on our teaching system. Warning: it may not always be easy to convert between LaTeX and lyx formats - use at your own risk!
- Texmaker** (not installed) is a free cross-platform LaTeX editor
- LEd** is a free integrated development environment (IDE) for use with Windows



95/98/Me/NT4/2000/XP/2003/Vista operating systems

- writelatex** lets you write LaTeX docs and work collaboratively without needing to install anything.
- The emacs editor offers extra menus when a LaTeX file is loaded in

Miscellaneous

- [Installing LaTeX Packages](#)
- [Configuring LaTeX](#)
- [Extending LaTeX](#)
- [Travels in TeX Land: Tweaking LaTeX](#) (David Walden)
- [LaTeX tips](#) (Volker Koch)
- Postscript, PDF and LaTeX versions of local documentation are [online](#).

Updates NEW

- January 2014 - [new PhD/MPhil Thesis](#) (with LyX support)
- March 2013 - minitoc installed
- July 2012 - TeXLive 2011 installed
- May 2011 - biblatex installed
- May 2009 - LaTeX removed from gate. Use one of the [Linux servers](#)

- May 2009 - IIB project classes (also for LyX users)
- February 2009 - latexdiff program installed - to determine and mark up differences between two latex files. Type man latexdiff for details.
- January 2009 - glossaries package installed, to supercede glossary. See the [glossaries documentation](#) for details.
- September 2008 - The TeX Live distribution has replaced the teTeX distribution. Users shouldn't notice any difference.
- September 2007 - nomencl (nomenclature package) updated to version 4.2. It's incompatible with the old version - use \usepackage[compatible]{nomencl} if you want the old behaviour.
- August 2007 - Metapost (mpost) and purifyeps installed
- July 2007 - TeTeX 3.0 installed on the teaching system
- 23/10/06 - Harish Bhanderi's [CUED PhD/MPhil Thesis Style](#)

Example

One way to get started with LaTeX is to look at a simple example. A short document is reproduced below. Engineering Department users can find a file with a similar structure in /export/Examples/LaTeX/demo0.tex. Further examples (a letter, a CV, etc) are in the same directory.

```
\documentclass{article}
\begin{document}

\section{Simple Text} % THIS COMMAND MAKES A SECTION TITLE.

Words are separated by one or more spaces. Paragraphs are separated by one or more blank lines. The output is not affected by adding extra spaces or extra blank lines to the input file.

Double quotes are typed like this: ``quoted text''.
Single quotes are typed like this: `single-quoted text'.

Long dashes are typed as three dash characters---like this.

Italic text is typed like this: \textit{this is italic text}.
Bold text is typed like this: \textbf{this is bold text}.

\subsection{A Warning or Two} % THIS COMMAND MAKES A SUBSECTION

If you get too much space after a mid-sentence period---abbreviations like etc.\ are the common culprits)---then type a backslash followed by a space after the period, as in this sentence.

Remember, don't type the 10 special characters (such as dollar sign and backslash) except as directed! The following seven are printed by typing a backslash in front of them: \$ & # \% \_ \{ and \}. The manual tells how to make other symbols.

\end{document} % THE INPUT FILE ENDS WITH THIS COMMAND
```

Once you have created a LaTeX source file it must be processed by LaTeX before it can be printed out. On systems that offer a command line you can try the command

`pdflatex myfile.tex`

while in the same folder as the saved LaTeX file. It will produce a number of files including myfile.log, myfile.aux and myfile.pdf. If you are using various sorts of cross referencing then you may have to run LaTeX more than once. If you want an automated bibliography you will also have to run bibtex.

When this procedure is complete you will have a file myfile.pdf to print out or preview.

© Cambridge University, Engineering Department, Trumpington Street, Cambridge CB2 1PZ, UK ([map](#))
Tel: +44 1223 332600, Fax: +44 1223 332662
Contact: [tl136](#) (with help from jpmg, etc)

Generating high-quality portable PDF files

The usual way to compile a TeX source file is to generate a .dvi file with the `tex` or `latex` command and then convert it into a PostScript file with `dvips`. If a PDF file is required, it can be generated from the PostScript by `ps2pdf`. This can be problematic in two respects: the quality of images may degrade for no apparent reason, and the resulting PDF file may not display correctly on other systems.

A long time ago, I found on someone else's home page a `dvips` command line which prevents both problems. (It seems to be extremely well hidden, as I did not manage to find it again. However, I made a note of it.) Here it is:

```
ps2pdf -sPAPERSIZE=a4 -dCompatibilityLevel=1.3 \
-dEmbedAllFonts=true -dSubsetFonts=true -dMaxSubsetPct=100 \
-dAutoFilterColorImages=false -dColorImageFilter=/FlateEncode \
-dAutoFilterGrayImages=false -dGrayImageFilter=/FlateEncode \
-dAutoFilterMonoImages=false -dMonoImageFilter=/CCITTFaxEncode \
document.ps document.pdf
```

I have since learned to understand the options. They are named, but hardly explained in the `ps2pdf` documentation which consists of the file `Ps2pdf.htm` in the Ghostscript documentation directory (use `locate Ps2pdf.htm` to find it).

The important thing for the image quality is `AutoFilter...Images=false` and `...ImageFilter=/FlateEncode`. The first disables the automatic determination by Ghostscript of the "best" compression format, which tends to favour `/DCTEncode`, lossy JPEG encoding. The second set of options manually set the compression method to the lossless (de)flate encoding for colour and greyscale images and to CCITT encoding for monochrome images.

The other options are for maximum compatibility of the generated PDF file. `CompatibilityLevel` sets the PDF version. The remaining options concern embedding of fonts into the generated PDF. `EmbedAllFonts=true` is self-explanatory and causes the output file to be readable even on systems which lack some of the fonts used. `SubsetFonts=true` together with `MaxSubsetPct=100` causes the fonts to be embedded partly only, however many characters from them may be used. This protects you from lawsuits if you use copyrighted fonts, as embedding a font in full amounts to an illegal copy. Last, the option `-sPAPERSIZE=a4` doesn't seem necessary unless you convert from some other size; replace `a4` by `letter` if that is the paper size you use.

An alternative way to arrive at a PDF file, if you do not require a PostScript file, is to use `pdftex` or `pdflatex` instead of `tex` or `latex`. In my experience, `pdflatex` embeds all fonts by default, as subsets, so you are safe on both the compatibility and the copyright issue. However, to be able to use `pdflatex`, you have to convert graphics into PDF format (or PNG for pixel graphics). To avoid any loss of quality, this should be done with the same `ps2pdf` command line shown above. The options relating to font embedding should not be omitted, as vector graphics can contain text which requires fonts. The paper

size option should be omitted.

As an aside, the options of `ps2pdf` above can be required in different contexts as well. That is because `ps2pdf` is just a script calling the Ghostscript interpreter (`gs`) and passes its options to it unchanged. `gs` can be used for tasks as diverse as concatenating PDF files, with the command line

```
gs -dBATCH -dNOPAUSE -dSAFER -sDEVICE=pdfwrite -sOUTPUTFILE=output.pdf \
<ps2pdf options> source1.pdf source2.pdf ...
```

where `<ps2pdf options>` stands for the options given above. The options conserving image quality are especially useful when putting the scanned pages of a document together (even the large copier at my office outputs single-page PDF files unless you can put a stack of loose pages into its automatic feed). You can use `gs` with the same command line and only one source file to embed fonts into a PDF document without regenerating it, provided the fonts are available on the system where you do it. Unfortunately the resulting document can be significantly larger, not because of the embedded fonts, but because `gs` is inefficient at re-encoding the images (you can see that it is not due to the fonts by trying `-dEmbedAllFonts=false`).

You can use the `pdffonts` command to find out which of the fonts used in a PDF document are embedded, and whether they are embedded as subsets.

UNIVERSITY *of* VIRGINIA LIBRARY

UNIVERSITYOFVIRGINIALIBRARY

Search Virgo for books, articles, digital materials, and

[Search](#)

ALSO TRY ▾

NAVIGATE TO:

(-- Select Page --)

[University of Virginia Library](#) > [Our Organization](#) > The
Electronic Text Center

The Electronic Text Center

The Electronic Text Center (1992-2007), known to many as “Etext,” served the University community’s teaching and research needs in the areas of humanities text encoding for fifteen years. Many of the resources once available on Etext are now available via [VIRGO](#), the Library’s online catalog and the primary access point for all U.Va. Library digital texts and images.

In the course of migrating thousands of texts from Etext to [VIRGO](#), we determined that certain resources were not eligible for inclusion, most often due to copyright issues. Many of the texts that were not migrated can be found among other university online text collections, [Google Books](#), [HathiTrust](#) and [Project Gutenberg](#). We regret any inconvenience this may cause you and we wish you the best with your research. Some pages from the Etext center have been preserved at the [Internet Archive](#).

If you have questions about the location of older resources, please send your inquiry to [Virgo Feedback](#).

HAVE QUESTIONS? NEED HELP?



Phone: 434.924.3021



Email: library@virginia.edu



[Chat now](#)

[Hours](#)

[Staff Directory](#)

[Jobs](#)

[Fellowships & Internships](#)

[Press](#)

[**GIVE TO THE LIBRARY**](#)

USING THE LIBRARY

[Library Use Policies](#)

[Off-Grounds Access](#)

[Library & ITS Accounts](#)

[Accessibility Services](#)

INITIATIVES

[Libra](#)

[Project Hydra](#)

OTHER SITES

[UVaCollab](#)

[SIS](#)

[ITS](#)

[Cavalier Advantage](#)

[U.Va. Home](#)

[Library Staff Site](#)

© 2014 by the Rector and Visitors of the [University of Virginia](#)



This library is a Congressionally designated depository for [U.S. Government documents](#). Public access to the Government documents is guaranteed by public law.

[Tracking Opt-out](#)

University of Virginia Library, P.O. Box 400113, Charlottesville, VA 22904-4113
(Parcel Service Delivery: Alderman Library, 160 McCormick Road, Charlottesville,
VA 22903)

Regular Expression Quick Reference v1.00

Online RegEx Resources: <http://gmckinney.info/regex>

Literal Characters

\f	Form feed
\n	Newline (Use \p in UltraEdit for platform independent line end)
\r	Carriage return
\t	Tab
\v	Vertical tab
\a	Alarm (beep)
\e	Escape
\xxx	The ASCII character specified by the octal number xxx
\xnn	The ASCII character specified by the hexadecimal number nn
\cx	The control character ^X. For example, \cl is equivalent to \t and \cJ is equivalent to \n

Character Classes

[...]	Any one character between the brackets.
[^...]	Any one character not between the brackets.
.	Any character except newline. Equivalent to [^\n]
\w	Any word character. Equivalent to [a-zA-Z0-9_] and [:alnum:]_
\W	Any non-word character. Equivalent to [^a-zA-Z0-9_] and [*[:alnum:]_]
\s	Any whitespace character. Equivalent to [\t\n\r\f\v] and [:space:]
\S	Any non-whitespace. Equivalent to [^\t\n\r\f\v] and [*[:space:]] Note: \w != \\$
\d	Any digit. Equivalent to [0-9] and [:digit:]
\D	Any character other than a digit. Equivalent to [^0-9] and [*[:digit:]]
[\\b]	A literal backspace (special case)
[[:class:]]	alnum alpha ascii blank cntrl digit graph lower print punct space upper xdigit

Replacement

\	Turn off the special meaning of the following character.
\n	Restore the text matched by the nth pattern previously saved by \l and \l. n is a number from 1 to 9, with 1 starting on the left.
\&	Reuse the text matched by the search pattern as part of the replacement pattern.
\~	Reuse the previous replacement pattern in the current replacement pattern. Must be the only character in the replacement pattern. (ex and vi).
\\$	Reuse the previous replacement pattern in the current replacement pattern. Must be the only character in the replacement pattern. (ed).
\u	Convert first character of replacement pattern to uppercase.
\U	Convert entire replacement pattern to uppercase.
\l	Convert first character of replacement pattern to lowercase.
\L	Convert entire replacement pattern to lowercase.

Repetition

{n,m}	Match the previous item at least n times but no more than m times.
{n,}	Match the previous item n or more times.
{n}	Match exactly n occurrences of the previous item.
?	Match zero or one occurrences of the previous item. Equivalent to {0,1}
+	Match one or more occurrences of the previous item. Equivalent to {1,}
*	Match zero or more occurrences of the previous item. Equivalent to {0,}
{ } ?	Non-greedy match - will not include the next match's characters.
??	Non-greedy match.
+?	Non-greedy match.
?	Non-greedy match. E.g. ^(.+?) \s \$ the grouped expression will not include trailing spaces.

Options

g	Perform a global match. That is, find all matches rather than stopping after the first match.
i	Do case-insensitive pattern matching.
m	Treat string as multiple lines (^ and \$ match internal '\n').
s	Treat string as single line (^ and \$ ignore '\n', but . matches '\n').
x	Extend your pattern's legibility with whitespace and comments.

Extended Regular Expression

(?#...)	Comment. "... is ignored.
(?:...)	Matches but doesn't return "..."
(?=...)	Matches if expression would match "... next
(?!...)	Matches if expression wouldn't match "... next
(?imsx)	Change matching rules (see options) midway through an expression.

Grouping

(...)	Grouping. Group several items into a single unit that can be used with *, +, ?, , and so on, and remember the characters that match this group for use with later references.
	Alternation. Match either the subexpressions to the left or the subexpression to the right.
\n	Match the same characters that were matched when group number n was first matched. Groups are subexpressions within (possibly nested) parentheses.

Anchors

^	Match the beginning of the string, and, in multiline searches, the beginning of a line.
\$	Match the end of the string, and, in multiline searches, the end of a line.
\b	Match a word boundary. That is, match the position between a \w character and a \W character. (Note, however, that \b matches backspace.)
\B	Match a position that is not a word boundary.

BNF and EBNF: What are they and how do they work?

By: [Lars Marius Garshol](#)

Contents

- [Introduction](#)
 - [What is this?](#)
 - [What is BNF?](#)
- [How it works](#)
 - [The principles](#)
 - [A real example](#)
 - [EBNF: What is it, and why do we need it?](#)
 - [An EBNF sample grammar](#)
- [Uses of BNF and EBNF](#)
 - [Common uses](#)
 - [How to use a formal grammar](#)
- [Parsing](#)
 - [The easiest way](#)
 - [Top-down parsing \(LL\)](#)
 - [An LL analysis example](#)
 - [An LL transformation example](#)
 - [The slightly harder way](#)
 - [Bottom-up parsing \(LR\)](#)
 - [LL or LR?](#)
 - [More information](#)
- [Appendices](#)
 - [Acknowledgements](#)

Introduction

What is this?

This is a short article that attempts to explain what BNF is, based on message <wkwwagbzn.fsf@ifi.uio.no> posted to comp.text.sgml on

16.Jun.98. Because of this it is a little rough, so if it leaves you with any unanswered questions, email me and I'll try to explain as best I can.

It has been filled out substantially since then and has grown quite large. However, you needn't fear. The article gets more and more detailed as you read on, so if you don't want to dig really deep into this, just stop reading when the questions you are interested in have been answered and things start getting boring.

What is BNF?

Backus-Naur notation (more commonly known as BNF or Backus-Naur Form) is a formal mathematical way to describe a language, which was developed by John Backus (and possibly Peter Naur as well) to describe the syntax of the Algol 60 programming language.

(Legend has it that it was primarily developed by John Backus (based on earlier work by the mathematician Emil Post), but adopted and slightly improved by Peter Naur for Algol 60, which made it well-known. Because of this Naur calls BNF Backus Normal Form, while everyone else calls it Backus-Naur Form.)

It is used to formally define the grammar of a language, so that there is no disagreement or ambiguity as to what is allowed and what is not. In fact, BNF is so unambiguous that there is a lot of mathematical theory around these kinds of grammars, and one can actually mechanically construct a parser for a language given a BNF grammar for it. (There are some kinds of grammars for which this isn't possible, but they can usually be transformed manually into ones that can be used.)

Programs that do this are commonly called "compiler compilers". The most famous of these is YACC, but there are many more.

How it works

The principles

BNF is sort of like a mathematical game: you start with a symbol (called the start symbol and by convention usually named S in examples) and are then given rules for what you can replace this symbol with. The language defined by the BNF grammar is just the set of all strings you can produce by following these rules.

The rules are called production rules, and look like this:

```
symbol := alternative1 | alternative2 ...
```

A production rule simply states that the symbol on the left-hand side of the := must be replaced by one of the alternatives on the right hand side. The alternatives are separated by |s. (One variation on this is to use ::= instead

of `:=`, but the meaning is the same.) Alternatives usually consist of both symbols and something called terminals. Terminals are simply pieces of the final string that are not symbols. They are called terminals because there are no production rules for them: they terminate the production process. (Symbols are often called non-terminals.)

Another variation on BNF grammars is to enclose terminals in quotes to distinguish them from symbols. Some BNF grammars explicitly show where whitespace is allowed by having a symbol for it, while other grammars leave this for the reader to infer.

There is one special symbol in BNF: `@`, which simply means that the symbol can be removed. If you replace a symbol by `@`, you do it by just removing the symbol. This is useful because in some cases it is difficult to end the replacement process without using this trick.

So, the language described by a grammar is the set of all strings you can produce with the production rules. If a string cannot in any way be produced by using the rules the string is not allowed in the language.

A real example

Below is a sample BNF grammar:

```
S  := '-' FN |
     FN
FN := DL |
     DL '.' DL
DL := D |
     D DL
D  := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

The different symbols here are all abbreviations: S is the start symbol, FN produces a fractional number, DL is a digit list, while D is a digit.

Valid sentences in the language described by this grammar are all numbers, possibly fractional, and possibly negative. To produce a number, start with the start symbol S:

S

Then replace the S symbol with one of its productions. In this case we choose not to put a `'-'` in front of the number, so we use the plain FN production and replace S by FN:

FN

The next step is then to replace the FN symbol with one of its productions. We want a fractional number, so we choose the production that creates two decimal lists with a `'. '` between them, and after that we keep choosing replacing a symbol with one of its productions once per line in the example below:

DL . DL

D . DL

```
3 . DL  
3 . D DL  
3 . D D  
3 . 1 D  
3 . 1 4
```

Here we've produced the fractional number 3.14. How to produce the number -5 is left as an exercise for the reader. To make sure you understand this you should also study the grammar until you understand why the string 3..14 cannot be produced with these production rules.

EBNF: What is it, and why do we need it?

In DL I had to use recursion (ie: DL can produce new DLs) to express the fact that there can be any number of Ds. This is a bit awkward and makes the BNF harder to read. Extended BNF (EBNF, of course) solves this problem by adding three operators:

- ?: which means that the symbol (or group of symbols in parenthesis) to the left of the operator is optional (it can appear zero or one times)
- * : which means that something can be repeated any number of times (and possibly be skipped altogether)
- + : which means that something can appear one or more times

An EBNF sample grammar

So in extended BNF the above grammar can be written as:

```
S := '-'? D+ ('.' D+)?  
D := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

which is rather nicer. :)

Just for the record: EBNF is not more powerful than BNF in terms of what languages it can define, just more convenient. Any EBNF production can be translated into an equivalent set of BNF productions.

Uses of BNF and EBNF

Common uses

Most programming language standards use some variant of EBNF to define the grammar of the language. This has two advantages: there can be no disagreement on what the syntax of the language is, and it makes it much easier to make compilers, because the parser for the compiler can

be generated automatically with a compiler-compiler like YACC.

EBNF is also used in many other standards, such as definitions of protocol formats, data formats and markup languages such as XML and SGML. (HTML is not defined with a grammar, instead it is defined with an SGML DTD, which is sort of a higher-level grammar.)

You can see a collection of BNF grammars at the [BNF web club](#).

How to use a formal grammar

OK. Now you know what BNF and EBNF are, what they are used for, but perhaps not why they are useful or how you can take advantage of them.

The most obvious way of using a formal grammar has already been mentioned in passing: once you've given a formal grammar for your language you have completely defined it. There can be no further disagreement on what is allowed in the language and what is not. This is extremely useful because a syntax description in ordinary prose is much more verbose and open to different interpretations.

Another benefit is this: formal grammars are mathematical creatures and can be "understood" by computers. There are actually lots of programs that can be given (E)BNF grammars as input and automatically produce code for parsers for the given grammar. In fact, this is the most common way to produce a compiler: by using a so-called compiler-compiler that takes a grammar as input and produces parser code in some programming language.

Of course, compilers do much more checking than just grammar checking (such as type checking) and they also produce code. None of these things are described in an (E)BNF grammar, so compiler-compilers usually have a special syntax for associating code snippets (called actions) with the different productions in the grammar.

The best-known compiler-compiler is YACC (Yet Another Compiler Compiler), which produces C code, but others exist for C++, Java, Python as well as many other languages.

Parsing

The easiest way

Top-down parsing (LL)

The easiest way of parsing something according to a grammar in use today is called LL parsing (or top-down parsing). It works like this: for each production find out which terminals the production can start with. (This is called the start set.)

Then, when parsing, you just start with the start symbol and compare the start sets of the different productions against the first piece of input to see which of the productions have been used. Of course, this can only be done if no two start sets for one symbol both contain the same terminal. If they do there is no way to determine which production to choose by looking at the first terminal on the input.

LL grammars are often classified by numbers, such as LL(1), LL(0) and so on. The number in the parenthesis tells you the maximum number of terminals you may have to look at at a time to choose the right production at any point in the grammar. So for LL(0) you don't have to look at any terminals at all, you can always choose the right production. This is only possible if all symbols have only one production, and if they only have one production the language can only have one string. In other words: LL(0) grammars are not interesting.

The most common (and useful) kind of LL grammar is LL(1) where you can always choose the right production by looking at only the first terminal on the input at any given time. With LL(2) you have to look at two symbols, and so on. There exist grammars that are not LL(k) grammars for any fixed value of k at all, and they are sadly quite common.

An LL analysis example

As a demonstration, let's do a start set analysis of the sample grammar above. For the symbol D this is easy: all productions have a single digit as their start set (the one they produce) and the D symbol has the set of all ten digits as its start set. This means that we have at best an LL(1) grammar, since in this case we need to look at one terminal to choose the right production.

With DL we run into trouble. Both productions start with D and thus both have the same start set. This means that one cannot see which production to choose by looking at just the first terminal of the input. However, we can easily get round this problem by cheating: if the second terminal on input is *not* a digit we must have used the first production, but if they both are digits we must have used the second one. In other words, this means that this is at best an LL(2) grammar.

I actually simplified things a little here. The productions for DL alone don't tell us which terminals are allowed after the first terminal in the D @ production, because we need to know which terminals are allowed after a DL symbol. This set of terminals is called the follow set of the symbol, and in this case it is '.' and the end of input.

The FN symbol turns out to be even worse, since both productions have all digits as their start set. Looking at the second terminal doesn't help since we need to look at the first terminal after the last digit in the digit list (DL) and we don't know how many digits there are until we've read them all. And since there is no limit on the number of digits there can be, this isn't an LL(k) grammar for any value of k at all (there can always be more digits than k , no matter which value of k value you choose).

Somewhat surprisingly perhaps, the S symbol is easy. The first production has '-' as its start set, the second one has all digits. In other words, when you start parsing you'll start with the S symbol and look at the input to

decide which production was used. If the first terminal is '-' you know that the first production was used. If not, the second one was used. It's only the FN and DL productions that cause problems.

An LL transformation example

However, there is no need to despair. Most grammars that are not LL(k) can fairly easily be converted to LL(1) grammars. In this case we'll need to change two symbols: FN and DL.

The problem with FN is that both productions begin with DL, but the second one continues with a '.' and another DL after the initial DL. This is easily solved: we change FN to have just one production that starts with DL followed by FP (fractional part), where FP can be nothing or '.' followed by a DL, like this:

```
FN := DL FP
FP := @ | '.' DL
```

Now there are no problems with FN anymore, since there's just one production, and FP is unproblematic because the two productions have different start sets. End of input and '.', respectively.

The DL is a tougher nut to crack, since the problem is the recursion and it's compounded by the fact that we need at least one D to result from the DL. The solution is to give DL a single production, a D followed by DR (digits rest). DR then has two productions: D DR (more digits) or @ (no more digits). The first production has a start set of all digits, while the second has '.' and end of input as its start set, so this solves the problem.

This is the complete LL(1) grammar as we've now transformed it:

```
S := '-' FN | FN
FN := DL FP
FP := @ | '.' DL
DL := D DR
DR := D DR | @
D := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

The slightly harder way

Bottom-up parsing (LR)

A harder way to parse is the one known as shift-reduce or bottom-up parsing. This technique collects input until it finds that it can reduce an input sequence with a symbol. This may sound difficult, so I'll give an example to clarify. We'll parse the string '3.14' and see how it was produced from the grammar. We start by reading 3 from the input:

3

and then we look to see if we can reduce it to the symbol it was produced from. And indeed we can, it was produced from the D symbol, which we replace the 3 with. Then we note that we can produce the D from DL and replace the D with DL. (The grammar is ambiguous, which means that we

can reduce further to FN, which would be wrong. For simplicity we just skip the wrong steps here, but an unambiguous grammar would not allow these wrong choices.) After that we read the . from the input and try to reduce it, but fail:

```
D  
DL  
DL .
```

This can't be reduced to anything, so we read the next character from the input: 1. We then reduce that to a D and read the next character, which is 4. 4 can be reduced to D, then to DL, and then the "D DL" sequence can be further reduced to a DL.

```
DL .  
DL . 1  
DL . D  
DL . D 4  
DL . D D  
DL . D DL  
DL . DL
```

Looking at the grammar we quickly note that FN can produce just this "DL . DL" sequence and do a reduction. We then note that FN can be produced from S and reduce the FN to S and then stop, as we've completed the parse.

```
DL . DL  
FN  
S
```

As you may have noted we could often choose whether to do a reduction now or wait until we had more symbols and then do a different reduction. There are more complex variations on this shift-reduce parsing algorithm, in increasing complexity and power: LR(0), SLR, LALR and LR(1). LR(1) usually needs unpractically large parse tables, so LALR is the most commonly used algorithm, since SLR and LR(0) are not powerful enough for most programming languages.

LALR and LR(1) are too complex for me to cover here, but you get the basic idea.

LL or LR?

This question has already been answered much better by someone else, so I'm just quoting his news message in full here:

```
I hope this doesn't start a war...
```

First -- Frank, if you see this, don't shoot me. (My boss is Frank DeRemer, the creator of LALR parsing...)

(I borrowed this summary from Fischer&LeBlanc's "Crafting a Compiler")

Simplicity -- LL
Generality -- LALR
Actions -- LL
Error repair -- LL
Table sizes -- LL
Parsing speed -- comparable (me: and tool-dependent)

Simplicity -- LL wins
=====

The workings of an LL parser are much simpler. And, if you have to debug a parser, looking at a recursive-descent parser (a common way to program an LL parser) is much simpler than the tables of a LALR parser.

Generality -- LALR wins
=====

For ease of specification, LALR wins hands down. The big difference here between LL and (LA)LR is that in an LL grammar you must left-factor rules and remove left recursion.

Left factoring is necessary because LL parsing requires selecting an alternative based on a fixed number of input tokens.

Left recursion is problematic because a lookahead token of a rule is always in the lookahead token on that same rule. (Everything in set A is in set A...) This causes the rule to recurse forever and ever and ever and ever...

To see ways to convert LALR grammars to LL grammars, take a look at my page on it:

<http://www.jguru.com/thetick/articles/lalrtoll.html>

Many languages already have LALR grammars available, so you'd have to translate. If the language doesn't have a grammar available, then I'll say it's not really any harder to write a LL grammar from scratch. (You just have to be in the right "LL" mindset, which usually involves watching 8 hours of Dr. Who before writing the grammar... I actually prefer LL if you didn't know...)

Actions -- LL wins
=====

In an LL parser you can place actions anywhere you want without introducing a conflict

Error repair -- LL wins
=====

LL parsers have much better context information (they are top-down parsers) and therefore can help much more in repairing an error, not to mention reporting errors.

Table sizes -- LL
=====

Assuming you write a table-driven LL parser, its tables are nearly half the size. (To be fair, there are ways to optimize LALR tables to make them smaller, so I think this one washes...)

Parsing speed -- comparable (me: and tool-dependent)

--Scott Stanchfield in article
<33C1BDB9.FC6D86D3@scruz.net> on

More information

John Aycock has developed an unusually nice and simple to use parsing framework in Python called [SPARK](#), which is described in his [very readable paper](#).

The definitive work on parsing and compilers is 'The Dragon Book', or [Compilers : Principles, Techniques, and Tools](#), by Aho, Sethi and Ullman. Beware, though, that this is a rather advanced and mathematical book.

A free online alternative, which looks rather good, is [this book](#), but I can't comment on the quality, since I haven't read it yet.

Henry Baker has written [an article about parsing in Common Lisp](#), which presents a simple, high-performant and very convenient framework for parsing. The approach is similar to that of compiler-compilers, but instead relies on the very powerful macro system of Common Lisp.

One syntax for specifying BNF grammars can be found in [RFC 2234](#). Another can be found in [the ISO 14977 standard](#).

Appendices

Acknowledgements

Thanks to:

- [Jelks Cabaniss](#), for encouraging me to turn the news article into a web article, and for providing very useful criticism of the article once it appeared in web form.
- C. M. Sperberg-McQueen for extra historical information about the name of BNF.
- [Scott Stanchfield](#) for writing the great comparison of LALR and LL. I have asked for permission to quote this, but have received no reply, unfortunately.
- James Huddleston for correcting me on John Backus' name.
- [Dave Pawson](#) for correcting a bad link.

Selected Windows Keyboard Shortcuts

CTRL + C	Copy
CTRL + X	Cut
CTRL + V	Paste
CTRL + Z	Undo
CTRL + Y	Redo
CTRL + A	Select All
CTRL + F4	Close the Active Document
CTRL + Esc	Display the Start Menu
CTRL + Dragging Item	Copy the Selected Item
F1	Display Help
F2	Rename the Selected Item
F3	Search for a File or Folder
F4	Display the Address Bar in My Computer
F5	Update the Active Window
F6	Cycle through Screen Elements
F10	Activate the Menu Bar in Active Program
ALT + Enter	Display Properties of Selected Item
ALT + Spacebar	Open Shortcut for Active Window
ALT + F4	Close the Active Document
ALT + Tab	Switch Between Open Items
ALT + Esc	Cycle Through Items in Order Opened
SHIFT + Delete	Permanently Delete an Item
SHIFT + F10	Display selected Item's Shortcut Menu
SHIFT + Insert CD	Prevent CD from Automatically Playing
WIN	Display Start Menu
WIN + D	Minimize All Windows
WIN + Shift + M	Undo Minimize All Windows
WIN + E	Display Windows Explorer
WIN + F	Display Search Utility
WIN + R	Display Run utility
WIN + L	Lock Computer
WIN + U	Open Utility Manager

THE ONE PAGE *LINUX* MANUAL

A summary of useful Linux commands

Version 3.0

May 1999

squadron@powerup.com.au

Starting & Stopping

shutdown -h now	Shutdown the system now and do not reboot
halt	Stop all processes - same as above
shutdown -r 5	Shutdown the system in 5 minutes and reboot
shutdown -r now	Shutdown the system now and reboot
reboot	Stop all processes and then reboot - same as above
startx	Start the X system

Accessing & mounting file systems

mount -t iso9660 /dev/cdrom /mnt/cdrom	Mount the device cdrom and call it cdrom under the /mnt directory
mount -t msdos /dev/hdd /mnt/ddrive	Mount hard disk d as a msdos file system and call it ddrive under the /mnt directory
mount -t vfat /dev/hdal /mnt/cdrive	Mount hard disk a as a VFAT file system and call it cdrive under the /mnt directory
umount /mnt/cdrom	Unmount the cdrom

Finding files and text within files

find / -name fname	Starting with the root directory, look for the file called fname
find / -name "*fname*"	Starting with the root directory, look for the file containing the string fname
locate missingfilename	Find a file called missingfilename using the locate command - this assumes you have already used the command updatedb (see next)
updatedb	Create or update the database of files on all file systems attached to the linux root directory
which missingfilename	Show the subdirectory containing the executable file called missingfilename
grep textstringtofind /dir	Starting with the directory called dir , look for and list all files containing textstringtofind

The X Window System

xvidtune	Run the X graphics tuning utility
XF86Setup	Run the X configuration menu with automatic probing of graphics cards
Xconfigurator	Run another X configuration menu with automatic probing of graphics cards
xf86config	Run a text based X configuration menu

Moving, copying, deleting & viewing files

ls -l	List files in current directory using long format
ls -F	List files in current directory and indicate the file type
ls -laC	List all files in current directory in long format and display in columns

rm name	Remove a file or directory called name
rm -rf name	Kill off an entire directory and all it s includes files and subdirectories
cp filename /home dirname	Copy the file called filename to the /home/dirname directory
mv filename /home dirname	Move the file called filename to the /home/dirname directory
cat filetoview	Display the file called filetoview
man -k keyword	Display man pages containing keyword
more filetoview	Display the file called filetoview one page at a time, proceed to next page using the spacebar
head filetoview	Display the first 10 lines of the file called filetoview
head -20 filetoview	Display the first 20 lines of the file called filetoview
tail filetoview	Display the last 10 lines of the file called filetoview
tail -20 filetoview	Display the last 20 lines of the file called filetoview

Installing software for Linux

rpm -ihv name.rpm	Install the rpm package called name
rpm -Uhv name.rpm	Upgrade the rpm package called name
rpm -e package	Delete the rpm package called package
rpm -l package	List the files in the package called package
rpm -ql package	List the files and state the installed version of the package called package
rpm -i --force package	Reinstall the rpm package called name having deleted parts of it (not deleting using rpm -e)
tar -zxf archive.tar.gz or tar -zvf archive.tgz	Decompress the files contained in the zipped and tarred archive called archive
./configure	Execute the script preparing the installed files for compiling

User Administration

adduser accountname	Create a new user call accountname
passwd accountname	Give accountname a new password
su	Log in as superuser from current login
exit	Stop being superuser and revert to normal user

Little known tips and tricks

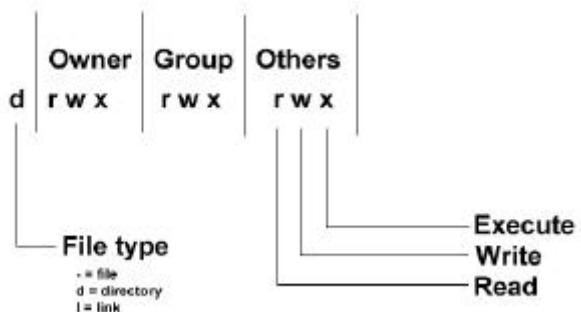
ifconfig	List ip addresses for all devices on the machine
apropos subject	List manual pages for subject
usermount	Executes graphical application for mounting and unmounting file systems

/sbin/e2fsck hda5	Execute the filesystem check utility on partition hda5
fdformat /dev/fd0H1440	Format the floppy disk in device fd0
tar -cMf /dev/fd0	Backup the contents of the current directory and subdirectories to multiple floppy disks
tail -f /var/log/messages	Display the last 10 lines of the system log.
cat /var/log/dmesg	Display the file containing the boot time messages - useful for locating problems. Alternatively, use the dmesg command.
*	wildcard - represents everything. eg. cp from/* to will copy all files in the from directory to the to directory
?	Single character wildcard. eg. cp config.? /configs will copy all files beginning with the name config. in the current directory to the directory named configs.
[xyz]	Choice of character wildcards. eg. ls [xyz]* will list all files in the current directory starting with the letter x, y, or z.
linux single	At the lilo prompt, start in single user mode. This is useful if you have forgotten your password. Boot in single user mode, then run the passwd command.
ps	List current processes
kill 123	Kill a specific process eg. kill 123

Configuration files and what they do

/etc/profile	System wide environment variables for all users.
/etc/fstab	List of devices and their associated mount points. Edit this file to add cdroms, DOS partitions and floppy drives at startup.
/etc/motd	Message of the day broadcast to all users at login.
/etc/rc.d/rc.local	Bash script that is executed at the end of login process. Similar to autoexec.bat in DOS.
/etc/HOSTNAME	Contains full hostname including domain.
/etc/cron.*	There are 4 directories that automatically execute all scripts within the directory at intervals of hour, day, week or month.
/etc/hosts	A list of all known host names and IP addresses on the machine.
/etc/httpd/conf	Parameters for the Apache web server
/etc/inittab	Specifies the run level that the machine should boot into.
/etc/resolv.conf	Defines IP addresses of DNS servers.
/etc/smb.conf	Config file for the SAMBA server. Allows file and print sharing with Microsoft clients.
~/.Xdefaults	Defines configuration for some X-applications. ~ refers to user's home directory.
/etc/X11/XF86Config	Config file for X-Windows.
~/.xinitrc	Defines the windows manager loaded by X. ~ refers to user's home directory.

File permissions



If the command `ls -l` is given, a long list of file names is displayed. The first column in this list details the permissions applying to the file. If a permission is missing for a owner, group or other, it is represented by - eg. `drwxr-x x`

Read = 4	File permissions are altered by giving the <code>chmod</code> command and the appropriate octal code for each user type. eg
Write = 2	<code>chmod 7 6 4 filename</code> will make the file called filename R+W+X for the owner, R+W for the group and R for others.
Execute = 1	<code>chmod 7 5 5</code> Full permission for the owner, read and execute access for the group and others.
	<code>chmod +x filename</code> Make the file called filename executable to all users.

X Shortcuts - (mainly for Redhat)

Control Alt + or -	Increase or decrease the screen resolution. eg. from 640x480 to 800x600
Alt escape	Display list of active windows
Shift Control F8	Resize the selected window
Right click on desktop background	Display menu
Shift Control AltR	Refresh the screen
Shift Control AltX	Start an xterm session

Printing

/etc/rc.d/init.d/lpd start	Start the print daemon
/etc/rc.d/init.d/lpd stop	Stop the print daemon
/etc/rc.d/init.d/lpd status	Display status of the print daemon
lpq	Display jobs in print queue
lprm	Remove jobs from queue
lpr	Print a file
lpc	Printer control tool
man subject lpr	Print the manual page called subject as plain text
man -t subject lpr	Print the manual page called subject as Postscript output
printtool	Start X printer setup interface



Get your own Official Linux Pocket Protector - includes handy command summary. Visit: www.powerup.com.au/~squadron



Mac OS X Lion Power User Keyboard Shortcuts

Visit MacMost.com for Mac and iPhone video tutorials, news, reviews and commentary.

v2.0.1

⇧ shift ⌘ control ⌘ option ⌘ command → tab ↵ return ⌋ delete ⌄ eject ⌂ esc

Finder Commands	
⌘ space	Spotlight menu
⌘ ⌄ space	Spotlight window
⌘ ⌋	Move to trash
⇧ ⌘ ⌋	Empty Trash
⇧ ⌄ ⌘ ⌋	Force Empty Trash
⌘ N	New Finder window
⇧ ⌘ N	New Folder
⌘ I	Get Info
⌄ ⌘ I	Show Inspector
⌘ Y or space	Quick Look
⌘ E	Eject selected volume
⌘ J	Show view options
⌘ K	Connect to server
⇧ ⌘ A	Open Applications folder
⇧ ⌘ C	Open Computer folder
⌘ ↑	Open enclosing folder
⌘ F	Find

Mission Control	
⌃ ↑	View Mission Control
⌃ ↓	Show app's windows
F11	Show desktop
F12	Show Dashboard
⌃ ←	Move between spaces
→ ↵	Show windows for next app after ⌃ ↓
spacebar	Enlarge window under cursor after ⌃ ↑

Keyboard Control Focus	
⌃ F2	Focus on menu bar
⌃ F3	Focus on Dock
⌃ F5	Focus on window toolbar
⌃ F8	Focus on menu bar status icons
Then, use ← → ↑ ↓ to navigate, ↵ to select, and esc to exit	

Switching Applications and Windows	
⌘ → ↵	Advance to next app
⌘ `	Next window in current app
⌄ ⌘ D	Hide/Show Dock

Universal Access Display Controls	
⌄ ⌘ 8	Toggle zoom feature (turn on to use zoom)
⌄ ⌘ = ⌄ ⌘ -	Zoom in / out (also ⌃ and mouse scroll)
⌃ ⌄ ⌘ 8	Reverse screen

Application Commands			
⌘ N	New window	⌘ ,	App preferences
⌘ O	Open file	⌘ H	Hide app
⌘ W	Close window	⌄ ⌘ H	Hide others
⌘ S	Save	⌘ T	Show fonts panel
⇧ ⌘ S	Save As	⇧ ⌘ C	Show colors panel
⌘ P	Print	⇧ ⌘ /	Help
⌘ Q	Quit	⌃ ⌘ F	Full Screen Mode

Startup Keys	
⌄	Choose boot volume
⌄ ⌘ P R	Reset PRAM
⌘ S	Single user mode boot
⌘ R	Disk Utility and Internet recovery
T	Go into Target disk mode

Note: Keyboard shortcuts can be disabled or customized in the System Preferences.

See <http://macmost.com/j-keyshort> for more shortcuts and to learn how to create your own.

Copyright © 2011 MacMost

Power Shortcuts	
⌄ ⌘ ▲	Sleep
⌃ ⌘ ▲	Restart
⌄ ⌄ ⌘ ▲	Shut down
⇧ ⌄ ▲	Sleep display
⌃ ▲	Power options dialog

Screen Capture	
⇧ ⌘ 3	Screen to file
⇧ ⌄ ⌘ 3	Screen to clipboard
⇧ ⌘ 4	Area to file (then space to get window)
⇧ ⌄ ⌘ 4	Area to clipboard (then space to get window)

How To Properly Comment Your Code

Uncommented Code The 6.189 staff received this code to grade.

```
city=raw_input("Enter a city: ")
while city[-1]==" ":
    city = city[:-1]
temp=raw_input("Enter a temperature in Farenheit: ")
temp = float(temp)
temp = (temp - 32.0)*(100.0/180.0)
temp = round(temp,3)
temp = str(temp)
print "In "+city+" it is "+temp+" degrees Celcius!"
```

There's a bunch of problems with it. A quick scan of it reveals no info as to which of our 250 students wrote this code. What is the name of the file? What does it do? At a glance, we are lost. This code would receive a grade of a ✓.

Commented Code The staff next received this file.

```
#Alyssa P. Hacker
#fah_to_celsius.py

#collect a city name from user
city=raw_input("Enter a city: ")

#trunacte whitespace
while city[-1]==" ":
    city = city[:-1]

#collect a temp from user
temp=raw_input("Enter a temperature in Farenheit: ")

#convert string to float
temp = float(temp)

#convert Farenheit temp to Celsius temp
temp = (temp - 32.0)*(100.0/180.0)

#truncate to 3 decimal places
temp = round(temp,3)

#recast as string so we can concatenate
temp = str(temp)

#print result!
print "In "+city+" it is "+temp+" degrees Celcius!"
```

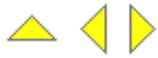
We can clearly see Alyssa's name and the name of her file. Further she has well-commented what the lines of her program do. This code would receive a grade of +. You should be like Alyssa! You should comment wherever you can - put comments that explain what you're doing, and if you're doing something tricky or unique be sure to explain that, as well. A good goal is to have 1 comment for every 1-4 lines of code. Be sure to not only document what your code is doing, but, as you begin writing more advanced code, you should document what was intentionally left out, optimized away, tried and discarded, etc - basically, any design decision you make.

MIT OpenCourseWare
<http://ocw.mit.edu>

6.189 A Gentle Introduction to Programming

January IAP 2011

For information about citing these materials or our Terms of Use, visit: <http://ocw.mit.edu/terms>.



2. Overview of the four main programming paradigms

In this section we will characterize the four main programming paradigms, as identified in [Section 1.2](#).

As the main contribution of this exposition, we attempt to trace the *basic discipline* and the *idea* behind each of the main programming paradigms.

With this introduction to the material, we will also be able to see how the functional programming paradigm corresponds to the other main programming paradigms.

- | | |
|---|--|
| 2.1 Overview of the imperative paradigm | 2.3 Overview of the logic paradigm |
| 2.2 Overview of the functional paradigm | 2.4 Overview of the object-oriented paradigm |

2.1. Overview of the imperative paradigm

[Contents](#) [Up](#) [Previous](#) [Next](#) [Slide](#) [Subject index](#) [Program index](#) [Exercise index](#)

**First do this and next
do that**

The 'first do this, next do that' is a short phrase which really in a nutshell describes the spirit of the imperative paradigm. The basic idea is the command, which has a measurable effect on the program state. The phrase also reflects that the order to the commands is important. 'First do that, then do this' would be different from 'first do this, then do that'.

In the itemized list below we describe the main properties of the imperative paradigm.

- Characteristics:
 - Discipline and idea
 - Digital hardware technology and the ideas of Von Neumann
 - Incremental *change of the program state* as a function of *time*.
 - Execution of computational steps in an order governed by *control structures*
 - We call the steps for *commands*
 - Straightforward abstractions of the way a traditional Von Neumann

computer works

- Similar to descriptions of everyday routines, such as food recipes and car repair
- Typical commands offered by imperative languages
 - Assignment, IO, procedure calls
 - Language representatives
 - Fortran, Algol, Pascal, Basic, C
- The natural abstraction is the procedure
 - Abstracts one or more actions to a procedure, which can be called as a single command.
 - "Procedural programming"

We use several names for the computational steps in an imperative language. The word *statement* is often used with the special computer science meaning 'a elementary instruction in a source language'. The word *instruction* is another possibility; We prefer to devote this word the computational steps performed at the machine level. We will use the word 'command' for the imperatives in a high level imperative programming language.

A procedure abstracts one or more actions to a procedure, which can be activated as a single action.

2.2. Overview of the functional paradigm

[Contents](#) [Up](#) [Previous](#) [Next](#) [Slide](#) [Subject index](#) [Program index](#) [Exercise index](#)

We here introduce the functional paradigm at the same level as imperative programming was introduced in [Section 2.1](#).

Functional programming is in many respects a simpler and more clean programming paradigm than the imperative one. The reason is that the paradigm originates from a purely mathematical discipline: the theory of functions. As described in [Section 2.1](#), the imperative paradigm is rooted in the key technological ideas of the digital computer, which are more complicated, and less 'clean' than mathematical function theory.

Below we characterize the most important, overall properties of the functional programming paradigm. Needless to say, we will come back to most of them in the remaining chapters of this material.

Evaluate an expression and use the resulting value
for something

- Characteristics:
 - Discipline and idea
 - Mathematics and the theory of functions
 - The values produced are *non-mutable*
 - Impossible to change any constituent of a composite value
 - As a remedy, it is possible to make a revised copy of composite value
 - Atemporal
 - Time only plays a minor role compared to the imperative paradigm
 - Applicative
 - All computations are done by applying (calling) functions
 - The natural abstraction is the function
 - Abstracts a single expression to a function which can be evaluated as an expression
 - Functions are first class values
 - Functions are full-fledged data just like numbers, lists, ...
 - Fits well with computations driven by needs
 - Opens a new world of possibilities

2.3. Overview of the logic paradigm

[Contents](#) [Up](#) [Previous](#) [Next](#) [Slide](#) [Subject index](#) [Program index](#) [Exercise index](#)

The logic paradigm is dramatically different from the other three main programming paradigms. The logic paradigm fits extremely well when applied in problem domains that deal with the extraction of knowledge from basic facts and relations. The logical paradigm seems less natural in the more general areas of computation.

**Answer a question via search for
a solution**

Below we briefly characterize the main properties of the logic programming paradigm.

- Characteristics:
 - Discipline and idea
 - Automatic proofs within artificial intelligence
 - Based on axioms, inference rules, and queries.
 - Program execution becomes a systematic search in a set of facts, making use of a set of inference rules

2.4. Overview of the object-oriented paradigm

[Contents](#) [Up](#) [Previous](#) [Next](#) [Slide](#) [Subject index](#) [Program index](#) [Exercise index](#)

The object-oriented paradigm has gained great popularity in the recent decade. The primary and most direct reason is undoubtedly the strong support of encapsulation and the logical grouping of program aspects. These properties are very important when programs become larger and larger.

The underlying, and somewhat deeper reason to the success of the object-oriented paradigm is probably the conceptual anchoring of the paradigm. An object-oriented program is constructed with the outset in concepts, which are important in the problem domain of interest. In that way, all the necessary technicalities of programming come in second row.

Send messages between objects to simulate the temporal evolution of a set of real world phenomena

As for the other main programming paradigms, we will now describe the most important properties of object-oriented programming, seen as a school of thought in the area of computer programming.

- Characteristics:
 - Discipline and idea
 - The theory of concepts, and models of human interaction with real world phenomena
 - Data as well as operations are encapsulated in objects
 - Information hiding is used to protect internal properties of an object
 - Objects interact by means of message passing
 - A metaphor for applying an operation on an object

- In most object-oriented languages objects are grouped in classes
 - Objects in classes are similar enough to allow programming of the classes, as opposed to programming of the individual objects
 - Classes represent concepts whereas objects represent phenomena
- Classes are organized in inheritance hierarchies
 - Provides for class extension or specialization

This ends the overview of the four main programming paradigms. From now on the main focus will be functional programming in Scheme, with special emphasis on examples drawn from the domain of web program development.

Generated: Tuesday July 2, 2013,
09:27:19



5.1.16 Real Programmers (Ed Post), see also Sec. 2.1

First reference occurs in *Real Programmers use FORTRAN*, see Section 2.1 on page 10.

5.1.17 Seven Topics in Python, see also Sec. 5.1.1

First reference occurs in *Seven Topics in Python*, see Section 5.1.1 on page 28.

5.1.18 Seven Topics in Python (Haug-Warberg), see also Sec. 5.1.1

First reference occurs in *Seven Topics in Python*, see Section 5.1.1 on page 28.

Epydoc

Automatic API Documentation Generation for Python

Overview

Epydoc is a tool for generating API documentation for Python modules, based on their docstrings. For an example of epydoc's output, see the API documentation for epydoc itself ([html](#), [pdf](#)). A lightweight markup language called [epytext](#) can be used to format docstrings, and to add information about specific fields, such as parameters and instance variables. Epydoc also understands docstrings written in [reStructuredText](#), Javadoc, and plaintext. For a more extensive example of epydoc's output, see the API documentation for [Python 2.5](#).

Latest Release

The latest stable release is [Epydoc 3.0](#). If you wish to keep up on the latest developments, you can also get epydoc from the [subversion repository](#). See [Installing Epydoc](#) for more information.

Screenshots



News

Epydoc 3.0 released [January 2008]

Epydoc version 3.0 is now available on the [SourceForge download page](#). See the [What's New](#) page for details. Epydoc is under active development; if you wish to keep up on the latest developments, you can get epydoc from the [subversion repository](#). If you find any bugs, or have suggestions for improving it, please report them on sourceforge.

Presentation at PyCon [March 2004]

Epydoc was presented at [PyCon](#) by Edward Loper. [Video and audio from the presentation](#) are available for download.

Documentation

Epydoc manual

- [Installing Epydoc](#)
- [Using Epydoc](#)
- [Python Docstrings](#)
- [The Epytext Markup Language](#)
- [Epydoc Fields](#)
- [reStructuredText and Javadoc](#)
- [Reference Documentation](#)

Related Information

- [Open Source License](#)
- [Change Log](#)
- [History](#)
- [Future Directions](#)
- [Related Projects](#)
- [Regression Tests](#)

API Documentation

Frequently Asked Questions

Feedback

- [Report a bug](#)
- [Suggest a feature](#)
- Author: [Edward Loper](#)

[Home](#)

[Installing Epydoc](#)

[Using Epydoc](#)

[Epytext](#)

[SOURCEFORGE.NET](#)

The Epytext Markup Language

A Brief Introduction

Epytext is a simple lightweight markup language that lets you add formatting and structure to docstrings. Epydoc uses that formatting and structure to produce nicely formatted API documentation. The following example (which has an unusually high ratio of documentation to code) illustrates some of the basic features of epytext:

```
def x_intercept(m, b):
    """
    Return the x intercept of the line M{y=m*x+b}.  The X{x intercept}
    of a line is the point at which it crosses the x axis (M{y=0}).

    This function can be used in conjunction with L{z_transform} to
    find an arbitrary function's zeros.

    @type m: number
    @param m: The slope of the line.
    @type b: number
    @param b: The y intercept of the line.  The X{y intercept} of a
              line is the point at which it crosses the y axis (M{x=0}).
    @rtype: number
    @return: the x intercept of the line M{y=m*x+b}.
    """
    return -b/m
```

You can compare this function definition with the [API documentation](#) generated by epydoc. Note that:

- Paragraphs are separated by blank lines.
- Inline markup has the form " $x\{\dots\}$ ", where " x " is a single capital letter. This example uses inline markup to mark mathematical expressions (" $M\{\dots\}$ "); terms that should be indexed (" $X\{\dots\}$ "); and links to the documentation of other objects (" $L\{\dots\}$ ").
- Descriptions of parameters, return values, and types are marked with "`@field:`" or "`@field arg:`", where "field" identifies the kind of description, and "arg" specifies what object is described.

Epytext is intentionally very lightweight. If you wish to use a more expressive markup language, I recommend [reStructuredText](#).

Epytext Language Overview

Epytext is a lightweight markup language for Python docstrings. The epytext markup language is used by epydoc to parse docstrings and create structured API documentation. Epytext markup is broken up into the following categories:

- **Block Structure** divides the docstring into nested blocks of text, such as *paragraphs* and *lists*.
 - **Basic Blocks** are the basic unit of block structure.
 - **Hierarchical blocks** represent the nesting structure of the docstring.
- **Inline Markup** marks regions of text within a basic block with properties, such as italics and hyperlinks.

Block Structure

Block structure is encoded using indentation, blank lines, and a handful of special character sequences.

- Indentation is used to encode the nesting structure of hierarchical blocks. The indentation of a line is defined as the number of leading spaces on that line; and the indentation of a block is typically the indentation of its first line.
- Blank lines are used to separate blocks. A blank line is a line that only contains whitespace.
- Special character sequences are used to mark the beginnings of some blocks. For example, '-' is used as a bullet for unordered list items, and '>>>' is used to mark [doctest blocks](#).

The following sections describe how to use each type of block structure.

Paragraphs

A paragraph is the simplest type of basic block. It consists of one or more lines of text. Paragraphs must be left justified (i.e., every line must have the same indentation). The following example illustrates how paragraphs can be used:

Docstring Input	Rendered Output
<pre>def example(): """ This is a paragraph. Paragraphs can span multiple lines, and can contain I{inline markup}. This is another paragraph. Paragraphs are separated by blank lines. """ *[...]*</pre>	<p>This is a paragraph. Paragraphs can span multiple lines, and can contain <i>inline markup</i>.</p> <p>This is another paragraph. Paragraphs are separated from each other by blank lines.</p>

Lists

Epytext supports both ordered and unordered lists. A list consists of one or more consecutive *list items* of the same type (ordered or unordered), with the same indentation. Each list item is marked by a *bullet*. The bullet for unordered list items is a single dash character (-). Bullets for ordered list items consist of a series of numbers followed by periods, such as 1.2. or 1.2.8..

List items typically consist of a bullet followed by a space and a single paragraph. The paragraph may be indented more than the list item's bullet; often, the paragraph is intended two or three characters, so that its left margin lines up with the right side of the bullet. The following example illustrates a simple ordered list.

Docstring Input	Rendered Output
<pre>def example(): """ 1. This is an ordered list item. 2. This is a another ordered list item. 3. This is a third list item. Note that the paragraph may be indented more than the bullet. """ *[...]*</pre>	<ol style="list-style-type: none">1. This is an ordered list item.2. This is another ordered list item.3. This is a third list item. Note that the paragraph may be indented more than the bullet.

List items can contain more than one paragraph; and they can also contain sublists, *literal blocks*, and *doctest blocks*. All of the blocks contained by a list item must all have equal indentation, and that indentation must be greater than or equal to the indentation of the list item's bullet. If the first contained block is a paragraph, it may appear on the same line as the bullet, separated from the bullet by one or more spaces, as shown in the previous example. All other block types must follow on separate lines.

Every list must be separated from surrounding blocks by indentation:

Docstring Input	Rendered Output
<pre>def example(): """ This is a paragraph. 1. This is a list item. 2. This a second list item. - This is a sublist """ [...]</pre>	<p>This is a paragraph.</p> <ol style="list-style-type: none">1. This is a list item.2. This is a second list item.<ul style="list-style-type: none">- This is a sublist.

Note that sublists must be separated from the blocks in their parent list item by indentation. In particular, the following docstring generates an error, since the sublist is not separated from the paragraph in its parent list item by

indentation:

Docstring Input	Rendered Output
<pre>def example(): """ 1. This is a list item. Its paragraph is indented 7 spaces. - This is a sublist. It is indented 7 spaces. """ #[...]</pre>	L5: Error: Lists must be indented.

The following example illustrates how lists can be used:

Docstring Input	Rendered Output
<pre>def example(): """ This is a paragraph. 1. This is a list item. - This is a sublist. - The sublist contains two items. - The second item of the sublist has its own sublist. 2. This list item contains two paragraphs and a doctest block. >>> print 'This is a doctest block' This is a doctest block This is the second paragraph. """ #[...]</pre>	<p>This is a paragraph.</p> <ol style="list-style-type: none">1. This is a list item.<ul style="list-style-type: none">o This is a sublist.o The sublist contains two items.<ul style="list-style-type: none">■ The second item of the sublist has its own own sublist.2. This list item contains two paragraphs and a doctest block. <div style="border: 1px solid black; padding: 5px; margin-top: 10px;">>>> print 'This is a doctest block' This is a doctest block</div> <p>This is the second paragraph.</p>

Epytext will treat any line that begins with a bullet as a list item. If you want to include bullet-like text in a paragraph, then you must either ensure that it is not at the beginning of the line, or use [escaping](#) to prevent epytext from treating it as markup:

Docstring Input	Rendered Output
<pre>def example(): """ This sentence ends with the number 1. Epytext can't tell if the "1." is a bullet or part of the paragraph, so it generates an error. """ #[...]</pre>	L4: Error: Lists must be indented.
<pre>def example(): """ This sentence ends with the number 1. This sentence ends with the number E{1}. """ #[...]</pre>	<p>This sentence ends with the number 1.</p> <p>This sentence ends with the number 1.</p>

Sections

A section consists of a heading followed by one or more child blocks.

- The heading is a single underlined line of text. Top-level section headings are underlined with the '=' character; subsection headings are underlined with the '-' character; and subsubsection headings are underlined with the '~' character. The length of the underline must exactly match the length of the heading.
- The child blocks can be paragraphs, lists, literal blocks, doctest blocks, or sections. Each child must have

equal indentation, and that indentation must be greater than or equal to the heading's indentation.

The following example illustrates how sections can be used:

Docstring Input	Rendered Output
<pre>def example(): """ This paragraph is not in any section. Section 1 ====== This is a paragraph in section 1. Section 1.1 ----- This is a paragraph in section 1.1. Section 2 ====== This is a paragraph in section 2. """ #[...]</pre>	<h2>Section 1</h2> <p>This is a paragraph in section 1.</p> <h3>Section 1.1</h3> <p>This is a paragraph in section 1.1.</p> <h2>Section 2</h2> <p>This is a paragraph in section 2.</p>

Literal Blocks

Literal blocks are used to represent "preformatted" text. Everything within a literal block should be displayed exactly as it appears in plaintext. In particular:

- Spaces and newlines are preserved.
- Text is shown in a monospaced font.
- Inline markup is not detected.

Literal blocks are introduced by paragraphs ending in the special sequence "`::`". Literal blocks end at the first line whose indentation is equal to or less than that of the paragraph that introduces them. The following example shows how literal blocks can be used:

Docstring Input	Rendered Output
<pre>def example(): """ The following is a literal block::: Literal / / Block This is a paragraph following the literal block. """ #[...]</pre>	<p>The following is a literal block:</p> <p> Literal / / Block</p> <p>This is a paragraph following the literal block.</p>

Literal blocks are indented relative to the paragraphs that introduce them; for example, in the previous example, the word "Literal" is displayed with four leading spaces, not eight. Also, note that the double colon ("`::`") that introduces the literal block is rendered as a single colon.

Doctest Blocks

Doctest blocks contain examples consisting of Python expressions and their output. Doctest blocks can be used by the doctest module to test the documented object. Doctest blocks begin with the special sequence "`>>>`". Doctest blocks are delimited from surrounding blocks by blank lines. Doctest blocks may not contain blank lines. The following example shows how doctest blocks can be used:

Docstring Input	Rendered Output
<pre>def example(): """ The following is a doctest block: >>> print (1+3, ... 3+5) (4, 8) >>> 'a-b-c-d-e'.split('-')</pre>	<p>The following is a doctest block:</p> <pre>>>> print (1+3, ... 3+5) (4, 8) >>> 'a-b-c-d-e'.split('-')</pre>

```

...
      3+5)
(4, 8)
>>> 'a-b-c-d-e'.split('-')
['a', 'b', 'c', 'd', 'e']

This is a paragraph following the
doctest block.
"""

#[...]

```

```
[ 'a', 'b', 'c', 'd', 'e' ]
```

This is a paragraph following the doctest block.

Fields

Fields are used to describe specific properties of a documented object. For example, fields can be used to define the parameters and return value of a function; the instance variables of a class; and the author of a module. Each field is marked by a *field tag*, which consist of an at sign ('@') followed by a *field name*, optionally followed by a space and a *field argument*, followed by a colon (:). For example, '@return:' and '@param x:' are field tags.

Fields can contain paragraphs, lists, literal blocks, and doctest blocks. All of the blocks contained by a field must all have equal indentation, and that indentation must be greater than or equal to the indentation of the field's tag. If the first contained block is a paragraph, it may appear on the same line as the field tag, separated from the field tag by one or more spaces. All other block types must follow on separate lines.

Fields must be placed at the end of the docstring, after the description of the object. Fields may be included in any order.

Fields do not need to be separated from other blocks by a blank line. Any line that begins with a field tag followed by a space or newline is considered a field.

The following example illustrates how fields can be used:

Docstring Input	Rendered Output
<pre> def example(): """ @param x: This is a description of the parameter x to a function. Note that the description is indented four spaces. @type x: This is a description of x's type. @return: This is a description of the function's return value. It contains two paragraphs. """ #[...] </pre>	<p>Parameters: x - This is a description of the parameter x to a function. Note that the description is indented four spaces. <i>(type=This is a description of x's type.)</i></p> <p>Returns: This is a description of the function's return value. It contains two paragraphs.</p>

For a list of the fields that are supported by epydoc, see the *epydoc fields* chapter.

Inline Markup

Inline markup has the form '**x{...}**', where **x** is a single capital letter that specifies how the text between the braces should be rendered. Inline markup is recognized within paragraphs and section headings. It is *not* recognized within literal and doctest blocks. Inline markup can contain multiple words, and can span multiple lines. Inline markup may be nested.

A matching pair of curly braces is only interpreted as inline markup if the left brace is immediately preceded by a capital letter. So in most cases, you can use curly braces in your text without any form of escaping. However, you do need to escape curly braces when:

1. You want to include a single (un-matched) curly brace.
2. You want to preceed a matched pair of curly braces with a capital letter.

Note that there is no valid Python expression where a pair of matched curly braces is immediately preceeded by a capital letter (except within string literals). In particular, you never need to escape braces when writing Python dictionaries. See also [escaping](#).

Basic Inline Markup

Epytext defines four types of inline markup that specify how text should be displayed:

- `I{...}`: Italicized text.
- `B{...}`: Bold-faced text.
- `C{...}`: Source code or a Python identifier.
- `M{...}`: A mathematical expression.

By default, source code is rendered in a fixed width font; and mathematical expressions are rendered in italics. But those defaults may be changed by modifying the CSS stylesheet. The following example illustrates how the four basic markup types can be used:

Docstring Input	Rendered Output
<pre>def example(): """ I{B{Inline markup} may be nested; and it may span} multiple lines. - I{Italicized text} - B{Bold-faced text} - C{Source code} - M{Math} Without the capital letter, matching braces are not interpreted as markup: C{my_dict={1:2, 3:4}}. """ #[...]</pre>	<p>Inline markup may be nested; and it may span multiple lines.</p> <ul style="list-style-type: none">• <i>Italicized text</i>• Bold-faced text• Source code• Math: m^*x+b <p>Without the capital letter, matching braces are not interpreted as markup: <code>my_dict={1:2, 3:4}</code>.</p>

URLs

The inline markup construct `U{text<url>}` is used to create links to external URLs and URIs. '`text`' is the text that should be displayed for the link, and '`url`' is the target of the link. If you wish to use the URL as the text for the link, you can simply write "`U{url}`". Whitespace within URL targets is ignored. In particular, URL targets may be split over multiple lines. The following example illustrates how URLs can be used:

Docstring Input	Rendered Output
<pre>def example(): """ - U{www.python.org} - U{http://www.python.org} - U{The epydoc homepage<http:// epydoc.sourceforge.net>} - U{The B{Python} homepage <www.python.org>} - U{Edward Loper<mailto:edloper@ gradient.cis.upenn.edu>} """ #[...]</pre>	<ul style="list-style-type: none">• www.python.org• http://www.python.org• The epydoc homepage• The Python homepage• Edward Loper

Documentation Crossreference Links

The inline markup construct '`L{text<object>}`' is used to create links to the documentation for other Python objects. '`text`' is the text that should be displayed for the link, and '`object`' is the name of the Python object that should be linked to. If you wish to use the name of the Python object as the text for the link, you can simply write `L{object}```. Whitespace within object names is ignored. In particular, object names may be split over multiple lines. The following example illustrates how documentation crossreference links can be used:

Docstring Input	Rendered Output
<pre>def example(): """ - L{x_transform} - L{search<re.search>} - L{The I{x-transform} function}</pre>	<ul style="list-style-type: none">• x_transform• search• The x-transform function

```
<x_transform>
"""
#[...]
```

In order to find the object that corresponds to a given name, epydoc checks the following locations, in order:

1. If the link is made from a class or method docstring, then epydoc checks for a method, instance variable, or class variable with the given name.
2. Next, epydoc looks for an object with the given name in the current module.
3. Epydoc then tries to import the given name as a module. If the current module is contained in a package, then epydoc will also try importing the given name from all packages containing the current module.
4. Epydoc then tries to divide the given name into a module name and an object name, and to import the object from the module. If the current module is contained in a package, then epydoc will also try importing the module name from all packages containing the current module.
5. Finally, epydoc looks for a class name in any module with the given name. This is only returned if there is a single class with such name.

If no object is found that corresponds with the given name, then epydoc issues a warning.

Indexed Terms

Epydoc automatically creates an index of term definitions for the API documentation. The inline markup construct '`x{...}`' is used to mark terms for inclusion in the index. The term itself will be italicized; and a link will be created from the index page to the location of the term in the text. The following example illustrates how index terms can be used:

Docstring Input	Rendered Output								
<pre>def example(): """ An X{index term} is a term that should be included in the index. """ #[...]</pre>	<p>An <i>index term</i> is a term that should be included in the index.</p> <table border="1"> <thead> <tr> <th colspan="2">Index</th> </tr> </thead> <tbody> <tr> <td>index term</td><td><i>example</i></td></tr> <tr> <td>x intercept</td><td><i>x_intercept</i></td></tr> <tr> <td>y intercept</td><td><i>x_intercept</i></td></tr> </tbody> </table>	Index		index term	<i>example</i>	x intercept	<i>x_intercept</i>	y intercept	<i>x_intercept</i>
Index									
index term	<i>example</i>								
x intercept	<i>x_intercept</i>								
y intercept	<i>x_intercept</i>								

Symbols

Symbols are used to insert special characters in your documentation. A symbol has the form '`s{code}`', where code is a symbol code that specifies what character should be produced. The following example illustrates how symbols can be used to generate special characters:

Docstring Input	Rendered Output
<pre>def example(): """ Symbols can be used in equations: - S{sum}S{alpha}/x S{<=} S{beta} S{<-} and S{larr} both give left arrows. Some other arrows are S{rarr}, S{uarr}, and S{darr}. """ #[...]</pre>	<p>Symbols can be used in equations:</p> <ul style="list-style-type: none"> • $\sum \alpha/x \leq \beta$ <p>\leftarrow and \leftarrow both give left arrows. Some other arrows are \rightarrow, \uparrow, and \downarrow.</p>

Although symbols can be quite useful, you should keep in mind that they can make it harder to read your docstring in plaintext. In general, symbols should be used sparingly. For a complete list of the symbols that are currently supported, see the reference documentation for [epytext.SYMBOLS](#).

Escaping

Escaping is used to write text that would otherwise be interpreted as epytext markup. Epytext was carefully constructed to minimize the need for this type of escaping; but sometimes, it is unavoidable. Escaped text has the form '`E{code}`', where code is an escape code that specifies what character should be produced. If the escape code is a single character (other than '{' or '}'), then that character is produced. For example, to begin a paragraph with a

dash (which would normally signal a list item), write '`E{-}`'. In addition, two special escape codes are defined: '`E{lb}`' produces a left curly brace ('{''); and '`E{rb}`' produces a right curly brace ('}''). The following example illustrates how escaping can be used:

Docstring Input	Rendered Output
<pre>def example(): """ This paragraph ends with two colons, but does not introduce a literal blockE{ : }E{ : } E{-} This is not a list item. Escapes can be used to write unmatched curly braces: E{rb}E{lb} """ #[...]</pre>	<p>This paragraph ends with two colons, but does not introduce a literal block::</p> <ul style="list-style-type: none"> - This is not a list item. <p>Escapes can be used to write unmatched curly braces: }{</p>

Graphs

The inline markup construct '`G{graphtype args...}`' is used to insert automatically generated graphs. The following graphs generation constructions are currently defines:

Markup	Description
<code>G{classtree classes...}</code>	Display a class hierarchy for the given class or classes (including all superclasses & subclasses). If no class is specified, and the directive is used in a class's docstring, then that class's class hierarchy will be displayed.
<code>G{packagetree modules...}</code>	Display a package hierarchy for the given module or modules (including all subpackages and submodules). If no module is specified, and the directive is used in a module's docstring, then that module's package hierarchy will be displayed.
<code>G{importgraph modules...}</code>	Display an import graph for the given module or modules. If no module is specified, and the directive is used in a module's docstring, then that module's import graph will be displayed.
<code>G{callgraph functions...}</code>	Display a call graph for the given function or functions. If no function is specified, and the directive is used in a function's docstring, then that function's call graph will be displayed.

Characters

Valid Characters

Valid characters for an epytext docstring are space (\040); newline (\012); and any letter, digit, or punctuation, as defined by the current locale. Control characters (\000-\010` and ``\013-\037) are not valid content characters. Tabs (\011) are expanded to spaces, using the same algorithm used by the Python parser. Carridge-return/newline pairs (\015\012) are converted to newlines.

Content Characters

Characters in a docstring that are not involved in markup are called *content characters*. Content characters are always displayed as-is. In particular, HTML codes are not passed through. For example, consider the following example:

Docstring Input	Rendered Output
<pre>def example(): """ test</pre>	<p>test</p>

```
"""  
#[...]
```

The docstring is rendered as `test`, and not as the word "test" in bold face.

Spaces and Newlines

In general, spaces and newlines within docstrings are treated as soft spaces. In other words, sequences of spaces and newlines (that do not contain a blank line) are rendered as a single space, and words may wrapped at spaces. However, within literal blocks and doctest blocks, spaces and newlines are preserved, and no word-wrapping occurs; and within URL targets and documentation link targets, whitespace is ignored.

[Home](#)

[Installing Epydoc](#)

[Using Epydoc](#)

[Epytext](#)

SOURCEFORGE
.net



Python Docstrings

Python documentation strings (or *docstrings*) provide a convenient way of associating documentation with Python modules, functions, classes, and methods. An object's docstring is defined by including a string constant as the first statement in the object's definition. For example, the following function defines a docstring:

```
def x_intercept(m, b):
    """
    Return the x intercept of the line y=m*x+b.  The x intercept of a
    line is the point at which it crosses the x axis (y=0).
    """
    return -b/m
```

Docstrings can be accessed from the interpreter and from Python programs using the "`__doc__`" attribute:

```
>>> print x_intercept.__doc__
Return the x intercept of the line y=m*x+b.  The x intercept of a
line is the point at which it crosses the x axis (y=0).
```

The [pydoc](#) module, which became part of [the standard library](#) in Python 2.1, can be used to display information about a Python object, including its docstring:

```
>>> from pydoc import help

>>> help(x_intercept)
Help on function x_intercept in module __main__:

x_intercept(m, b)
    Return the x intercept of the line y=m*x+b.  The x intercept of a
    line is the point at which it crosses the x axis (y=0).
```

For more information about Python docstrings, see the [Python Tutorial](#) or the O'Reilly Network article [Python Documentation Tips and Tricks](#).

Variable docstrings

Python don't support directly docstrings on variables: there is no attribute that can be attached to variables and retrieved interactively like the `__doc__` attribute on modules, classes and functions.

While the language doesn't directly provides for them, Epydoc supports *variable docstrings*: if a variable assignment statement is immediately followed by a bare string literal, then that assignment is treated as a docstring for that variable. In classes, variable assignments at the class definition level are considered class variables; and assignments to instance variables in the constructor (`__init__`) are considered instance variables:

```

class A:
    x = 22
    """Docstring for class variable A.x"""

    def __init__(self, a):
        self.y = a
        """Docstring for instance variable A.y"""

```

Variables may also be documented using *comment docstrings*. If a variable assignment is immediately preceded by a comment whose lines begin with the special marker '#:', or is followed on the same line by such a comment, then it is treated as a docstring for that variable:

```

#: docstring for x
x = 22
x = 22 #: docstring for x

```

Notice that variable docstrings are only available for documentation when the source code is available for *parsing*: it is not possible to retrieve variable

Items visibility

Any Python object (modules, classes, functions, variables...) can be *public* or *private*. Usually the object name decides the object visibility: objects whose name starts with an underscore and doesn't end with an underscore are considered private. All the other objects (including the "magic functions" such as `__add__`) are public.

For each module and class, Epydoc generates pages with both public and private methods. A Javascript snippet allows you to toggle the visibility of private objects.

If a module wants to hide some of the objects it contains (either defined in the module itself or imported from other modules), it can explicitly list the names if its [public names](#) in the `__all__` variable.

If a module defines the `__all__` variable, Epydoc uses its content to decide if the module objects are public or private.

[Home](#)

[Installing
Epydoc](#)

[Using Epydoc](#)

[Epytext](#)





NumPy

Scipy.org

NumPy

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- tools for integrating C/C++ and Fortran code
- useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data. Arbitrary data-types can be defined. This allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

Numpy is licensed under the [BSD license](#), enabling reuse with few restrictions.

Getting Started

- [Getting Numpy](#)
- [Installing the SciPy Stack](#)
- [NumPy and SciPy documentation page](#)
- [NumPy Tutorial](#)
- [NumPy for MATLAB® Users](#)
- [NumPy functions by category](#)
- [NumPy Mailing List](#)

More Information

- [NumPy Sourceforge Home Page](#)
- [SciPy Home Page](#)
- [Interfacing with compiled code](#)
- [Older python array packages](#)

About Numpy

[License](#)

[Old array packages](#)

© Copyright 2013 Numpy developers. Created using [Sphinx](#) 1.1.3.

Exercise 1

Preisig, H A

Chemical Engineering, NTNU

Topology :

A jacketed stirred tank – no mixing effects

Question

Suggest a very simple physical topology for a jacketed stirred tank. Label all components of the graph.

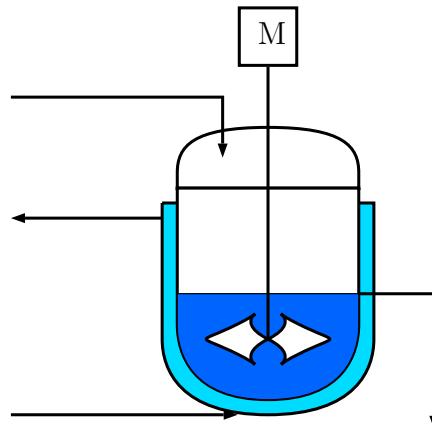


Figure 1: A simple jacketed stirred tank reactor with an overflow

Topology: Tank with external tubular heat exchanger

Question

Suggest a very simple physical topology for a stirred tank with a heat exchanger in the form of half-pipes welded onto the surface. Label all components of the graph.

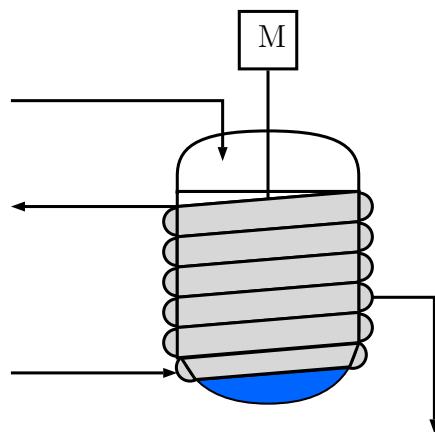


Figure 2: A simple stirred tank reactor with a welded-on heat exchanger and an overflow

Topology:

A jacketed stirred tank – with mixing effects

Question:

Suggest for a jacketed stirred tank reactor a topology that includes some distribution effects in the jacket and a network mixing model for the liquid contents.

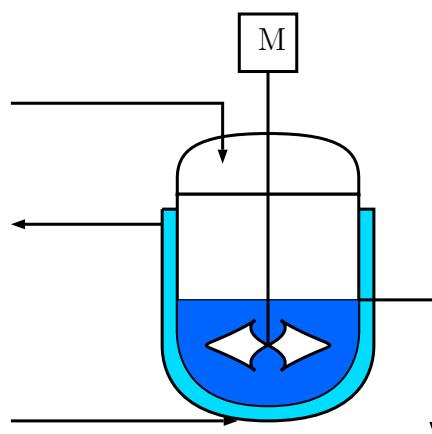


Figure 3: A simple jacketed stirred tank reactor with an overflow

Topology: A generic simple heat exchanger

Question

Suggest a very simple topology for a generic heat exchanger.

Topology: A single-tube heat exchanger

Question

Suggest a topology that approximates the behaviour of a single-tube heat exchanger by "slicing" the tubes and thus generate a chain of paired lumped systems.

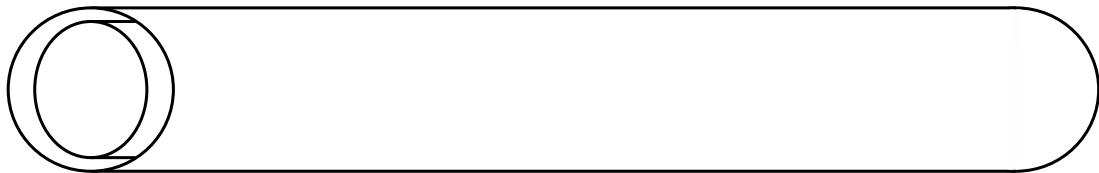


Figure 4: A basic single-tube heat exchanger

Topology: A simple distillation tray model

Question

Suggest a simple topology for a distillation tray.

Topology: Butter in frying pan

Question

Suggest a topology that describes the behaviour of a piece of butter melting in the pan.

Purpose: how long does it take to melt it.

Regular Expression Search-and-replace (TKP4106)



Zooball/Dove

"There is no reason anyone would want a computer in their home."

Ken Olsen, founder of DEC (1977)

Assignments

1. Read [A Smalltalk about Modelling](#). The paper explains some of the reasons why you should learn about computer languages in your natural science study.
2. Install either Vim, Emacs, Smultron or TextPad on your computer. Change the color preferences to light grey or pastel background, black text and low brightness highlight colors. Never use a gleaming white background and bright red, blue, green, etc. colors. The contrast will affect your eyes badly. The reason is that you will at times be staring very intensively on the screen for a long time to think hard about an algorithm or to find a bug. Now, this work mode is very different from what you have experienced before using e.g. word processors so you must learn to take care of your eyes!
3. Convert [critical data](#) from XML (eXtensible Markup language) to CSV (Comma Separated Variables) format. Often, it is safer to use semicolon rather than comma as the field separator, especially if the fields themselves contain commas (like many chemical component names do). Or, you can enclose the field name in double quotes and still use comma as the separator.

Note: There is a difference in line endings on Windows (carriage return + newline), Mac (carriage return) and Unix (newline). In computer jargon these characters are given ASCII codes 13 (CR) and 10 (NL) respectively. Their regular expression equivalents are \r and \n. Modern editors are aware this problem and you can change the newline character(s) to whatever you like before saving the file. This will become important when you are matching strings that span several lines in the file.

XML belongs to a world of its own, but we do not need to know much about the language to solve this task. We only need to identify the repetitive pattern that are used to store our

our data. The characteristic encoding of the XML-file is:

```
<?xml version="1.0" encoding="UTF-8"?>
<?doc id="" time="Wed Apr 23 21:51:57 CEST 2008" file="foobar"?>
<new class="Hash" length="464">
<hash class="String" value="ACETIC ANHYDRIDE"/>
<push class="Array" length="4">
<push class="DBpair" name="tc" value="569 K Reid77"/>
<push class="DBpair" name="pc" value="46.2 atm Reid77"/>
<push class="DBpair" name="vc" value="290 cc mol^{-1} Reid77"/>
<push class="DBpair" name="zc" value="0.287 Reid77"/>
</push>
<hash class="String" value="2,4-DIMETHYLHEXANE"/>
<push class="Array" length="4">
...
</new>
```

The output shall be on the form:

```
Name, Tc, Pc, Vc, Zc
, K, atm, cc mol^{-1},
"ACETIC ANHYDRIDE", 569, 46.2, 290, 0.287
...
```

4. Convert all files in [Archive](#) from their non-standard in-house format to CSV format.

In programming, working with multiple source files is more like a rule than an exception. For a couple of files I would probably edit the changes by hand, but if the files grows in number to 5 or maybe 10 I would definitely look for a pattern to see if it is possible to make simultaneous changes to all the files. The encoding of the data files does in this case follow a very simple pattern:

```
DALEX76B
Alexandrov, A.A., Khasanahin, T.S., and Larkin, D.K.
Paper to the Working Group 1 of the IAPS, Kyoto, Japan, (1976).
T90(K) P(MPa) d(kg/m3)
96
423.114 55.568000 945.20639
423.114 40.152000 938.01591
...
```

The output shall be on the form:

```
T90, P, d
K, MPa, kg/m3
423.114, 55.568000, 945.20639
423.114, 40.152000, 938.01591
...
```

5. Make sure the output files can be opened without trouble in Excel or OpenOffice.

Regular expressions belong to the simplest of all languages. An excerpt from Wikipedia informs us that: "In computing, a regular expression, also referred to as regex or regexp, provides a concise and flexible means for matching strings

of text, such as particular characters, words, or patterns of characters. A regular expression is written in a formal language that can be interpreted by a regular expression processor." Regular expressions are of widespread use for analyzing text, defining programming language syntax and for generic search-and-replace in editors. A very short overview of the basic commands is given below:

```
^      Start of a string
$      End of a string
.      Any character (except \n)
*      0 or more of previous expression
+      1 or more of previous expression
?      0 or 1 of previous expression

\w     Matches any word character
\W     Matches any non-word character
\s     Matches any white-space character
\S     Matches any non-white-space character
\d     Matches any decimal digit
\D     Matches any nondigit

[abc]  Matches any single character included in the set
[^abc]  Matches any single character not in the set
[a-z]   Contiguous character ranges
(a|b)   a or b
ab{2}   Matches two b characters

(expr) Makes a backreference of whatever is matched.
         The backreference is made available as \1 or $1
         in many search-and-replace routines.
```

A few examples follow. The text string we want to analyze is: "Hello TKP4106!"

```
^.*$          Matches 'Hello TKP4106!'
^[a-zA-Z0-9 !]*$  Matches 'Hello TKP4106!'
^.*(o T).*$    Matches 'Hello TKP4106!' (\1=>'o T')
\w+            Matches 'Hello'
\s\w+          Matches ' TKP4106'
\d+            Matches '4106'
\w             Matches ''
\w*(\w+)\w*(\w+)  Matches 'Hello TKP4106!' (\1=> ' ' and \2=> ' !')
```

You remember maybe the "burglar's language" from your childhood? It was a simple translation of all consonants b, c, d, etc. into bob, coc, dod, etc. So, "Python" would become "popytothohonon". This is hard for your tongue but it is very easy to achieve with regular expressions:

```
Search for: ([^aeiouy\W])
Replace by: \lo\1
```

There are tons of regex documentation on the Web. This link to [Regular](#)

[Expressions](#) seems quite OK. Note, however, that there are many flavors of regular expressions and that the syntax can (will) differ when you switch between two different editors, operating systems or programming languages.

A Smalltalk[†] about modelling

Tore Haug-Warberg
Department of Chemical Engineering
NTNU (Norway)

5 June 2009

1 Background

Humanity is deeply rooted in the Enlightenment and the contemporary search for a rational description of Nature. Man is the only animal on planet Earth that systematically investigate, interpret, and employ the basic laws of nature to his own benefit. It is not too much to say that the understanding of the laws of nature has paved our road to technological success, and to proliferation beyond our own control. But, notwithstanding the tremendous success we have had with our technology there is always room for a better understanding of natural phenomena and in particular those of complex nature.

We tend to think that a complex system must be technically intricate as well. That is wrong. For example: Life at the kitchen sink is quite simple (technically), but at the same time so complex (mathematically) that it is possible to enjoy a full academic career trying to explain all the physical phenomena that are observed: Drop formation, water twirls, shock fronts, bubble coalescence, foams, vortices, etc. This daily experience, which we rarely appreciate, is quite contrary to the situation in the laboratory. There, we try to eliminate all random factors in order to understand one particular phenomenon. The outcome of the study can be a measured value of some kind, or the input to a refined *model* of the phenomenon being studied. Actually, the old saying “seeing is believing” is for us akin to “observing is explaining”. Every observable physical phenomenon must find a rational explanation. There is no easy escape from this dilemma because we believe so hard in our present understanding of the physics. But, there are unsurmountable problems in explaining all the nitty-gritty details of Nature. We pretend, therefore, that our models are too simple still.

Collecting many small pieces of information make us able to understand and model parts of the world around us. At this point the use of computers has strengthened our capabilities of formulating and solving complex physico-mathematical models for a diverse set of industrial operations like fluid transport, chemical reaction, separation, casting, electrolysis, extrusion and rolling. The continuum description of a full-sized control volume with stress-strain interactions and complicated geometry may now be formulated

[†]Smalltalk is a purely object-oriented programming language invented in the 1980s. It has later inspired the development of Ruby—a modern scripting language of the same breed as Perl and Python.

and solved as systems of equations with millions of unknowns. Weather forecasting is maybe the ultimate example.

2 Computer science

Modelling does also depend on numerical issues like rounding error, computation speed, memory capacity and discretization schemes. Focus is thereby lifted from the understanding of the laws of nature to the understanding of numerics and computer languages. Most important maybe, is the observation that a physical model can be refined indefinitely without coming to a full answer of “life, universe and everything”. All models have to give in at some point of refinement. This has to do with the *granularity* of the model. The calculation of fluid flow, for instance, does normally ignore the propagation of sound waves. So, if sound waves are important, the model will fail. It does not matter how many parameters we introduce, or how clever we are tweaking the numbers. It does simply fail. We say that the model must be *validated* against experiments to be trusted. Another unfortunate situation occurs when the model gives consistently wrong results. Changing the direction of gravity for instance would cause a stone to fall upwards. Apart from this flaw all the derived results could be correct. There is no way a computer can understand or check this out without human interaction. The programmer must *verify* that the equations are solved correctly. Our first statement about modelling is therefore:

Validation: The model is made right (experiment decides)

Verification: The right model is made (programmer decides)

The secret is to make sure that the model has the right granularity with respect to what it is supposed to do, and to choose an implementation that makes the best out of the time available and the human resources. The old rule of thumb that one line of code is equivalent to one working hour is still valid. For bigger projects devoted to advanced modelling this number may easily drop to two lines per day. It is impossible to give a totally satisfactory implementation guide to all kinds of physical problems, but it pays to keep a close eye at the physics (mostly conservation laws), the solution methods, and the program structure. Ideally, a physico-mathematical model consists of four main parts:

1. A deterministic* function (the model)
2. Model parameters (perhaps quite many and ill-organized)
3. A numerical solver (normally linearized)
4. Calculated results (vector fields or matrices maybe)

Considering these four parts of the model from the very beginning will inevitably limit the modelling task to comply with the available human resources. But even the best

*Quantum mechanics makes an interesting case in physical modelling since it is not strictly deterministic.

modelling practise gives no clue about *how* the model is going to be used. Should it be a stand-alone tool or made part of a program library? Is it required to make a compiled program or will an interpreted script do? In higher education it would be very beneficial if the joint modelling efforts from all the math and science classes were put into a small *toolbox* that the students could bring *out* from university *into* their future jobs. The current situation is nearly the opposite and that is not prosperous for academia. To shed some light on this topic I shall like to present a somewhat personal view on the links between programming languages, modelling and model uses:

$$\begin{array}{llll} \text{Languages} & \models & \text{Mathematics} & \models \text{Physics} \\ & \models & \text{Modelling} & \models \text{Simulation} \\ & \models & \text{Animation} & \models \text{GUI} \end{array}$$

The binary operator \models means a dependency—in the sense that Mathematics rely on a (formal) Language, Physics rely on Mathematics, Modelling rely on Physics, etc. In the late medieval period European universities taught natural languages (Greek and Latin mostly), medicine, theology and astronomy. About 300 years ago mathematics and physics entered the scene as subjects of their own, while modelling and simulation were not commonplace till after WWII. These subjects were quite early moved out of the university, however, and safely placed in governmental research institutes, mostly connected to defense and aero-space industries. Animation belongs to the computer science era, and Graphical User Interfaces (GUI) had basically to await the introduction of the Windows 3.1 operating system in the late 1980s.

3 Natural sciences

As a consequence of our expanding knowledge it becomes increasingly harder to give priority to one particular subject on the cost of the others. Like Figure 1 says: What is the most important subject to teach first? Languages or GUI? Not an easy question because mathematics is a language of its own and a textbook is a kind of a graphical user interface. Or, perhaps the subjects should be taught in parallel? There are no definite answers to these questions, yet we must choose what to teach, when to teach and how to teach it. It is interesting to note that our education system which started out teaching *natural* languages several hundreds of years ago has by now ended up as a big consumer of *formal* language procedures and computer programs.

Classic knowledge has in a way been replaced by synthetic know-how. Just think about the use of Internet as a platform for collecting and retrieving information. The funny thing is that this change has not been taken into account in the natural science curriculums we see today. Retrospectively, the computer was born in a top secret physics lab but quickly moved out to become an everyday entertainment machine. It shall be our challenge to bring it back into scientific teaching as a mind extender—not a mind boggler. In order to do this we need to understand the buzzwords mentioned above, and we need to make a choice about where we should put our efforts. The worst scenario is doing a little of everything which easily ends up in nothing.

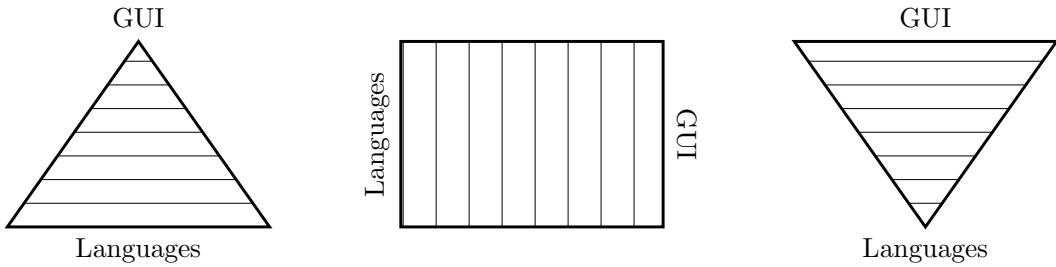


Figure 1: What is the most important subject to teach first? Languages or GUI? Or, perhaps the subjects should be taught in parallel?

Let it be my bold statement that the university must focus on the teaching of formal Languages, Mathematics and Physics. This is a very conservative approach, but on top of this we should introduce Modelling as a separate issue from *day one* at the university. This does not mean that the students shall run commercial software with advanced graphical interfaces. It means, however, that the computer (language) development has come to a point where it is possible to solve (non-linear) physical problems at a pace that was unimaginable 15 years ago. So, rather than talking about models—not to say model simplifications—we can teach the students *how* to model. Our focus can thereby be shifted from mathematical details[†] to physical insight.

At the same time it is important to make a sharp distinction between *modelling* and *simulation*. Modelling is the mathematical description of a physical event into a formal language, while simulation is the systematic use of models to study a complete process. Simulation is great for validation purposes and for our understanding of complex systems, but it should definitely be kept out of the classroom because it does not bring in any new understanding of the basics. The control people may disagree with me here, but I am talking about basics in the sense of physics—not about systems behaviour.

The situation is a somewhat different when it comes to Animation and GUI since these subjects are touched upon already in the elementary school. Moreover, the World Wide Web is a gigantic software enterprise which impossibly can be kept out of the classroom. It is also true that the Ministries of Education worldwide think these topics are especially important, maybe because “seeing is believing”. I believe these simplistic thoughts are harmful, however, because only a small fraction of the resources spent on developing computer games, movies, music and entertainment find its way back to where it all started; namely increasing the knowledge of the world around us. E.g. the *Avatar* (2009) movie, which by all means was a trendsetter, is a good example on how reality and fiction can be seamlessly merged using a good deal of computing power. But, however breathtaking the movie is, it does not increase our understanding of the world around us.

It is also a common misconception that kids in general get very excited, and want to learn science, by simply watching animations and simulations on the computer screen.

[†]The mathematicians do not need to worry. There is plenty of room for a thorough mathematical underpinning in all physical disciplines.

This is simply not true as virtually all students today have watched animated TV programs and fabulous action movies since they were 3 years old. The professors are enthusiastic, but the students think it is downright boring. However, it *is* our duty to teach the students natural sciences, and even though it is sad to watch how the universities in Norway are lacking a good strategy on how to cope with this undertaking—now that we definitely have entered the computer age, we must do something. In my opinion this something should be a mix of traditional mathematics, physics and chemistry, interspersed with modelling as a tool for learning. The second statement about modelling (and computer science in natural science education) is therefore that we should limit our focus to:

$$\text{Languages} \models \text{Mathematics} \models \text{Physics} \models \text{Modelling}$$

It is necessary to put some emphasis on the learning of *formal languages* to understand *what* can be done on a computer, not only *how* it can be done. The common double-clicking-machine is good for everyday surfing on the web and manipulating song lists, but it has nothing to do with scientific computing. The situation today is that all students are trained in their mother tongue, and in one or two foreign languages. This is very good but it is worth a second thought that they are not equally well trained in speaking any of the computer languages. Quite interestingly though, since they may easily spend 3–8 hours behind the screen every day. Some people would claim that there are more than 5000 computer languages today and that the students cannot learn everything, but formal languages are quite simplistic and follow the same basic ideas: Alphabet, vocabulary, syntax and semantics. The crucial point is that the students must learn how to express their thoughts (model = structure + physics + math) in at least *one* such language. To ignore this focus is like traveling to a foreign country without knowing the local lingo: You will be nothing but a tourist. In my opinion students of natural sciences at NTNU should definitely not be computer tourists. They should know how to master their new frontier.

5.3.2 Regular Expressions, see also Sec. 5.1.8

First reference occurs in *Regex* (Stephen Ramsay), see Section 5.1.8 on page 71.

Exercise 2

1 Physical topology 2

1.1 How to do it

Suggest for all the below-defined processes an abstraction showing

- Lumped capacities (storage) of extensive conserved quantities (component mass, mass, energy, momentum) as circles. These are capacities where the intensive properties are assumed constant over the whole spatial domain they occupy. Lumps for which a steady-state assumption is made, that is one assumes the lump to be of negligible capacity, a line is used instead of the circle. Latter assumption is often called a pseudo-steady state assumption. It is a time-scale assumption!
- Connections
 - mass connection with straight lines
 - heat connections as wiggled lines
 - work connections as dashed lines

Show more than one possible abstraction (physical topology) when ever you can and argue why you do it as you do, thus give the assumptions that underlay your topology.

1.2 The different plants

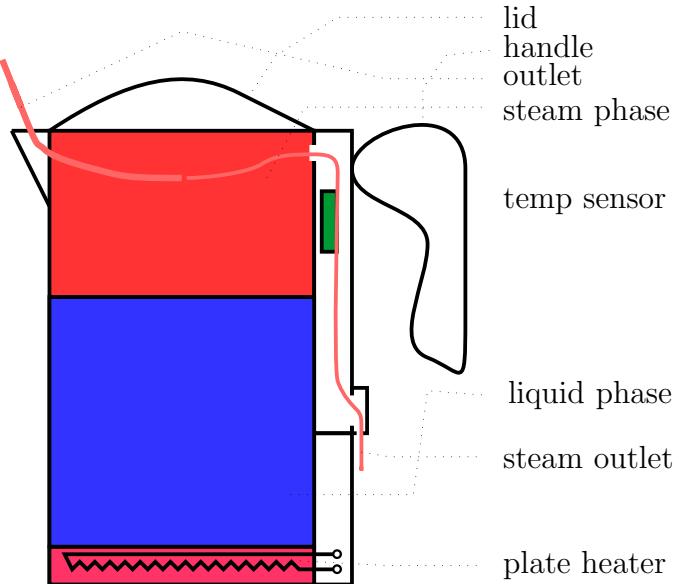
- Kettle, a hot water heater you may have in your kitchen. The kettle is of the type with the heating plate on the bottom.
Purpose: switching it off when water is boiling.
- Transporting fruit in containers is a non-trivial problem as the fruit must breathe and ripens or rots, thus undergoes chemical changes, which are associated with a thermal effect. Latter can be quite significant. Think about a hay stack, for example.
Purpose: Dynamics of the changes in the fruit's quality.
- Hot glass of water.
Purpose 1: Dynamics of cooling down.
Purpose 2: Loosing mass.
- Hot glass of water covered with a lid.
Purpose as above.
- Piece of butter melting in the pan.
Purpose: how long does it take to melt it.

1 Suggested solution

The solutions given below are sample solutions. Such modelling does not have a unique answer, thus the given topologies are possible variations of the theme.

1.1 Kettle

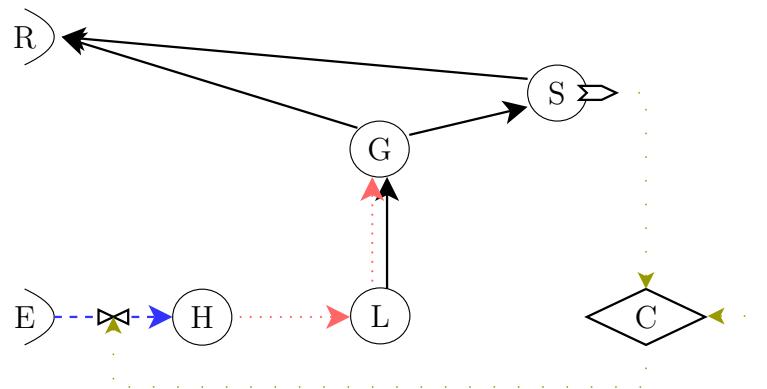
The process is shown in the figure below and so is a possible abstraction.



This abstraction assumes that

- The contents of the tank consists of two phases
- Each ideally mixed
- The jacket is ideally mixed
- The heat transfer is uniform
- The heater has a significant capacity
- The wall has an insignificant effect on the dynamics
- No heat losses to the environment
- No capacity effects of the construction
- Two gas streams are essential, the one that escapes through the top directly into the room and the other which is pushed passed the temperature sensor. The latter usually being quite small.
- The conditions in the room do not change with time.

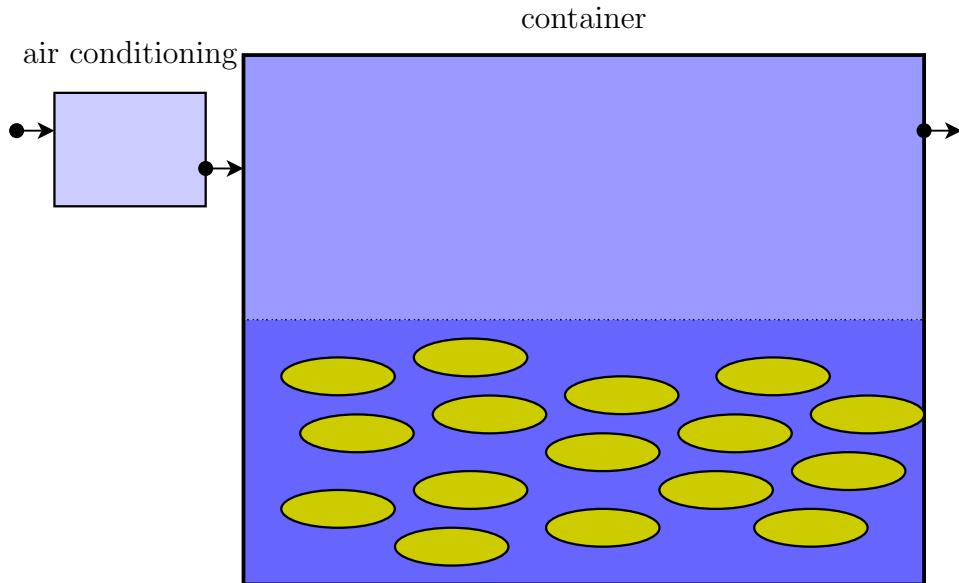
A possible abstraction could look something like this:



Here E:: energy reservoir, R:: room, H:: heater, L:: liquid, G:: gas, steam, S:: sensor, C:: controller.

1.2 Fruit Container

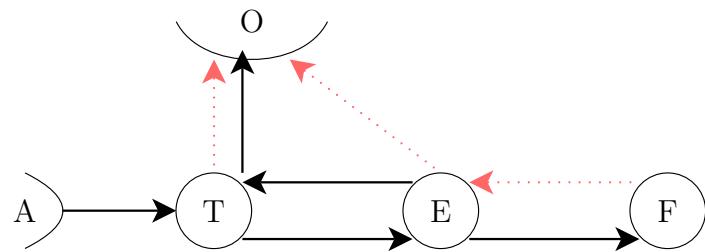
The container is equipped with an air-conditioning unit, which adjusts the conditions in the container. The solids, here the fruit is loosely packed into the container.



This abstraction assumes that

- The dynamics of the air conditioning is neglected in fact it is modelled as a reservoir of conditioned air.
- The air space is split into two parts:
 - a volume which is exchanged fast, the part which has little resistance between the air inlet and the outlet.

- a volume for the air of the free room between the solids.
- The solids as one big lump together, thus all behave the same.
- Each of these lumps is well mixed
- The fruit exchanges heat and mass
- The container exchanges heat with the environment
- Air is ejected into the environment

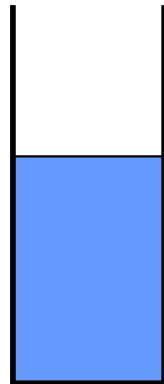


Here A :: air conditioning unit (idealised), O :: outside, T :: fast-reacting air space of the container, E :: Air in the immediate environment of the fruit, F :: fruit, lumped together as a single phase or system, thus all are assumed to behave the same.

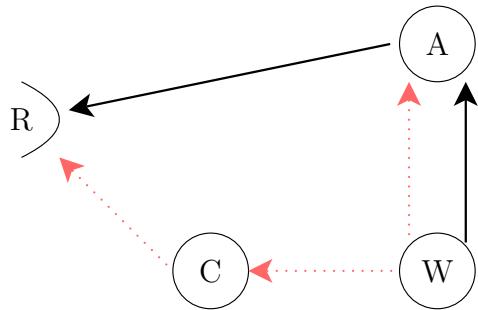
1.3 Water glass

This is a seemingly simple system. If we are interested in how it is cooling down, we look into a shorter time scale than when we are interested in evaporation being the only way it can lose mass.

The process is:



The topology capturing the cooling-down process may be:



With R :: the room, C :: the container's material, here probably glass, W :: the water body, assumed to be uniform in all intensive properties, A :: the air above the water in the glass, also assumed to be uniform.

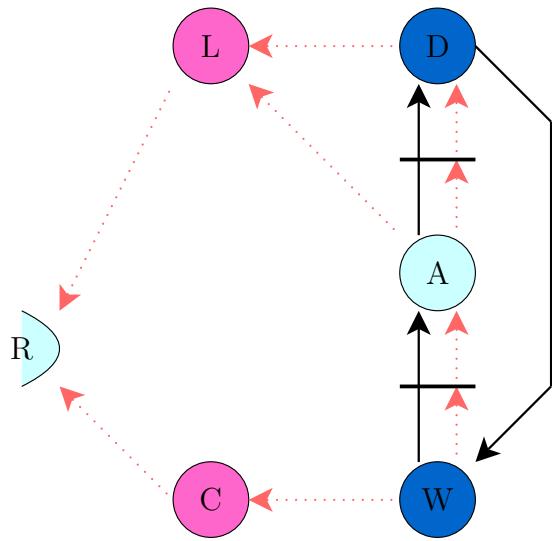
Obviously this is a simple representation of the process. Improvements can be made, in terms of adding descriptive power. This can for example be that one views the air above the water as a one-dimensional distributed air body and the heat and mass flow as diffusion processes.

Another meaningful extension could be to model the heat loss to the support surface of the glass. This can make a lot of sense if it is heated or cooled, purposely or not.

Changing the time scale and looking at the evaporation process, the exchanged heat is probably not making a whole of a lot of difference, except than that the initial evaporation rate is higher if one does not model the heat losses through conduction.

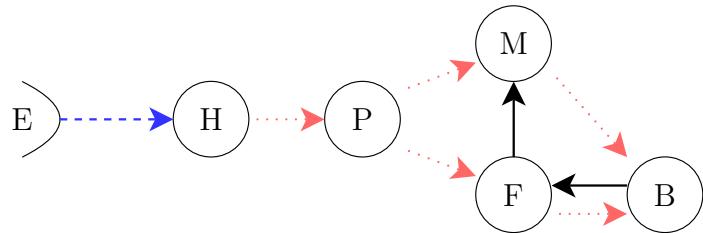
1.4 Water glass with lid

Adding a lid inhibits the convective mass transfer and thus the energy loss through this mechanism. the heat loss is now mainly driven by condensing the steam on the lid, which then flows, drips back into the water. Thus we have now two mass streams in and out of the air space above the water and we have a heat stream from the condensation to the lid inside the glass. Outside, the heat is lost through conduction into the surrounding air. This latter process is by no means simple. Increasing the air flow around the glass will affect the rate of energy transfer from the glass' surface to the passing-by air. The heat conduction stream shown from the water to the air space above is probably insignificant compared to the convective streams as long as the water is hot. Later it may become the main exchange.



1.5 Frying pan

The pan is on a hot plate. It has a thick metal bottom, and a piece of butter in it. The butter will form a film as it melts.



E :: electricity reservoir, H :: heater, hot plate, P :: bottom of the pan, F :: liquid butter film between pan's bottom and solid butter, M :: molten butter not under the solid butter.

As the butter is melting, it forms a film and thus gets separated from the pan's bottom. The melted butter flows out into a body of liquid butter that is also heated by the pan and since the butter is in this melt, it will likely also get some heat from the molten butter again.

Documenting your Code (TKP4106)



Zooball/Chicken

"Real Programmers write programs, not documentation."

The real programmer

Assignments

1. Instal Python v2.7.x on your computer. You are going to run Python from the terminal also called the command window (we don't use IDE's — do we?). I suggest you change the color preferences of your terminal to black screen and amber or green text. This sounds like an echo from the old days of monochrome displays, but it stands the test even today. The terminal is for punching in cryptic commands and have maybe thousands of lines of output pour over your screen. It is mainly for your information, not for producing readable code. A black screen is more relaxing to the eye than a bright screen.
2. Instal [epydoc](#) on your computer.
3.
 - a. Download the Python stub program [atoms.py](#).
 - b. Run epydoc on the stub file. Use stylesheet [TKP4106.css](#). The syntax is explained further down the page.
 - c. Learn how epydoc uses [epytext](#) for rendering its output.
 - d. Publish the HTML output from epydoc on your home page.
4. Download the Python scripts [morse.py](#) and [antimorse.py](#) for translating back and forth between the Latin and Morse alphabets. Learn how you can run these scripts in the terminal window. Study [Python strings](#) in general and method calls like `sys.stdin`, `re.sub` and keywords like `import`, `if-elif-else` and `print` in particular.

The source code documentation can be made at two levels. The traditional approach is to write lucid comments directly in the code — either above a block of code of major significance, say an `if-else` test or a `for` loop — or in-line to the right of each code statement. The block comment is easier to format and can be shaped into a paragraph of its own, while the in-line comment has the nicety that it vanishes if the statement should ever be deleted (a comment which is out of sync with the source code is incredibly misleading). I tend to use both comment styles in my programming of small stand-alone scripts like the Matlab script shown below.

Note that all the comments have flush right margin. This helps the reading a lot. Especially if you have a context sensitive editor which is almost certainly the case.

```
%Simplex algorithm applied to solve a limited LP-problem. The syntax is [x,b,A,it] = LP(x,b,A,c). The starting point is a minimization problem on the form
%
%      min(c'*y_{k+1})
%      A*y_{k+1} = A*y_k
%      y_{k+1}>= 0
%
% where:
%      y(b) = x (basis variables)
%      y(f) = 0 (free variables)
%
% x = solution vector (basis variables)          [m x 1]
% b = column indices of basis variables in A      [m x 1]
% A = coefficient matrix where rank(A) = m >1     .      [m x n]
% c = cost vector                                [n x 1]
% it = number of iterations spent in this function
%
%Copyright Tore Haug-Warberg 2008 (course TKP4175, KP8108, NTNU)
%
function [x,b,A,it] = LP(x,b,A,c)
%
f = 1:length(c); % temporary list of all variable indices
f(b) = []; . % remove basis variables =>free variables
%
for it=1:prod(size(A)) % restricted no of iterations for simplex
    dldx = c(f)' - c(b)']*A(:,f); % derivatives of d(c'*x)/dx(f)
    if all(dldx>=0) % all derivatives are non-negative
        return % converged, further progress impossible
    else % there is at least one negative derivative
        i = find(dldx<0); % consider negative derivatives only
        B = repmat(x,1,length(i)); % repeated right hand side
        H = A(:,f(i))./(B+eps); % reciprocal limiting factors
        if all(max(H)<=0) % entire column is zero or unbounded prob
            error('Unbounded problem') % no solution to this problem
        else % there is at least one regularly bounded free variable
            j = find(max(H)>0); % find new basis candidate
            [tmp,k] = max(dldx(i(j))); % most promising basis variable
            [tmp,l] = max(H(:,j(k))); % corresponding row index in A
            fpiv = i(j(k)); % f-index of new basis variable
            bpiv = find(A(l,b)==1); % b-index of old basis variable
            tmp = b(bpiv); % temporary storage
            b(bpiv) = f(fpiv); % replace old basis var with new one
            f(fpiv) = tmp; % replace old free variable with new one
            x = A(:,b)\x; % calculate new basis solution
            A = A(:,b)\A; % calculate new coefficient matrix
        end
    end
end
error(['Not converged in ',num2str(prod(size(A))), ' iterations'])
```

The second approach is to use some kind of lightweight formatting, called mark-up. This makes it possible to produce stand-alone documentation without intervening the code itself. This approach is suitable for larger projects but it requires a bit of metaprogramming, i.e. there is "coding in the coding". It is important, therefore, that

the mark-up stays out of the way without cluttering the code. This is the documentation form used in many programming languages today and tools like e.g. Doxygen makes it possible to churn out PDF and HTML documentation from many different sources of code written in C, C++, Fortran, Ruby, Python, etc. A simpler tool that goes with Python is epydoc. It builds on epytext, a kind of [docstring](#) format. An example is shown below. The code is admittedly polluted by artifacts like @summary, @author and other so-called metacommands, but the benefit of doing this extra formatting more than outweighs the drawback. From running the source code through epydoc

```
$ epydoc -v --css=TKP4106.css --parse-only atoms.py
```

an HTML [Epydoc output file](#) is generated. Realize how the documentation looks quite the same independent of the programmer's personal coding style.

```
"""
@summary:      Chemical formula parsing suite. Bla-bla.
@author:       Tore Haug-Warberg
@organization: Department of Chemical Engineering, NTNU, Norway
@contact:      haugwarb@nt.ntnu.no
@license:      GPLv3
@requires:     Python 2.3.5 or higher
@since:        2011.06.30 (THW)
@version:      0.9
@todo 1.0:    Bla-bla.
@change:       started (2011.06.30)
@change:       continued (2011.07.12)
@note:         Bla-bla.

"""

import re

def atoms(formula, debug=False, stack=[{}], \
          atom=r'([A-Z][a-z]?)(\d+)?', ldel=r'\(', rdel=r'\)(\d+)?'):

    """
    The 'atoms' parser takes a chemical formula on standard form - something
    like 'COOH(C(CH3)2)3CH3' - and breaks it into a dictionary of recognized
    atoms and their respective occurrences {'C': 11, 'H': 22, 'O': 2}. The
    parsing is performed left-to-right in a recursive manner which means it
    can handle nested parentheses.

    @param formula: a chemical formula 'COOH(C(CH3)2)3CH3'
    @param debug:   True or False flag
    @param stack:   an initial list of dictionaries
    @param atom:    string equivalent of RE matching atom name including an
                   optional number 'He', 'N2', 'H3', etc.
    @param ldel:    string equivalent of RE matching the left delimiter '('
    @param rdel:    string equivalent of RE matching the right delimiter in-
                   cluding an optional number ')', ')3', etc.

    @type formula: aString
    @type debug:   aBoolean
    @type stack:   aList
    @type atom:    aRE on raw string format
    @type ldel:    aRE on raw string format
    @type rdel:    aRE on raw string format
    """

    import re
    import collections

    def parse(formula):
        """Parse a chemical formula into a dictionary of atoms and their
        occurrences. The formula is assumed to be in standard form, i.e.
        atoms are separated by single spaces and there are no spaces
        between atoms and their counts. Nested parentheses are handled
        correctly.
        """
        stack = []
        atoms = {}
        current_atom = None
        current_count = 1
        current_depth = 0
        depth_stack = [0]
        depth_index = 0

        for char in formula:
            if char.isupper():
                if current_atom:
                    atoms[current_atom] += current_count
                current_atom = char
                current_count = 1
            elif char.islower():
                current_atom += char
            elif char.isdigit():
                current_count *= 10
                current_count += int(char)
            elif char == '(':
                stack.append((current_atom, current_depth))
                current_atom = None
                current_depth += 1
                depth_stack.append(current_depth)
            elif char == ')':
                current_atom, current_depth = stack.pop()
                current_depth = depth_stack.pop()
                if current_atom:
                    atoms[current_atom] += current_count
                current_atom = None
                current_count = 1
            else:
                raise ValueError("Unknown character in formula: " + char)

        if current_atom:
            atoms[current_atom] += current_count

        return atoms

    atoms_dict = parse(formula)

    if debug:
        print("Atoms dictionary: ", atoms_dict)

    # Now we have a dictionary of atoms and their occurrences. We
    # need to convert it into a list of dictionaries where each
    # dictionary represents an atom and its count. This is done
    # by creating a list of lists, where the index of the inner
    # list corresponds to the atom name and the value is the count.
    # For example, if the atoms dictionary is {'C': 11, 'H': 22, 'O': 2},
    # the resulting list would be [[{'C': 1}, {'H': 1}, {'O': 1}], [{}]*11, [{}]*22, [{}]*2].
    # This is then converted into a single list of dictionaries
    # where each dictionary contains both the atom name and its count.
    # Finally, the stack is updated with these new dictionaries.
    stack = []
    for atom, count in atoms_dict.items():
        for _ in range(count):
            stack.append({atom: 1})

    return stack
```

```
@return:           aDictionary e.g. {'C': 11, 'H': 22, 'O': 2}
"""

```

The secret of documentation lies in documenting your code from day one. Always make ready for documentation. Never wait. It will be too late before you know. In your future job you will be constantly assigned new tasks, which of course are more important than the one you are doing at the moment. By adopting a suitable documentation style you will always be able to return to your programs after a shorter or longer break. Without such a standard you will be lost. As a spin-off you can also produce documents that are valuable to your colleagues. **It does not matter how clever you are in programming if things only work on your desktop!**

5.5.1 The real programmer, see also Sec. 2.1

First reference occurs in *Real Programmers use FORTRAN*, see Section 2.1 on page 10.

5.5.2 epydoc, see also Sec. 5.1.19

First reference occurs in *Epydoc* ([sourceforge](#)), see Section 5.1.19 on page 101.

5.5.3 Verbatim: “atoms.py”

```

1 """
2 @summary:      Chemical formula parser. <pass: your description>
3 @author:       <pass: your name>
4 @organization: Department of Chemical Engineering, NTNU, Norway
5 @contact:      <pass: your address>
6 @license:      <pass: GPLv3 or whatever>
7 @requires:     Python <pass: x.y.z> or higher
8 @since:        <pass: yyyy.mm.dd> (<pass: your initials>)
9 @version:      <pass: x.y.z>
10 @todo 1.0:     <pass: bla-bla>
11 @change:       started (<pass: yyyy.mm.dd>)
12 @change:       <pass: last change description> (<pass: yyyy.mm.dd>)
13 @note:         <pass: bla-bla>
14 """
15
16 def atoms(formula, debug=False, stack=[], delim=0, \
17           atom=r'<pass>', ldel=r'<pass>', rdel=r'<pass>'):
18 """
19     The 'atoms' parser <pass: your description>.
20
21     @param formula: a chemical formula 'COOH(C(CH3)2)3CH3'
22     @param debug:   True or False flag
23     @param stack:   list of dictionaries { 'atom name': int, ... }
24     @param delim:   number of left-delimiters that have been opened and not yet
25                   closed.
26     @param atom:    string equivalent of RE matching atom name including an
27                   optional number 'He', 'N2', 'H3', etc.
28     @param ldel:    string equivalent of RE matching the left-delimiter '('
29     @param rdel:    string equivalent of RE matching the right-delimiter
30                   including an optional number ')', ')3', etc.
31
32     @type formula: <pass>
33     @type debug:   aBoolean
34     @type stack:   <pass>
35     @type delim:   <pass>
36     @type atom:    aRE on raw string format
37     @type ldel:    <pass>
38     @type rdel:    <pass>
39
40     @return:       aList [ aDictionary, aDictionary, ... ]
41                 e.g. [{C': 11, 'H': 22, 'O': 2}]
42 """
43
44 import re
45
46 # Empty strings do always pose problems. Test explicitly.
47 pass
48
49 # Initialize the dictionary stack. Can't be done in the function header be-
50 # cause Python initializes only once. Subsequent calls to this function will
51 # then increment the same dictionary rather than making a new one.
52 stack = stack or []
53
54 # Python has no switch - case construct. Match all possibilities first and
55 # test afterwards:
56 re_atom = pass
57 re_ldel = pass
58 re_rdel = pass
59
60 # Atom followed by an optional number (default is 1).
61 if re_atom:
62     tail = formula[len(re_atom.group()):]
63     head = pass
64     num = pass
65
66     if stack[-1].get(head, True):                      # verbose testing of Hash key
67         pass                                         # increment occurrence
68     else:
69         pass                                         # initialization

```

```

70         if debug: print [head, num, tail]
71
72     # Left-delimiter.
73     elif re_ldel:
74         tail    = pass
75         delim += pass
76
77         stack.append({})      # will be popped from stack by next right-delimiter
78
79         if debug: print ['left-delimiter', tail]
80
81     # Right-delimiter followed by an optional number (default is 1).
82     elif re_rdel:
83         tail    = pass
84         num     = pass
85         delim -= pass
86
87         if delim < 0:
88             raise SyntaxError("un-matched right parenthesis in '%s' % (formula,))
89
90         for (k, v) in stack.pop().iteritems():
91             stack[-1][k] = pass
92
93         if debug: print ['right-delimiter', num, tail]
94
95     # Wrong syntax.
96     else:
97         raise SyntaxError("'%s' does not match any regex" % (formula,))
98
99     # The formula has not been consumed yet. Continue recursive parsing.
100    if len(tail) > pass:
101        atoms(pass, pass, pass, pass, pass, pass)
102        return stack
103
104    # Nothing left to parse. Stop recursion.
105    else:
106        if delim > 0:
107            raise SyntaxError("un-matched left parenthesis in '%s' % (formula,))
108        if debug: print stack[-1]
109        return stack
110

```

5.5.4 epytext, see also Sec. 5.1.20

First reference occurs in *Epytext markup (sourceforge)*, see Section 5.1.20 on page 102.

5.5.5 docstring, see also Sec. 5.1.21

First reference occurs in *Python Docstrings (Sourceforge)*, see Section 5.1.21 on page 112.

Module atoms_stub

Author: <pass: your name>

Organization: Department of Chemical Engineering, NTNU, Norway

Contact: <pass: your address>

License: <pass: GPLv3 or whatever>

Requires: Python <pass: x.y.z> or higher

Since: <pass: yyyy.mm.dd> (<pass: your initials>)

Version: <pass: x.y.z>

To Do (1.0): <pass: bla-bla>

Change Log:

- started (<pass: yyyy.mm.dd>)
- <pass: last change description> (<pass: yyyy.mm.dd>)

Note: <pass: bla-bla>

Functions

[hide private]

```
atoms(formula, debug=False, stack=[], delim=0, atom=r'<pass>',
      ldel=r'<pass>', rdel=r'<pass>')
```

The 'atoms' parser <pass: your description>.

Function Details

[hide private]

```
atoms(formula, debug=False, stack=[], delim=0, atom=r'<pass>',
      ldel=r'<pass>', rdel=r'<pass>')
```

The 'atoms' parser <pass: your description>.

Parameters:

- **formula** (<pass>) - a chemical formula 'COOH(C(CH3)2)3CH3'
- **debug** (aBoolean) - True or False flag
- **stack** (<pass>) - list of dictionaries { 'atom name': int, ... }
- **delim** (<pass>) - number of left-delimiters that have been opened and not yet closed.
- **atom** (aRE on raw string format) - string equivalent of RE matching atom name including an optional number 'He', 'N2', 'H3', etc.
- **ldel** (<pass>) - string equivalent of RE matching the left-delimiter '('
- **rdel** (<pass>) - string equivalent of RE matching the right-delimiter including an optional number ')', ')3', etc.

Returns:

aList [aDictionary, aDictionary, ...] e.g. [{C: 11, H: 22, O: 2}]

Exercise 3

1 Question: Linear algebra 01

Refresh basic linear algebra

1.1 Problem Definition

Given the objects:

scalar	a
column vector	$\underline{x} := \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} := [x_i]_{i:=1,\dots,n}$
column vector	$\underline{b} := \begin{bmatrix} 1 \\ 2 \\ -3 \end{bmatrix}$
row vector	$\underline{x}^T := [x_1 \ x_2 \ \dots \ x_n]$
matrix	$\underline{\underline{A}} := \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m} \\ \vdots & \vdots & \dots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m} \end{bmatrix} := [a_{i,j}]_{i:=1,\dots,n; j:=1,\dots,m}$
matrix	$\underline{\underline{B}} := \begin{bmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,p} \\ b_{2,1} & b_{2,2} & \dots & b_{2,p} \\ \vdots & \vdots & \dots & \vdots \\ b_{o,1} & b_{o,2} & \dots & b_{o,p} \end{bmatrix} := [b_{i,j}]_{i:=1,\dots,o; j:=1,\dots,p}$
matrix	$\underline{\underline{C}} := \begin{bmatrix} -1 & 0 & 2 & -3 & 1.5 \\ 3 & -2 & 4 & 1 & 2.5 \\ 2 & 1.5 & -2 & 0 & -3 \\ 0 & 1 & 3.5 & -1.2 & -4 \end{bmatrix}$
matrix	$\underline{\underline{D}} := \begin{bmatrix} 3 & 6 & 5 & -5 \\ 4 & -2 & -1 & 7 \\ 5 & 3 & 4 & 1 \\ 3 & -4 & 4 & 2 \\ 10 & -2 & 2 & 1 \\ 3 & -2 & 4 & 1 \end{bmatrix}$
matrix	$\underline{\underline{E}} := \begin{bmatrix} e_{1,1} & e_{1,2} & e_{1,3} & e_{1,4} \\ e_{2,1} & e_{2,2} & e_{2,3} & e_{2,4} \\ e_{3,1} & e_{3,2} & e_{3,3} & e_{3,4} \\ e_{4,1} & e_{4,2} & e_{4,3} & e_{4,4} \end{bmatrix}$

matrix $\underline{\underline{\mathbf{F}}} := \begin{bmatrix} 1 & 2 & 3 \\ 3 & 0 & 4 \\ 2 & 1 & 5 \end{bmatrix}$

matrix $\underline{\underline{\mathbf{A}}}_1 := \begin{bmatrix} 3 & 6 & 5 \\ 4 & -2 & -1 \\ 5 & 3 & 4 \end{bmatrix}$

matrix $\underline{\underline{\mathbf{A}}}_2 := \begin{bmatrix} 2 & 2 & 3 \\ 4 & -1 & 2 \\ -6 & -6 & -9 \end{bmatrix}$

matrix $\underline{\underline{\mathbf{A}}}_3 := \begin{bmatrix} 2 & 2 & 3 \\ -6 & -6 & -9 \\ 4 & -1 & 2 \end{bmatrix}$

1.2 What to do

Fill in the details of the operations indicating the conditions that apply for the dimensions of the objects and what the dimensions are of the results.

dimensions	$\dim a := ?$
dimensions	$\dim \underline{\mathbf{x}} := ?$
dimensions	$\dim \underline{\underline{\mathbf{A}}} := ?$
dimensions	$\dim \underline{\underline{\mathbf{B}}} := ?$
dimensions	$\dim \underline{\underline{\mathbf{C}}} := ?$
scalar	$a \underline{\mathbf{x}} := ?$
transposition	$\underline{\mathbf{x}}^T := ?$
transposition	$\underline{\underline{\mathbf{A}}}^T := ?$
sum	$\underline{\underline{\mathbf{A}}} + \underline{\underline{\mathbf{B}}} := ?$
product	$\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{B}}} := ?$
product	$\underline{\mathbf{x}}^T \underline{\mathbf{x}} := ?$
product	$\underline{\underline{\mathbf{A}}}^T \underline{\underline{\mathbf{A}}} := ?$
product	$\underline{\underline{\mathbf{A}}} \underline{\underline{\mathbf{B}}} := ?$
element (2,3) and (3,4) of product	$\underline{\underline{\mathbf{D}}} \underline{\underline{\mathbf{C}}} := ?$
inverse	$\underline{\underline{\mathbf{E}}}^{-1} := ?$
inverse	$\underline{\underline{\mathbf{F}}}^{-1} := ?$
Solution for $\underline{\underline{\mathbf{A}}}_1 \underline{\mathbf{x}} = \underline{\mathbf{b}}$	$\underline{\mathbf{x}} := ?$
Solution for $\underline{\underline{\mathbf{A}}}_2 \underline{\mathbf{x}} = \underline{\mathbf{b}}$	$\underline{\mathbf{x}} := ?$
Solution for $\underline{\underline{\mathbf{A}}}_3 \underline{\mathbf{x}} = \underline{\mathbf{b}}$	$\underline{\mathbf{x}} := ?$

2 Programming: Matrix input

2.1 Objectives

Expand on your programming skills in particular on regular expressions

2.2 Assignment

Write a python program that takes the command line input, a string, and converts it into an internal object, a list of lists.

The command line input, a string, is a definition of a vector or matrix, whereby we use the syntax of MatLab. Your Python-internal representation is a list of lists which is also the output.

- matrix delimiters are the two rectangular brackets, ([]) and ()

- column separators are either a comma (,) or a space
- row delimiters are either semicolon (;) or a new line

Show that your code works for the inputs:

- [1, 2, 3; 4, 5, 6]
- [1.0, 2.9 3; 4, 0.5E1, 6.0]

Gives a proper error message :

- if the dimension conditions are not met [1, 2; 3, 4, 5]
- if the notation is wrong: (1, 2; 3, 4)

1 Suggested solution: Linear algebra 01

$$\dim a := 1 , \quad (1)$$

$$\dim \underline{\mathbf{x}} := n , \quad (2)$$

$$\dim \underline{\underline{\mathbf{A}}} := n \times m , \quad (3)$$

$$\dim \underline{\underline{\mathbf{B}}} := o \times p , \quad (4)$$

$$\dim \underline{\underline{\mathbf{C}}} := 4 \times 5 . \quad (5)$$

$$a \underline{\mathbf{x}} := a \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a x_1 \\ a x_2 \\ \vdots \\ a x_n \end{bmatrix} \quad (6)$$

The dimension of this result is n (or $n \times 1$).

$$\underline{\mathbf{x}}^T := [x_1 \ x_2 \ \dots \ x_n] \quad (7)$$

The dimension of this result is n (or $1 \times n$).

$$\underline{\underline{\mathbf{A}}}^T := \begin{bmatrix} a_{1,1} & a_{2,1} & \dots & a_{n,1} \\ a_{1,2} & a_{2,2} & \dots & a_{n,2} \\ \vdots & \vdots & \dots & \vdots \\ a_{1,m} & a_{2,m} & \dots & a_{n,m} \end{bmatrix} \quad (8)$$

The dimension of this result is $m \times n$.

The condition necessary for calculating a sum of matrices is that the dimensions of both matrices must be equal, therefore in this case: $o = n$ and $p = m$.

$$\underline{\underline{\mathbf{A}}} + \underline{\underline{\mathbf{B}}} := \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m} \\ \vdots & \vdots & \dots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m} \end{bmatrix} + \begin{bmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,m} \\ b_{2,1} & b_{2,2} & \dots & b_{2,m} \\ \vdots & \vdots & \dots & \vdots \\ b_{n,1} & b_{n,2} & \dots & b_{n,m} \end{bmatrix} \quad (9)$$

$$:= \begin{bmatrix} a_{1,1} + b_{1,1} & a_{1,2} + b_{1,2} & \dots & a_{1,m} + b_{1,m} \\ a_{2,1} + b_{2,1} & a_{2,2} + b_{2,2} & \dots & a_{2,m} + b_{2,m} \\ \vdots & \vdots & \dots & \vdots \\ a_{n,1} + b_{n,1} & a_{n,2} + b_{n,2} & \dots & a_{n,m} + b_{n,m} \end{bmatrix} \quad (10)$$

The dimension of this result is $n \times m$.

The condition necessary for calculating a product of matrices is that the column-dimension of the first matrix equals the row-dimension of the second matrix, therefore in this case: $o = m$.

$$\underline{\underline{\mathbf{A}}}\underline{\underline{\mathbf{B}}} := \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,m} \\ a_{2,1} & a_{2,2} & \dots & a_{2,m} \\ \vdots & \vdots & \dots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,m} \end{bmatrix} \begin{bmatrix} b_{1,1} & b_{1,2} & \dots & b_{1,p} \\ b_{2,1} & b_{2,2} & \dots & b_{2,p} \\ \vdots & \vdots & \dots & \vdots \\ b_{m,1} & b_{m,2} & \dots & b_{m,p} \end{bmatrix} \quad (11)$$

$$:= \begin{bmatrix} a_{1,1}b_{1,1} + a_{1,2}b_{2,1} + \dots + a_{1,m}b_{m,1} & \dots & a_{1,1}b_{1,p} + a_{1,2}b_{2,p} + \dots + a_{1,m}b_{m,p} \\ a_{2,1}b_{1,1} + a_{2,2}b_{2,1} + \dots + a_{2,m}b_{m,1} & \dots & a_{2,1}b_{1,p} + a_{2,2}b_{2,p} + \dots + a_{2,m}b_{m,p} \\ \vdots & \dots & \vdots \\ a_{n,1}b_{1,1} + a_{n,2}b_{2,1} + \dots + a_{n,m}b_{m,1} & \dots & a_{n,1}b_{1,p} + a_{n,2}b_{2,p} + \dots + a_{n,m}b_{m,p} \end{bmatrix} \quad (12)$$

The dimension of this result is $n \times p$.

The condition necessary for making a product of matrices is also applied for this case, because the column-dimension of the first is the row-dimension of the second: n .

$$\underline{\mathbf{x}}^T \underline{\mathbf{x}} := \begin{bmatrix} x_1 & x_2 & \dots & x_n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad (13)$$

$$:= \begin{bmatrix} x_1^2 + x_2^2 + \dots + x_n^2 \end{bmatrix} \quad (14)$$

The dimension of this result is 1.

The dimension of $\underline{\underline{\mathbf{A}}}^T$ is $m \times n$, therefore the necessary condition in this case is: $m = n$.

$$\underline{\underline{\mathbf{A}}}^T \underline{\underline{\mathbf{A}}} := \begin{bmatrix} a_{1,1} & a_{2,1} & \dots & a_{n,1} \\ a_{1,2} & a_{2,2} & \dots & a_{n,2} \\ \vdots & \vdots & \dots & \vdots \\ a_{1,n} & a_{2,n} & \dots & a_{n,n} \end{bmatrix} \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & \dots & a_{2,n} \\ \vdots & \vdots & \dots & \vdots \\ a_{n,1} & a_{n,2} & \dots & a_{n,n} \end{bmatrix} \quad (15)$$

$$:= \begin{bmatrix} a_{1,1}^2 + a_{2,1}^2 + \dots + a_{n,1}^2 & \dots & a_{1,1}a_{1,n} + a_{2,1}a_{2,n} + \dots + a_{n,1}a_{n,n} \\ a_{1,2}a_{1,1} + a_{2,2}a_{2,1} + \dots + a_{n,2}a_{n,1} & \dots & a_{1,2}a_{1,n} + a_{2,2}a_{2,n} + \dots + a_{n,2}a_{n,n} \\ \vdots & \dots & \vdots \\ a_{1,1}a_{1,n} + a_{2,1}a_{2,n} + \dots + a_{n,1}a_{n,n} & \dots & a_{1,n}^2 + a_{2,n}^2 + \dots + a_{n,n}^2 \end{bmatrix} \quad (16)$$

The dimension of this result is $n \times n$.

The multiplication of matrices $\underline{\underline{\mathbf{C}}}$ and $\underline{\underline{\mathbf{D}}}$ is as below

$$\underline{\underline{\mathbf{D}}} \underline{\underline{\mathbf{C}}} := \begin{bmatrix} 25 & -9.5 & 2.5 & 3 & 24.5 \\ -12 & 9.5 & 26.5 & -22.4 & -24 \\ 12 & 1 & 17.5 & -13.2 & -1 \\ -7 & 16 & -11 & -15.4 & -25.5 \\ -12 & 8 & 11.5 & -33.2 & 0 \\ -1 & 11 & -6.5 & -12.2 & -16.5 \end{bmatrix} \quad (17)$$

To get the inverse of a matrix the matrix has to be square and the determinant non-zero. At least the former is the case. Therefore:

$$\underline{\underline{\mathbf{E}}}^{-1} = \frac{\text{adj}(\underline{\underline{\mathbf{E}}})}{|\underline{\underline{\mathbf{E}}}|}, \quad (18)$$

with

$$\text{adj}(\underline{\underline{\mathbf{E}}}) = [r_{ij}]^T,$$

and

$$r_{ij} := (-1)^{i+j} \det(\underline{\underline{\mathbf{E}}}_{ij}).$$

$$\begin{aligned} |\underline{\underline{\mathbf{E}}}| = & e_{1,1}e_{2,2}e_{3,3}e_{4,4} - e_{1,1}e_{2,2}e_{3,4}e_{4,3} - e_{1,1}e_{3,2}e_{2,3}e_{4,4} - e_{1,1}e_{3,2}e_{2,4}e_{4,3} \\ & + e_{1,1}e_{4,2}e_{2,3}e_{3,4} - e_{1,1}e_{4,2}e_{2,4}e_{3,3} - e_{2,1}e_{1,2}e_{3,3}e_{4,4} \\ & + e_{2,1}e_{1,2}e_{3,4}e_{4,3} + e_{2,1}e_{3,2}e_{1,3}e_{4,4} - e_{2,1}e_{3,2}e_{1,4}e_{4,3} \\ & - e_{2,1}e_{4,2}e_{1,3}e_{3,4} + e_{2,1}e_{4,2}e_{1,4}e_{3,3} + e_{3,1}e_{1,2}e_{2,3}e_{4,4} \\ & - e_{3,1}e_{1,2}e_{2,4}e_{4,3} - e_{3,1}e_{2,2}e_{1,3}e_{4,4} + e_{3,1}e_{2,2}e_{1,4}e_{4,3} \\ & + e_{3,1}e_{4,2}e_{1,3}e_{2,3} - e_{3,1}e_{4,2}e_{1,4}e_{2,3} - e_{4,1}e_{1,2}e_{2,3}e_{3,4} \\ & + e_{4,1}e_{1,2}e_{2,4}e_{3,3} + e_{4,1}e_{2,2}e_{1,3}e_{3,4} - e_{4,1}e_{2,2}e_{1,4}e_{3,3} \\ & - e_{4,1}e_{3,2}e_{1,3}e_{2,4} + e_{4,1}e_{3,2}e_{1,4}e_{2,3} \end{aligned} \quad (19)$$

$$adj(\mathbf{E}) = \begin{bmatrix} \begin{vmatrix} e_{2,2} & e_{2,3} & e_{2,4} \\ e_{3,2} & e_{3,3} & e_{3,4} \\ e_{4,2} & e_{4,3} & e_{4,4} \end{vmatrix} & \begin{vmatrix} e_{2,1} & e_{2,3} & e_{2,4} \\ e_{3,1} & e_{3,3} & e_{3,4} \\ e_{4,1} & e_{4,3} & e_{4,4} \end{vmatrix} & \begin{vmatrix} e_{2,1} & e_{2,2} & e_{2,4} \\ e_{3,1} & e_{3,2} & e_{3,4} \\ e_{4,1} & e_{4,2} & e_{4,4} \end{vmatrix} & \begin{vmatrix} e_{2,1} & e_{2,2} & e_{2,3} \\ e_{3,1} & e_{3,2} & e_{3,3} \\ e_{4,1} & e_{4,2} & e_{4,3} \end{vmatrix} \\ \begin{vmatrix} e_{1,2} & e_{1,3} & e_{1,4} \\ e_{3,2} & e_{3,3} & e_{3,4} \\ e_{4,2} & e_{4,3} & e_{4,4} \end{vmatrix} & \begin{vmatrix} e_{1,1} & e_{1,3} & e_{1,4} \\ e_{3,1} & e_{3,3} & e_{3,4} \\ e_{4,1} & e_{4,3} & e_{4,4} \end{vmatrix} & \begin{vmatrix} e_{1,1} & e_{1,2} & e_{1,4} \\ e_{3,1} & e_{3,2} & e_{3,4} \\ e_{4,1} & e_{4,2} & e_{4,4} \end{vmatrix} & \begin{vmatrix} e_{1,1} & e_{1,2} & e_{1,3} \\ e_{3,1} & e_{3,2} & e_{3,3} \\ e_{4,1} & e_{4,2} & e_{4,3} \end{vmatrix} \\ \begin{vmatrix} e_{1,2} & e_{1,3} & e_{1,4} \\ e_{2,2} & e_{2,3} & e_{2,4} \\ e_{4,2} & e_{4,3} & e_{4,4} \end{vmatrix} & \begin{vmatrix} e_{1,1} & e_{1,3} & e_{1,4} \\ e_{2,1} & e_{2,3} & e_{2,4} \\ e_{4,1} & e_{4,3} & e_{4,4} \end{vmatrix} & \begin{vmatrix} e_{1,1} & e_{1,2} & e_{1,4} \\ e_{2,1} & e_{2,2} & e_{2,4} \\ e_{4,1} & e_{4,2} & e_{4,4} \end{vmatrix} & \begin{vmatrix} e_{1,1} & e_{1,2} & e_{1,3} \\ e_{2,1} & e_{2,2} & e_{2,3} \\ e_{4,1} & e_{4,2} & e_{4,3} \end{vmatrix} \\ \begin{vmatrix} e_{1,2} & e_{1,3} & e_{1,4} \\ e_{2,2} & e_{2,3} & e_{2,4} \\ e_{3,2} & e_{3,3} & e_{3,4} \end{vmatrix} & \begin{vmatrix} e_{1,1} & e_{1,3} & e_{1,4} \\ e_{2,1} & e_{2,3} & e_{2,4} \\ e_{3,1} & e_{3,3} & e_{3,4} \end{vmatrix} & \begin{vmatrix} e_{1,1} & e_{1,2} & e_{1,4} \\ e_{2,1} & e_{2,2} & e_{2,4} \\ e_{3,1} & e_{3,2} & e_{3,4} \end{vmatrix} & \begin{vmatrix} e_{1,1} & e_{1,2} & e_{1,3} \\ e_{2,1} & e_{2,2} & e_{2,3} \\ e_{3,1} & e_{3,2} & e_{3,3} \end{vmatrix} \end{bmatrix} \quad (20)$$

The two first elements in the adjoint matrix are calculated:

$$\det \begin{pmatrix} e_{2,2} & e_{2,3} & e_{2,4} \\ e_{3,2} & e_{3,3} & e_{3,4} \\ e_{4,2} & e_{4,3} & e_{4,4} \end{pmatrix} = e_{2,2}(e_{3,3}e_{4,4} - e_{3,4}e_{4,3}) - e_{2,3}(e_{3,2}e_{4,4} - e_{3,4}e_{4,2}) + e_{2,4}(e_{3,2}e_{4,3} - e_{3,3}e_{4,2})$$

$$\det \begin{pmatrix} e_{2,1} & e_{2,3} & e_{2,4} \\ e_{3,1} & e_{3,3} & e_{3,4} \\ e_{4,1} & e_{4,3} & e_{4,4} \end{pmatrix} = e_{2,1}(e_{3,3}e_{4,4} - e_{3,4}e_{4,3}) - e_{2,3}(e_{3,1}e_{4,4} - e_{3,4}e_{4,1}) + e_{2,4}(e_{3,1}e_{4,3} - e_{3,3}e_{4,1})$$

The dimension of this matrix is the same as the dimension of $\underline{\underline{\mathbf{E}}}$, namely 4×4 .

$$\underline{\underline{\mathbf{F}}}^{-1} = \begin{bmatrix} 0.444 & 0.778 & -0.889 \\ 0.778 & 0.111 & -0.556 \\ -0.333 & -0.333 & 0.667 \end{bmatrix} \quad (21)$$

$$\underline{\mathbf{x}} = \underline{\underline{\mathbf{A}_1}}^{-1} \underline{\mathbf{b}} = \begin{bmatrix} 1.129 \\ 3.742 \\ -4.968 \end{bmatrix} \quad (22)$$

The determinant of $\underline{\underline{\mathbf{A}_2}}$ and $\underline{\underline{\mathbf{A}_3}}$ are zero. So, $\underline{\mathbf{x}}$ cannot be calculated. The second matrix equation has infinite solutions, because the first and the last are linear dependent, so the system is underdetermined, whilst the third matrix equation $\underline{\underline{\mathbf{A}_3}} \underline{\mathbf{x}} = \underline{\mathbf{b}}$ has no feasible solution.

2 Suggested solution: Matrix input

Design

The input to the program is a string, which must be processed using a Matlab-like syntax.

How to go about it:

- Check if input is a string
- Extract string between the two out bracktes, [*string of rows*]
- Split string of rows into rows
- Loop through all rows
 - split current row into columns, which are matrix elements
 - add a new empty rows
 - loop through each element in the rows converting each element into a float
 - check if number of elements in the current row is the same as in the previous row
- return the list of lists

2.1 A program

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""
@summary: Convert a madlab matrix into a nested python list
@author: Eivind Haug-Warberg
@copyright: Tore Haug-Warberg
@organization: Department of Chemical Engineering, NTNU, Norway
@contact: haugwarb@nt.ntnu.no
@license: GPLv3
@requires: Python 2.7.1 or higher
@since: 2012.09.12 (EHW)
@version: 1.0
@todo 2.0:
@change: started (2012.09.20)
"""

def m2py(matstring, padding = None, debug = False):
    """
    Convert a Matlab matrix string into a nested python list. The string must
    start with optional white spaces and a left square bracket and end with
    optional white spaces and a right square bracket. Inside the brackets,
    there should be numbers (integer or float). Use commas "," or spaces " "
    to terminate columns and semicolon ";" to terminate rows.

    @param matstring: Input matrix given as a Matlab string
    @param padding: Object used for padding missing matrix elements
    @param debug: True or False flag

    @type matstring: aString
    @type padding: anObject
    @type debug: aBoolean

    @return: aList [ aList [ aFloat, aFloat, ... ] ]
             e.g. [[1.0, 0.0, ...], [0.0, 1.0, ...],
                   [-1.0, 0.0, ...], [0.0, -1.0, ...], ...]
    """
    import re

    # check for string
    if type(matstring) != str:
        print 'Input must be string'
        return None

    # extract string between the opening [ and closing ]
    tokenlist = re.match(r'^\s*\[.*\]\s*$', matstring)
    if tokenlist == None:
        print 'Input format is: [num1, num2, ...; num3, num4, ...; ...].'
        return None
```

```

# for an empty string we return an empty list of lists
if tokenlist.group(1) == '':
    return []

# define regular expression for the splitting into columns and rows
colsep = re.compile(r'\s*,?\s*') # column delimiters are ',' and ' '
rowsep = re.compile(r'\s*;\s*') # row delimiter is ';''

# split into rows first
rows = rowsep.split(tokenlist.group(1))

output = [] # allocate an empty list for the output

# Display the number of rows.
if debug: print 'Number of rows: %s'%len(rows)

# loop through all rows
for row in range(len(rows)):
    output.append([]) # add new row
    cols = colsep.split(rows[row]) # split into columns

    for col in cols: # loop through columns
        # Display the position in the nested list.
        if debug: print 'Value of row ' + str(len(output)) + ', column ' + \
            str(len(output[-1])) + ': ' + col

    try:
        output[-1].append(float(col))
    except: # handle conversion exception
        print 'Elements in matrix must be integers or floats'
        return None

# compare current length of the column with the previous one
# in the first instance the two are the same.
if len(output[row]) != len(output[row-1]):
    print 'matrix must have equal-length rows'
    return None

return output

if __name__ == '__main__':
    print '\n -----'
    print 'Matlab matrix input to be converted into a list of lists'

    temp_in = '\ncase : %s'
    temp_out = 'result: %s'
    temp_issue = 'issue : %s'

    a = '[1,2,3;4,5,6]'
    print temp_in %a

```

```
print temp_out %m2py(a)

a = '[1.0, 2.9 3; 4, 0.5E-1,6.0]'
print temp_in %a
print temp_out %m2py(a)

a = '[1,2.; 3, 4, 5]'
print temp_in %a
print temp_out %m2py(a)
print temp_issue %'problem with the definition of the float'

a = '1,2;3,4'
print temp_in %a
print temp_out %m2py(a)

a = '[1,2.0 E-2,3;4, 1.5,2.0,-1]'
print temp_in %a
print temp_out %m2py(a)
print temp_issue %'problem with the definition of the float'

a = '[1,2.0;1.5,2.0,-1]'
print temp_in %a
print temp_out %m2py(a)
print temp_issue %'row length'
```

2.2 Test run

```
-----
Matlab matrix input to be converted into a list of lists

case  : [1,2,3;4,5,6]
result: [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]]

case  : [1.0, 2.9 3; 4, 0.5E-1,6.0]
result: [[1.0, 2.9, 3.0], [4.0, 0.05, 6.0]]

case  : [1,2.; 3, 4, 5]
matrix must have equal-length rows
result: None
issue : problem with the definition of the float

case  : 1,2;3,4
Input format is: [num1, num2, ...; num3, num4, ...; ...].
result: None

case  : [1,2.0 E-2,3;4, 1.5,2.0,-1]
Elements in matrix must be integers or floats
result: None
issue : problem with the definition of the float

case  : [1,2.0;1.5,2.0,-1]
matrix must have equal-length rows
result: None
issue : row length
```

S

Parsing a Molecular Formula (TKP4106)



Zooball/Lion

A language that doesn't affect the way you think about programming, is not worth knowing.

[Alan J. Perlis \(1982\)](#)

Assignments

1.
 - a. Download the stub program [atoms.py](#). Save the file in your local Python folder. Keep the file name as indicated.
 - b. Learn about [Python dictionaries](#) and lists in general and about method calls like `re.match`, `len` and keywords like `def`, `pass`, `return` in particular. We shall also make use of a programming concept called "recursiveness". A simple example is the calculation of, say, 5 factorial. We can either program it like this:

```
def factorial(n=5):  
    m = 1  
    for i in range(1,n+1):  
        m *= i  
    return m
```

or, using recursive function calls:

```
def factorial(n=5):  
    if n > 1:  
        return n*factorial(n-1)  
    else:  
        return 1
```

Recursiveness gives beatiful albeit hard-to-debug computer code. There are special languages devoted entirely to so-called functional programming, like e.g. Lisp and Haskell, but Python is also quite well-suited for such tasks.

2. Write a chemical formula parser called `atoms` that takes a string input and returns a dictionary (hash table) of atom names (keys) and stoichiometric numbers (values). Like for instance:

```
atoms('COOH(C(CH3)2)3CH3') == [{H: 22, C: 11, O: 2}]
```

Use `atoms.py` as template. Do not change any of the variable names because this makes student's assistance and co-operation much harder!

Chemical formulas are — from a mass balance perspective — simple linear algebraic expressions. This sounds perhaps a little strange at first, but the algebraic rules for summation and multiplication are implicitly given by the formula. Take e.g. water (H_2O). The mass of one water molecule is $\text{H}^2 + \text{O}^1$ where H and O stand for the atomic masses of hydrogen and oxygen. So, when we write H_2O we really mean $\text{H}^2 + \text{O}^1$. The same rule applies to more complicated molecules like for instance $\text{COOH}(\text{C}(\text{CH}_3)_2)_3\text{CH}_3$. The mass is $\text{C}^1 + \text{O}^1 + \text{O}^1 + (\text{C}^1 + (\text{C}^1 + \text{H}^3)^2)^3 + \text{C}^1 + \text{H}^3$. We see that the use of parentheses are just like in everyday algebra. This means that it is possible to interpret — we shall hereafter call it parse — the formula into a list of atoms and a corresponding list of stoichiometric numbers. These two lists are conveniently held together in what is called a dictionary (hash table). In programming lingo we would say:

```
'COOH(C(CH3)2)3CH3' -> [{ 'H': 22, 'C': 11, 'O': 2 }]
```

To make the syntax straight `[{}]` means a list of length one which contains an empty dictionary. Note that for technical reasons the hash table is put inside a list (an array). This makes later use of the code easier (the exact reason is not visible at the moment). To write a parser we must know a little about [Backus-Naur Formalism](#) (BNF). The idea is quite simple, but it is hard to explain in words. An example serves better. Here is the BNF description of a floating decimal number:

```

S   := FN | '-' FN
FN  := DL | DL '.' DL
DL  := D  | D DL
D   := '0' | '1' | ... | '9'

```

Here `S` stands for sentence, `FN` for floating number, `DL` for digit list and `D` for digit. These are called the production rules. They are on the form `SYMBOL := SYMBOL | TERMINAL`. A symbol is something that is defined by `:=` while a terminal is a literal string in quotes. We see that our number is composed of the terminals `-`, `.`, `0`, `1`, `...`, `9`. OK, fine. Let's see if the BNF can represent a number for us. Starting at the top of the production list we continue making arbitrary decisions till there is nothing more to decide:

```

S           <- the starting point
'-' FN      <- used 2nd rule for S
'-' DL '.' DL <- used 2nd rule for FN
'-' D '.' D DL <- used 1st rule for left DL and 2nd rule for right DL
'-' D '.' D D <- used 1st rule for last DL
'-' '3' '.' '1' '4' <- used 4th, 2nd and 5th rules for the three D's

```

The outcome of the random process is `-3.14` which is a perfectly legal float. However, the BNF is quite tedious and therefore EBNF has been developed (E stands for Extended). It uses `?` for zero or one (occurrences), `+` for one or many, and `*` for zero or many. The same number defined in EBNF is:

```

S := '-' ? D + ('.' D +) ?
D := '0' | '1' | ... | '9'

```

This is definitely simpler and it is also quite close to [Regular Expressions](#) (re) notation in Python. Actually, there are many dialects of RE but they are all close to this form:

```
s := (-)?([0-9]+)(\.( [0-9]+))?
```

or even simpler:

```
s := -?\d+(\.\d+)?
```

The idea is now to use s inside a program to match **all** occurrences of floating point numbers. This is an incredible strong concept as it opens up for the programming of programming languages (making parsers and compilers). Now, back to our chemical formula we need only three regular expressions:

- 1) An atom name (chemical symbol) followed by nothing or an integer.
- 2) A left delimiter (left parenthesis).
- 3) A right delimiter (right parenthesis) followed by nothing or an integer.

At the moment these expressions will do all right:

```
ATOM := ([A-Z][a-z]?) (\d+)?  
LDEL := \\(  
RDEL := \\) (\d+)?
```

I have'nt mentioned it yet, but there are a few reserved characters in RE's. These include: ., -, +, (,), [,], {, }, ?, |, ^ and \$. Any use of these characters as terminal strings must be preceeded by \ (a backspace). The technique is called "escaping" in the local lingo.

The trick is now to make use of ATOM, LDEL and RDEL to break the chemical formula into bits and pieces using recursive function calls starting at the left end of the formula. Exactly how this procedure should be written is made part of your assignment (but you have got the license to ask).

5.7.1 Verbatim: “atoms.py”

```

1 """
2 @summary:      Chemical formula parser. <pass: your description>
3 @author:       <pass: your name>
4 @organization: Department of Chemical Engineering, NTNU, Norway
5 @contact:      <pass: your address>
6 @license:      <pass: GPLv3 or whatever>
7 @requires:     Python <pass: x.y.z> or higher
8 @since:        <pass: yyyy.mm.dd> (<pass: your initials>)
9 @version:      <pass: x.y.z>
10 @todo 1.0:     <pass: bla-bla>
11 @change:       started (<pass: yyyy.mm.dd>)
12 @change:       <pass: last change description> (<pass: yyyy.mm.dd>)
13 @note:         <pass: bla-bla>
14 """
15
16 def atoms(formula, debug=False, stack=[], delim=0, \
17           atom=r'<pass>', ldel=r'<pass>', rdel=r'<pass>'):
18 """
19     The 'atoms' parser <pass: your description>.
20
21     @param formula: a chemical formula 'COOH(C(CH3)2)3CH3'
22     @param debug:   True or False flag
23     @param stack:   list of dictionaries { 'atom name': int, ... }
24     @param delim:   number of left-delimiters that have been opened and not yet
25                   closed.
26     @param atom:    string equivalent of RE matching atom name including an
27                   optional number 'He', 'N2', 'H3', etc.
28     @param ldel:    string equivalent of RE matching the left-delimiter '('
29     @param rdel:    string equivalent of RE matching the right-delimiter
30                   including an optional number ')', ')3', etc.
31
32     @type formula: <pass>
33     @type debug:   aBoolean
34     @type stack:   <pass>
35     @type delim:   <pass>
36     @type atom:    aRE on raw string format
37     @type ldel:    <pass>
38     @type rdel:    <pass>
39
40     @return:       aList [ aDictionary, aDictionary, ... ]
41                 e.g. [{C': 11, 'H': 22, 'O': 2}]
42 """
43
44 import re
45
46 # Empty strings do always pose problems. Test explicitly.
47 pass
48
49 # Initialize the dictionary stack. Can't be done in the function header be-
50 # cause Python initializes only once. Subsequent calls to this function will
51 # then increment the same dictionary rather than making a new one.
52 stack = stack or []
53
54 # Python has no switch - case construct. Match all possibilities first and
55 # test afterwards:
56 re_atom = pass
57 re_ldel = pass
58 re_rdel = pass
59
60 # Atom followed by an optional number (default is 1).
61 if re_atom:
62     tail = formula[len(re_atom.group()):]
63     head = pass
64     num = pass
65
66     if stack[-1].get(head, True):                      # verbose testing of Hash key
67         pass                                         # increment occurrence
68     else:
69         pass                                         # initialization

```

```

70         if debug: print [head, num, tail]
71
72     # Left-delimiter.
73     elif re_ldel:
74         tail    = pass
75         delim += pass
76
77         stack.append({})      # will be popped from stack by next right-delimiter
78
79         if debug: print ['left-delimiter', tail]
80
81     # Right-delimiter followed by an optional number (default is 1).
82     elif re_rdel:
83         tail    = pass
84         num     = pass
85         delim -= pass
86
87         if delim < 0:
88             raise SyntaxError("un-matched right parenthesis in '%s' % (formula,))
89
90         for (k, v) in stack.pop().iteritems():
91             stack[-1][k] = pass
92
93         if debug: print ['right-delimiter', num, tail]
94
95     # Wrong syntax.
96     else:
97         raise SyntaxError("'%s' does not match any regex" % (formula,))
98
99     # The formula has not been consumed yet. Continue recursive parsing.
100    if len(tail) > pass:
101        atoms(pass, pass, pass, pass, pass, pass)
102        return stack
103
104    # Nothing left to parse. Stop recursion.
105    else:
106        if delim > 0:
107            raise SyntaxError("un-matched left parenthesis in '%s' % (formula,))
108        if debug: print stack[-1]
109        return stack
110

```

5.7.2 Backus-Naur Formalism, see also Sec. 5.1.10

First reference occurs in *BNF and EBNF* (*L. M. Garshol*), see Section 5.1.10 on page 75.

Exercise 4

Preisig, H A

Chemical Engineering, NTNU

1 Question: Networks 1.1

Given the topology for a stirred tank:

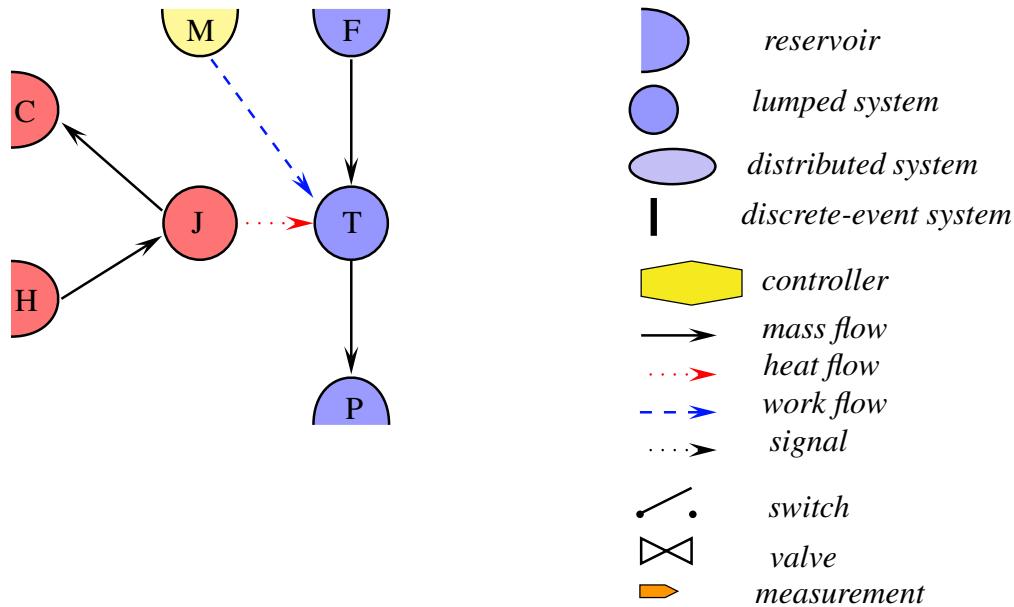


Figure 1: A simple topology of a model for a jacketed stirred tank reactor

1. Complete labelling of the graph by adding the labels for the streams
2. Write the incidence list for
 - all connections
 - mass only
 - heat only
 - work only
3. What are the incidence matrices for the mass flow network, the energy-flow network?
4. Write a python program that takes the incidence list and generates the incidence matrices. Give the incidence list as tuples of strings, where the strings are the identifiers for the source and sink nodes. Generate the index map for the systems, the numerical incidence list and then the incidence matrix. So input is a string representing the list of tuples of strings. The output is the numerical incidence matrix in printed form.

2 Question: Networks 1.2

In the lecture notes, the section on networks, a set of equations appears as an example, which formulates a transfer network as a function of the effort variables in the connected systems.

$$\begin{aligned}\dot{\Phi}_a &= -\hat{\Phi}_{a|b} \\ \dot{\Phi}_b &= +\hat{\Phi}_{a|b} + \hat{\Phi}_{c|b} - \hat{\Phi}_{b|d} \\ \dot{\Phi}_c &= -\hat{\Phi}_{c|b} \\ \dot{\Phi}_d &= +\hat{\Phi}_{b|d}\end{aligned}$$

which when substituted becomes:

$$\begin{aligned}\dot{\Phi}_a &= +c_{a|b}(\pi_b - \pi_a) \\ \dot{\Phi}_b &= -c_{a|b}(\pi_b - \pi_a) - c_{c|b}(\pi_b - \pi_c) + c_{b|d}(\pi_d - \pi_b) \\ \dot{\Phi}_c &= +c_{c|b}(\pi_b - \pi_c) \\ \dot{\Phi}_d &= -c_{b|d}(\pi_d - \pi_b)\end{aligned}$$

Please explain how you can formulate in the abstract form which is shown below:

$$\dot{\Phi} = -\underline{\underline{\mathbf{F}}} \underline{\underline{\mathbf{C}}} \underline{\underline{\mathbf{F}}^T} \boldsymbol{\pi}$$

3 Question: Networks 1.3

Figure 2 shows the conceptual process flowsheet for production of crude methanol. As the reactor only converts a limited amount of syngas into methanol, the un-reacted gas is recycled. A purge makes sure that the impurities do not accumulate in the recycle. The feed specifications are shown in Table below and the main reaction going on in the reactor are:

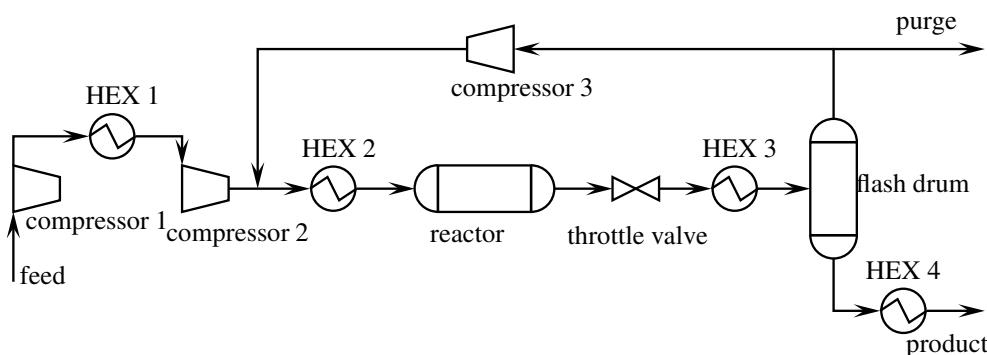
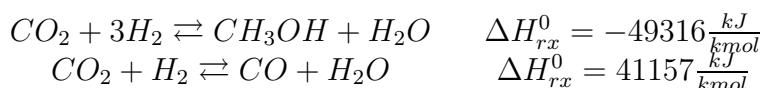


Figure 2: Simple flowsheet of a methanol plant

Table 1: Feed specifications

Component	Formula	content
Methane	CH_4	10
Carbon Monoxide	CO	12
Carbon Dioxide	CO_2	8
Hydrogen	H_2	70

3.1 Tasks

- Sketch a very simple topology of the methanol flowsheet. For the system definition, stay on the unit level where appropriate.
- Identify the species in each node in the topology.
- Generate the global species set.
- Write a python program that gets the species set in the respective nodes as input. Each species set is a vector of strings, whereby each string represents a species. For example, the species set for the feed would be $[CH4', CO', CO2', H2']$. Several nodes contain the same set of species. You may define domains, which contain nodes with the same species sets. The output shall be the index map for each species set in the respective domain. The following tasks should be done:
 - process the input string
 - generate the global species set as a list of strings
 - generate the index map: $[1, \dots, \# \text{ species}]$
 - generate selection matrix for each domain
 - compute the index set for the species in each domain
 - generate the output

4 Question: Shell balance 01

Assume a heating wire with a cylindrical insulation in a uniform environment. The dimension of wire is negligible. Derive the model equation, a PDE in radial co-ordinates, for the computation of the temperature profile in the insulation. State clearly your assumptions.

1 Solution: Networks 1.1

The topology graph contains the following arcs:

$$\mathcal{A} := [\hat{n}_{H|J}, \hat{n}_{J|C}, \hat{n}_{F|T}, \hat{n}_{T|P}, \hat{q}_{J|T}, \hat{w}_{M|T}]$$

For the mass only, it is the first 4 elements, the heat only is the 5th and the work only is the last.

The incidence matrices is constructed quickly by generating a table with the rows labelled with the systems and the columns with the streams. Since the streams carry the information of the reference direction (source | sink), one only needs to fill in the -1 in the source row and the $+1$ in the sink row. For readability we label the rows and the columns.

For the mass:

	$\hat{n}_{H J}$	$\hat{n}_{J C}$	$\hat{n}_{F T}$	$\hat{n}_{T P}$
T			+1	-1
J	+1	-1		
H	-1			
C		+1		
M				
F			-1	
P				+1

For energy we need all, the mass streams, as they carry energy in the form of internal, kinetic and potential energy, and we need the heat and work streams:

	$\hat{n}_{H J}$	$\hat{n}_{J C}$	$\hat{n}_{F T}$	$\hat{n}_{T P}$	$\hat{q}_{J T}$	$\hat{w}_{M T}$
T			+1	-1	+1	+1
J	+1	-1			-1	
H	-1					
C		+1				
M						-1
F			-1			
P				+1		

Note that the "mass" incidence matrix is part of the "energy" incidence matrix reflecting the fact that mass flow induces energy flow.

2 Solution: Networks 1.2

In a first step we can write:

$$\dot{\Phi} = \underline{\underline{F}} \underline{\underline{C}} \begin{bmatrix} \pi_b - \pi_a \\ \pi_c - \pi_b \\ \pi_d - \pi_c \\ \pi_d - \pi_b \end{bmatrix}$$

with:

	$a b$	$b c$	$c d$	$b d$
a	-1			
b	+1	-1		-1
c		+1	-1	
d			+1	+1

and:

$$\underline{\underline{C}} := \text{diag}[c_{a|b} \ c_{b|c} \ c_{c|d} \ c_{b|d}]$$

The vector of differences in the effort variables contains the negative signs of the respective column in the F-matrix. So this vector can be generated by:

$$\begin{bmatrix} \pi_b - \pi_a \\ \pi_c - \pi_b \\ \pi_d - \pi_c \\ \pi_d - \pi_b \end{bmatrix} := \underline{\underline{F}}^T \boldsymbol{\pi}$$

3 Solution: Networks 1.3

1)

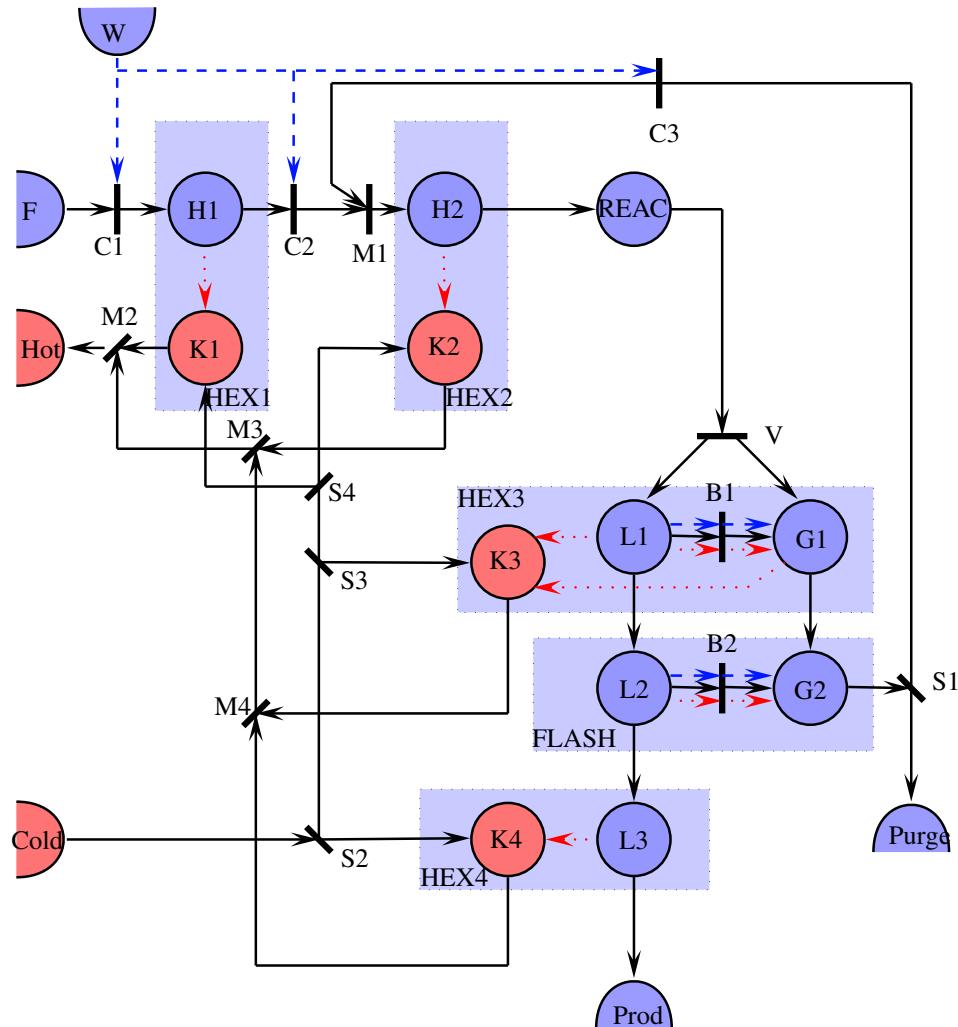


Figure 1: Topology of a methanol plant

2) What is where?

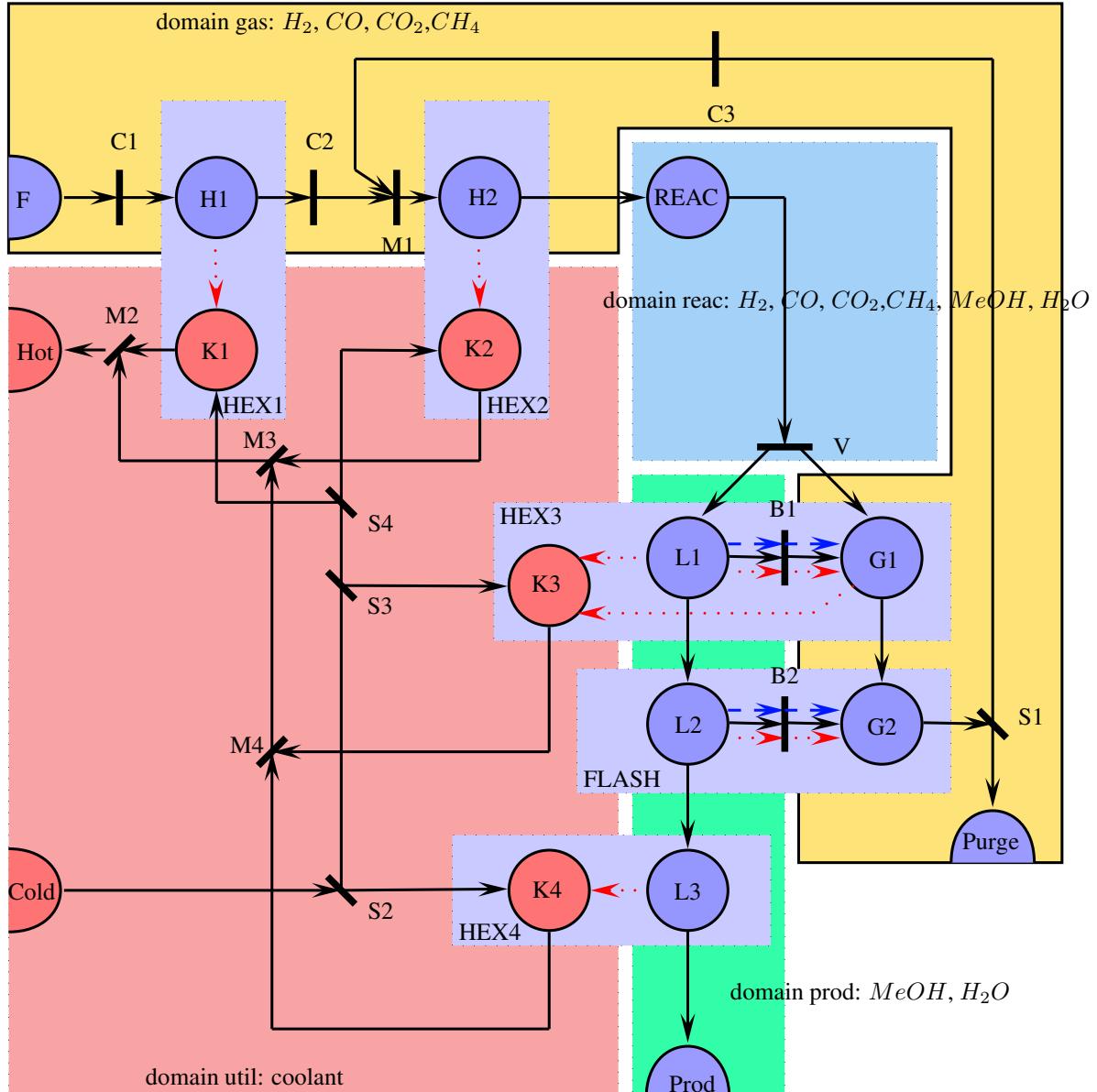


Figure 2: Domains with the same components in the topology of a methanol plant

3) Species set:= [$H_2 \ CO \ CO_2 \ CH_4 \ CH_3OH \ H_2O \ Cooling\ Fluid$]

The selection matrices are defined for the four domains which have common set of species:

Domain gas: H_2, CO, CO_2, CH_4

$$S_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Domain reac: H_2 , CO , CO_2 , CH_4 , $MeOH$, H_2O

$$S_2 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Domain Prod: $MeOH$, H_2O

$$S_3 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Domain util: *Coolant*

$$S_4 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

By multiplying the selection matrix to the vector of components, the existence of species in each node will be checked. For example,

$$a_1 = S_1 a = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

where a in the index vector of the species.

3.0.1 Program

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def make_stream(si,stot):
    descr = si[0]
    species = si[1]
    mat = [[int(i == j) for i in stot[1]] for j in si[1]]
    indices = mprod(mat, [[i+1] for i in range(0, len(stot[1]))])
    return {'Description': descr, 'Species': species, 'Matrix': mat, \
            'Indices': indices}

from mprod import mprod

def methprod(domains,debug):

    # Makes a list of all the species.
    stot = set([])
    [[stot.add(specie) for specie in domain[1]] for domain in domains]
    stot = list(stot); stot.sort()
    # Displays all the species.
    if debug:
        print 'Debugging :'
        print 'Total species: %s'%stot
    # Makes the index dictionary.
    indices = {}
    for specie in stot:
        indices[specie] = indices.get(specie,len(indices) + 1)
    # Dispaly the indices.
    if debug: print 'Index values: %s'%indices
    # Makes an empty matrix (this will be the incidence matrix soon.
    mat = [[0 for i in stot] for j in domains]
    # Displays the empty matrix.
    if debug: print 'Empty matrix: %s'%mat
    # Fills '1' into the places where a specie occures in a domain.
    for i in range(0, len(domains)):
        for specie in domains[i][1]:
            mat[i][indices[specie]-1] = 1
    # Displays the incidence matrix.
    if debug: print 'Incidence matrix: %s'%mat

    return stot, mat

if __name__ == '__main__':
    print '\n-----',
    print 'Incidence matrices test run',
```

```

print '\n-----',
species, mat = (methprod([('Domain 1',['H2O','CO']),('Domain 2',['CH4','NO3'])]),T
print '\n-----',
print 'Resulting incidence matrix for the species set \n%s \n\nis:\n%s' %species
for i in range(len(mat)):
    print 'row %s:%s'%(i,mat[i])
print '\n-----',

```

3.0.2 Output with debugging on

Incidence matrices test run

Debugging :
Total species: ['CH4', 'CO', 'H2O', 'NO3']
Index values: {'CO': 2, 'H2O': 3, 'CH4': 1, 'NO3': 4}
Empty matrix: [[0, 0, 0, 0], [0, 0, 0, 0]]
Incidence matrix: [[0, 1, 1, 0], [1, 0, 0, 1]]

Resulting incidence matrix for the species set
['CH4', 'CO', 'H2O', 'NO3']

is:

row 0:[0, 1, 1, 0]
row 1:[1, 0, 0, 1]

4 Solution: Shell balance 01

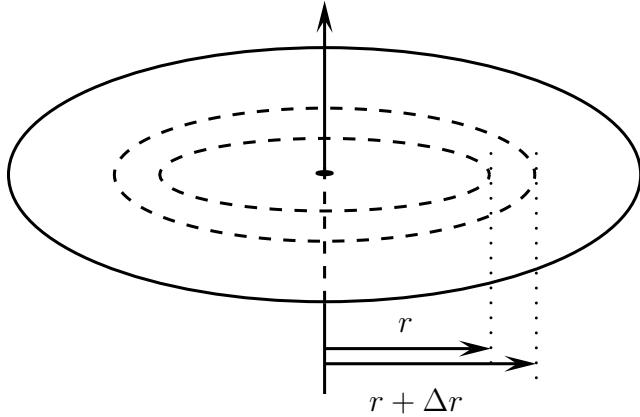
Conforming with the assumption of uniform environment, the temperature changes only in the radial direction. The geometry is shown in Figure 4:

Let the extensive quantity be Φ and its flux $\hat{\varphi}$. Then a balance over a small volume element ΔV is

$$\frac{d\Phi}{dt} := A_r \hat{\varphi}|_r - A_{r+\Delta r} \hat{\varphi}|_{r+\Delta r}$$

with

$$A_{r+\Delta r} \hat{\varphi}|_{r+\Delta r} \approx A_r \hat{\varphi}|_r + \left. \frac{\partial A \hat{\varphi}}{\partial r} \right|_{r+\Delta r}$$



substitution yields:

$$\begin{aligned}
 \frac{d\Phi}{dt} &:= A_r \hat{\varphi}|_r - \left(A_r \hat{\varphi}|_r + \left. \frac{\partial A \hat{\varphi}}{\partial r} \right|_r \Delta r \right) \\
 &:= - \left. \frac{\partial A \hat{\varphi}}{\partial r} \right|_r \Delta r \\
 &:= - \left. \frac{\partial A}{\partial r} \right|_r \hat{\varphi} \Delta r - A_r \left. \frac{\partial \hat{\varphi}}{\partial r} \right|_r \Delta r \\
 &:= - \frac{A_r}{r} \hat{\varphi} \Delta r - A_r \left. \frac{\partial \hat{\varphi}}{\partial r} \right|_r \Delta r \\
 &:= - \left(\frac{1}{r} \hat{\varphi} + \left. \frac{\partial \hat{\varphi}}{\partial r} \right|_r \right) A_r \Delta r
 \end{aligned}$$

where we used the fact that the area being a linear function of r .

Next we divide by the volume and take the limit:

$$\begin{aligned}
 \lim_{\Delta V \rightarrow 0} \frac{d\Phi/\Delta V}{dt} &:= - \left(\frac{1}{r} \hat{\varphi} + \left. \frac{\partial \hat{\varphi}}{\partial r} \right|_r \right) \\
 \frac{\partial \varphi}{\partial t} &:= - \left(\frac{1}{r} \hat{\varphi} + \left. \frac{\partial \hat{\varphi}}{\partial r} \right|_r \right)
 \end{aligned}$$

For the energy balance at constant pressure, the extensive quantity is enthalpy H and the partial enthalpy is $c_p \rho T$ with c_p :: the specific heat capacity and ρ :: the density. The flux $\hat{\varphi}$ for an isotropic material is $-k \frac{\partial T}{\partial r}$, with k being the heat conductivity. Assuming the properties being constant not only with respect to the direction (isotropic) but also to the state, in this case temperature, then:

$$\begin{aligned}
 \frac{\partial \varphi}{\partial t} &:= - \left(\frac{1}{r} \hat{\varphi} + \left. \frac{\partial \hat{\varphi}}{\partial r} \right|_r \right) \\
 \frac{\partial c_p \rho T}{\partial t} &:= \frac{1}{r} k \left. \frac{\partial T}{\partial r} \right|_r + \left. \frac{\partial}{\partial r} \right|_r k \left. \frac{\partial T}{\partial r} \right|_r \\
 c_p \rho \frac{\partial T}{\partial t} &:= \frac{k}{r} \left. \frac{\partial T}{\partial r} \right|_r + k \left. \frac{\partial^2 T}{\partial r^2} \right|_r \\
 \frac{\partial T}{\partial t} &:= \frac{k}{c_p \rho} \left(\frac{1}{r} \left. \frac{\partial T}{\partial r} \right|_r + \left. \frac{\partial^2 T}{\partial r^2} \right|_r \right)
 \end{aligned}$$

$$\frac{\partial T}{\partial t} := \alpha \left(\frac{1}{r} \frac{\partial T}{\partial r} \Big|_r + \frac{\partial^2 T}{\partial r^2} \Big|_r \right)$$

introducing the definition $\alpha := \frac{k}{c_p \rho}$:: the heat diffusivity.

The Atom Matrix (TKP4106)



Zooball/Penguin

"Spell Check Song"

I have a spelling checker.
It came with my PC.
It plane lee marks four my revue
Miss steaks aye can knot see.

Eye ran this poem threw it.
Your sure real glad two no.
Its very polished in its weigh,

...

Spell Check Song

Assignments

1. Write a procedure `atom_matrix` for calculating the formula matrix of an ordered set a substances from their chemical formulas (given as a list of strings). Make the output a list of lists of integers `[[int11, int12, ...], [in21, int22, ...], ...]`. Use the stub program [atom_matrix.py](#) as template.
2. Spin-off (not compulsory): Write a procedure `molecular_weight` for calculating the molecular weight of a substance given its chemical formula (string). Make the output a list of two integers `[int1,int2]` where $M_w = \text{int1}/\text{int2}$ and all the digits of `int1` are significant. Use the stub program [molecular_weight.py](#) as template.
3. Learn about [Python sets](#) (as in "set" theory) and about method calls like `str.sort` and keywords like `list` in particular. We shall also start talking about the list iterator `for x in xlist` and the [List comprehension](#) `[a+b for (a, b) in zip(alist, blist)]`.

Python is a programming language which to a large extent is built on the concept of lists and list comprehensions. Mix it with recursive function calls and you have a powerful programming environment! About the difference between for-loops, list comprehension and recursive function calls I shall say this much:

1. For-loops are for casual problems without any particular data structure.

2. List comprehension is a Good Thing if you are dealing entirely with lists.
3. Recursive programming is The Way of making lists of arbitrary length when termination (convergence) can be guaranteed.

Three stylistic examples follow. Let `args` be a list, or any other data structure with an **iterator** implemented, that is a method which visits the members of the list once - and exactly once. objects `arg` of unknown types. `fun` is a function that takes one `arg` and do something about it, and `err` is a second function that evaluates the convergence criterion for the sequence:

```
# Imperative for-loop:
for arg in args:
    fun(arg)
    pass

# List comprehension:
[fun(arg) for arg in args]

# Recursive function call:
def rc(arg, fun, err, seq=[]):
    if err(arg, fun): rc(fun(arg), fun, err, seq)
    seq.insert(0, arg)
    return seq
```

Note that in the two first cases `fun` appears as a function in the mathematical sense. In the last case, however, `fun` (and `err`) appear as function objects given to `rc`. They are sometimes called **functors** to remind you of functionals in mathematics. Think about integrals. This is a mathematical operation awaiting your function of interest in order to produce a number. `rc` is doing the same. It awaits a starting point `arg` and two functors `fun` and `crit` in order to produce the convergence sequence `seq`. If you are new to Python this sounds Greek maybe, but give it a chance! Invent a few problems and increase your knowledge... A minimal example is the convergence of $x_{n+1} = x_n * x_n \Rightarrow 0$ for $x_0 < 1$ and $n \Rightarrow \infty$. A possible implementation is:

```
# Perfectly general Fixed Point Iteration.
def rc(arg, fun, err, seq=[]):
    if err(arg, fun): rc(fun(arg), fun, err, seq)
    seq.insert(0, arg)
    return seq

# Your function implementation.
def myfun(arg):
    return arg**2

# Your termination criteria.
def myerr(arg, fun):
    if abs(arg-fun(arg)) > 0: return True
    return False

args = rc(0.999, myfun, myerr)
print args
```

The sequence converges beatifully to zero (make sure to run the program yourself in order to achieve a better understanding of the matter):

```
[ 0.999,
  0.9980010000000003,
  0.99600599600100004,
  etc.

  3.3406915454655646e-29,
  1.1160220001945103e-57,
  1.2455051049181556e-114,
  1.5512829663771860e-228,
  0.0 ]
```

Back to business... The formula (atom) matrix of a mixture — an ordered set of substances called a component list — is defined as a stoichiometry matrix where each of the columns is assigned to a substance and each of the rows is assigned to a chemical element (atom). The column sequence must correspond to the given component list, while the rows may come in any order. One simple example illustrates the concept:

```
[
[2, 4, 0, 2],  # H
[1, 0, 2, 0],  # O
[0, 1, 1, 0]   # C
]
```

This is the formula (atom) matrix corresponding to the component list: H2O, CH4, CO2 and H2. The generalization into more complex mixtures is straightforward. We shall, however, calculate the matrix by first parsing each formula into a dictionary telling how many atoms there are of each kind and then transcribe the dictionaries into a list of lists of stoichiometric numbers. For the simple example given above the programmatic actions look like:

```
['H2O', 'CH4', 'CO2', 'H2']

=>

[
{'H':2, 'O':1},
{'C':1, 'H':4},
{'C':1, 'O':2},
{'H':2}
]

=>
```

```
[  
[2, 4, 0, 2], # H  
[1, 0, 2, 0], # O  
[0, 1, 1, 0] # C  
]
```

In order to do so we need to learn about lists and dictionaries, and about iterators and list comprehensions in Python. Recursive functions are also into this picture since our formula parser is built on that principle.

5.9.1 Verbatim: “atom_matrix.py”

```
1  """
2  @summary:      Return the ( atoms x species ) formula matrix for a given list of
3  chemical formulas.
4  @author:       Tore Haug-Warberg
5  @organization: Department of Chemical Engineering, NTNU, Norway
6  @contact:      haugwarb@nt.ntnu.no
7  @license:      GPLv3
8  @requires:     Python 2.3.5 or higher
9  @since:        2011.08.30 (THW)
10 @version:      0.9
11 @todo 1.0:
12 @change:       started (2011.08.30)
13 """
14
15 def atom_matrix(formulas, debug=False):
16     """
17     Calculate an atom stoichiometry matrix which is conformal to the chemical
18     formulas given in list 'formulas'.
19
20     @param formulas: list of chemical formulas e.g. ['H2O', 'CO2', ... ]
21     @param debug:    True or False flag
22
23     @type formulas: <pass>
24     @type debug:    aBoolean
25
26     @return:         aList [ aList [ aNumber, aNumber, ... ] ]
27                     e.g. [[2, 0, ...], [1, 2, ...], [0, 1, ...], ...]
28     """
29
30     from atoms import atoms
31
32     import sys
33
34     if sys.version_info < (2, 4):
35         from sets import Set                      # deprecated since version 2.4
36         stack = [] # list of parsed formulas (dictionaries) e.g. {'H':2, 'O':1'}
37         syms  = Set()                           # set of unique atom names (chemical symbols)
38     else:
39         stack = [] # list of parsed formulas (dictionaries) e.g. {'H':2, 'O':1'}
40         syms  = set()                           # set of unique atom names (chemical symbols)
41
42     # Build 'stack' and 'syms'.
43     for formula in formulas:
44         stack.append({})
45         pass                                     # update chemical symbols Set
46
47     syms = list(syms)                         # transform set into list before sorting!
48     syms.sort()                             # sort atom names lexicographically (in-place sorting)
49
50     arr = []                                  # the atom stoichiometry 'matrix'
51
52     # Build 'arr'.
53     for sym in syms:                         # for all atoms
54         arr.append([])                      # make a new row of stoichiometric coefficients
55         for hsh in stack:                  # for all formulas
56             pass                            # fill in with values in the last row
57
58     return arr      # size is (m x n) where n = len(formulas) and m = len(syms)
```

5.9.2 Verbatim: “molecular_weight.py”

```

1  """
2  @summary:      Return molecular weight tuple (val, err) for a given formula.
3  @author:       Tore Haug-Warberg
4  @organization: Department of Chemical Engineering, NTNU, Norway
5  @contact:      haugwarb@nt.ntnu.no
6  @license:      GPLv3
7  @requires:     Python 2.3.5 or higher
8  @since:        2011.08.30 (THW)
9  @version:      0.9
10 @todo 1.0:
11 @change:       started (2011.08.30)
12 """
13
14 def molecular_weight(formula, debug=False, mw=[]):
15     """
16     Calculate molecular weight (mass per mole) of a substance with chemical
17     composition equal to 'formula'. The atomic masses of the elements are (by
18     default) taken from: M. E. Wieser, Atomic Weights of the Elements 2005, Pure
19     Appl. Chem., Vol. 78, No. 11, pp. 2051-2066, 2006 (see code), unless explic-
20     itly provided by the user (in list 'mw'). The calculated molecular weight
21     is returned as a scaled integer, i.e. val[0], where all the digits are sign-
22     ificant. The order of magnitude of the scaling is returned as a second value
23     val[1] such that the actual Mw = val[0]/val[1].
24
25     @param formula: a chemical formula 'COOH(C(CH3)2)3CH3'
26     @param debug:   True or False flag
27     @param mw:      list of tuple ('name', 'symbol', number, mass, uncertainty)
28
29     @type formula: <pass>
30     @type debug:   aBoolean
31     @type mw:      <pass>
32
33     @return:        theList [ anInt, anInt ]
34     """
35
36     # Chemical formula parser and transcendental math.
37     from atoms import atoms
38     import math
39
40     stack = pass          # parse formula into [{Symbol:int, 'Symbol':int, ...}]
41
42     if not stack: return [0, 1]           # no atom stoichiometry is available
43
44     hsh = pass                # continue with {Symbol:int, 'Symbol':int, ...}
45
46     # Enter periodic table information: The 'mw' list is either given as input
47     # to the function 'molecular_weight' or it is an empty list in which case it
48     # must be properly defined here.
49     mw = mw or \
50         [
51             ('carbon',      'C',    6,    12.0107,    8E-5),
52             ('hydrogen',    'H',    1,    1.00794,   7E-6)
53         ]
54
55     val = 0.0                  # molecular weight [amu]
56     err = 0.0                  # truncation error (approx. uncertainty)
57     m   = 0                     # number of elements recognized in the formula
58
59     # Calculate 'val', 'err' og 'm'.
60     for tup in mw:
61         if hsh.has_key(tup[1]):           # increment molecular weight
62             pass
63             pass                         # increment error (uncertainty)
64             pass                         # increment the number of elements in the formula
65         else:
66             pass
67
68     if m != len(hsh): raise SyntaxError("weird atom in %s" % (formula,))      #
69

```

```
70 |     n = abs(int(math.log10(err)))      # calculate order of magnitude (abs value)
71 |     if debug: print [val, err, n]
72 |
73 |     return [int(round(val*10**n)), 10**n] # make sure last digit is significant
```

Exercise 5

Preisig, H A

Chemical Engineering, NTNU

1 Question

The Plant

Subject of the exercise is the plant shown below. The plant mixes two components, namely A and W. The W is supplied from a reservoir, say water from the tap, whilst A is taken from a tank. The A tank is periodically filled, which is not shown on the scheme. A is pumped into the mixing tank, whilst W is flowing in freely. The product is taken from the bottom and communicates to the storage tank freely. No control on the latter, the flow is driven by the level differences. The S tank is a storage tank, which is somewhat larger than the mixing tank. A discrete consumer is drawing from the storage, whereby discrete implies that the consumer takes material at a constant rate over a given period of time. All tanks have an overflow for safety. M overflows on the top into S, whilst the A-tank and the S-tank overflow into a catchment.

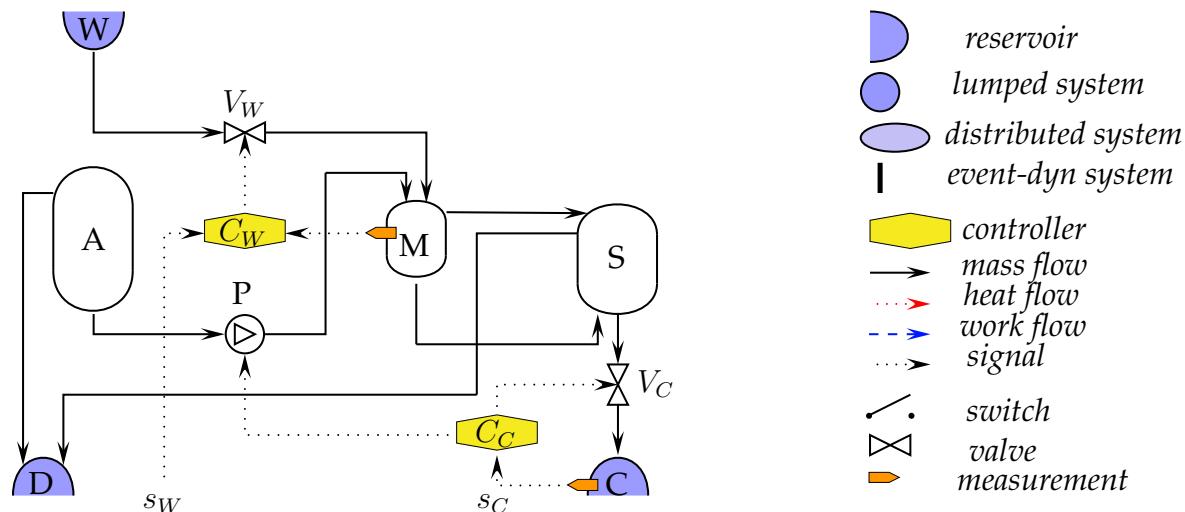


Figure 1: A complex mixing plant

The units in the plant are:

- W :: infinite source of solvent W
- A :: finite large source of component A, is refurbished periodically, which is not shown.
- P :: pump for moving material from A-tank to M-tank
- M :: mixing tank
- S :: product storage tank
- C :: consumer

- D :: catchment for possible spill
- V_W :: valve W flow
- V_C :: valve product flow controlled by the consumer
- C_W :: controller for W flow
- C_C :: controller for product flow
- s_W :: set point for level
- s_C :: set point for product flow set by the consumer

1.1 Topology

Provide a simple topology for the overall process. No evaporation takes place.

1.2 Model of a part

We isolate a part we already know, namely the infinite source W , and the mixing tank M extended with the storage tank and the overflow.

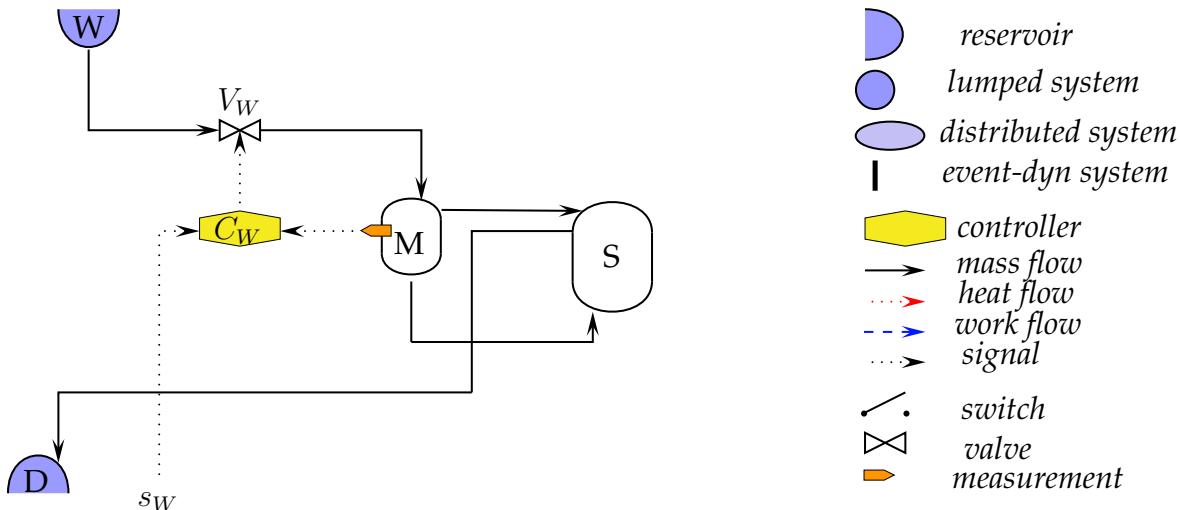


Figure 2: The core of the plant with a single feed and an overflow

- Establish topology for this part
- Write a scalar mass balance for the M and S . Scalar because we have only one component, namely W .
- Add transfer laws for the four mass streams.
- Add all the additional equations that link the mass in the tank with the observations and the effort variables. The tanks are cylindrical with a given cross sectional area. The density of the material is constant, the reservoir pressures are known, the

controller is a proportional controller with negative feedback, thus $u := -k_c(s_M - h_M)$, with s_M :: the setpoint for the level, h_M :: the level in tank M and k_c a proportional gain.

- NOTE: do not substitute the equations into the mass balances. Instead leave them as separate algebraic equations.
- Define state x , input u and output y for the M-S plant section. Wrap the secondary variables such as flows \hat{x} , the effort variables into a vector \underline{z} and the characteristic quantities like cross sectional area of the tanks, we collect in a vector $\underline{\Theta}$
- Rewrite model in an extended state-space notation using the above-defined symbols.

1.3 Dynamics

We now look at a simplified plant and focus on its dynamics.

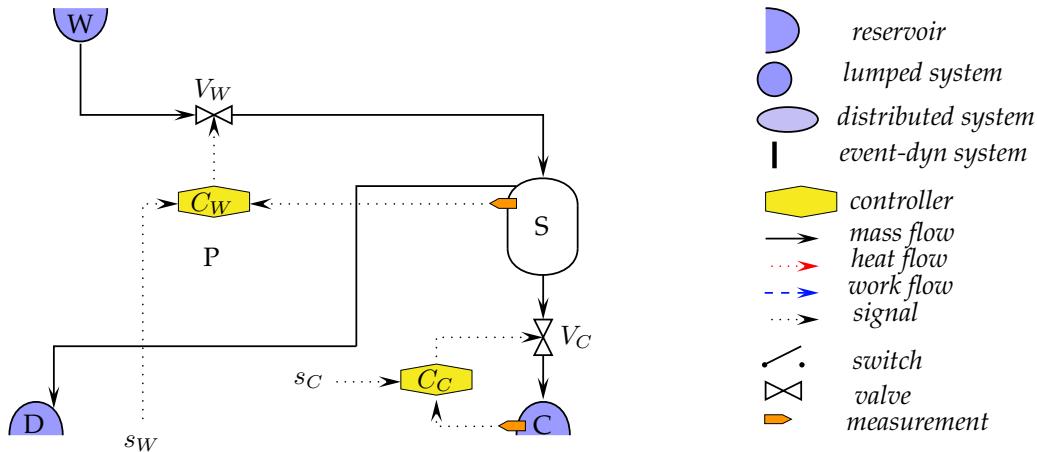


Figure 3: The simplified plant to simulate

To simplify the plant we have removed the A component handling completely and combined the mixing with the storage tank. So we de facto deal only with a tank for which we define a geometry because we measure the level in the tank. Again to simplify the problem we assume a cylindrical container with a given cross section area and a given maximum volume. For the simulation, we assume that we have a history of the consumer behaviour, thus a how much mass flow is leaving the S container.

The simplified model is:

$$\dot{x}(t) = \hat{x}_W(t) - \hat{x}_C(t) \quad (1)$$

$$\hat{x}_W(t) := \Theta_W u_W(t) \quad (2)$$

$$y(t) := \Theta_h x(t) \quad (3)$$

$$y(k) := y(k \Delta t) \quad t \in [k t, (k + 1) \Delta t] \quad (4)$$

$$u_W(k) := \Theta_p (y_s(k) - y(k)) \quad (5)$$

$$u_W(t) := u_W(k) \quad t \in [k t, (k + 1) \Delta t] \quad (6)$$

$$\hat{x}_C(t) := u_C(k) \quad t \in [k t, (k + 1) \Delta t] \quad (7)$$

How does this work? The dynamics are given by a continuous function (1). The rest of the plant works in discrete time, meaning that the values only change at the time $k \Delta t$ and stay constant until just before the next samples are taken. So at the beginning of the interval, one knows the measurement and can compute the state and also the consumption is known at that point in time, remaining constant until the next time event. Equation (4,6,7) say that things stay constant from the beginning of the time interval until right before the next time event, when the sample is taken and the controls are computed.

- Sketch the signals to get an insight on how this works.
- Substitute to get a single ordinary differential equation.
- Integrate both sides over the arbitrary time interval $t \in [k t, (k + 1) \Delta t]$.
- The result is a difference equation
- Write a program that computes the history of the state given
 - the initial state $x(t := 0) := 0.5$ and $\underline{\Theta} := [0.4, 3, 0.6]^T$
 - the history of the input u_C as a vector of numbers: $u_C(k := 1, \dots, 100) := [...]$
The first 10 values are 0.1, the next 10 are 0.2, the next 20 are 0.15, the next 10 are 0.25, the next 20 are 0.05 and the rest 0.2.
 - the setpoint $y_s(k) := 0.6$ to be constant over the time.
 - change the controller gain Θ_2 to see what happens. What happens if the product $|1 - \Theta_W \Theta_p \Theta_h|$ gets larger than 1 ?
 - Output data as text file so you can plot it.

1 Suggested Solutions

1.1 Topology

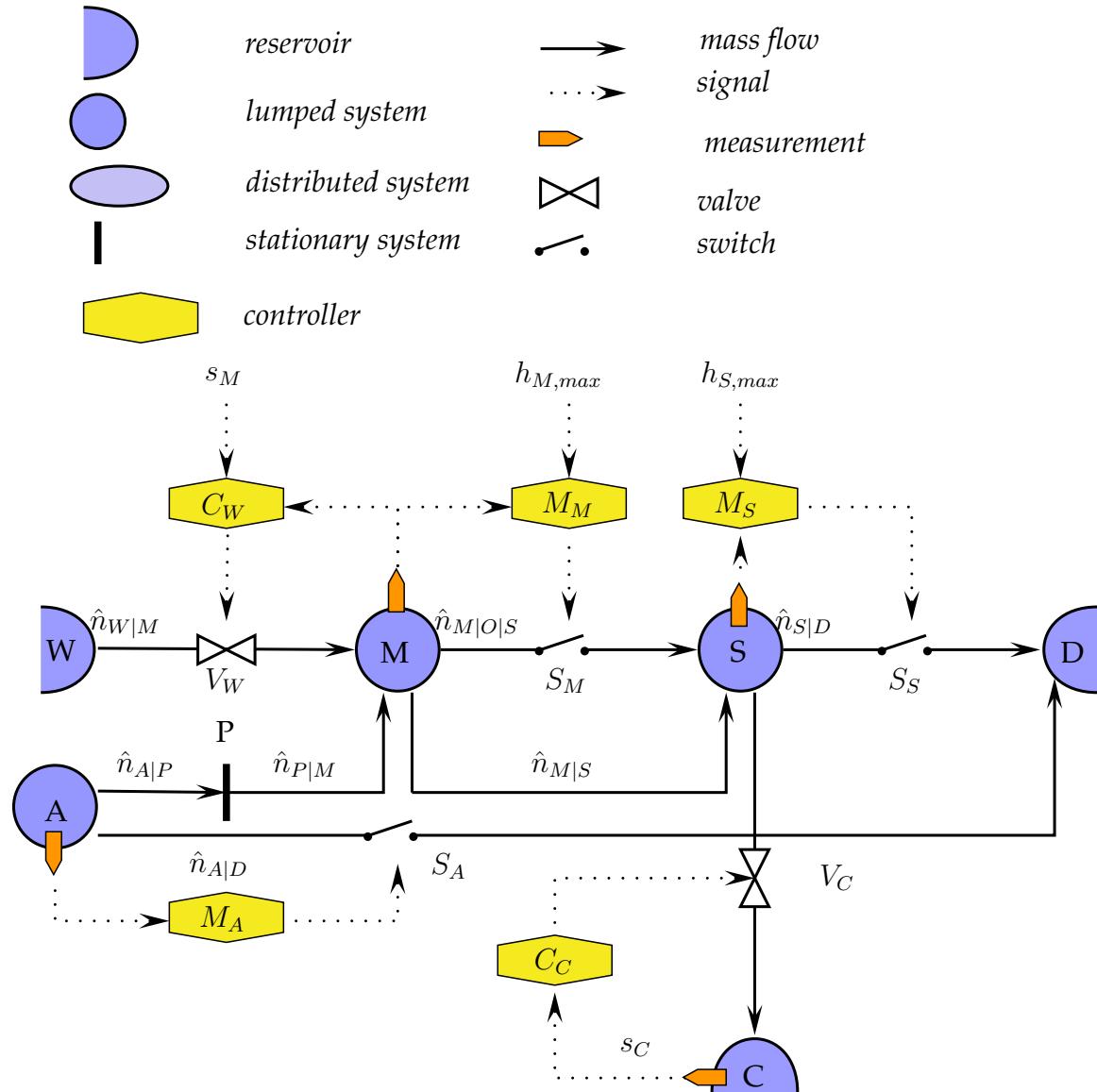


Figure 1: Topology of the complex mixing plant

1.2 Simplified plant

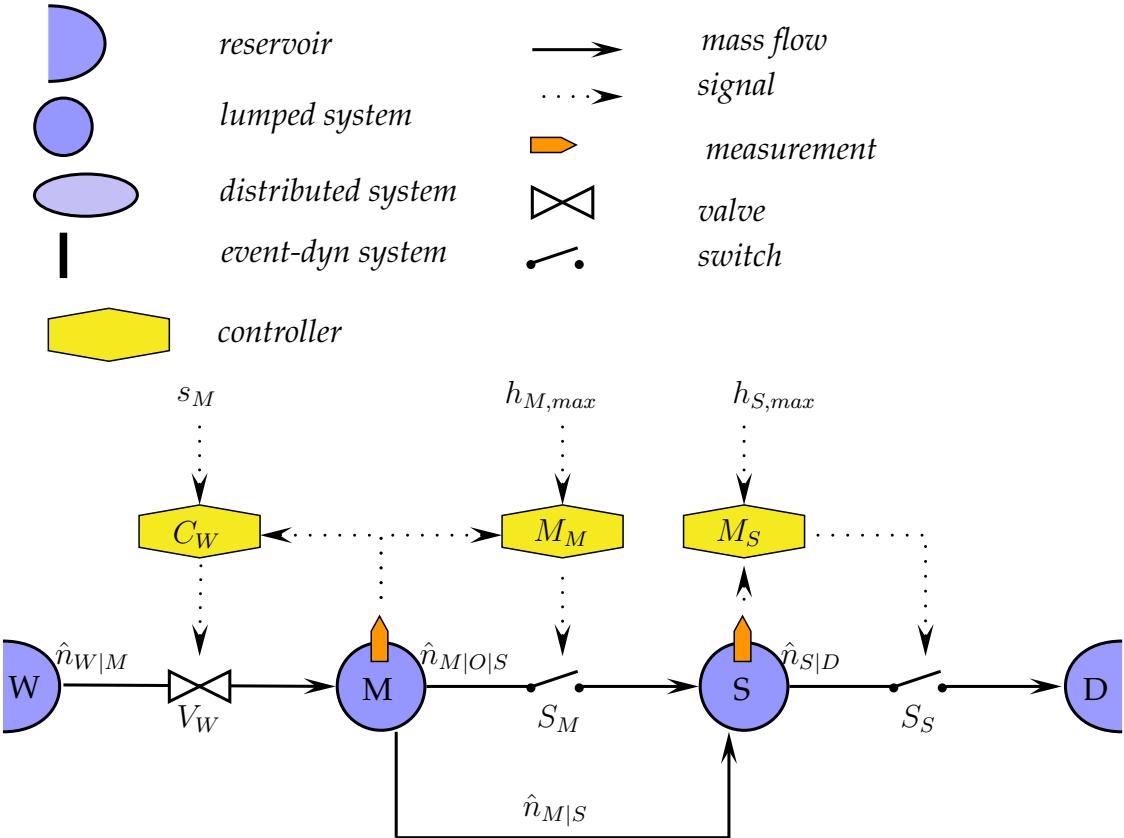


Figure 2: Topology of the simplified mixing plant

The two mass balances are:

$$\begin{aligned}\dot{n}_M &= +\hat{n}_{W|M} - \hat{n}_{M|S} - \hat{n}_{M|O|S} \\ \dot{n}_S &= +\hat{n}_{M|S} + \hat{n}_{M|O|S} - \hat{n}_{S|D}\end{aligned}$$

The flows:

$$\begin{aligned}d(\lambda_a, \lambda_b) &:= \text{sign}(\lambda_b - \lambda_a) \sqrt{|\lambda_b - \lambda_a|} \\ s(\lambda) &:= 1/2(1 + \text{sign}(\lambda)) \\ \hat{n}_{W|M} &:= -k_{W|M} u_W d(p_W, p_M) \\ \hat{n}_{M|S} &:= -k_{M|S} d(p_M, p_S) \\ \hat{n}_{M|O|S} &:= s(h_M - h_{M,max}) \hat{n}_{W|M} \\ \hat{n}_{S|D} &:= s(h_S - h_{S,max}) (\hat{n}_{M|S} + \hat{n}_{M|O|S})\end{aligned}$$

The measurements and state limits for the systems $s \in [W, M]$:

$$\begin{aligned}V_s &:= \rho^{-1} n_S \\ h_s &:= \frac{V_s}{A_s}\end{aligned}$$

The driving forces:

$$\begin{aligned} p_W &:= \text{constant} \\ p_M &:= \rho g h_M \\ p_S &:= \rho g h_S \end{aligned}$$

Control:

$$u_W := -k_c (s_M - h_M)$$

Given:

$$\begin{aligned} p_W &:: \text{pressure of reservoir} \\ k_{a|b} &:: \text{for the two pressure-driven streams} \\ A_s &:: \text{cross sectional area for } s \in [M.S] \\ s_W &:: \text{set point for the level controller} \end{aligned}$$

State space notation:

$$\begin{aligned} \text{state} \quad \underline{x} &:= [n_M, n_S]^T \\ \text{measurements} \quad \underline{y} &:= [h_M, h_S]^T \\ \text{input} \quad \underline{u} &:= [p_W, p_D, s_M, h_{M,max}, h_{S,max}, u_W] \\ \text{secondary states} \quad \underline{z} &:= [\hat{n}_{W|M}, \hat{n}_{M|S}, \hat{n}_{M|O|S}, \hat{n}_{S|D}, V_M, V_S, h_M, h_S, p_M, p_S]^T \\ \text{parameters} \quad \underline{\Theta} &:= [k_{W|M}, k_{M|S}, A_M, A_S, \rho, g, k_c]^T \end{aligned}$$

1.2.1 State-Space version

Balances:

$$\begin{aligned} \dot{x}_1 &= +z_1 - z_2 - z_3 \\ \dot{x}_2 &= +z_2 + z_3 - z_4 \end{aligned}$$

Flows:

$$\begin{aligned} z_1 &:= -\Theta_1 u_6 d(u_1, z_9) \\ z_2 &:= -\Theta_2 d(z_9, z_{10}) \\ z_3 &:= s(z_7 - u_4) z_1 \\ z_4 &:= s(z_8 - u_5) (z_1 + z_3) \end{aligned}$$

Volumes:

$$\begin{aligned} z_5 &:= \Theta_5^{-1} x_1 \\ z_6 &:= \Theta_5^{-1} x_2 \end{aligned}$$

Levels:

$$\begin{aligned}z_7 &:= z_5 \Theta_3^{-1} \\z_8 &:= z_6 \Theta_4^{-1}\end{aligned}$$

Pressures

$$\begin{aligned}z_9 &:= \Theta_5 \Theta_6 z_7 \\z_{10} &:= \Theta_5 \Theta_6 z_8\end{aligned}$$

$$\begin{aligned}u_1 &:: \text{constant \& given} \\u_2 &:: \text{constant \& given} \\u_3 &:: \text{constant \& given} \\u_4 &:: \text{constant \& given} \\u_5 &:: \text{constant \& given} \\u_6 &:= -\Theta_7 d(u_3, z_7)\end{aligned}$$

2 Suggested Solutions: Dynamics

Crimp model to a single ODE:

$$\begin{aligned}
\dot{x} &:= \hat{x}_W(t) - \hat{x}_C(t) \\
&:= \Theta_W u_W(t) - u_C(k) \\
&:= \Theta_W u_W(k) - u_C(k) \\
&:= \Theta_W \Theta_p (y_s(k) - y(k)) - u_C(k) \\
&:= \Theta_W \Theta_p (y_s(k) - \Theta_h x(k)) - u_C(k)
\end{aligned}$$

The right-hand-side is now constant for the duration of the time interval $t \in [k \Delta t, (k+1) \Delta t]$ and we get:

$$\begin{aligned}
\int_{k \Delta t + \epsilon}^{(k+1) \Delta t} \dot{x} dt &:= \int_{k \Delta t + \epsilon}^{(k+1) \Delta t} \Theta_W \Theta_p (y_s(k) - \Theta_h x(k)) - u_C(k) dt \\
(x(k+1) - x(k)) \Delta t &:= (\Theta_W \Theta_p (y_s(k) - \Theta_h x(k)) - u_C(k)) \int_{k \Delta t + \epsilon}^{(k+1) \Delta t} dt \\
&:= (\Theta_W \Theta_p (y_s(k) - \Theta_h x(k)) - u_C(k)) \Delta t \\
x(k+1) - x(k) &:= \Theta_W \Theta_p (y_s(k) - \Theta_h x(k)) - u_C(k) \\
x(k+1) &:= x(k) + \Theta_W \Theta_p (y_s(k) - \Theta_h x(k)) - u_C(k) \\
&:= x(k) - \Theta_W \Theta_p \Theta_h x(k) + \Theta_W \Theta_p y_s(k) - u_C(k) \\
&:= (1 - \Theta_W \Theta_p \Theta_h) x(k) + \Theta_W \Theta_p y_s(k) - u_C(k)
\end{aligned}$$

A program in Matlab

```
% dynamics (solution)
%
% 2012-09-25 Preisig, H A

% parameters
% flow constant, controller gain, geometry
P = [0.4, 3, 0.8]';
1-P(1)*P(2)*P(3)

% initial conditions & allocate array
x = zeros(100,1);
x(1) = 0.5;

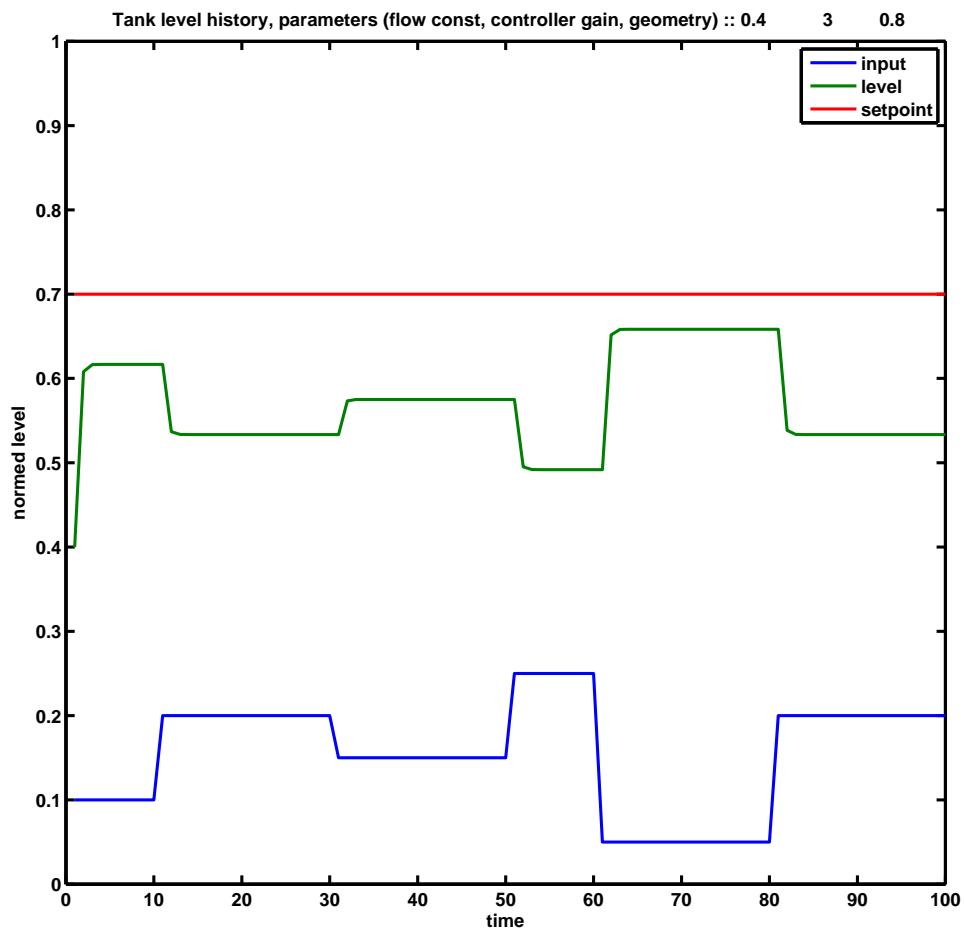
ys = 0.7;

u = [0.1 * ones(10,1)
      0.2 * ones(20,1)
      0.15 * ones(20,1)
      0.25 * ones(10,1)
      0.05 * ones(20,1)
      0.20 * ones(20,1)];

for k = 1:99
    x(k+1) = (1-P(1)*P(2)*P(3)) *x(k) + P(1) * P(2) * ys - u(k);
end

plot([u, P(3) * x, ys*ones(100,1)])
legend('input','level','setpoint')
s = num2str(P')
title(['Tank level history, parameters (flow const, controller gain, geometry) :: ', s])
xlabel('time')
ylabel('normed level')
ax = axis();
ax(3:4) = [0,1]
axis(ax)
```

Results



A program in Python

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
@summary:      Simulates a simple tank with P control
@author:       Preisig, H A
@copyright:    Preisig, H A
@organization: Department of Chemical Engineering, NTNU, Norway
@contact:      heinz.preisig@chemeng.ntnu.no
@license:      GPLv3
@requires:     Python 2.7.1 or higher
@since:        2012.10.05
@version:      1.0
@todo:         2.0:
@change:       2012.10.08
"""

def matnums(val, cols):
    return [val for i in range(0,cols)]

def matfunc(f='dynamics_01.dat', debug = False):
    txtfile = open(f,'w')
    # Parameters
    # Flow constant, controller gain, geometry

    P = [0.4, 3, 0.8]
    range_k = range(0,100)

    print 'Stability argument: ',1-P[0]*P[1]*P[2]

    # Initial condition
    x = [0.5]
    y = [P[2] * x[0]]

    ys = 0.7

    u = matnums(0.1, 10) + matnums(0.2, 20) + matnums(0.15, 20) + \
        matnums(0.25, 10) + matnums(0.05, 20) + matnums(0.2, 20)

    for k in range_k:
        x.append((1 - P[0] * P[1] * P[2]) * x[k] + P[0] * P[1] * ys - u[k])
        y.append(P[2] * x[-1])

        if debug:
            print str(u[k]) + ', ' + str(y[k]) + ', ' + str(ys)
        else:
            print>>txtfile, (k+1), str(u[k]), str(y[k]), str(ys)
```

```
if not debug: print 'Data written to file :', f
txtfile.close()

return (range_k, \
        [x[i] for i in range_k], \
        [y[i] for i in range_k], \
        ys, \
        [u[i] for i in range_k], \
        P \
        )

if __name__ == '__main__':
    matfunc(debug=False)
```

Independent Reactions (TKP4106)



Zooball/Fish

Reasons computers are male:

- In order to get their attention, you have to turn them on.
- They have a lot of data but are still clueless.
- They are supposed to help you solve your problems, but half the time they cause the problem.
- As soon as you commit to one, you realize that if you had waited a little longer, you could have had a better model.

Computers are male

Assignments

1. Write a procedure `rref` for calculating the row-reduced-echelon $\text{rref}(A) = \text{inv}(G)*A$ of a given matrix A . Matrix G is formally required, but it will never show up in the code. The return values shall be matrix $B = [B1^T, 0]^T$ where B is of the same shape as A , $\text{rank}(A) = \text{rank}(B1)$, and $\text{pivots}(B) = [\text{None} | \text{anInt}, \dots]$ identifying the pivot elements used in the elimination process (row pivoting only). That is $\text{rref}(A) \Rightarrow B, \text{rank}(B1), \text{pivots}(B)$. Use the stub program [rref.py](#) as template.
2. Based on the output of `rref` write another procedure for calculating the nullspace N of A such that $[A^T, N]$ makes an invertible basis for the vector space. That is $\text{null}(A) \Rightarrow N, \text{rank}(N)$ where $\text{rank}(A)+\text{rank}(N)=\text{rowdim}(A)$. Use the stub program [null.py](#) as template.

Read this whitepaper about [The mass balance](#) if you need a more thorough explanation of the nullspace theory than you will find on the current page.

From formula matrix A we can calculate a row-reduced-echelon form $B1$ by doing Gauss elimination on the rows of A . This process will require row permutations if one of the pivot elements becomes zero, but it does without any column permutations. Let $\text{inv}(G)$ be a matrix that is doing the steps needed. Then, by definition $\text{rref}(A) = \text{inv}(G)A$. The shape of $\text{rref}(A)$ is the same as A but it may have one or more rows being fully zero (filling out the lower part of the matrix) even when A is dense. Hence, $\text{rref}(A) = [B1^T, 0]^T$ where

the 0 matrix may or may not exist.

The next operation is to make an elementary matrix $E1$ by putting all non-pivot columns in $B1$ to zero. These are the columns that have not been fully row-reduced (an invertible matrix has, by the way, no such columns). This process is hard to explain in words, but the examples below are quite illuminating.

From $B1$ and $E1$ we can easily calculate $E1*B1-I$. This matrix has the property that $B1(E1*B1-I) = 0$. Prove it! Furthermore, we can show (after a second or maybe third thought) that $A(E1*B1-I) = 0$. This means the non-zero columns of $E1*B1-I$ define the null space of A . Hence $N = (E1*B1-I)$.

Our first example is a one-component mixture of water. Water (H_2O) has 2 hydrogen atoms and 1 oxygen atom. The formula matrix and the corresponding Gauss elimination is shown below:

```
A      = [ [ 2 ]   'H'  
          [ 1 ] ] 'O'  
  
Step #1: 0.5*R1  
Step #2: R2 - R1  
  
rref(A) = [ [ 1 ]  
            [ 0 ] ]  
  
inv(G)  = [ [ 0.5 0 ]  
            [ -0.5 1 ] ]  
  
B1      = [ [ 1 ] ]  
  
rank    = 1  
  
pivots  = [ 0 ]  
  
E1      = [ [ 1 ] ]  
  
E1*B1-I = [ [ 0 ] ]  
  
N      = [ [ ] ]
```

The second example is a binary mixture of water monomer and water dimer (H_2O , $(H_2O)_2$). Note that A is a square matrix, albeit with two linearly dependent rows. $rref(A)$ has therefore a zero row at the end which means $B1$ has only 1 row while A got 2. We say that A is rank deficient, which means there is the possibility of a chemical reaction in the mixture. From the stoichiometry of N we can deduce $2*H_2O - 1*(H_2O)_2 = 0$ or $2*H_2O = (H_2O)_2$. The two forms are equivalent.

```
A      = [ [ 2 4 ]   'H'  
          [ 1 2 ] ] 'O'  
  
Elimination step 1: 0.5*R1
```

```

Elimination step 2: R2 - R1

rref(A) = [ [ 1 2 ]
            [ 0 0 ] ]

inv(G)  = [ [ 0.5 0 ]
            [ -0.5 1 ] ]

B1      = [ [ 1 2 ] ]

rank    = 1

pivots = [ 0 None ]

E1      = [ [ 1 ]
            [ 0 ] ]

E1*B1-I = [ [ 0 2 ]
             [ 0 -1 ] ]

N      = [ [ 2 ]
            [ -1 ] ]

```

The third example is a binary mixture of hydrogen and oxygen (H₂, O₂). Again, A is a square matrix but this time it is non-singular. This means there are no chemical reaction possible.

```

A      = [ [ 2 0 ]      'H'
          [ 0 2 ] ]      'O'

Elimination step 1: 0.5*R1
Elimination step 2: 0.5*R2

rref(A) = [ [ 1 0 ]
            [ 0 1 ] ]

inv(G)  = [ [ 0.5 0 ]
            [ 0 0.5 ] ]

B1      = [ [ 1 0 ]
            [ 0 1 ] ]

rank    = 2

pivots = [ 0 1 ]

E1      = [ [ 1 0 ]
            [ 0 1 ] ]

E1*B1-I = [ [ 0 0 ]
             [ 0 0 ] ]

N      = [ [ ]
            [ ] ]

```

The fourth example is a quinary mixture of formaldehyde, carbon monoxide, hydrogen, water and oxygen (CHOH, CO, H2, H2O, O2). This is an almost fullblown example (it does not require row permutations though) because the elimination process leaves a non-pivot column in the middle of A. The rank of A is 3 (all rows are independent) and the row-size is 5. That means there are 2 degrees of freedom which manifest themselves as chemical reactions. From the stoichiometry matrix N we get: $1*\text{CHOH} - 1*\text{CO} - 1*\text{H}_2 = 0$ and $-2*\text{CHOH} + 2*\text{CO} + 2*\text{H}_2\text{O} - 1*\text{O}_2 = 0$, or, alternatively, CHOH = CO + H2 and 2*CO + 2*H2O = 2*CHOH + O2.

```

A      = [ [ 1 1 0 0 0 ]      'C'
          [ 2 0 2 2 0 ]      'H'
          [ 1 1 0 1 2 ] ]  'O'

Elimination step 1: R2 - 2*R1
Elimination step 2: R3 - 1*R1
Elimination step 3: -0.5*R2
Elimination step 4: R1 - 1*R2
Elimination step 5: R1 - 1*R3
Elimination step 6: R2 + 1*R3

rref(A) = [ [ 1 0 1 0 -2 ]
            [ 0 1 -1 0 2 ]
            [ 0 0 0 1 2 ] ]

inv(G)  = [ [ 1 0.5 -1 ]
            [ 0 -.5 1 ]
            [ -1 0 1 ] ]

B1      = [ [ 1 0 1 0 -2 ]
            [ 0 1 -1 0 2 ]
            [ 0 0 0 1 2 ] ]

rank    = 3

pivots  = [ 0 1 None 3 None ]

E1      = [ [ 1 0 0 ]
            [ 0 1 0 ]
            [ 0 0 0 ]
            [ 0 0 1 ]
            [ 0 0 0 ] ]

E1*B1 - I = [ [ 0 0 1 0 -2 ]
               [ 0 0 -1 0 2 ]
               [ 0 0 -1 0 0 ]
               [ 0 0 0 0 2 ]
               [ 0 0 0 0 -1 ] ]

N      = [ [ 1 -2 ]
          [ -1 2 ]
          [ -1 0 ]
          [ 0 2 ]
          [ 0 -1 ] ]

```

5.11.1 Verbatim: “rref.py”

```

1  """
2  @summary:      Calculate the row-reduced echelon form of a given matrix.
3  @author:       Tore Haug-Warberg
4  @organization: Department of Chemical Engineering, NTNU, Norway
5  @contact:      haugwarb@nt.ntnu.no
6  @license:      GPLv3
7  @requires:     Python 2.3.5 or higher
8  @since:        2011.08.30 (THW)
9  @version:      0.8
10 @todo 1.0:
11 @change:       started (2011.08.30)
12 """
13
14 def rref(amat, debug=False):
15     """
16     Calculate the row-reduced-echelon of 'amat' using Gauss elimination of the
17     rows. There is partial pivoting only - ie no column permutations. The output
18     is a matrix of the same shape as 'amat'::
19
20         | 0 ... 0 1 * ... 0 * ... 0 * ... * |
21         | 0 ... 0 0 0 ... 1 * ... 0 * ... * |
22         | 0 ... 0 0 0 ... 0 0 ... 0 * ... * |
23     rref(amat) = | :   :   :   :   :   :   :   :   |
24         | 0 ... 0 0 0 ... 0 0 ... 1 * ... * |
25         | 0 ... 0 0 0 ... 0 0 ... 0 0 ... 0 |
26         | :   :   :   :   :   :   :   :   :   |
27         | 0 ... 0 0 0 ... 0 0 ... 0 0 ... 0 |
28
29     Notice the zero blocks at the left and bottom of 'rref(amat)'. For chemical
30     formula matrices the left block is always missing while the bottom block is
31     present in the case 'amat' is rank deficient (more atoms than components for
32     example). The 'rank' of 'rref(amat)' is equal to the number of non-zero
33     rows. The 'pivots' list holds the position of all the pivot elements used in
34     the elimination, i.e. [None, ..., None, i, None, ..., j, None, ..., k, None,
35     ..., None] in the example above. Note: The output matrix elements are con-
36     verted to Float irrespective of what comes in (Int or Float).
37
38     @param amat: Input matrix given as a list of lists of numbers
39     @param debug: True or False flag
40
41     @type amat: aList [ aList [ aNumber, aNumber, ... ], ... ]
42     @type debug: aBoolean
43
44     @return:    aList [ rref(amat), anInt, aList [ None | anInt, ... ] ]
45     """
46
47     if not(amat) or not(amat[0]):
48         raise ArithmeticError("zero rows in amat '%s'"%(amat,))
49
50     amat = pass                      # make work copy and convert to float
51     pivots = range(0, len(amat[0]))    # assume len(amat[0]) = len(amat[1]) = ...
52     rank = 0                          # initialize number of non-zero rows in amat
53
54     if debug: print '\nrref()\n' + \
55                           '\ninputamat=' + str(amat)
56
57     for c in pivots:                  # consider all columns of amat
58         piv, val = 0, 0.0             # starting pivot row, pivot value
59         for r in range(pass, pass)   # partial pivoting of remaining rows
60             arc = pass                # current amat[row,column] element
61             if abs(arc) > abs(val):  # new pivot candidate found
62                 pass                  # change pivot row, pivot value
63
64             if debug:
65                 print '\namat:' + str(amat) + \
66                               '\ncolumn:' + str(c) + \
67                               '\npivot_element:' + str(piv) + \
68                               '\npivot_value:' + str(val)
69

```

```

70     if val != 0.0:                      # a non-zero pivot value was found
71         pass                                # swap rows
72
73     for j in range(pass, pass)            # start pivot row scaling
74         pass                                # make amat[rank][c] = 1
75
76     # Note reversed order in row elimination. You either has to do this,
77     # or use a temporary variable. If you use j in range(c,len(pivots))
78     # then amat[i][c] is changed at the very beginning of the loop which
79     # screws up the algorithm.
80     for i in range(pass, pass)          # start row elimination
81         if i == rank: continue        # ignore pivot row
82         for j in range(pass, pass)    # reversed row elimination
83             pass                                # make amat[i][c] = 0
84
85     rank += 1                           # increase the rank
86 else:                                # zero pivot value
87     pivots[c] = None                  # current column is not a free variable
88
89 if debug:
90     print '\noutputamat:\n' + str(amat) + \
91         '\nrank:\n' + str(rank) + \
92         '\npivots:\n' + str(pivots)
93
94 return [amat, rank, pivots]

```

5.11.2 Verbatim: “null.py”

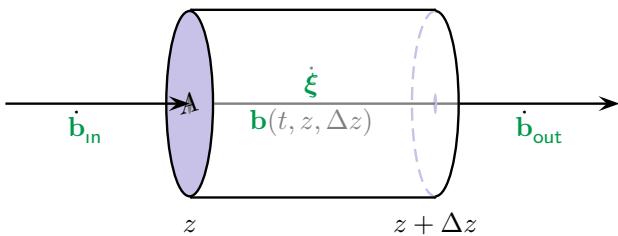
```
1  """
2  @summary:      Calculate the nullspace of a given matrix.
3  @author:       Tore Haug-Warberg
4  @organization: Department of Chemical Engineering, NTNU, Norway
5  @contact:      haugwarb@nt.ntnu.no
6  @license:      GPLv3
7  @requires:     Python 2.3.5 or higher
8  @since:        2011.08.30 (THW)
9  @version:      0.9
10 @todo 1.0:
11 @change:       started (2011.08.30)
12 """
13
14 def null(amat, debug=False):
15     """
16     Calculate the nullspace of 'amat' from rref(amat) and fiddling around with
17     the Gauss elimination structure. The result is that amat*null(amat) = zero.
18     That's all. No fancy mathematics like e.g. orthonormalization of the null-
19     space.
20
21     @param amat: Input matrix given as a list of lists of numbers
22     @param debug: True or False flag
23
24     @type amat: aList [ aList [ aNumber, aNumber, ... ], ... ]
25     @type debug: aBoolean
26
27     @return:      aList [ aList [ aFloat, aFloat, ... ] ]
28                 e.g. [[1.0, 0.0], [0.0, 1.0], [-1.0, 0.0], [0.0, -1.0], ...]
29
30     """
31
32     # Row-reduced-echelon-form.
33     from rref import rref
34
35     bmat, rank, pivots = rref(amat, debug)
36
37     if debug:
38         print '\nnull()\n' + \
39             '\ninput_bmat=' + str(bmat) + \
40             '\ninput_rank=' + str(rank) + \
41             '\ninput_pivots=' + str(pivots)
42
43     # Insert -1 along the main diagonal for each of the dependent variables.
44     for r in [i for i in range(0,len(pivots)) if pivots[i] == None]:
45         pass
46
47     # Strip off rows that have been pushed outside the matrix boundary (they are
48     # anyway fully zero).
49     pass
50
51     # Remove the columns corresponding to independent variables in the nullspace
52     # solution.
53     for r in range(0,len(pivots)):
54         if debug:
55             print '\nbmat=' + str(bmat) + \
56                 '\nrow=' + str(r)
57
58     # Remove independent variables by popping from right to left.
59     for c in [pass]:
60         pass
61
62     if debug: print '\noutput_bmat=' + str(bmat)
63
64     return bmat
```

Plug Flow Reactor. Part I

Tore Haug-Warberg
Department of Chemical Engineering
NTNU (Norway)

23 August 2011
(completed after 120 hours of writing, programming and testing)

1 The mass balance



From Einstein's mass-energy equivalence $E = mc^2$ we know that energy and mass are in principle convertible state properties. At least so for relativistic processes and nuclear reactions. In everyday physics and chemistry the mass changes are so small, however, that

we are not able to measure them correctly, and for all practical purposes we may therefore assume that mass and energy are independent properties. The mass balance of an open system can then be written

$$M(t, z, \Delta z) = \int_0^t \dot{M}_{in} d\tau - \int_0^t \dot{M}_{out} d\tau .$$

In this equation M is used (rather than m) for the total mass to conform with thermodynamic practise where extensive quantities are designated by capital letters. The balance of total mass is an absolute must for all non-nuclear systems, but for multicomponent mixtures of chemical origin we can go a bit further. The balance principle does not only apply to the total mass, but to the mass of each individual atom in the mixture. Or, we may consider the mole number B_i of each atom since the atomic masses are constant properties of the atoms. This means that the mass $M_i = B_i * M_{w,i}$ of atom i is conserved if B_i is conserved. Let $\mathbf{b} \triangleq [B_1, B_2, \dots]$ be a vector of mole numbers for all the atoms in the mixture. The mass balance of an open chemical system is then

$$\mathbf{b}(t, z, \Delta z) = \int_0^t \dot{\mathbf{b}}_{in} d\tau - \int_0^t \dot{\mathbf{b}}_{out} d\tau$$

To proceed we need to embroider the concepts of *chemical formulas* and *chemical reactions*. Quite interestingly, we can in the present context look upon chemical formulas as algebraic expressions written on a very condensed form. Take for instance iron(II)-acetate: $\text{Fe}(\text{CH}_3\text{COO})_2 \cdot 4\text{H}_2\text{O}$. Using standard rules of operation (from IUPAC) the formula expands to:

$$\begin{aligned}\text{Fe}(\text{CH}_3\text{COO})_2 \cdot 4\text{H}_2\text{O} &= \text{Fe} + 2 \cdot (2\text{C} + 3\text{H} + 2\text{O}) + 4 \cdot (2\text{H} + \text{O}) \\ &= \text{Fe} + 4\text{C} + 14\text{H} + 8\text{O}\end{aligned}$$

Convince yourself that this expression evaluates to the molecular weight of iron(II)-acetate provided the symbols Fe, C, H and O are assigned to the atomic masses of the chemical elements in question. You can also verify that the summation of pair products (a number times a symbol) are the only operations needed in the calculation. This makes matrix algebra a useful tool since the inner product of matrix algebra is just that—a summation of pair products. By considering a mixture of *known* chemical substances it is possible to make a corresponding list of all atoms encountered in the mixture. The link between these two lists is the so-called formula matrix. Let again $\mathbf{b} \triangleq [B_1, B_2, \dots]$ and this time also $\mathbf{n} \triangleq [N_1, N_2, \dots]$ where N_i is the mole number of compound i often referred to as substance i . Using matrix algebra we can now write:

$$\mathbf{b} = \mathbf{A}\mathbf{n}$$

The stoichiometric coefficients of each substance, of which iron(II)-acetate is one example, are collected into the corresponding columns of \mathbf{A} . Albeit quite trivial, the principle is best served by a concret example. Take e.g. the combustion of methane (CH_4) in air (0.78 N_2 , 0.21 O_2 and 0.01 Ar) to the reaction products CO , CO_2 , H_2O , H_2 , OH , H and NO . Altogether there are 11 substances and 5 atoms in the mixture:

$$\mathbf{A} = \begin{pmatrix} \text{CH}_4 & \text{N}_2 & \text{O}_2 & \text{Ar} & \text{CO} & \text{CO}_2 & \text{H}_2\text{O} & \text{H}_2 & \text{OH} & \text{H} & \text{NO} \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 2 & 2 & 1 & 1 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 & 1 & 2 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{matrix} \text{C} \\ \text{H} \\ \text{N} \\ \text{O} \\ \text{Ar} \end{matrix}, \quad (1)$$

and, to make what we are talking about absolutely clear:

$$\mathbf{n} = (N_{\text{CH}_4} \ N_{\text{N}_2} \ N_{\text{O}_2} \ N_{\text{Ar}} \ N_{\text{CO}} \ N_{\text{CO}_2} \ N_{\text{H}_2\text{O}} \ N_{\text{H}_2} \ N_{\text{OH}} \ N_{\text{H}} \ N_{\text{NO}})^T,$$

$$\mathbf{b} = (B_{\text{C}} \ B_{\text{H}} \ B_{\text{N}} \ B_{\text{O}} \ B_{\text{Ar}})^T.$$

The mass balance is now written

$$\mathbf{A}\mathbf{n}(t, z, \Delta z) = \int_0^t \mathbf{A}\dot{\mathbf{n}}_{\text{in}} \, d\tau - \int_0^t \mathbf{A}\dot{\mathbf{n}}_{\text{out}} \, d\tau, \quad (2)$$

but \mathbf{A} is usually a singular matrix (except for mixtures of pure elements) which prohibits a simple solution to these equations. The physical reasoning is that there can occur chemical transpositions in the system taking one set of substances (reactants) into another set of substances (products). This transposition is called *chemical reaction*. It is known by experiment that chemical reactions can change the composition of the system without altering the mole numbers of the atoms. The mathematical explanation of the phenomena lies in the nullspace of \mathbf{A} . It is defined as a matrix \mathbf{N} such that $\mathbf{AN} = \mathbf{0}$ and where $(\mathbf{A}^T \mathbf{N})$ constitutes an invertible matrix of full rank. From the definition of the nullspace it is clear that whatever happens in the column space of \mathbf{N} it will not affect the atoms vector \mathbf{b} . To make this situation very clear we shall consider a *closed* system that is changed from one compositional state 1 to another state 2 . The equations describing the changes are listed below:

$$\begin{aligned}\mathbf{b}_2 &= \mathbf{b}_1 \\ \mathbf{An}_2 &= \mathbf{An}_1 \\ \mathbf{A}(\mathbf{n}_2 - \mathbf{n}_1) &= \mathbf{0} \\ \mathbf{A}\Delta\mathbf{n} &= \mathbf{0}\end{aligned}$$

If we now calculate $\Delta\mathbf{n}$ as a linear combination of the columns of \mathbf{N} we have a full-blown solution to the mass balance problem of the closed system:

$$\Delta\mathbf{n} = \mathbf{N}\xi \quad \Rightarrow \quad \mathbf{A}\Delta\mathbf{n} = \mathbf{AN}\xi = \mathbf{0}$$

The elements ξ_i of the solution vector ξ are the extents of reaction for each *independent* reaction in the system. With this understanding in mind we can recast the mass balance into

$$\mathbf{n}(t, z, \Delta z) = \int_0^t \dot{\mathbf{n}}_{in} d\tau - \int_0^t \dot{\mathbf{n}}_{out} d\tau + \int_0^t \int_z^{z+\Delta z} A \mathbf{N} \dot{\xi} d\zeta d\tau, \quad (3)$$

where A stands for the cross-sectional area of the reactor (perpendicular to the flow) and $\dot{\xi}$ is the vector of independent reaction rates (moles per unit time and volume). It is easy to verify that Eq. 3 is a solution of Eq. 2. Multiplying by \mathbf{A} on both sides of the equation makes the chemical reaction integral drop out because $\mathbf{AN} = \mathbf{0}$. Eq. 2 is thereby reduced to Eq. 3.

To calculate actual numbers for ξ_i we need to model either the reaction kinetics or the thermodynamic equilibria (or both) in the mixture, and to do this we must couple the mass balance equations with the energy and impulse balances of the system. This is our ultimate goal explained in the Part III of this paper entitled *Modelling Issues*.

We must first concentrate on the nullspace calculation, however, and find a clear-cut and solid way to do the matrix operations that are needed. There are several nullspace algorithms on the market but we shall define our own. The reasons are twofold: Firstly, the problems we are dealing with are on a tiny scale (5–20 variables) and there is no need for a very fast and numerically secure algorithm. Secondly, bringing in an advanced nullspace algorithm has the disadvantage that we do not learn much about simpler things

like Gauss-elimination, row dependencies and matrix ranks. Calculating the *row reduced echelon* (starcaise) form $\mathbf{B} = \text{rref}(\mathbf{A}) = \mathbf{G}^{-1}\mathbf{A}$ is one way to define the nullspace. Let \mathbf{G} be an invertible matrix doing a sequence of zero or more steps of Gauss-elimination to reach the following result:

$$\mathbf{B} \triangleq \left(\begin{array}{c} \mathbf{B}_1 \\ \mathbf{0} \end{array} \right) = \mathbf{G}^{-1}\mathbf{A} = \left(\begin{array}{ccc|cccccccccc} 0 & \cdots & 0 & 1 & * & \cdots & 0 & * & \cdots & 0 & * & \cdots \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 1 & * & \cdots & 0 & * & \cdots \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & \cdots & 1 & * & \cdots \\ \hline 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots \end{array} \right)$$

The matrix element $*$ can be any real number (i.e. not necessarily 0 or 1) or a missing element (in which case the whole column is missing).

The elimination process is properly defined for all matrices regardless their shape and content, but columns that are fully zero have no meaning in thermodynamics (they correspond to chemical formulas without any atoms). Rows that are fully zero are on the other hand physically acceptable, and is in fact quite inevitable for single component systems with two or more atoms. Note also that there are two special cases of \mathbf{B} : If \mathbf{A} is invertible then $\mathbf{B}_1 = \mathbf{I}$ and $\mathbf{G} = \mathbf{A}^{-1}$. If $\mathbf{A} = \mathbf{0}$ then \mathbf{B}_1 is empty and $\mathbf{G} = \mathbf{I}$. From \mathbf{B}_1 we can define the elementary matrix

$$\mathbf{E}_1^T = \left(\begin{array}{ccc|cccccccccc} 0 & \cdots & 0 & 1 & 0 & \cdots & 0 & 0 & \cdots & 0 & 0 & \cdots \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 1 & 0 & \cdots & 0 & 0 & \cdots \\ \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots \\ 0 & \cdots & 0 & 0 & 0 & \cdots & 0 & 0 & \cdots & 1 & 0 & \cdots \end{array} \right)$$

by putting all $*$ to zero. Thus $\dim(\mathbf{E}_1^T) = \dim(\mathbf{B}_1)$. The product of \mathbf{E}_1 and \mathbf{B}_1 is thereby a *square* matrix with either 0 or 1 along the diagonal. Hence $\mathbf{E}_1\mathbf{B}_1 - \mathbf{I}$ is a similarly shaped matrix with either -1 or 0 along the diagonal. In order to see this clearly we remove for a moment all ellipses \cdots , $:$ and \ddots from the matrix expression:

$$\mathbf{E}_1\mathbf{B}_1 - \mathbf{I} = \left(\begin{array}{ccccccccc} -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & * & 0 & * & 0 & * & * \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & * & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \end{array} \right)$$

The outcome of the manipulation is that $\mathbf{B}(\mathbf{E}_1\mathbf{B}_1 - \mathbf{I}) = \mathbf{0}$. This property follows from the definition of \mathbf{E}_1 which implies $\mathbf{B}_1\mathbf{E}_1 \triangleq \mathbf{I}_{\text{rank}(\mathbf{A}) \times \text{rank}(\mathbf{A})}$. Furthermore:

$$\mathbf{B}(\mathbf{E}_1\mathbf{B}_1 - \mathbf{I}) \triangleq \left(\begin{array}{c} \mathbf{B}_1 \\ \mathbf{0} \end{array} \right) (\mathbf{E}_1\mathbf{B}_1 - \mathbf{I}) = \left(\begin{array}{c} \mathbf{I} \\ \mathbf{0} \end{array} \right) \mathbf{B}_1 - \left(\begin{array}{c} \mathbf{B}_1 \\ \mathbf{0} \end{array} \right) = \mathbf{0}$$

It also means we have captured the nullspace of \mathbf{A} since $\mathbf{A} = \mathbf{GB}$. If $\mathbf{B}(\mathbf{E}_1\mathbf{B}_1 - \mathbf{I})$ is zero then $\mathbf{A}(\mathbf{E}_1\mathbf{B}_1 - \mathbf{I})$ is zero because \mathbf{G} is an invertible (non-singular) matrix. What remains now is to extract \mathbf{N} by selecting the non-zero columns of $\mathbf{E}_1\mathbf{B}_1 - \mathbf{I}$. Let \mathbf{E}_2 be an elementary selection matrix doing these operations. Then:

$$\mathbf{N} \doteq (\mathbf{E}_1\mathbf{B}_1 - \mathbf{I})\mathbf{E}_2$$

Each column of \mathbf{N} corresponds to a chemical reaction with coefficients taken from the elements of that column. From its physical interpretation \mathbf{N} is also called the reaction stoichiometry matrix of the system.

Let $\mathbf{A} = (1 \ 2)$ be the atomic matrix of a chemical system comprised of component A and its dimer A_2 . We shall find the reaction stoichiometry of this system using the matrix formulations above. The result is

$$\begin{aligned}\mathbf{A} &= (1 \ 2) \\ \mathbf{B}_1 &= \mathbf{B} = (1 \ 2) \\ \mathbf{E}_1^T &= (1 \ 0) \\ \mathbf{E}_1\mathbf{B}_1 &= \begin{pmatrix} 1 & 2 \\ 0 & 0 \end{pmatrix} \\ \mathbf{E}_1\mathbf{B}_1 - \mathbf{I} &= \begin{pmatrix} 0 & 2 \\ 0 & -1 \end{pmatrix} \\ \mathbf{N} \doteq (\mathbf{E}_1\mathbf{B}_1 - \mathbf{I})\mathbf{E}_2 &= \begin{pmatrix} 2 \\ -1 \end{pmatrix}\end{aligned}$$

where $\mathbf{B}_1\mathbf{E}_1 = (1 \ 2)(1 \ 0)^T = (1) \equiv \mathbf{I}_{\text{rank}(\mathbf{A}) \times \text{rank}(\mathbf{A})}$. Note: The stoichiometry matrix \mathbf{N} is in chemical lingo written $2A \leftrightarrow A_2$. Left as an exercise for the reader is finding all six(!) reactions in the methane–air system mentioned in Eq. 1.

After this lengthy digression of nullspaces and chemical reactions we shall finally continue with the mass balance in Eq. 3. The forthcoming discussion has much in common with the energy balance in Part II of this paper, but the mass balance is inherently simpler than seen from a modelling point of view. To continue we shall first require the partial derivative of \mathbf{n} at a fixed spatial position z with respect to time is:

$$\left(\frac{\partial \mathbf{n}}{\partial t} \right)_z = \dot{\mathbf{n}}_z - \dot{\mathbf{n}}_{z+\Delta z} + \int_z^{z+\Delta z} A \mathbf{N} \dot{\boldsymbol{\xi}} d\zeta$$

As is also explained in the second paper this equation has a very special meaning whenever the physical situation is such that it allows the left hand side to be put to zero. This is the celebrated *steady state* which reduces the differential equation to an algebraic equation on the form:

$$\dot{\mathbf{n}}_{z+\Delta z} - \dot{\mathbf{n}}_z = \int_z^{z+\Delta z} A \mathbf{N} \dot{\boldsymbol{\xi}} d\zeta$$

The mole flows can be factored into the flow of total mass and a composition term:

$$\dot{\mathbf{n}} = \dot{M} \mathbf{c}$$

The mass balance is then reduced to:

$$(\dot{M} \mathbf{c})_{z+\Delta z} - (\dot{M} \mathbf{c})_z = \int_z^{z+\Delta z} A \mathbf{N} \dot{\xi} d\zeta$$

From the mass conservation principle we know that (for steady-state flow):

$$\dot{M}_{z+\Delta z} - \dot{M}_z = 0$$

Division by $\dot{M}_{z+\Delta z} = \dot{M}_z \hat{=} \dot{M}$ on both sides of the equation yields:

$$\mathbf{c}_{z+\Delta z} - \mathbf{c}_z = \int_z^{z+\Delta z} A \mathbf{N} \dot{M}^{-1} \dot{\xi} d\zeta$$

In the limit of $\Delta z \rightarrow 0$ we get:

$$\lim_{\Delta z \rightarrow 0} (\mathbf{c}_{z+\Delta z} - \mathbf{c}_z) = A \mathbf{N} \dot{M}^{-1} \dot{\xi} \Delta z$$

or rearranged:

$$\lim_{\Delta z \rightarrow 0} \frac{\mathbf{c}_{z+\Delta z} - \mathbf{c}_z}{\Delta z} = A \mathbf{N} \dot{M}^{-1} \dot{\xi}$$

We immediately recognize the left hand side as the partial derivative of \mathbf{c} with respect to z . On the right hand side we can make the definition $\mathbf{r} \hat{=} \dot{M}^{-1} \dot{\xi}$ standing for the specific reaction yield (moles per unit mass and volume). The mass balance for a steady state reactor is finally written:

$$\left(\frac{\partial \mathbf{c}}{\partial z} \right)^{\text{s-s}} = A \mathbf{N} \mathbf{r}$$

To solve this equation we need \mathbf{N} and a kinetic model for $\mathbf{r}(z, \mathbf{c})$. An algorithm for calculating \mathbf{N} is discussed in this paper, but the calculation of \mathbf{r} has to await a more thorough discussion of thermodynamic state variables in Parts II and III of this paper. The reason is that \mathbf{r} is a strong function of thermodynamic variables like temperature and pressure in addition to the composition variable \mathbf{c} .

There is another formal issue here which must not be forgotten: The mass balance is written as a partial derivative with respect to the spatial co-ordinate. This is odd since \mathbf{c} is by no means a function of z . It only depends on z through the solution of the differential equation. The thread to this discussion will be picked up in conjunction with the energy balance in Part II.

Exercise 6

1 Question: Topology of a fermentation plant

Subject of the exercise is the plant shown in Figure 1. It represents a fermentation plant that is attached to a sugar manufacturing facility. When producing sugar, one of the main waste streams contains some remaining sugar besides a lot of organic waste materials, which are mostly solid. So this added facility is to ferment the remaining sugar into EtOH through a yeast fermentation. The plan section takes the waste stream "as is" into the fermenter.

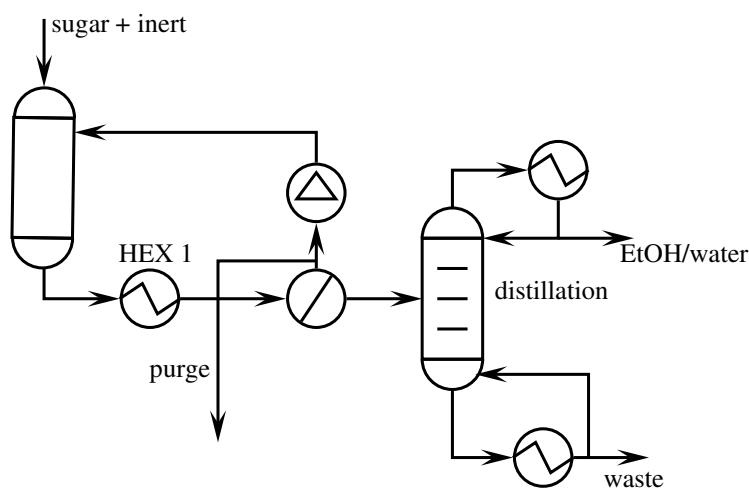
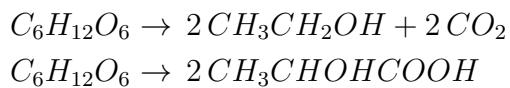


Figure 1: A biofuel production plant

For the purpose of the exercise we make a number of assumptions so as to simplify the overall picture:

- Feed contains water, dissolved sugar and solid inert impurities.
- The yeast is solid and added at the beginning. It acts like a catalyst and is thus not shown in the reaction, even though it is growing and dying.
- The filter operates ideally, meaning the solids are completely separated. None passes over into the liquid stream.
- The process operates in continuous mode. In practice the fermentation goes bad ever so often and the plant needs to be cleaned and the fermentation process must be re-initiated.
- The distillation is rather ideal, thus produces azeotrope EtOH and water on the top, whilst no EtOH leaves through the bottom stream, but all the other liquid waste products are appearing in the bottom.

The main reactions, or the ones we consider, are:



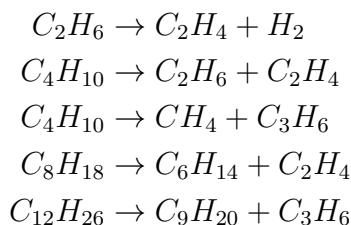
The second reaction produces lactic acid as the by-product of fermentation process

Tasks:

- Sketch the topology of the ethanol production plant.
- How would you show the initialisation process in the topology?
- Have a closer look at the filter. Any alternatives ? What was your assumption for the filter part? What will be different if we assume the filter is relatively large or small compared to the fermenter ?

2 Question: Reactions

Ethylene and propylene are two of the important products from petrochemical industries. They are produced by different methods, e.g. cracking. During cracking, the saturated hydrocarbons are cracked in a furnace with high temperature. The products include Hydrogen, Praffines, Olefines, Diolefines, Aromatics and water. The following equations are just to name a few of cracking reactions.



Write the species-atom matrix.

3 Question: Dynamics

Consider the process from the previous assignment as shown in Figure 2.

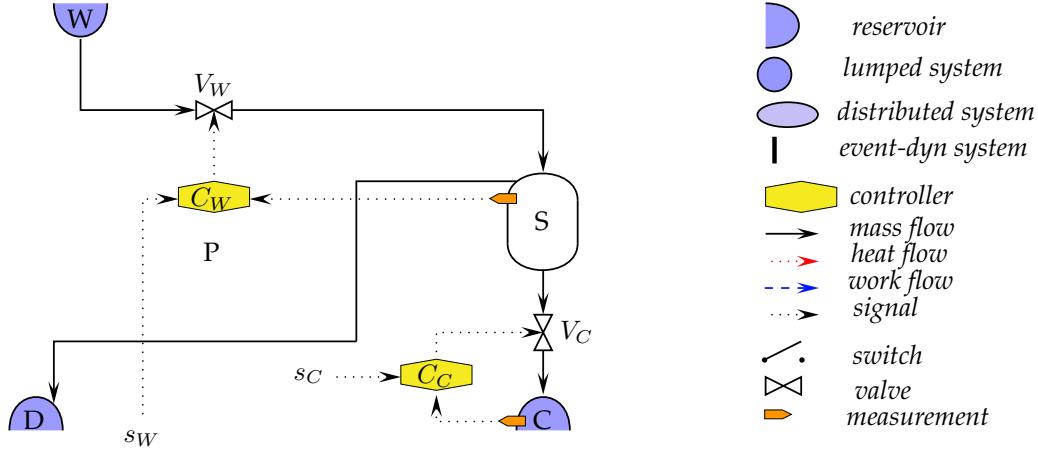


Figure 2: The simplified plant to simulate

The simplified model is:

$$\begin{aligned}
 \dot{x}(t) &= \hat{x}_W(t) - \hat{x}_C(t) \\
 \hat{x}_W(t) &:= \Theta_W u_W(t) \\
 y(t) &:= \Theta_h x(t) \\
 y(k) &:= y(k \Delta t) & t \in [k t, (k + 1) \Delta t] \\
 e(k) &:= y_s(k) - y(k) \\
 u_W(k) &:= u_W(k - 1) + \Theta_P \left(\left(1 + \frac{\Delta t}{\Theta_I} \right) e(k) - e(k - 1) \right) \\
 u_W(t) &:= u_W(k) & t \in [k t, (k + 1) \Delta t] \\
 \hat{x}_C(t) &:= u_C(k) & t \in [k t, (k + 1) \Delta t]
 \end{aligned}$$

In contrast to the first assignment, we now use a discrete PI controller so as to remove the offset from the response.

The parameter vector $\underline{\Theta} := [\Theta_W, \Theta_h, \Theta_P, \Theta_I]^T$

- Substitute to get a single ordinary differential equation.
- Integrate both sides over the arbitrary time interval $t \in [k t, (k + 1) \Delta t]$.
- The result is a difference equation
- Write a program that computes the history of the state given
 - the initial state $x(t := 0, 1) := [0.5, 0.45]$ and $\underline{\Theta} := [0.4, 0.8, 0.02, 0.01]^T$
 - sampling time is 1
 - the history of the input u_C as a vector of numbers: $u_C(k := 1, \dots, 100) := [...]$
The first 10 values are 0.1, the next 10 are 0.2, the next 20 are 0.15, the next 10 are 0.25, the next 20 are 0.05 and the rest 0.2.

- the setpoint $y_s(k) := 0.5$ to be constant over the time.
- change the controller proportional gain Θ_P and the integral constant Θ_I to see what happens.
- Output data as text file so you can plot it (k, y_s, y, u_C) .

3.1 On the side

The equation for continuous PI controller is

$$u := P \left(e + \frac{1}{T_i} \int e dt \right)$$

The derivative is approximated as

$$\frac{du}{dt} \approx \frac{u(k) - u(k-1)}{\Delta t} \approx \frac{1 - q^{-1}}{\Delta t} u(k)$$

where q is the shift operator defined by the equation

$$u(k+1) := q u(k)$$

So,

$$\begin{aligned} u(k) &:= P \left(e(k) + \frac{\Delta t}{T_i} \frac{1}{1 - q^{-1}} e(k) \right) \\ u(k) &:= P \left(1 + \frac{\Delta t}{T_i} \frac{1}{1 - q^{-1}} \right) e(k) \end{aligned}$$

$$\begin{aligned} u(k) (1 - q^{-1}) &= P \left((1 - q^{-1}) + \frac{\Delta t}{T_i} \right) e(k) \\ &= P \left(1 + \frac{\Delta t}{T_i} - q^{-1} \right) e(k) \end{aligned}$$

The final expression for $u(k)$ is

$$u(k) := u(k-1) - P e(k-1) + P \left(1 + \frac{\Delta t}{T_i} \right) e(k)$$

4 Question: Linear algebra 02

The following matrices are given:

$$\underline{\underline{\mathbf{A}}} := \begin{bmatrix} 2 & 2 & -4 & 3 & 8 & 1 \\ 3 & 1 & -2 & 1 & 2 & 2 \\ 2 & 1 & 3 & 0 & -3 & 1 \end{bmatrix}$$

$$\underline{\underline{B}} := \begin{bmatrix} 1 & 3 & -3 \\ -3 & 7 & -3 \\ -6 & 6 & -2 \end{bmatrix}$$

$$\underline{\underline{C}} := \begin{bmatrix} 10 & 2 & 8 \\ 2 & 1 & 1 \\ 4 & 7 & -3 \end{bmatrix}$$

$$\underline{\underline{D}} := [0]$$

$$\underline{\underline{E}} := [1]$$

Tasks

- Calculate the null space for $\underline{\underline{A}}, \underline{\underline{B}}$ and $\underline{\underline{C}}$:
- What does $N(\underline{\underline{B}}) = 0$ means?
- Find the eigen values and eigen vectors of $\underline{\underline{B}}$
- What is the size of null space matrices for $\underline{\underline{D}}$ and $\underline{\underline{E}}$

1 Suggested solution: Topology fermenter plant

1.1 Overall view

To make things easier to read, we approach the topology of the fermenter plant in bits and pieces, beginning with a global view whereby the units are mainly made visible. We use rectangular boxes for units or parts of them:

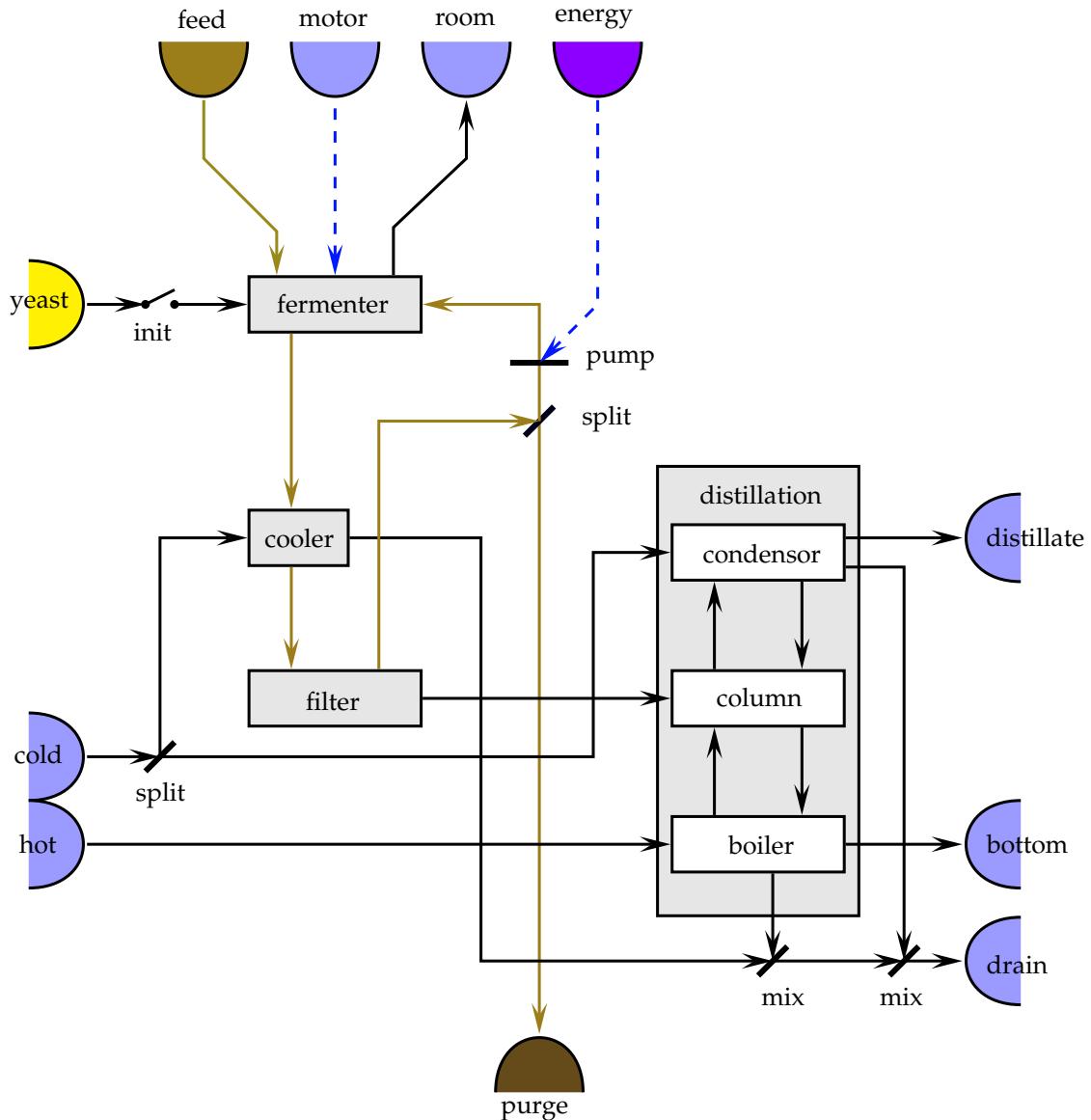


Figure 1: Topology of a biofuel production plant

Note that we use the sludge as a composite phase in this approach.

1.2 The reactor part

Next we zoom into reactor part. Notice that we now split into two phases. The sludge is split into a solid phase and a liquid phase assuming the reaction is taking place in the solid phase, rational being that the yeast is a biological cell structure, which appears in the "solid" phase, in contrast to the liquid phase that provides the sugar and takes the liquid products. The split into two phases may not be necessary. One could also handle

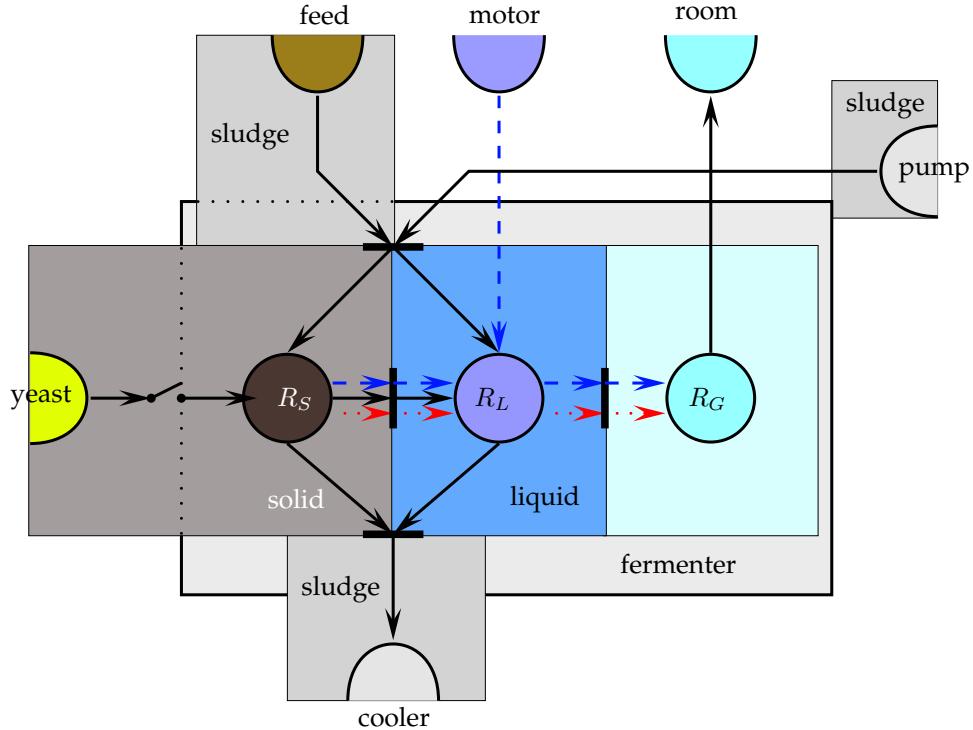


Figure 2: Topology of the fermenter

this in a pseudo-phase. This is an example on how one could define a topology that "sees" it as two phases.

1.3 Heat exchanger

The heat exchanger cools both phases and since the solid phase is dispersed in the liquid phase, it seems appropriate to again define it as a pseudo-phase, thus a sludge though now with a different composition.

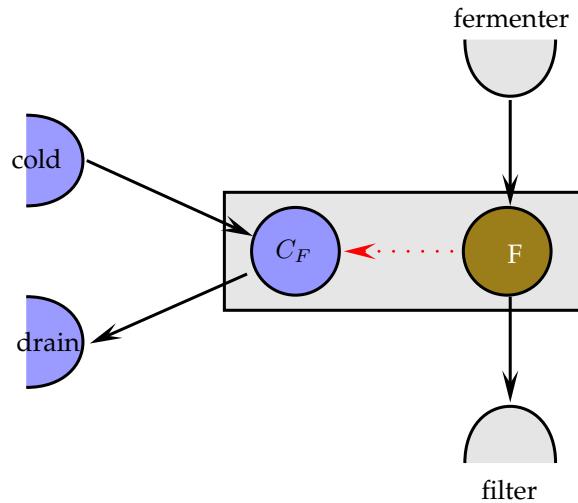


Figure 3: Topology of the cooler

1.4 Filter

From the topology point of view, this is a really interesting unit. If one thinks about to also describe the filter-cake build up, for example this unit becomes really complex but also interesting. Many alternatives are possible. The one below is just one of them.

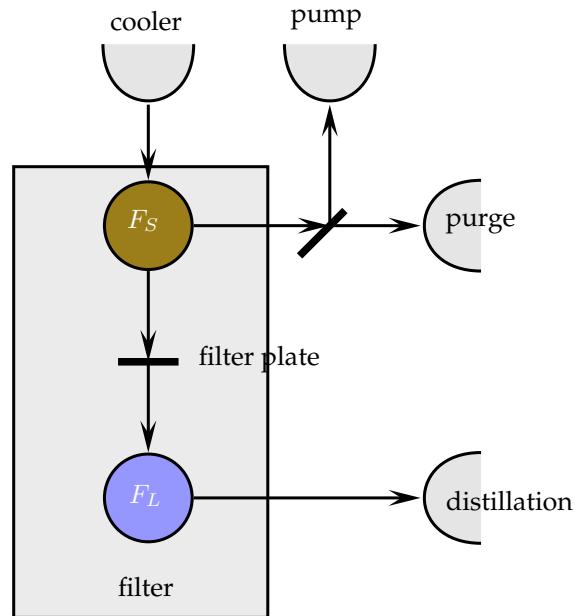


Figure 4: Topology of the filter

1.5 Distillation

A good old friend, the distillation column. Each tray simplified to two phases with varying volume, thus the volume-work arrow. The condenser is described as a drop-condensation unit in which the heat transfer occurs both through the gas phase and the liquid phase in contrast to film condensation, where it is mostly the liquid that is being cooled directly. The main heat transfer in the boiler is through the wall to the liquid phase. In some boilers there may also be a heat transfer into the gas phase, but that is usually a very small part in contrast to the other heat flow

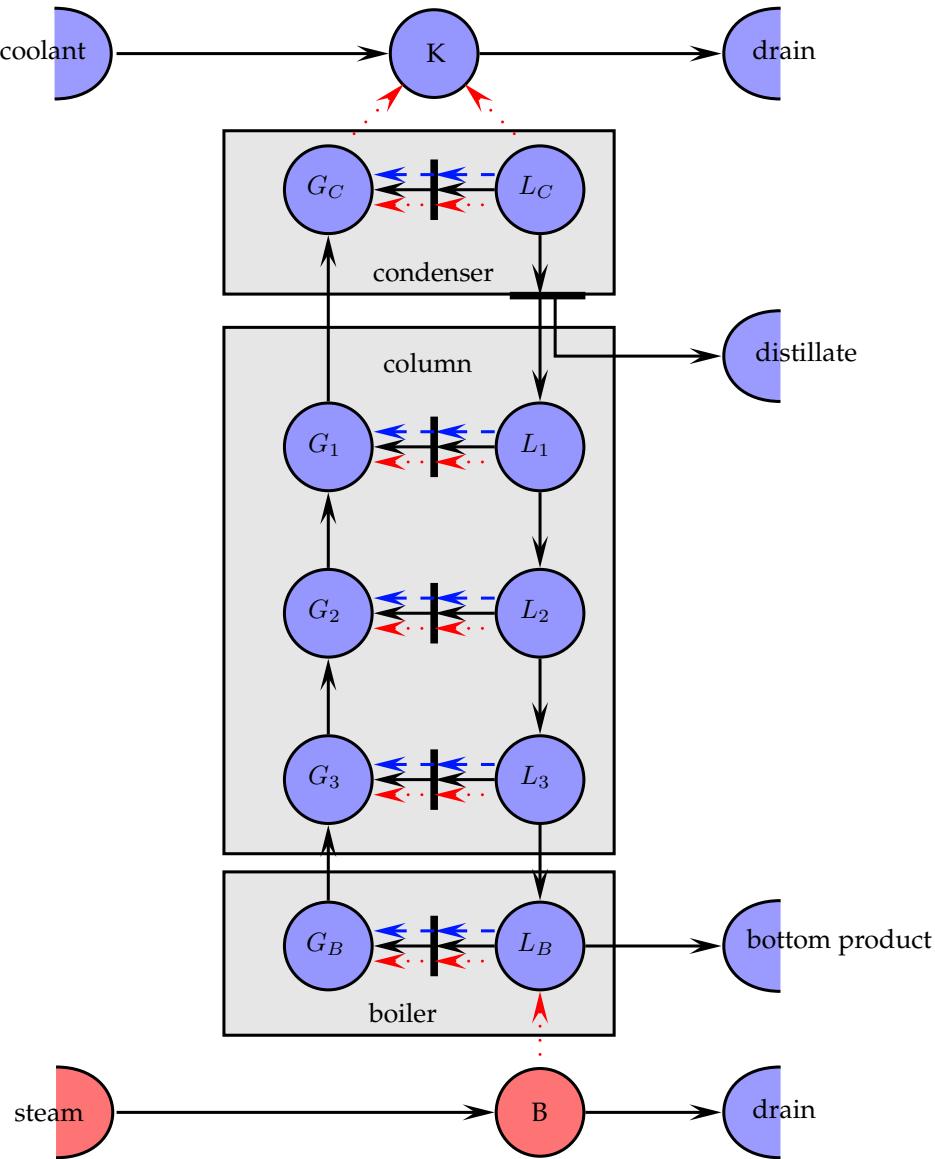


Figure 5: Topology of the distillation

2 Solution: Reactions

$$species = [C_2H_6 \ C_2H_4 \ H_2 \ C_4H_{10} \ CH_4 \ C_3H_6 \ C_8H_{18} \ C_6H_{14} \ C_{12}H_{26} \ C_9H_{20}]$$
$$index = [1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10]$$

	1	2	3	4	5	6	7	8	9	10
C	2	2	0	4	1	3	8	6	12	9
H	6	4	2	10	4	6	18	14	26	20

3 Dynamics

Like the previous assignment, the variables are substituted in the main equation:

$$\begin{aligned}
\dot{x} &:= \hat{x}_W(t) - \hat{x}_C(t) \\
&:= \Theta_W u_W(t) - u_C(k) \\
&:= \Theta_W u_W(k) - u_C(k) \\
&:= \Theta_W \left[u_W(k-1) - \Theta_P \left(\left(1 - \frac{\Delta t}{\Theta_I}\right) e(k-1) + e(k) \right) \right] - u_C(k) \\
&:= \Theta_W \left[u_W(k-1) - \Theta_P \left(\left(1 - \frac{\Delta t}{\Theta_I}\right) (y_s(k-1) - \Theta_h x(k-1)) + (y_s(k) - \Theta_h x(k)) \right) \right] - u_C(k)
\end{aligned}$$

The right-hand-side is now constant for the duration of the time interval $t \in [k \Delta t, (k+1) \Delta t]$ and we get:

$$\int_{k \Delta t + \epsilon}^{(k+1) \Delta t} \dot{x} dt := \int_{k \Delta t + \epsilon}^{(k+1) \Delta t} RHS dt$$

$$\begin{aligned}
x(k+1) &= \left(1 - \Theta_W \Theta_P \Theta_h \left(1 + \frac{\Delta t}{\Theta_I} \right) \right) x(k) \\
&\quad + \Theta_W \Theta_p \Theta_h x(k-1) \\
&\quad + \Theta_W u_W(k-1) \\
&\quad + \Theta_W \Theta_p \left(1 + \frac{\Delta t}{\Theta_I} \right) y_s(k) - \Theta_W \Theta_p y_s(k-1) \\
&\quad - u_C(k)
\end{aligned}$$

This should be solved in an iterative manner. For every step, we need to have the error $e(k)$ and $u_W(k)$, so that we can calculate $x(k+1)$.

$$\begin{aligned}
e(k) &= y_s - \Theta_h x(k) \\
u_W(k) &= u_W(k-1) + \Theta_P \left(\left(1 + \frac{\Delta t}{\Theta_I} \right) e(k) - e(k-1) \right) \\
x(k+1) &= x(k) + \Theta_W u_W(k) - u_C(k)
\end{aligned}$$

A program in Matlab

```
% dynamics_02 (solution)
% simple tank controlled by a discrete PI controller
%
% 2012-10-09 Preisig, H A

% parameters
% flow constant, controller gain, geometry
% parameters : water flow, geometry, prop const, integral const
P = [0.4, 0.8, 2, 0.4]';

% initial conditions & allocate array
dt = 1; % sampling time
x = zeros(100,1); % pre-allocate space

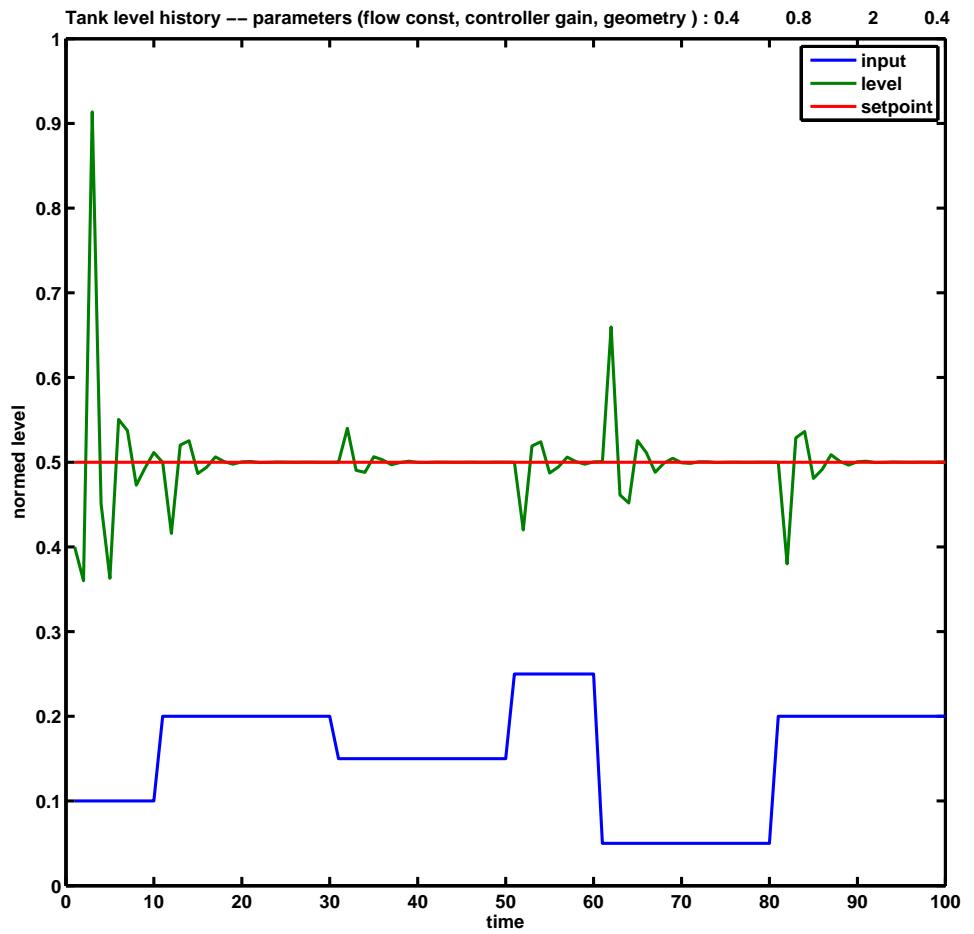
u_W(1) = 0;
x(1) = 0.5;
x(2) = .45;
ys = 0.5;
e(1) = ys-P(3)*x(1);

u = [0.1 * ones(10,1)
      0.2 * ones(20,1)
      0.15 * ones(20,1)
      0.25 * ones(10,1)
      0.05 * ones(20,1)
      0.20 * ones(20,1)];

for k = 2:99
    e(k) = ys - P(2) * x(k);
    u_W(k) = u_W(k-1) + P(3) * ( (1 + dt/P(4)) * e(k) - e(k-1));
    x(k+1) = x(k) + P(1) * u_W(k) - u(k);
end

y = P(2)*x;
plot([u, y, ys*ones(100,1)])
legend('input','level','setpoint')
s = num2str(P');
title(['Tank level history -- parameters (flow const, controller gain, geometry ) : ' s])
xlabel('time')
ylabel('normed level')
```

Results



A python program

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
@summary:      Simulates a simple tank with PI control
@author:       Preisig, H A
@copyright:    Preisig, H A
@organization: Department of Chemical Engineering, NTNU, Norway
@contact:      heinz.preisig@chemeng.ntnu.no
@license:      GPLv3
@requires:     Python 2.7.1 or higher
@since:        2012.10.05
@version:      1.0
@todo:         2.0:
@change:       2012.10.08
"""

def matnums(val, cols):
    return [val for i in range(0,cols)]

def matfunc(f='dynamics_02.dat', debug = False):
    txtfile = open(f,'w')
    # Parameters
    # Flow constant, geometry, controller gain, controller integral const

    P = [0.4, 0.8, 2, 0.4]
    range_k = range(0,100)
    dt = 1.0

    # Initial condition
    x = [0.5, 0.45]
    ys = 0.7
    y = [P[1] * x[0], P[1] * x[1]]
    u_W = [0]
    e = [ys-y[0]]

    u = matnums(0.1, 10) + matnums(0.2, 20) + matnums(0.15, 20) + \
        matnums(0.25, 10) + matnums(0.05, 20) + matnums(0.2, 20)

    for k in range_k[1:-1]:
        e.append(ys - P[1] * x[k])
        u_W.append(u_W[-1] + P[2] * ((1+dt/P[3]) * e[k] - e[k-1]))
        x.append(x[-1] + P[0] * u_W[k] - u[k])
        y.append(P[1]*x[-1])

        if debug:
            print str(u[k]) + ', ' + str(y[k]) + ', ' + str(ys)
```

```

    else:
        print>>txtfile, (k+1), str(u[k]), str(y[k]), str(ys)

    if not debug: print 'Data written to file :', f
    txtfile.close()

    return (range_k,
            [x[i] for i in range_k], \
            [y[i] for i in range_k], \
            ys, \
            [u[i] for i in range_k], \
            P \
            )
}

if __name__ == '__main__':
    matfunc(debug=False)

```

Launcher

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
@summary: launches the dynamics_02 simulation and plots results
@author: Preisig, H A
@copyright: Preisig, H A
@organization: Department of Chemical Engineering, NTNU, Norway
@contact: heinz.preisig@chemeng.ntnu.no
@license: GPLv3
@requires: Python 2.7.1 or higher
@since: 2012.10.05
@version: 1.0
@todo 2.0:
@change: 2012.10.08
"""

from dynamics_02 import matfunc
from gnuplot import gnuplot
import os

#files, extensions and title template
d = 'dynamics_02'
ext = '.dat'
title = 'Tank level with parameters: %s'

```

```

(k, x, y, ys, u, P) = matfunc()

p = gnuplot(title=title%P, xmin=0, xmax=100, ymin=0.0, ymax = 1.0)

p.add(d+ext, x=1, y=2, width=2, color='red', title='mass')
p.add(d+ext, x=1, y=3, width=2, color='blue', title ='level')
p.add(d+ext, x=1, y=4, width=2, color='green', title = 'setpoint')
p.plot(d)

try: os.remove(d+ext)
except: pass
try: os.remove(d+'.pyc')
except: pass
try: os.remove('gnuplot.pyc')
except: pass

```

GnuPlot wrapper

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""
@summary:      Wrapper for gnu plot
@author:       Eivind Haug-Warberg
@copyright:    Tore Haug-Warberg
@organization: Department of Chemical Engineering, NTNU, Norway
@contact:      haugwarb@nt.ntnu.no
@license:      GPLv3
@requires:     Python 2.7.1 or higher
@since:        2012.09.12 (EHW)
@version:      1.0
@todo 2.0:
@change:       2012.09.27
"""


```

```

import os
import string
import time
import subprocess
import re

# This is a template for the plot ${keyword} will be replaced.
TEMPLATE = \
'''
#!/sw/bin/gnuplot -persist

set terminal postscript \\


```

```

    eps noenhanced monochrome\\
    dashed defaultplex \"${textfont}\" ${textsize}

set output "${filename}.eps"

set title "${title}"
set size ${xsize},${ysize}
set xlabel "${xlabel}"
set ylabel "${ylabel}"

set xrange [${xmin}://${xmax}]
set yrange [${ymin}://${ymax}]
set mxtics ${xtics}
set mytics ${ytics}

set samples ${samples}

set key font ",${keytextsize}" spacing ${keylinespace} \\
    ${keyboxposition} ${keyshow}
set multiplot

,,,
# This is the plot template for a graph.
ADDGRAPH = \
,,,
set key at 0,${ykeybox}

plot ${plotobject} ${title} with ${style} linetype ${type} \\
    linewidth ${width} linecolor rgb \"${color}\"
,,,

ADDLIST = \
,,,
set key at 10,${ykeybox}

plot \'-\' using ${columns} ${title} with ${style} linetype ${type} \\
    linewidth ${width} linecolor rgb \"${color}\"
,,,

# Default input for gnuplot initialization.
Kwargs = {'title': 'GNUploat created: %s'%time.asctime(), 'samples': 3000, \
          'xsize': 1, 'xlabel': 'x', 'xmin': -1, 'xmax': 1, 'xtics': 1, \
          'ysize': 1, 'ylabel': 'f(x)', 'ymin': -1, 'ymax': 1, 'ytics': 1, \
          'textfont': 'Helvetica', 'textsize': 18, 'keytextsize': 10, \
          'keylinespace': 0.6, 'keyboxposition': 'right', 'keyshow': 'nobox' \
        }

class gnuplot():
    # CALLED WHEN A PLOT IS INITIALIZED.

```

```

def __init__(self, filename='', **kwargs):
    kw = {}                                # Creates a new dictionary to store input.
    kw.update(KWARGS)                      # Put the default input into the new dictionary.
    kw.update(kwargs)                      # Updates the default input with input from user.

    # This block will add an unformated, commented out python template on the
    # top of the plot template. Has no other function than documentation.
    self.template = '#'*80 + '\n# Template for the gnuplot class\n' + '#'*80
    for line in re.split(r'\n', TEMPLATE):
        self.template += '\n# %s'%line

    # Format the python template, and add it to our gnuplot template.
    self.template += string.Template(TEMPLATE).safe_substitute(kw)

# CALLED WHEN A PLOT IS ADDED.
def add(self, plotobject, x=1, y=2, error=False, ykeybox = False, \
        title ='notitle', **kwargs):

    if os.path.isfile(plotobject):      # Tests if the input object is a file.
        if error:          # "error" is the third data row when you plot error bars.
            error = ':%s%error' # Add a colon in the beginning of the error string.
            # Creates a dictionary for the default error plot layout.
            kw = {'style': 'yerrorbars', 'type': 1, 'width': 1}
        # If "error" has no value, you're doing a normal plot. we'll set error
        # to be an empty string, to generalize the code later, create a
        # dictionary for default the "y-to-x-plot" layout.
        else: error, kw = '', {'style': 'lines', 'type': 1, 'width': 2}

        # If title is set to auto, generate a automatic title.
        if title == 'auto':
            title = 'Datafile: %s (%s:%s%s)' % (plotobject, x, y, error)

        # Make a gnuplot formated string for doing a plot from file.
        plotobject = '\">%s\" using %s:%s%s' % (plotobject, x, y,error)

    else:
        # This block is executed when you don't plot from files, in other words,
        # when you plot from a mathematical function. Generate an automatic
        # title, just like on line 78, and create a dictionary for the default
        # function layout.
        if title == 'auto': title = 'Function: %s%plotobject'
        kw = {'style': 'lines', 'type': 1, 'width': 1}

    kw.update({'color': 'black'})           # All plots are black by default.
    kw.update(kwargs)                     # Add the input to the default layout dictionary.

    # If you're plotting something as with points, you may want to configurate
    # the point look too.
    if re.match(r'.*points',kw['style']):
        kw['style'] += ' pointtype %s pointsize %s' % (kw.get('pointtype',1), \

```

```

        kw.get('pointsize',1))

# If a title was given, format it right, so the legend can be added.
if title != 'notitle': title = 'title \"%s\"%title'

# Read the last line in the gnuplot template. If there is only white
# spaces in it, it can for sure not say "plot" there. A new plot must be
# started. If there was not only white spaces on the last line, a plot
# has been started earlier, you can continue adding graphs to this plot
# by using commas.

#if re.match(r'^\s*$',re.split(r'\n',self.template)[-1]):
#    self.template += 'plot '
#else: self.template += ', \\\\n      \\\\n      '
if not ykeybox:
    plots = 0
lines = re.split(r'\n',self.template)
for line in lines:
    if re.match('^\s*key\s*at.*$',line): plots += 1
print plots
ykeybox = 0.95 - plots * 0.07

# Update kw with all the local variables in this namespace, and format the
# python template to a gnuplot template, and add the the plot template to
# the main template.
kw.update(locals())
self.template += string.Template(ADDGRAPH).safe_substitute(kw)

# CALLED WHEN PLOT FROM LIST IS ADDED
def addlist(self, x, y=False, error=False, title='notitle', **kwargs):
    if not y: y = x, range(1, len(x) + 1)

    if error:
        columns = '1:2:3'
        kw = {'style': 'yerrorbars', 'type': 1, 'width': 1, 'color': 'black'}
        error += ['e']
        print 1
    else:
        columns = '1:2'
        kw = {'style': 'lines', 'type': 1, 'width': 2, 'color': 'black'}

    error = ['' for i in range(len(x)+1)]

kw.update(kwargs)

if title != 'notitle': title = 'title \"%s\"%title'

kw.update(locals())

```

```

if re.match(r'.*points',kw['style']):
    kw['style'] += ' pointtype %s pointsize %s' % (kw.get('pointtype',1),
                                                    kw.get('pointsize',1))

#if re.match(r'^\s*$',re.split(r'\n',self.template)[-1]):
# self.template += 'plot '
#else: self.template += ', \\n      \\n      '

self.template += string.Template(ADDLIST).substitute(kw)

x.append('e'); y.append('e')

for i in range(0,len(x)):
    self.template += '\n      %s %s %s' % (x[i], y[i], error[i])
    self.template += '\n'

# CALLED WHEN PLOT IS CREATED
def plot(self,filename, clean=False):
    # In order to do a plot with direct input (not from file), do you need to
    # start a gnuplot process and then enter the plotting commands. If we do
    # this by using os.system(), this program will pause until the plotting
    # process is finished, and then enter the plotting commands in the
    # terminal, not in gnuplot. What we need to do is to make a subprocess,
    # and give our input to that process, instead of the terminal.
    plot = subprocess.Popen(['gnuplot'], stdin=subprocess.PIPE)
    plot.communicate(string.Template(self.template).safe_substitute({'filename': fi

# ps2pdf is not installed by default, so catch the possible error.
try: os.system('ps2pdf %s.eps' %filename)
except: print 'Can not convert %s.ps to PDF by using ps2pdf.' % filename

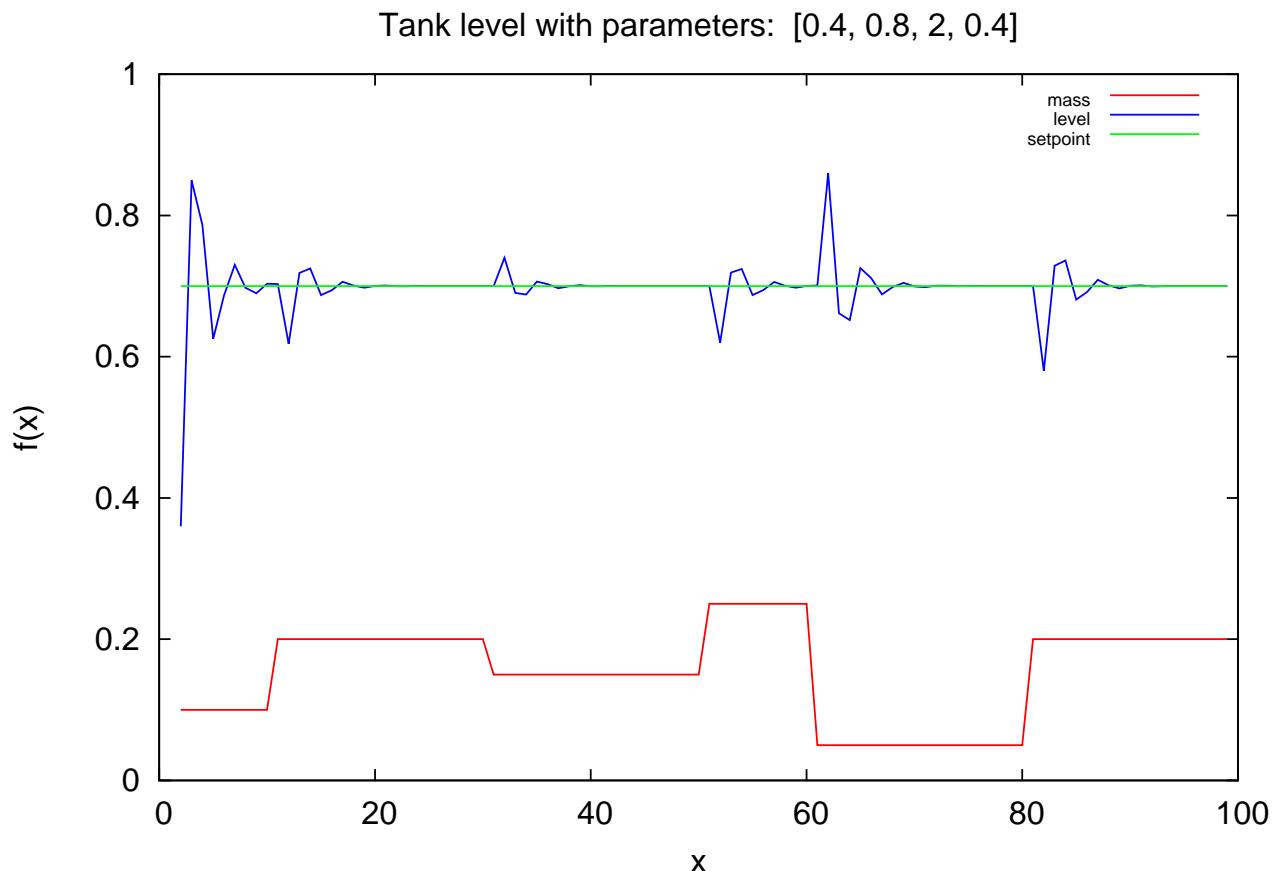
#if clean: os.remove('%s.ps' % filename) # Remove the .ps file if you want.

# CALLED WHEN PLOT IS SAVED
def save(self,filename,output = False):
    with open(filename, 'w') as template:                      # Open the given file.
        if output: template.write(string.Template(self.template \
                                         ).safe_substitute({'filename': output}))
    else: template.write(self.template)

# CALLED WHEN PLOT IS LOADED
def load(self,filename, clean = False):
    with open(filename, 'r') as template: # Open the given file in read mode.
        self.template = template.read()      # Read the text that it contains.
    if clean: os.remove(filename)       # Remove the template file if you want.

```


Results



4 Solution: Linear Algebra 02

$$\underline{\underline{\mathbf{A}}} := \begin{bmatrix} 2 & 2 & -4 & 3 & 8 & 1 \\ 3 & 1 & -2 & 1 & 2 & 2 \\ 2 & 1 & 3 & 0 & -3 & 1 \end{bmatrix}$$

The matrix is converted to the row echelon form via the row operations below

$$\begin{aligned} R_1 - \frac{2}{3}R_2 &\rightarrow R_2 \\ R_1 - R_3 &\rightarrow R_3 \end{aligned}$$

So,

$$\underline{\underline{\mathbf{A}}} := \begin{bmatrix} 2 & 2 & -4 & 3 & 8 & 1 \\ 0 & \frac{4}{3} & \frac{-8}{3} & \frac{7}{3} & \frac{20}{3} & \frac{-1}{3} \\ 0 & 1 & -7 & 3 & 11 & 0 \end{bmatrix}$$

The second element in Row three is removed by

$$R_2 - \frac{4}{3}R_3 \rightarrow R_3$$

$$\underline{\underline{\mathbf{A}}} := \begin{bmatrix} 2 & 2 & -4 & 3 & 8 & 1 \\ 0 & \frac{4}{3} & \frac{-8}{3} & \frac{7}{3} & \frac{20}{3} & \frac{-1}{3} \\ 0 & 0 & \frac{20}{3} & \frac{-5}{3} & -8 & \frac{-1}{3} \end{bmatrix}$$

Now we start with the upper triangular part

$$R_2 - \frac{2}{3}R_1 \rightarrow R_1$$

$$\underline{\underline{\mathbf{A}}} := \begin{bmatrix} \frac{-4}{3} & 0 & 0 & \frac{1}{3} & \frac{4}{3} & -1 \\ 0 & \frac{4}{3} & \frac{-8}{3} & \frac{7}{3} & \frac{20}{3} & \frac{-1}{3} \\ 0 & 0 & \frac{20}{3} & \frac{-5}{3} & -8 & \frac{-1}{3} \end{bmatrix}$$

$$R_3 + \frac{20}{8}R_2 \rightarrow R_2$$

$$\underline{\underline{\mathbf{A}}} := \begin{bmatrix} \frac{-4}{3} & 0 & 0 & \frac{1}{3} & \frac{4}{3} & -1 \\ 0 & \frac{10}{3} & 0 & \frac{25}{6} & \frac{26}{3} & \frac{-7}{6} \\ 0 & 0 & \frac{20}{3} & \frac{-5}{3} & -8 & \frac{-1}{3} \end{bmatrix}$$

We should have identity matrix in the left. So, each row is divided by the values in the diagonal of the reduced matrix.

$$\underline{\underline{\mathbf{A}}} := \begin{bmatrix} 1 & 0 & 0 & \frac{-1}{4} & -1 & \frac{3}{4} \\ 0 & 1 & 0 & \frac{5}{4} & \frac{26}{10} & \frac{-7}{20} \\ 0 & 0 & 1 & \frac{-1}{4} & \frac{-6}{5} & \frac{-1}{20} \end{bmatrix}$$

So,

$$N(\underline{\underline{\mathbf{A}}}) := \begin{bmatrix} \frac{1}{4} & 1 & \frac{-3}{4} \\ \frac{-5}{4} & \frac{-26}{10} & \frac{7}{20} \\ \frac{1}{4} & \frac{6}{5} & \frac{1}{20} \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\underline{\underline{\mathbf{B}}} := \begin{bmatrix} 1 & 3 & -3 \\ -3 & 7 & -3 \\ -6 & 6 & -2 \end{bmatrix}$$

For every matrix $\underline{\underline{\mathbf{B}}} \in \mathbb{R}^{m \times n}$, we have

$$\text{rank}(\underline{\underline{\mathbf{B}}}) + \dim(N(\underline{\underline{\mathbf{B}}})) = n$$

Since $\underline{\underline{\mathbf{B}}}$ is full rank, the null space of $\underline{\underline{\mathbf{B}}}$ is empty. This means that there is no degrees of freedom and there is a unique solution.

$$\underline{\underline{\mathbf{C}}} := \begin{bmatrix} 10 & 2 & 8 \\ 2 & 1 & 1 \\ 4 & 7 & -3 \end{bmatrix}$$

The Upper triangular matrix which is derived from $\underline{\underline{\mathbf{C}}}$ is

$$\underline{\underline{\mathbf{C}}} := \begin{bmatrix} 10 & 2 & 8 \\ 0 & -3 & 3 \\ 0 & 0 & 0 \end{bmatrix}$$

As it is seen, there is no pivot in the third column. Looking at $\underline{\underline{\mathbf{C}}}$, it is noted that the third column is actually a linear combination of the first two columns.

$$\text{Col 3} = \text{Col 1} - \text{Col 2}$$

We can for example choose $\begin{bmatrix} 1 & -1 & -1 \end{bmatrix}$ as a solution to null space. Note that all linear combinations of solutions to the null space are also solutions to the null space. The null space of $\underline{\mathbf{E}}$ is a matrix of size 1×1 and could be any number. The matrix $\underline{\underline{\mathbf{F}}}$ does not have any null space, since it is invertible.

Root solvers (TKP4106)



Zooball/Elephant

Reasons computers are female

- No one but their creator understands their internal logic.
- The native language they use to communicate with other computers is incomprehensible to everyone else.
- Even your smallest mistakes are stored in long-term memory for later retrieval.
- As soon as you make a commitment to one, you find yourself spending half your paycheck on accessories for it.

Computers are female

Assignments

1. Write a procedure `sqrt` for solving $x=\sqrt{y}$ using Newton-Raphson iteration. Variable y is supposed to be a known number taken in from the command line and you are asked to find x . Note: You cannot iterate on $x-\sqrt{y}=0$ directly because this problem already requires `sqrt()` which is an unknown function (without importing the `math` module in Python). Rather, you should consider iterating on $x^{**2}-y=0$. Use the stub program [sqrt.py](#) as template.
2. Play around with `sqrt()` and see if you can trick it somehow. Make it diverge in other words.
3. Write a procedure `pv` for solving $pv(p,t,v,ntot)=0$ using Newton-Raphson iteration. Variables p , t , v and $ntot$ are supposed to be known numbers taken in from the command line. However, v is a starting value only and will change during the iteration. Note: You must avoid unphysical solutions. That is to say negative volumes. Use the stub program [pv.py](#) as template.
4. Play around with `pv()` and learn more about Newton-Raphson iteration sequences. Run it a couple of thousand times at different starting values to see how stable it is. Observe that the iteration method is of 2nd order. I.e. that it doubles the significant digits in every iteration (at some point in the iteration history).

Start reading about [The energy balance](#) to get into the thinking of physical

problem formulations, equations of state and numerical solvers.

Most of the time we will be using Newton-Raphson iteration in this course for solving non-linear equations, but there is something called recursive iteration (using the Banach fix-point theorem) which can be very efficient. Perhaps you know this type of iteration as 'direct substitution'. It is worth while looking at - now that we know a little Python.

1. Write a recursive procedure for iterating $x_{k+1} = x_k^{**2}$ starting at $x_0 < 1$.

Have a look at [for_lc_rc.py](#) for some compelling thoughts on how this iteration can be achieved.

%Predefined.

HTML text.

5.13.1 Verbatim: “sqrt.py”

```
1  """
2  @summary:      Calculate the square root of any set of positive numbers using
3  Newton-Raphson iteration on::
4
5      x*x - y = 0
6
7      where y is the given number. In the implementation below y and x
8  are not plain numbers but lists of numbers.
9  @author:       Tore Haug-Warberg
10 @organization: Department of Chemical Engineering, NTNU, Norway
11 @contact:      haugwarb@nt.ntnu.no
12 @license:      GPLv3
13 @requires:     Python 2.3.5 or higher
14 @since:        2011.10.13 (THW)
15 @version:      1.0
16 @todo 2.0:    nothing
17 @change:       started (2011.10.13)
18 @note:         On a Unix terminal you can use the script like this:
19
20         >>> python sqrt.py
21         >>> python sqrt.py <y1> <y2> <...>
22
23     y1    = aNumber
24     y2    = aNumber
25     ...   = aNumber
26
27 """
28
29 def sqrt(y, x, debug=False, norm=1e999):
30
31     if debug:
32         print x
33
34     dy = pass                                # calc max(abs(residual))
35
36     if dy < 1.0e-8 and dy >= norm:           # iterate till the bitter end
37         return x
38
39     else:
40         return pass                            # sqrt(y, x_k+1, debug, dy)
41
42 # Test the code. Feed it pretty bad starting values...
43 #
44 if __name__ == '__main__':
45
46     import sqrt
47     import sys
48
49     # User problem.
50     if len(sys.argv) > 1:
51         y1    = [float(yi) for yi in sys.argv[1:]]
52         x0    = y1
53         debug = False
54
55     # Default problem.
56     else:
57         y1    = [2, 3, 4]
58         x0    = [1.0e-10, 1, 1.0e10]
59         debug = True
60
61     print sqrt.sqrt(y1, x0, debug)
```

5.13.2 Verbatim: “pv.py”

```

1  """
2  @summary:      Solve pig(v) = p1 using Newton-Raphson iteration.
3  @:           Step size is controlled in order to avoid v < 0.
4  @author:       Tore Haug-Warberg
5  @organization: Department of Chemical Engineering, NTNU, Norway
6  @contact:      haugwarb@nt.ntnu.no
7  @license:      GPLv3
8  @requires:     Python 2.3.5 or higher
9  @since:        2011.10.13 (THW)
10 @version:      0.9
11 @todo 1.0:
12 @change:       started (2011.10.13)
13 @note:         On a Unix terminal you can use the script like this:
14
15         >>> python pv.py
16         >>> python pv.py <p1> <t> <v0> <n_tot>
17
18     p1    = pressure [kbar]
19     t     = temperature [kK]
20     v0    = initial volume [dm3]
21     ntot = total number of moles [mol]
22
23 """
24
25 def pv(p1, t=0.29815, v0=1.0, ntot=1.0, debug=False):
26
27     converged = False                                # convergence flag
28     norm = 1.0                                       # convergence control variable
29     eps = 1.0e-8                                     # convergence tolerance
30     v = v0                                           # start volume
31     r = 0.083145119843087                           # gas constant [105 J mol-1 K-1]
32
33     # Solve p(v) = p1 using Newton's method.
34     while not converged:
35         dp_dv = pass                                 # Jacobian
36         dp   = pass                                 # pressure residual
37         dv   = pass                                 # volume change
38         converged = abs(dv) < eps and abs(dv) >= norm # decreasing norm?
39         norm = abs(dv)                             # new norm
40
41     # The model fails if 'v' becomes negative volume. Shorten the iteration
42     # step till the updated volume is positive. Raise an exception if the
43     # step becomes too small.
44     while v+dv < 0.0:
45         if abs(dv) < eps:
46             raise SyntaxError("cannot converge up(v)=up1 relation")
47             pass                                    # reduce the step length (heuristic rule)
48         pass                                      # update volume
49         if debug:
50             print "norm=%8.3g; v=%16.15g;" % (norm, v)
51
52     return v
53
54 # Test the code.
55 #
56 if __name__ == '__main__':
57
58     import pv
59     import sys
60
61     # User problem.
62     if len(sys.argv) == 5:
63         p1    = float(sys.argv[1])
64         t     = float(sys.argv[2])
65         v0    = float(sys.argv[3])
66         ntot = float(sys.argv[4])
67         debug = False
68
69     # Default problem.

```

```

70     else:
71         p1      = 0.2                      # given pressure [kbar]
72         t       = 0.8                      # temperature [kK]
73         v0      = 1.0                      # initial volume [dm3]
74         ntot    = 13.0                     # total mole number [mol]
75         debug   = True
76
77     print '\nInput:'
78     print 'p1=%8.6f; T=%8.6f; V0=%8.6f; Ntot=%8.6f\n' % (p1, t, v0, ntot)
79     print '\nOutput:\nV1=%8.6f\n' % (pv.pv(p1, t, v0, ntot, debug),)

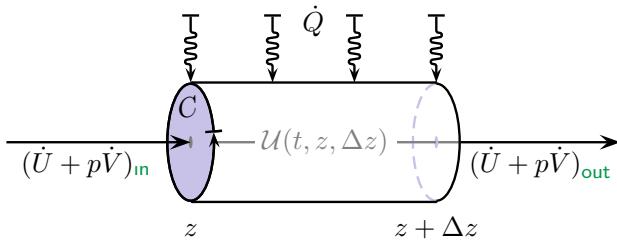
```

Plug Flow Reactor. Part II

Tore Haug-Warberg
 Department of Chemical Engineering
 NTNU (Norway)

16 October 2011
 (completed after 120 hours of writing, programming and testing)

1 The energy balance



The derivation of a rigorous energy balance for any real-life system, of which the idealized Plug Flow Reactor (PFR) is one simple example, demands a *tour de continuum mécanique* which definitely is beyond the scope of this little text. But, we cannot ignore

the energy balance altogether so we must somehow pick up a model description that is mathematically succinct and at the same time physically correct. The following derivation is a humble attempt to reach a reasonably clear disposition of the subject.

Let $\mathcal{U}(t, z, \Delta z)$ be the internal energy of a control volume with one inlet and one outlet. The material flow into the control volume, and out from it, is assumed to be perpendicular to the control surfaces which are situated at z and $z + \Delta z$. This simplification reduces the traditional inner product of the surface normal (vector) and the (vectorial) flows of heat, displacement work, and energy, into their scalar counterparts called \dot{Q} , $p\dot{V}$ and \dot{U} . Note that we shall only consider the flow of internal energy \dot{U} while in the general case we might need to include terms for potential energy, kinetic energy, surface energy, electromagnetic energy and so forth. But, because the picture becomes immensely complicated when every possible term is included, it is important to simplify the model as much as possible without loosing the grip of reality. According to the aforementioned simplifications and the principle of energy conservation we shall write

$$\mathcal{U}(t, z, \Delta z) = \mathcal{U}_o + \int_0^t (\dot{U} + p\dot{V})_z \, d\tau - \int_0^t (\dot{U} + p\dot{V})_{z+\Delta z} \, d\tau + \int_0^t (\dot{Q} - \dot{W}_s) \, d\tau$$

where \dot{W}_s is the mechanical “shaft” work applied to the reactor. Normally it is close to zero. Subscripts z and $z + \Delta z$ are used to denote physical properties that are calculated

at these two spatial positions. This is not to say that \dot{U} and $p\dot{V}$ are functions of z per se. They have co-ordinates of their own which in a way are defined at every point in space and time. This subtlety is discussed further down the text.

In the current context we may put the integration constant \mathcal{U}_0 to zero. It implies that a material system with zero mass has zero energy. This is an important thermodynamic consideration which is true for all chemical systems in the absence of strong electromagnetic radiation.

The symbols \dot{Q} , \dot{V} and \dot{U} stand for the transported heat, volume and energy (per unit time) and has nothing to do with the derivative of a mathematical function, say F , which is defined like:

$$\left(\frac{\partial F}{\partial t}\right) \triangleq \lim_{\Delta t \rightarrow 0} \left(\frac{F(t + \Delta t) - F(t)}{\Delta t} \right)_{x_1, x_2, \dots}$$

This means we need to distinguish clearly between the transportation \dot{F} and the time derivative $(\partial F / \partial t)$. The scientific units are the same but their interpretations are entirely different¹. In other papers you may find \hat{F} being used rather than the dotted form favored here. The meaning is the same though.

To continue, \mathcal{U} and U from which \dot{U} is derived look quite similar, but they do actually measure two different aspects of internal energy. \mathcal{U} is a mathematical construction (we may call it a functional) which has no simple physical description, while U is a thermodynamic state function $U(S, V, N_1, N_2, \dots)$ which by definition is independent of time. That is to say $U(\mathbf{x}, t_1, z_1) = U(\mathbf{x}, t_2, z_2) = \dots$ for fixed values of entropy, volume and mole numbers (collected into one vector \mathbf{x}). To be a state function U must represent the energy of an isotropic system in equilibrium with respect to certain restricted changes in the state variables S, V, N_1, N_2 , etc. (the definition of state variables is made broader later in this text). Hence, it is generally true that $(\partial U / \partial t) = 0$ while $(\partial \mathcal{U} / \partial t) \neq 0$. To proceed, we introduce from thermodynamic theory that $H \triangleq U + pV$. This definition also works for the transported enthalpy:

$$\dot{H} \triangleq \dot{U} + p\dot{V} \quad (1)$$

¹Formal arguments can be raised against this conjecture. Consider a functional \mathcal{F} that describes the amount of energy, mass or any other extensive property that has passed the control surface at z over the time period $[0, t]$. Then

$$\mathcal{F}(t, z) = \int_0^t A \dot{f} d\tau$$

where \dot{f} is the flux (amount per unit area and time) of F , and A is the cross-sectional area of the transport. The time derivative of \mathcal{F} is

$$\left(\frac{\partial \mathcal{F}}{\partial t}\right)_z = A \dot{f} \triangleq \dot{F}$$

So, in a sense \dot{F} is really a partial derivative, but it must be understood that \mathcal{F} has no explicit (and time independent) function expression like e.g. the thermodynamic and kinetic models we are using. Most students have problems in understanding the fundamental difference between dF/dt and $(\partial F / \partial t)$ and I therefore hesitate in calling \dot{F} a derivative because it will bring even more confusion into the subject.

It works because p (the pressure) is an intensive state variable which is independent of the magnitude of the volume flow. At the same time we want to integrate the total heat flux over the external surface of the reactor section

$$\dot{Q} = \int_z^{z+\Delta z} C\dot{q} d\zeta , \quad (2)$$

where C is the circumference of the reactor and \dot{q} is the heat flux (per unit time and surface area). Note that $d\zeta$ rather than dz is acting as an integrator for \dot{q} . We use this convention (Greek integrator—Latin variable) to make sure we do not mix up the integrator symbol with the symbol of either the upper or the lower limit of the integral². This makes the integral a function of z while ζ is consumed during the integration.

It is customary to neglect the heat flow in the axial direction which is why the integral is carried out over the outer surface only. However, strictly speaking there is an order-of-magnitude analysis missing here but this is left as an exercise for the reader. The internal energy of the control volume is then:

$$\mathcal{U}(t, z, \Delta z) = \int_0^t (\dot{H}_z - \dot{H}_{z+\Delta z}) d\tau + \int_0^t \int_z^{z+\Delta z} C\dot{q} d\zeta d\tau$$

This states the energy balance of a simple plug flow reactor. On the form given it is particularly useful for testing and verifying the accuracy of numerical integrators used in dynamic simulation studies, but this is not our goal. We shall proceed instead by calculating the partial derivative \mathcal{U} at a fixed spatial position z with respect to time:

$$\left(\frac{\partial \mathcal{U}}{\partial t} \right)_{z, \Delta z} = \dot{H}_z - \dot{H}_{z+\Delta z} + \int_z^{z+\Delta z} C\dot{q} d\zeta \quad (3)$$

On the current form Eq. 3 leads to a partial differential equation (PDE) in time and space which is considered to be a hard numerical task. But, there are relevant simplifications. In particular we shall study the behaviour of closed systems without throughput of mass and steady state (time independent) systems.

1.1 First law of thermodynamics

A special form of the energy balance applies to *closed systems*. Here, closed means $\dot{H}_z = \dot{H}_{z+\Delta z} = 0$. This appears to be outside the scope of our PFR model but it is still in reach of the thermodynamic formalism. In a system of this kind energy changes

²Dealing mostly with closed and definite integrals we may not even realise the problem, but as we move on to indefinite integrals (antiderivatives) the symbol clash becomes very noticeable. In thermodynamics we define for example the residual function $G^{r,p}(p) \triangleq \int_0^p (V(\pi) - V^{ig}(\pi)) d\pi$ where π is an integrator (over pressure) and p is the system pressure. The convolution integral $F(t) = \int_0^t \varphi(\tau)\psi(t-\tau) d\tau$ used in signal theory is another example. The mutual roles of τ and t must here be sorted out beforehand.

solely because heat is expelled to, or brought in from, the environment. For the change of \mathcal{U} we can then write:

$$(\text{d}\mathcal{U})^{\text{c-s}} = \int_z^{z+\Delta z} C\dot{q} \text{d}\zeta \text{d}t$$

Backsubstitution of \dot{Q} from Eq. 2 yields the simpler form: $\text{d}\mathcal{U} = \dot{Q} \text{d}t$. A similar argument holds also for any kind of external work even though it by coincidence has been excluded in Eq. 3. The reason is that the PFR model is not subject to any volume change nor is it equipped with a mechanical stirrer. If we had decided to include external work (positive when work is delivered by the system) the energy equation would have been extended to $\text{d}\mathcal{U} = \dot{Q} \text{d}t - \dot{W} \text{d}t$.

Taken a bit further it customary to say that $\dot{Q} \text{d}t = \delta Q$ and $\dot{W} \text{d}t = \delta W$ where δQ and δW stand for the non-exact differentials of Q and W . Non-exact means that \mathcal{U} does not depend on Q and W in a definite way. I.e. there exists no function $\mathcal{U}(Q, W)$ such that when Q and W are given then \mathcal{U} is also given. This should be quite intuitive all the time \mathcal{U} is the energy of a material system where the masses of the chemical constituents must also play a role.

In fact, Q and W are path dependent functions of the thermodynamic state, and also of the spatial co-ordinates and of time. They are not state functions in any way and they do not constitute a part of the system. Rather, they express the transportation of energy across the system border. Inside the system, however, heat and work can only be stored as internal energy. There are in other words no “heat content” or “energy content” of the system, only the ability to exchange heat and work with the environment. We therefore talk about “heat potential” and “work potential” to stress the fact that energy (the thermodynamic potential) has to be converted back and forth between heat and work all the time.

Finally, before we leave the discussion of the closed system we shall make a precise interpretation of \mathcal{U} and U . It has already been stated that \mathcal{U} is a constructed energy function—a functional—that serves the need of an accumulation term in the energy equation. From the discussion given above it is clear that \mathcal{U} does not change in a closed system unless there is heat or work exchange with the environment. If there are no interactions of any kind, then all experiments made over the past 200 years indicate that \mathcal{U} gradually becomes undistinguishable from U . That is:

$$\mathcal{U}_{\text{eq}} \hat{=} \lim_{t \rightarrow \infty} \mathcal{U} \rightarrow U$$

The two functions \mathcal{U} and U are identical whenever their function values are the same over the entire definition domain³. In this case \mathcal{U} is constant throughout the experiment so how can it then become *gradually* undistinguishable from U ? The *experiment* tells us that \mathcal{U} does not change in a closed system over time. Our *postulate* says that \mathcal{U} is identical to U when all internal agitation and transients have died out. Before that the measurements of any intensive variables like temperature, pressure and chemical

³E.g. the two functions $f(x) = \cos^2(x) + \sin^2(x)$ and $g(x) = 1$ are mathematically identical for $x \in \mathbb{R}$.

potentials give unreliable readings even though the function values are the same at any time. It is only then all the readings are stable we can say that $\mathcal{U} \equiv U$ in the mathematical understanding of the statement. We call this the equilibrium state of the system. It has an incredible simple representation in the sense that only $n+2$ macroscopic variables are needed in order to establish the value of $U(S, V, N_1, N_2, \dots, N_n)$. From a microscopic point of view this is really *incredible* because there are $6N_A \sum_i N_i$ mechanical degrees of freedom when all the particles in the system are considered as a Newtonian universe. Thermodynamic systems are much simpler, however, because experimentally only the statistically most relevant state is being observed, and since thermodynamics *is* a phenomenological science the observations and theory go hand in hand. This means we can write the energy balance of a closed system as

$$(\text{d}U)^{\text{c-s}} = \delta Q - \delta W$$

which is precisely the first law of thermodynamics. The energy balance in Eq. 3 fulfills in other words the requirements of the first law of thermodynamics albeit in disguise. It must be understood, however, that the usability of $U = \mathcal{U}_{\text{eq}}$ hinges on the fact that the relaxation time of the equilibrium process must be smaller than the time scale of the simulation. This may, or may not, be the case, but for the present purpose we shall assume that U has the meaning of \mathcal{U} ; at least locally for each point in space—if not for the entire system.

1.2 Steady state solution

Eq. 3 has another special meaning whenever the physical situation is such that it allows the left hand side to be put to zero. It is the celebrated *steady state* which reduces the differential equation to a time-independent algebraic equation on the form:

$$(\dot{H}_{z+\Delta z} - \dot{H}_z)^{\text{s-s}} = \int_z^{z+\Delta z} C\dot{q} \text{d}\zeta$$

Despite its simple form the last equation has a wide range of applicability. It is valid for any type of fluid flow, inviscid or not, gas or liquid, one-phase or multi-phase, and with or without chemical reactions.

Just like the displacement work in Eq. 1 was factored into $p\dot{V}$, the transported enthalpy can be factored into the transported mass and a term called the *specific* enthalpy h :

$$\dot{H} = h\dot{M}$$

The inherent scaling properties, namely that $\dot{W} = p\dot{V}$ and $\dot{H} = h\dot{M}$, are deeply rooted in thermodynamic theory and are examples of the so-called Euler homogeneous functions. The energy balance is then reduced to:

$$(h\dot{M})_{z+\Delta z} - (h\dot{M})_z = \int_z^{z+\Delta z} C\dot{q} \text{d}\zeta$$

From the mass conservation principle we know that (for steady-state flow):

$$\dot{M}_{z+\Delta z} - \dot{M}_z = 0$$

Division by $\dot{M}_{z+\Delta z} = \dot{M}_z \triangleq \dot{M}$ on both sides of the equation yields:

$$h_{z+\Delta z} - h_z = \int_z^{z+\Delta z} C \frac{\dot{q}}{\dot{M}} d\zeta$$

In the limit of $\Delta z \rightarrow 0$ we get:

$$\lim_{\Delta z \rightarrow 0} (h_{z+\Delta z} - h_z) = C \frac{\dot{q}}{\dot{M}} \Delta z$$

or rearranged:

$$\lim_{\Delta z \rightarrow 0} \frac{h_{z+\Delta z} - h_z}{\Delta z} = C \frac{\dot{q}}{\dot{M}}$$

We immediately recognize the left hand side as the partial derivative of h with respect to z . On the right hand side we can make the definition $q \triangleq \dot{q}/\dot{M}$ standing for the specific heat load (energy per unit mass and area). The energy balance for a steady state reactor with only internal energy flow is then:

$$\left(\frac{\partial h}{\partial z} \right)^{\text{s-s}} = Cq$$

The anti-derivative of the energy balance defines the so-called enthalpy equation (please note the integral on the right side is zero for an adiabatic reactor without external heat load):

$$h(z) = h(0) + \int_0^z C(\zeta) q(\zeta) d\zeta$$

At this point we need to worry about the mathematical notation we are using. The operations are formally correct up to the point where $\Delta z \rightarrow 0$, but here it stops. At some finite value of Δz it becomes smaller than the resolution of the measurement. Or, it may in fact become smaller than the effective size of the molecules comprising the system and on this tiny scale h loses its meaning since it requires a big number of colliding molecules to establish a thermodynamic state variable. Hence, the derivative $(\partial h / \partial z)$ does not exist in proper. It is only the finite difference $h_{z+\Delta z} - h_z$ that is physically measureable, and then only if Δz is *sufficiently* large. This is not a practical problem in most cases, but for e.g. high-vacuum systems we must take precautions because the distance covered between two successive collisions of the molecules can be of the order millimeters or even centimeters.

Our second worry is that h is not a function of the spatial co-ordinate z . It is in fact a function of the state variables $T, v \triangleq V/M, c_1 \triangleq N_1/M, c_2 \triangleq N_2/M$, etc. when any of the modern pressure explicit equations of state are being used in the modelling (most

of them are descendants of the Van der Waals equation of state from 1873). Hence, $(\partial h/\partial z)$ does not exist other than as a formal expression, but from differential calculus we know that $\text{d}h/\text{d}z$ takes the same numerical value as $(\partial h/\partial z)$ when all the degrees of freedom except one (i.e. z) are locked. However, the total differential of h is

$$\begin{aligned}\text{d}h = & \left(\frac{\partial h}{\partial T} \right)_{v,c_1,c_2,\dots} \text{d}T + \left(\frac{\partial h}{\partial v} \right)_{T,c_1,c_2,\dots} \text{d}v \\ & + \left(\frac{\partial h}{\partial c_1} \right)_{T,v,c_2,c_3,\dots} \text{d}c_1 + \left(\frac{\partial h}{\partial c_2} \right)_{T,v,c_1,c_3,\dots} \text{d}c_2 + \dots\end{aligned}$$

or given a more compact form:

$$\text{d}h = \partial_T h \cdot \text{d}T + \partial_v h \cdot \text{d}v + \partial_{c_1} h \cdot \text{d}c_1 + \partial_{c_2} h \cdot \text{d}c_2 + \dots$$

Inventing a new notation “over the night” is not something I usually recommend, but we will run out of paper pretty soon unless we do something about the partial derivatives flourishing all over the place. Dividing by $\text{d}z$ (which is an algebraic quantity remember—and by the way quite different from ∂_z which is an operator) gives the differential quotient:

$$\begin{aligned}\left(\frac{\text{d}h}{\text{d}z} \right) = & \left(\frac{\partial h}{\partial T} \right)_{v,c_1,c_2,\dots} \left(\frac{\text{d}T}{\text{d}z} \right) + \left(\frac{\partial h}{\partial v} \right)_{T,c_1,c_2,\dots} \left(\frac{\text{d}v}{\text{d}z} \right) \\ & + \left(\frac{\partial h}{\partial c_1} \right)_{T,v,c_2,c_3,\dots} \left(\frac{\text{d}c_1}{\text{d}z} \right) + \left(\frac{\partial h}{\partial c_2} \right)_{T,v,c_1,c_3,\dots} \left(\frac{\text{d}c_2}{\text{d}z} \right) + \dots\end{aligned}$$

or, using our shorter notation:

$$\nabla h = \partial_T h \cdot \nabla T + \partial_v h \cdot \nabla v + \partial_{c_1} h \cdot \nabla c_1 + \partial_{c_2} h \cdot \nabla c_2 + \dots$$

This is precisely the expression we are looking for. The crux of the matter is that ∇h takes the same numerical value as $(\partial h/\partial z)$, but to carry on we need to first solve an equation system that settles the values of ∇T , ∇v , ∇c_1 , ∇c_2 , etc. This is done by simultaneously solving the energy, momentum and mass balances at the inlet of the reactor and integrating the solution variables along the spatial co-ordinate z . The how’s and why’s are fully explained in Part III of this paper entitled *Modelling Issues*. The implicitness of the conservation statement is so fundamental to the thermodynamist, however, that it is really deserves an introductory example. The internal workings of the so-called Jacobian transformation is explained below.

1.3 Calculation example

Doing matrix algebra by hand is hard work but there is no other way we can get an understanding of how the linearization really works. So, to gain the insight we shall practise on a minimalistic 2×2 example. Assume a problem on the form:

$$\begin{aligned}H^{\text{ig}}(T, V) &\triangleq C_P^{\text{ig}} T = H_{\circ} \\ p^{\text{ig}}(T, V) &\triangleq \frac{NRT}{V} = p_{\circ}\end{aligned}$$

where N is constant, and H_\circ and p_\circ are conserved quantities. Let $\mathbf{x} \hat{=} (T \ V)$ and $\mathbf{y} \hat{=} (H \ p)$. To solve $\mathbf{y}(\mathbf{x}) = \mathbf{y}_\circ$ we first linearize $\mathbf{y}(\mathbf{x})$ and then attempt to solve the equations iteratively using the Newton–Raphson method:

$$\mathbf{y}_k + \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}^T} \right)_k (\mathbf{x}_{k+1} - \mathbf{x}_k) = \mathbf{y}_\circ$$

Rearrangement gives:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{J}_k^{-1}(\mathbf{y}_k - \mathbf{y}_\circ)$$

where

$$\mathbf{J}_k \hat{=} \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}^T} \right)_k = \begin{pmatrix} \left(\frac{\partial H}{\partial T} \right)_V & \left(\frac{\partial H}{\partial V} \right)_T \\ \left(\frac{\partial p}{\partial T} \right)_V & \left(\frac{\partial p}{\partial V} \right)_T \end{pmatrix}_k$$

so that:

$$\mathbf{J}_k^{-1} = \begin{pmatrix} C_P^{ig} & 0 \\ \frac{NR}{V} & -\frac{NRT}{V^2} \end{pmatrix}_k^{-1} = \frac{-1}{C_P^{ig} NRT} \begin{pmatrix} -\frac{NRT}{V^2} & 0 \\ -\frac{NR}{V} & C_P^{ig} \end{pmatrix}_k = \begin{pmatrix} \frac{1}{C_P^{ig}} & 0 \\ \frac{V}{C_P^{ig} T} & -\frac{V^2}{NRT} \end{pmatrix}_k$$

The remaining algebra is straightforward:

$$\begin{pmatrix} T \\ V \end{pmatrix}_{k+1} = \begin{pmatrix} T \\ V \end{pmatrix}_k - \begin{pmatrix} \frac{1}{C_P^{ig}} & 0 \\ \frac{V}{C_P^{ig} T} & -\frac{V^2}{NRT} \end{pmatrix}_k \left[\begin{pmatrix} C_P^{ig} T \\ \frac{NRT}{V} \end{pmatrix}_k - \begin{pmatrix} H \\ p \end{pmatrix}_\circ \right]$$

Iteration example: $H_\circ = 10^4 \text{ J}$, $p_\circ = 10^6 \text{ Pa}$, $N = 1 \text{ mol}$, $C_P^{ig} = \frac{5}{2}R$, $R = 8.3145 \text{ J mol}^{-1} \text{ K}^{-1}$.

k	$T \text{ [K]}$	$V \text{ [m}^3\text{]}$
0	298.15	0.001
1	481.087257201275	0.00221018092537634
2	481.087257201275	0.00319913692002833
3	481.087257201275	0.00383965458178457
4	481.087257201275	0.00399357233671433
6	481.087257201275	0.00399998967128617
7	481.087257201275	0.00399999999997333
8	481.087257201275	0.004

The Newton–Raphson iteration is a so-called second order method. One characteristic feature is that the number of significant digits will double in each iteration *sufficiently* close to the solution (iteration 3 onward). Verify this behaviour. From the table it is also clear that T converges in one step whilst V requires 8 iterations. Give a reason for this observation⁴. Finally, it should be mentioned that the Newton–Raphson method is sensitive to the starting values. E.g. try to start the iteration at $V = 0.01$ rather than $V = 0.001$. Suggest a possible fix to the algorithm in this case⁵.

⁴: $H(T, V)$ is strictly linear in both T and V .

⁵: Unphysical volume update. Step length restriction is necessary.

1.4 Epilogue

I have in this little text sought to establish a fairly rigorous derivation of the energy balance for an idealized plug flow reactor. It is neither highly sophisticated nor does it require advanced mathematics. Still, it is not of a kind that is eagerly agreed upon by the chemical engineering community—be it professors, students or working professionals. Many people find the painstaking calculations of differentials and partial derivatives confusing and of little practical interest, but the latter is definitely wrong. The very fact that ∇T , ∇v and ∇c_i are solution variables of a set of model equation whereas $\partial_T h$, $\partial_v h$ and $\partial_{c_i} h$ are explicit (or sometimes implicit) state functions establishing the coefficient matrix of the model equations is so important that it can hardly be overemphasized.

The culprit in this controversy might be the teaching of $\text{d}y/\text{d}x = y'$ in highschool mathematics. By doing so the students learn that $\text{d}y/\text{d}x$ is synonymous with $y' \triangleq (\partial y / \partial x)$ and that the rest of the story is just syntactic sugar. For one-variable systems I can agree that the difference is subtle, but for many-variable systems it is not. The discussion has much in common with the use of substantial derivatives in fluid mechanics which says: $\text{d}y/\text{d}t = (\partial y / \partial t) + (\partial y / \partial x_1) \text{d}x_1/\text{d}t + (\partial y / \partial x_2) \text{d}x_2/\text{d}t + \dots$. In this case I think it can hardly be misunderstood that $\text{d}y/\text{d}t$ and $(\partial y / \partial t)$ are different mathematical objects—and very different ones as well.

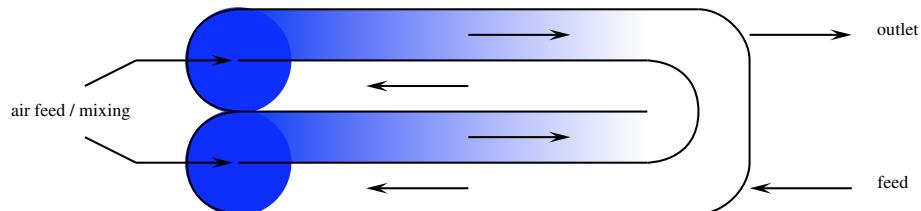
Exercise 7

Preisig, H A

Chemical Engineering, NTNU

1 Question: Water treatment plant

The activated sludge unit is part of any modern municipal waste-water treatment plant. It operates in two major modes, aerobic and anaerobic. Any such cleaning unit must provide for both modes of operation and preferably in a way so that the water is exposed to switching conditions several times. Combining this requested feature with space limitations brought about the design of circular reactors, which include aerobic and anaerobic sections in sequences. The design is called the carousel design. It builds on a circular main flow, feeding sewage water at one location and taking out cleaned water at the other location. Air is injected in one or several places to generate aerobic conditions in some parts of the reactor. The air injection also acts as mixer. Away from the air injection, the oxygen contents decreases quite rapidly leaving the rest of the reactor to operate in the anaerobic mode, so with no oxygen present.



We are to model the water treatment plant as a series of compartments for which the scheme is given in the Figure above. Relevant species are: A:: species to be oxidized, B:: species to be reduced, D:: inert species, O:: oxidant, R:: reductant, W :: water

- Establish an abstract topology (capacities and transport) reflecting the main dynamics of the process. Add labels to indicate capacities and streams.
- Show in a table on what balance equation has to be written for what capacity in terms of component masses and energy.

2 Question: Reactions 02

Consider the following reactions are present in an ideally stirred tank reactor:



(2)

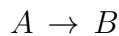
where $k_2 \gg k_1$ and the rate of reactions of first order functions of the concentrations.

Tasks

1. Write the species balances.
2. Use the order of magnitude assumption of $k_2 \gg k_1$ to do a singular perturbation on the balance of species B showing that it is equivalent to pseudo steady state for species B.
3. Eliminate the composition of species B from the balance equations.

3 Question: Shell balance 02

We want to model the diffusion of component A coupled with the following reaction in a spherical catalyst



where

$$\tilde{n}_A = -k c_A$$

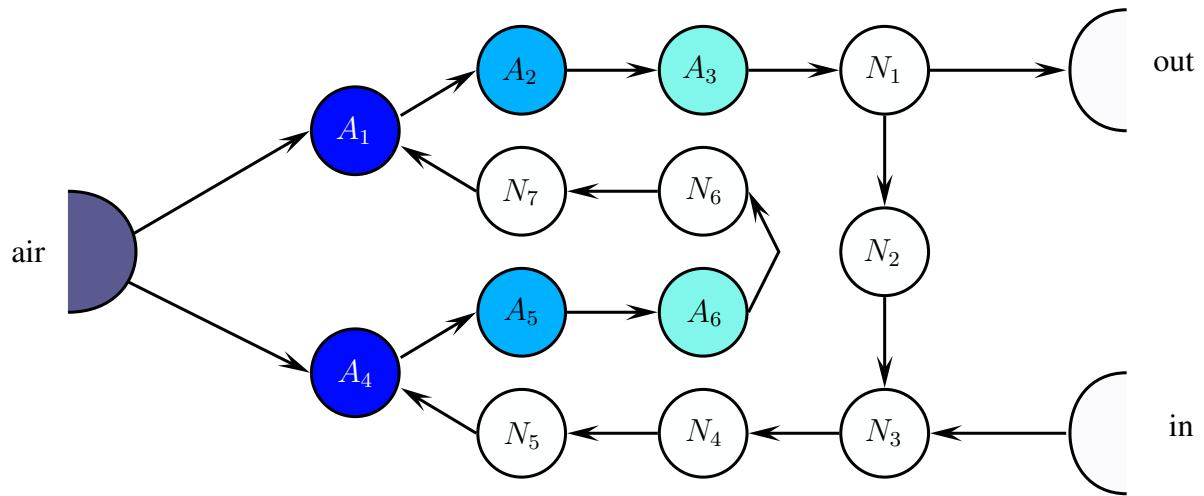
Our objective is to determine the variation of the concentration. The mass flux is only in the radial direction and is calculated from Fick's law:

$$\hat{\hat{n}}_A = -D_{AB} \frac{dc_A}{dr}$$

Note that the flux $\hat{\hat{n}}$ is the flow per unit area.

1 Solution: Carrousel

1.1 Topology



1.2 What balances

system	A	B	D	O	R	W
A_i	x	x	x	x	x	x
N_i	x	x	x		x	x

2 Solution Reactions 02

Writing the balance for each species we will have:

$$\underline{\dot{n}} = \underline{\tilde{n}}$$

The production rate is:

$$\begin{aligned}\underline{\tilde{n}} &:= V \underline{\underline{N}}^T \underline{\tilde{\xi}} \\ \underline{\tilde{\xi}} &:= \underline{\underline{K}} \underline{g}(\underline{c}) \\ \underline{c} &:= V^{-1} \underline{n}\end{aligned}$$

The stoichiometry is:

$$\underline{\underline{N}} := \begin{bmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \end{bmatrix}$$

The frequency functions are:

$$\underline{g}(\underline{c}) := \underline{c}$$

and the reaction constant matrix:

$$\underline{\underline{K}} := \text{diag} [k_1 \ k_2]$$

If we now substitute to get the extended equations:

$$\begin{aligned}\dot{n}_A &= -k_1 c_A \\ \dot{n}_B &= k_1 c_A - k_2 c_B \\ \dot{n}_c &= k_2 c_B\end{aligned}$$

The order of magnitude assumption is that $k_2 \gg k_1$. Thus we are going to use k_2 as the singular perturbation parameter by deviding the second balance equation, namely the one for the species B by k_2 .

$$\begin{aligned}\frac{1}{k_2} \dot{n}_B &= \frac{k_1}{k_2} c_A - c_B \\ \epsilon \dot{n}_B &= \frac{k_1}{k_2} c_A - c_B\end{aligned}$$

Taking the limit gives us the behaviour of the B-balance for larger time scales:

$$\lim_{\epsilon \rightarrow 0} \epsilon \dot{n}_B = \frac{k_1}{k_2} c_A - c_B$$

So,

$$0 = \frac{k_1}{k_2} c_A - c_B$$

and

$$c_B = \frac{k_1}{k_2} c_A$$

which simplifies the model to:

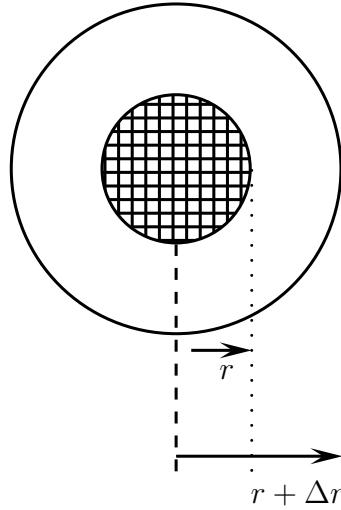
$$\dot{n}_A = -k_1 c_A$$

and

$$\begin{aligned}\dot{n}_c &= k_2 \frac{k_1}{k_2} c_A \\ &= k_1 c_A\end{aligned}$$

3 Solution: Shell balance 02

Conforming with the assumption of uniform environment, the temperature changes only in the radial direction. The geometry is shown in Figure 3:



Let the extensive quantity be Φ and its flux $\hat{\varphi}$. Then a balance over a small volume element ΔV is

$$\frac{d\Phi}{dt} := A_r \hat{\varphi}|_r - A_{r+\Delta r} \hat{\varphi}|_{r+\Delta r}$$

with

$$A_{r+\Delta r} \hat{\varphi}|_{r+\Delta r} := A_r \hat{\varphi}|_r + \left. \frac{\partial A \hat{\varphi}}{\partial r} \right|_{r+\Delta r}$$

substitution yields:

$$\begin{aligned}
\frac{d\Phi}{dt} &:= A_r \hat{\varphi}|_r - \left(A_r \hat{\varphi}|_r + \frac{\partial A \hat{\varphi}}{\partial r} \Big|_r \Delta r \right) \\
&:= - \frac{\partial A \hat{\varphi}}{\partial r} \Big|_r \Delta r \\
&:= - \frac{\partial A}{\partial r} \Big|_r \hat{\varphi}|_r \Delta r - A_r \frac{\partial \hat{\varphi}}{\partial r} \Big|_r \Delta r \\
&:= - \frac{2A_r}{r} \hat{\varphi}|_r \Delta r - A_r \frac{\partial \hat{\varphi}}{\partial r} \Big|_r \Delta r \\
&:= - \left(\frac{2}{r} \hat{\varphi}|_r + \frac{\partial \hat{\varphi}}{\partial r} \Big|_r \right) A_r \Delta r
\end{aligned}$$

where we used the fact that the area being a quadratic function of r . Thus we used:

$$\frac{\partial A}{\partial r} := \frac{\partial 4\pi r^2}{\partial r} := 8\pi r$$

Next we divide by the volume and take the limit:

$$\begin{aligned}
\lim_{\Delta V \rightarrow 0} \frac{d\Phi/\Delta V}{dt} &:= - \left(\frac{2}{r} \hat{\varphi}|_r + \frac{\partial \hat{\varphi}}{\partial r} \Big|_r \right) \\
\frac{\partial \varphi}{\partial t} &:= - \left(\frac{2}{r} \hat{\varphi} + \frac{\partial \hat{\varphi}}{\partial r} \Big|_r \right)
\end{aligned}$$

The flux $\hat{\varphi}$ for an isotropic material is $-D_{AB} \frac{\partial c_A}{\partial r}$, with D_{AB} being the diffusivity parameter. Thus substituting this flux relation into the above equation gives

$$k \frac{\partial c_A}{\partial t} := D_{AB} \left(\frac{2}{r} \frac{\partial c_A}{\partial r} \Big|_r + \frac{\partial^2 c_A}{\partial r^2} \Big|_r \right)$$

Solving a Set of Non-Linear Equations (TKP4106)



[Zooball/Beaver](#)

"... one of the main causes of the fall of the Roman Empire was that, lacking zero, they had no way to indicate successful termination of their C programs."

[Robert Firth](#)

Assignments

1. Write a procedure `solve` for solving a set of linear equations using the Row-Reduced-Echelon form of matrix `A`. Hint: For the linear equation system $A \cdot X = B$ we get `rref([A | B]) = [I | X]` according to the definition of `rref`. Object `B` is a "matrix" in this case. If it so happens that `B` has a single column `b` we end up with the special case $A \cdot x = b$, but there is not much to save, neither in time nor in programming lines, from disregarding the general solution. So, go for it! Use the stub program [solve.py](#) as template.
2. Linearize the energy balance and the pressure specification of the Plug Flow Reactor. Combine it with the mass balance into one simultaneous set of linear(ized) equations. Write a solver that iterates on `T`, `v`, `c_1`, `c_2`, ... to find a thermodynamic state which is constrained by `h`, `p`, `c_1`, `c_2`, Use the stub program [hpn.py](#) as template.
3. It can also be worth while programming the matrix (inner) product for later use. Use the stub program [mprod.py](#) as template.

Continue reading about [The energy balance](#) if you need further guidance to the understanding of energy, enthalpy, thermodynamics and the mapping between different co-ordinate systems.

%Predefined.

HTML text.

5.15.1 Verbatim: “solve.py”

```
1  """
2  @summary:      Calculate xmat from amat * xmat = bmat.
3  @author:       Tore Haug-Warberg
4  @organization: Department of Chemical Engineering, NTNU, Norway
5  @contact:      haugwarb@nt.ntnu.no
6  @license:      GPLv3
7  @requires:     Python 2.3.5 or higher
8  @since:        2011.08.30 (THW)
9  @version:      0.8
10 @todo 1.0:
11 @change:       started (2011.08.30)
12 """
13
14 def solve(amat, bmat, debug=False):
15     """
16     Solve the linear equation system amat * xmat = bmat using rref(augm) where
17     augm = [amat | bmat] is the row augmented matrix [amat[0] + bmat[0], ...].
18
19     @param amat:  Input matrix given as a list of lists of numbers
20     @param bmat:  Right hand specification given as a list of lists of numbers
21     @param debug: True or False flag
22
23     @type amat:   aList [ aList [ aNumber, aNumber, ... ], ... ]
24     @type bmat:   <pass>
25     @type debug:  <pass>
26
27     @return:      aList [ aList [ aFloat, aFloat, aFloat ] ]
28                 e.g. [[1.0, 2.0, ...], [3.0, 4.0, ...], [5.0, 6.0, ...], ...]
29     """
30
31     # Row-reduced-echelon-form.
32     from rref import rref
33
34     if not(amat) or not(amat[0]):
35         pass                                         # raise exception
36
37     if len(amat) != len(amat[0]):
38         pass                                         # raise exception
39
40     if not(bmat) or not(bmat[0]):
41         pass                                         # raise exception
42
43     if len(bmat) != len(amat[0]):
44         pass                                         # raise exception
45
46     augm = pass                                     # augmented matrix [amat | bmat]
47
48     augm, rank, pivots = rref(augm, debug)
49
50     if rank != len(amat):
51         pass                                         # raise exception
52
53     return pass                                     # return solution
```

5.15.2 Verbatim: “hpн.py”

```

1      """
2      @summary: Solve (H, p, N1, N2, ..., N5) versus (T, V, N1, N2, ..., N5) for the
3          ideal gas equation of state. The pertinent equations are::
4
5          H      = sum_i ( h(T)_i * N_i )
6          h(T)_i = h0_i + int_{T0}^T ( cp(T)_i * dT )
7          Cp     = sum_i ( cp(T)_i * N_i )
8          cp(T)_i = c1_i + c2_i*t + c3_i*t**2 + c4_i*t**3
9          p      = ntot *R * T / V
10         ntot   = sum_i ( N_i )
11
12         The strategy is to implement a standard Newton-Raphson iterator and
13         solve::
14
15         (tvn)^{k+1} = (tvn)^{k} + d(tvn)
16         d(tvn)       = inv(jac) * (y1 - hpн)
17
18         repeatedly until the norm of d(tvn) is not decreasing anymore. On
19         the right side 'y1' is a given constraint "matrix":-
20
21             [ [ H1    ],
22               [ p1    ],
23               y1 =  [ N1_1 ],
24               [ N1_2 ],
25               ...
26             ]
27
28         and 'hpн' is a similarly shaped "matrix" of ideal gas properties
29         calculated as functions of T, V, and N_1, ... N_5:-
30
31             [ [ H   ],
32               [ p   ],
33               hpн = [ N1  ],
34               [ N2  ],
35               ...
36             ]
37
38         The Jacobian of H, p, N1, N2, ... with respect to T, V, N1, N2, ...
39         is on the form:-
40
41             [ [ dH/dT,   dH/dV,   dH/dN1,   dH/dN2,   ... ],
42               [ dp/dT,   dp/dV,   dp/dN1,   dp/dN2,   ... ],
43               jac =  [dN1/dT, dN1/dV, dN1/dN1, dN1/dN2, ... ],
44               [dN2/dT, dN2/dV, dN2/dN1, dN2/dN2, ... ],
45               ...
46             ]
47
48         @author:      Tore Haug-Warberg
49         @organization: Department of Chemical Engineering, NTNU, Norway
50         @contact:     haugwarb@nt.ntnu.no
51         @license:     GPLv3
52         @requires:    Python 2.3.5 or higher
53         @since:       2011.10.13 (THW)
54         @version:     0.0.1
55         @todo 1.0:
56         @change:      started (2011.11.13)
57         @note:
58
59         Test the program entering one of the following lines from the command line:-
60
61         >>> python hpн.py
62         >>> python hpн.py <H1> <p1> <N1_1> ... <N1_5>
63         >>> python hpн.py <H1> <p1> <N1_1> ... <N1_5> <T0> <V0> <N0_1> ... <N0_5>
64
65         H1      = final enthalpy [10^5 J]
66         p1      = final pressure [kbar]
67         N1_1   = final mole number of component 1 [mol]
68         ...
69         N1_5   = final mole number of component 5 [mol]

```

```

70     TO    = initial temperature [kK]
71     VO    = initial volume [dm3]
72     NO_1 = initial mole number of component 1 [mol]
73     ...
74     NO_5 = initial mole number of component 5 [mol]
75
76     """
77
78     import tkp4106
79
80     def hpn_vs_tvn_solver(y1, x0, eps=1.0e-8, maxiter=50):
81
82         fix_rgass = 0.083145119843087                      # gas constant
83         var_t     = x0[0][-1]                                # temperature [kK]
84         var_v     = x0[1][-1]                                # volume [dm3]
85         var_n     = [ni[-1] for ni in x0[2:]]               # mole numbers [mol]
86         par_h0   = [-.45898, 0.00000, 0.00000, 0.00000, -.74520] # h0 [10^5 J/mol]
87         par_c1_cp = [0.27310, 0.31150, 0.27140, 0.20786, 0.01925] # Cp coefficient
88         par_c2_cp = [0.23830, -.13570, 0.09274, 0.00000, 0.52130] # Cp coefficient
89         par_c3_cp = [0.17070, 0.26800, -.13810, 0.00000, 0.11970] # Cp coefficient
90         par_c4_cp = [-.11850, -.11680, 0.07645, 0.00000, -.11320] # Cp coefficient
91
92         converged = False                                  # convergence flag
93         norm = 1.0                                       # convergence control variable
94         ni = 0                                           # number of iterations
95         nc = len(var_n)                                 # number of components in mixture
96
97         while not converged:
98             ni += 1
99
100            t = var_t
101            v = var_v
102            n = var_n
103            r = fix_rgass
104
105            ntot = sum(n)
106
107            # Initialization of enthalpy and its derivatives.
108            state_h = 0.0
109            state_h_t = 0.0
110            state_h_v = 0.0
111            state_h_n = [0.0]*nc
112
113            state_p = pass                                # p(T,V,n)
114            state_p_t = pass                             # (dp/dT)_{V,n}
115            state_p_v = pass                             # (dp/dV)_{T,n}
116            state_p_n = pass                            # (dp/dn[i])_{T,V,n[j]}
117
118            state_n = n
119            state_n_t = pass                           # (dn/dT)_{V,n}
120            state_n_v = pass                           # (dn/dV)_{T,n}
121            state_n_n = [int(i==j) for i in xrange(0,nc) for j in xrange(0,nc)]
122
123            t0 = 0.29815                                 # standard state temperature
124
125            for i in xrange(0, nc):
126                hti = par_h0[i] + \
127                    pass + \
128                    pass + \
129                    pass + \
130                    pass                                # int_{t0}^{T} cp[i](t) dt
131                cpi = par_c1_cp[i] + \
132                    par_c2_cp[i]*t + \
133                    par_c3_cp[i]*t**2 + \
134                    par_c4_cp[i]*t**3                         # cp[i](T)
135                state_h += pass                           # H(T,V,n)
136                state_h_t += pass                          # (dH/dT)_{V,n}
137                state_h_v += pass                          # (dH/dV)_{T,n}
138                state_h_n[i] = pass                      # (dH/dn[i])_{T,V,n[j]}
139
140            hpn = [[state_h]] + [[[state_p]] + [[ni] for ni in state_n]
141
```

```

142     dh = [state_h_t] + [state_h_v] + state_h_n                      # dH/d(T,V,n)
143     dp = pass                                         # dp/d(T,V,n)
144     dn = [\]
145         [state_n_t[i]] +
146         [state_n_v[i]] + \
147             state_n_n[i*nc:(i+1)*nc] for i in xrange(0, nc) \
148         ]                                         # dn/d(T,V,n)
149
150     jac = pass                                         # d(H,p,n)/d(T,V,n)
151
152     dy = pass                                         # y1 - (H,p,n)
153     dx = tkp4106.solve(jac, dy)
154     tmp = max([abs(dxi[-1]) for dxi in dx])
155     converged = abs(tmp) < eps and abs(tmp) >= norm
156     norm = abs(tmp)
157     print "norm=%8.3g;" % (norm,)
158     if not converged and ni >= abs(maxiter):
159         raise SyntaxError("maxIterations(%s) exceeded" % (ni,))
160     var_t += pass                                         # update temperature
161     var_v += pass                                         # update volume
162     var_n = pass                                         # update mole numbers
163
164     tvn = [[var_t]] + [[var_v]] + [[ni] for ni in var_n]
165     hpn = [[state_h]] + [[state_p]] + [[ni] for ni in state_n]
166
167     return [tvn, hpn]
168
169 # Test the code.
170 #
171 if __name__ == '__main__':
172
173     import hpn
174     import sys
175
176     # Read in H1, p1 and n1, plus T0, V0 and n0 from the command line.
177     if len(sys.argv) == 7+7+1:
178         x0 = [[float(x0i)] for x0i in sys.argv[8:]]           # T, V, n
179         y1 = [[float(y1i)] for y1i in sys.argv[1:8]]           # H, p, n
180
181     # Read in H1, p1 and n1 from the command line. Use default T0, V0 and n0.
182     elif len(sys.argv) == 7+1:
183         x0 = [[0.29815], [0.001], [2.0], [1.5], [0.5], [3.0], [1.0]]    # T, V, n
184         y1 = [[float(y1i)] for y1i in sys.argv[1:]]           # H, p, n
185
186     # Use default H1, p1 and n1, plus default T0, V0 and n0.
187     else:
188         x0 = [[0.29815], [0.001], [2.0], [1.5], [0.5], [3.0], [1.0]]    # T, V, n
189         y1 = [[0], [0.1], [1.0], [2.5], [1.5], [2.0], [3.0]]           # H, p, n
190
191     tvn, hpn = hpn.hpn_vs_tvn_solver(y1, x0)
192
193     print '\nInput:'
194     print "T0=%12.6g; V0=%12.6g; n0=%s;" % (x0[0][-1], x0[1][-1], x0[2:])
195     print "H1=%12.6g; p1=%12.6g; n1=%s;" % (y1[0][-1], y1[1][-1], y1[2:])
196
197     print '\nOutput:'
198     print "T=%12.6g; V=%12.6g; n=%s;" % (tvn[0][-1], tvn[1][-1], tvn[2:])
199     print "H=%12.6g; p=%12.6g; n=%s;" % (hpn[0][-1], hpn[1][-1], hpn[2:])

```

5.15.3 Verbatim: “mprod.py”

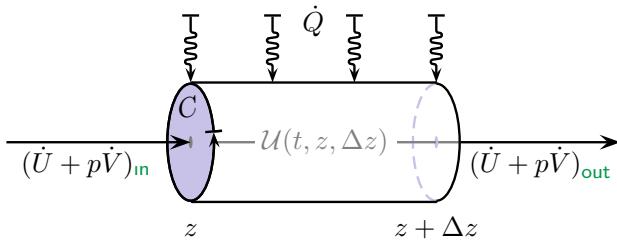
```
1  """
2  @summary:      Calculate the full matrix product cmat = amat * bmat.
3  @author:       Tore Haug-Warberg
4  @organization: Department of Chemical Engineering, NTNU, Norway
5  @contact:      haugwarb@nt.ntnu.no
6  @license:      GPLv3
7  @requires:     Python 2.3.5 or higher
8  @since:        2011.08.30 (THW)
9  @version:      0.9
10 @todo 1.0:
11 @change:       started (2011.08.30)
12 """
13
14 def mprod(amat, bmat, debug=False):
15     """
16     Matrix multiplication of amat * bmat = cmat.
17
18     @param amat:  <pass>
19     @param bmat:  <pass>
20     @param debug: <pass>
21
22     @type amat:   aList [ aList [ aNumber, aNumber, ... ], ... ]
23     @type bmat:   <pass>
24     @type debug:  <pass>
25
26     @return:      aList [ aList [ aFloat, aFloat, aFloat ] ]
27                 e.g. [[1.0, 2.0, ...], [3.0, 4.0, ...], [5.0, 6.0, ...], ...]
28 """
29
30     if not(amat) or not(amat[0]):
31         pass                                # raise exception
32
33     if not(bmat) or not(bmat[0]):
34         pass                                # raise exception
35
36     if len(bmat) != len(amat[0]):
37         pass                                # raise exception
38
39     # Output matrix has dimension: rows(amat) x columns(bmat).
40     cmat = [[0 for b in bmat[0]] for a in amat]
41
42     for i pass                         # rows in amat = rows in cmat
43         for j pass                      # columns in bmat = columns in cmat
44             for k pass                    # columns in amat = rows in bmat
45                 pass                      # calculate cmat[i][j]
46
47     return cmat
```

Plug Flow Reactor. Part II

Tore Haug-Warberg
 Department of Chemical Engineering
 NTNU (Norway)

16 October 2011
 (completed after 120 hours of writing, programming and testing)

1 The energy balance



The derivation of a rigorous energy balance for any real-life system, of which the idealized Plug Flow Reactor (PFR) is one simple example, demands a *tour de continuum mécanique* which definitely is beyond the scope of this little text. But, we cannot ignore

the energy balance altogether so we must somehow pick up a model description that is mathematically succinct and at the same time physically correct. The following derivation is a humble attempt to reach a reasonably clear disposition of the subject.

Let $\mathcal{U}(t, z, \Delta z)$ be the internal energy of a control volume with one inlet and one outlet. The material flow into the control volume, and out from it, is assumed to be perpendicular to the control surfaces which are situated at z and $z + \Delta z$. This simplification reduces the traditional inner product of the surface normal (vector) and the (vectorial) flows of heat, displacement work, and energy, into their scalar counterparts called \dot{Q} , $p\dot{V}$ and \dot{U} . Note that we shall only consider the flow of internal energy \dot{U} while in the general case we might need to include terms for potential energy, kinetic energy, surface energy, electromagnetic energy and so forth. But, because the picture becomes immensely complicated when every possible term is included, it is important to simplify the model as much as possible without loosing the grip of reality. According to the aforementioned simplifications and the principle of energy conservation we shall write

$$\mathcal{U}(t, z, \Delta z) = \mathcal{U}_o + \int_0^t (\dot{U} + p\dot{V})_z \, d\tau - \int_0^t (\dot{U} + p\dot{V})_{z+\Delta z} \, d\tau + \int_0^t (\dot{Q} - \dot{W}_s) \, d\tau$$

where \dot{W}_s is the mechanical “shaft” work applied to the reactor. Normally it is close to zero. Subscripts z and $z + \Delta z$ are used to denote physical properties that are calculated

at these two spatial positions. This is not to say that \dot{U} and $p\dot{V}$ are functions of z per se. They have co-ordinates of their own which in a way are defined at every point in space and time. This subtlety is discussed further down the text.

In the current context we may put the integration constant \mathcal{U}_0 to zero. It implies that a material system with zero mass has zero energy. This is an important thermodynamic consideration which is true for all chemical systems in the absence of strong electromagnetic radiation.

The symbols \dot{Q} , \dot{V} and \dot{U} stand for the transported heat, volume and energy (per unit time) and has nothing to do with the derivative of a mathematical function, say F , which is defined like:

$$\left(\frac{\partial F}{\partial t}\right) \triangleq \lim_{\Delta t \rightarrow 0} \left(\frac{F(t + \Delta t) - F(t)}{\Delta t} \right)_{x_1, x_2, \dots}$$

This means we need to distinguish clearly between the transportation \dot{F} and the time derivative $(\partial F / \partial t)$. The scientific units are the same but their interpretations are entirely different¹. In other papers you may find \hat{F} being used rather than the dotted form favored here. The meaning is the same though.

To continue, \mathcal{U} and U from which \dot{U} is derived look quite similar, but they do actually measure two different aspects of internal energy. \mathcal{U} is a mathematical construction (we may call it a functional) which has no simple physical description, while U is a thermodynamic state function $U(S, V, N_1, N_2, \dots)$ which by definition is independent of time. That is to say $U(\mathbf{x}, t_1, z_1) = U(\mathbf{x}, t_2, z_2) = \dots$ for fixed values of entropy, volume and mole numbers (collected into one vector \mathbf{x}). To be a state function U must represent the energy of an isotropic system in equilibrium with respect to certain restricted changes in the state variables S, V, N_1, N_2 , etc. (the definition of state variables is made broader later in this text). Hence, it is generally true that $(\partial U / \partial t) = 0$ while $(\partial \mathcal{U} / \partial t) \neq 0$. To proceed, we introduce from thermodynamic theory that $H \triangleq U + pV$. This definition also works for the transported enthalpy:

$$\dot{H} \triangleq \dot{U} + p\dot{V} \quad (1)$$

¹Formal arguments can be raised against this conjecture. Consider a functional \mathcal{F} that describes the amount of energy, mass or any other extensive property that has passed the control surface at z over the time period $[0, t]$. Then

$$\mathcal{F}(t, z) = \int_0^t A \dot{f} d\tau$$

where \dot{f} is the flux (amount per unit area and time) of F , and A is the cross-sectional area of the transport. The time derivative of \mathcal{F} is

$$\left(\frac{\partial \mathcal{F}}{\partial t}\right)_z = A \dot{f} \triangleq \dot{F}$$

So, in a sense \dot{F} is really a partial derivative, but it must be understood that \mathcal{F} has no explicit (and time independent) function expression like e.g. the thermodynamic and kinetic models we are using. Most students have problems in understanding the fundamental difference between dF/dt and $(\partial F / \partial t)$ and I therefore hesitate in calling \dot{F} a derivative because it will bring even more confusion into the subject.

It works because p (the pressure) is an intensive state variable which is independent of the magnitude of the volume flow. At the same time we want to integrate the total heat flux over the external surface of the reactor section

$$\dot{Q} = \int_z^{z+\Delta z} C\dot{q} d\zeta , \quad (2)$$

where C is the circumference of the reactor and \dot{q} is the heat flux (per unit time and surface area). Note that $d\zeta$ rather than dz is acting as an integrator for \dot{q} . We use this convention (Greek integrator—Latin variable) to make sure we do not mix up the integrator symbol with the symbol of either the upper or the lower limit of the integral². This makes the integral a function of z while ζ is consumed during the integration.

It is customary to neglect the heat flow in the axial direction which is why the integral is carried out over the outer surface only. However, strictly speaking there is an order-of-magnitude analysis missing here but this is left as an exercise for the reader. The internal energy of the control volume is then:

$$\mathcal{U}(t, z, \Delta z) = \int_0^t (\dot{H}_z - \dot{H}_{z+\Delta z}) d\tau + \int_0^t \int_z^{z+\Delta z} C\dot{q} d\zeta d\tau$$

This states the energy balance of a simple plug flow reactor. On the form given it is particularly useful for testing and verifying the accuracy of numerical integrators used in dynamic simulation studies, but this is not our goal. We shall proceed instead by calculating the partial derivative \mathcal{U} at a fixed spatial position z with respect to time:

$$\left(\frac{\partial \mathcal{U}}{\partial t} \right)_{z, \Delta z} = \dot{H}_z - \dot{H}_{z+\Delta z} + \int_z^{z+\Delta z} C\dot{q} d\zeta \quad (3)$$

On the current form Eq. 3 leads to a partial differential equation (PDE) in time and space which is considered to be a hard numerical task. But, there are relevant simplifications. In particular we shall study the behaviour of closed systems without throughput of mass and steady state (time independent) systems.

1.1 First law of thermodynamics

A special form of the energy balance applies to *closed systems*. Here, closed means $\dot{H}_z = \dot{H}_{z+\Delta z} = 0$. This appears to be outside the scope of our PFR model but it is still in reach of the thermodynamic formalism. In a system of this kind energy changes

²Dealing mostly with closed and definite integrals we may not even realise the problem, but as we move on to indefinite integrals (antiderivatives) the symbol clash becomes very noticeable. In thermodynamics we define for example the residual function $G^{r,p}(p) \doteq \int_0^p (V(\pi) - V^{ig}(\pi)) d\pi$ where π is an integrator (over pressure) and p is the system pressure. The convolution integral $F(t) = \int_0^t \varphi(\tau)\psi(t-\tau) d\tau$ used in signal theory is another example. The mutual roles of τ and t must here be sorted out beforehand.

solely because heat is expelled to, or brought in from, the environment. For the change of \mathcal{U} we can then write:

$$(\text{d}\mathcal{U})^{\text{c-s}} = \int_z^{z+\Delta z} C\dot{q} \text{d}\zeta \text{d}t$$

Backsubstitution of \dot{Q} from Eq. 2 yields the simpler form: $\text{d}\mathcal{U} = \dot{Q} \text{d}t$. A similar argument holds also for any kind of external work even though it by coincidence has been excluded in Eq. 3. The reason is that the PFR model is not subject to any volume change nor is it equipped with a mechanical stirrer. If we had decided to include external work (positive when work is delivered by the system) the energy equation would have been extended to $\text{d}\mathcal{U} = \dot{Q} \text{d}t - \dot{W} \text{d}t$.

Taken a bit further it customary to say that $\dot{Q} \text{d}t = \delta Q$ and $\dot{W} \text{d}t = \delta W$ where δQ and δW stand for the non-exact differentials of Q and W . Non-exact means that \mathcal{U} does not depend on Q and W in a definite way. I.e. there exists no function $\mathcal{U}(Q, W)$ such that when Q and W are given then \mathcal{U} is also given. This should be quite intuitive all the time \mathcal{U} is the energy of a material system where the masses of the chemical constituents must also play a role.

In fact, Q and W are path dependent functions of the thermodynamic state, and also of the spatial co-ordinates and of time. They are not state functions in any way and they do not constitute a part of the system. Rather, they express the transportation of energy across the system border. Inside the system, however, heat and work can only be stored as internal energy. There are in other words no “heat content” or “energy content” of the system, only the ability to exchange heat and work with the environment. We therefore talk about “heat potential” and “work potential” to stress the fact that energy (the thermodynamic potential) has to be converted back and forth between heat and work all the time.

Finally, before we leave the discussion of the closed system we shall make a precise interpretation of \mathcal{U} and U . It has already been stated that \mathcal{U} is a constructed energy function—a functional—that serves the need of an accumulation term in the energy equation. From the discussion given above it is clear that \mathcal{U} does not change in a closed system unless there is heat or work exchange with the environment. If there are no interactions of any kind, then all experiments made over the past 200 years indicate that \mathcal{U} gradually becomes undistinguishable from U . That is:

$$\mathcal{U}_{\text{eq}} \hat{=} \lim_{t \rightarrow \infty} \mathcal{U} \rightarrow U$$

The two functions \mathcal{U} and U are identical whenever their function values are the same over the entire definition domain³. In this case \mathcal{U} is constant throughout the experiment so how can it then become *gradually* undistinguishable from U ? The *experiment* tells us that \mathcal{U} does not change in a closed system over time. Our *postulate* says that \mathcal{U} is identical to U when all internal agitation and transients have died out. Before that the measurements of any intensive variables like temperature, pressure and chemical

³E.g. the two functions $f(x) = \cos^2(x) + \sin^2(x)$ and $g(x) = 1$ are mathematically identical for $x \in \mathbb{R}$.

potentials give unreliable readings even though the function values are the same at any time. It is only then all the readings are stable we can say that $\mathcal{U} \equiv U$ in the mathematical understanding of the statement. We call this the equilibrium state of the system. It has an incredible simple representation in the sense that only $n+2$ macroscopic variables are needed in order to establish the value of $U(S, V, N_1, N_2, \dots, N_n)$. From a microscopic point of view this is really *incredible* because there are $6N_A \sum_i N_i$ mechanical degrees of freedom when all the particles in the system are considered as a Newtonian universe. Thermodynamic systems are much simpler, however, because experimentally only the statistically most relevant state is being observed, and since thermodynamics *is* a phenomenological science the observations and theory go hand in hand. This means we can write the energy balance of a closed system as

$$(\text{d}U)^{\text{c-s}} = \delta Q - \delta W$$

which is precisely the first law of thermodynamics. The energy balance in Eq. 3 fulfills in other words the requirements of the first law of thermodynamics albeit in disguise. It must be understood, however, that the usability of $U = \mathcal{U}_{\text{eq}}$ hinges on the fact that the relaxation time of the equilibrium process must be smaller than the time scale of the simulation. This may, or may not, be the case, but for the present purpose we shall assume that U has the meaning of \mathcal{U} ; at least locally for each point in space—if not for the entire system.

1.2 Steady state solution

Eq. 3 has another special meaning whenever the physical situation is such that it allows the left hand side to be put to zero. It is the celebrated *steady state* which reduces the differential equation to a time-independent algebraic equation on the form:

$$(\dot{H}_{z+\Delta z} - \dot{H}_z)^{\text{s-s}} = \int_z^{z+\Delta z} C\dot{q} \text{d}\zeta$$

Despite its simple form the last equation has a wide range of applicability. It is valid for any type of fluid flow, inviscid or not, gas or liquid, one-phase or multi-phase, and with or without chemical reactions.

Just like the displacement work in Eq. 1 was factored into $p\dot{V}$, the transported enthalpy can be factored into the transported mass and a term called the *specific* enthalpy h :

$$\dot{H} = h\dot{M}$$

The inherent scaling properties, namely that $\dot{W} = p\dot{V}$ and $\dot{H} = h\dot{M}$, are deeply rooted in thermodynamic theory and are examples of the so-called Euler homogeneous functions. The energy balance is then reduced to:

$$(h\dot{M})_{z+\Delta z} - (h\dot{M})_z = \int_z^{z+\Delta z} C\dot{q} \text{d}\zeta$$

From the mass conservation principle we know that (for steady-state flow):

$$\dot{M}_{z+\Delta z} - \dot{M}_z = 0$$

Division by $\dot{M}_{z+\Delta z} = \dot{M}_z \triangleq \dot{M}$ on both sides of the equation yields:

$$h_{z+\Delta z} - h_z = \int_z^{z+\Delta z} C \frac{\dot{q}}{\dot{M}} d\zeta$$

In the limit of $\Delta z \rightarrow 0$ we get:

$$\lim_{\Delta z \rightarrow 0} (h_{z+\Delta z} - h_z) = C \frac{\dot{q}}{\dot{M}} \Delta z$$

or rearranged:

$$\lim_{\Delta z \rightarrow 0} \frac{h_{z+\Delta z} - h_z}{\Delta z} = C \frac{\dot{q}}{\dot{M}}$$

We immediately recognize the left hand side as the partial derivative of h with respect to z . On the right hand side we can make the definition $q \triangleq \dot{q}/\dot{M}$ standing for the specific heat load (energy per unit mass and area). The energy balance for a steady state reactor with only internal energy flow is then:

$$\left(\frac{\partial h}{\partial z} \right)^{\text{s-s}} = Cq$$

The anti-derivative of the energy balance defines the so-called enthalpy equation (please note the integral on the right side is zero for an adiabatic reactor without external heat load):

$$h(z) = h(0) + \int_0^z C(\zeta) q(\zeta) d\zeta$$

At this point we need to worry about the mathematical notation we are using. The operations are formally correct up to the point where $\Delta z \rightarrow 0$, but here it stops. At some finite value of Δz it becomes smaller than the resolution of the measurement. Or, it may in fact become smaller than the effective size of the molecules comprising the system and on this tiny scale h loses its meaning since it requires a big number of colliding molecules to establish a thermodynamic state variable. Hence, the derivative $(\partial h / \partial z)$ does not exist in proper. It is only the finite difference $h_{z+\Delta z} - h_z$ that is physically measureable, and then only if Δz is *sufficiently* large. This is not a practical problem in most cases, but for e.g. high-vacuum systems we must take precautions because the distance covered between two successive collisions of the molecules can be of the order millimeters or even centimeters.

Our second worry is that h is not a function of the spatial co-ordinate z . It is in fact a function of the state variables $T, v \triangleq V/M, c_1 \triangleq N_1/M, c_2 \triangleq N_2/M$, etc. when any of the modern pressure explicit equations of state are being used in the modelling (most

of them are descendants of the Van der Waals equation of state from 1873). Hence, $(\partial h/\partial z)$ does not exist other than as a formal expression, but from differential calculus we know that $\text{d}h/\text{d}z$ takes the same numerical value as $(\partial h/\partial z)$ when all the degrees of freedom except one (i.e. z) are locked. However, the total differential of h is

$$\begin{aligned}\text{d}h = & \left(\frac{\partial h}{\partial T} \right)_{v,c_1,c_2,\dots} \text{d}T + \left(\frac{\partial h}{\partial v} \right)_{T,c_1,c_2,\dots} \text{d}v \\ & + \left(\frac{\partial h}{\partial c_1} \right)_{T,v,c_2,c_3,\dots} \text{d}c_1 + \left(\frac{\partial h}{\partial c_2} \right)_{T,v,c_1,c_3,\dots} \text{d}c_2 + \dots\end{aligned}$$

or given a more compact form:

$$\text{d}h = \partial_T h \cdot \text{d}T + \partial_v h \cdot \text{d}v + \partial_{c_1} h \cdot \text{d}c_1 + \partial_{c_2} h \cdot \text{d}c_2 + \dots$$

Inventing a new notation “over the night” is not something I usually recommend, but we will run out of paper pretty soon unless we do something about the partial derivatives flourishing all over the place. Dividing by $\text{d}z$ (which is an algebraic quantity remember—and by the way quite different from ∂_z which is an operator) gives the differential quotient:

$$\begin{aligned}\left(\frac{\text{d}h}{\text{d}z} \right) = & \left(\frac{\partial h}{\partial T} \right)_{v,c_1,c_2,\dots} \left(\frac{\text{d}T}{\text{d}z} \right) + \left(\frac{\partial h}{\partial v} \right)_{T,c_1,c_2,\dots} \left(\frac{\text{d}v}{\text{d}z} \right) \\ & + \left(\frac{\partial h}{\partial c_1} \right)_{T,v,c_2,c_3,\dots} \left(\frac{\text{d}c_1}{\text{d}z} \right) + \left(\frac{\partial h}{\partial c_2} \right)_{T,v,c_1,c_3,\dots} \left(\frac{\text{d}c_2}{\text{d}z} \right) + \dots\end{aligned}$$

or, using our shorter notation:

$$\nabla h = \partial_T h \cdot \nabla T + \partial_v h \cdot \nabla v + \partial_{c_1} h \cdot \nabla c_1 + \partial_{c_2} h \cdot \nabla c_2 + \dots$$

This is precisely the expression we are looking for. The crux of the matter is that ∇h takes the same numerical value as $(\partial h/\partial z)$, but to carry on we need to first solve an equation system that settles the values of ∇T , ∇v , ∇c_1 , ∇c_2 , etc. This is done by simultaneously solving the energy, momentum and mass balances at the inlet of the reactor and integrating the solution variables along the spatial co-ordinate z . The how’s and why’s are fully explained in Part III of this paper entitled *Modelling Issues*. The implicitness of the conservation statement is so fundamental to the thermodynamist, however, that it is really deserves an introductory example. The internal workings of the so-called Jacobian transformation is explained below.

1.3 Calculation example

Doing matrix algebra by hand is hard work but there is no other way we can get an understanding of how the linearization really works. So, to gain the insight we shall practise on a minimalistic 2×2 example. Assume a problem on the form:

$$H^{\text{ig}}(T, V) \triangleq C_P^{\text{ig}} T = H_{\circ}$$

$$p^{\text{ig}}(T, V) \triangleq \frac{NRT}{V} = p_{\circ}$$

where N is constant, and H_\circ and p_\circ are conserved quantities. Let $\mathbf{x} \hat{=} (T \ V)$ and $\mathbf{y} \hat{=} (H \ p)$. To solve $\mathbf{y}(\mathbf{x}) = \mathbf{y}_\circ$ we first linearize $\mathbf{y}(\mathbf{x})$ and then attempt to solve the equations iteratively using the Newton–Raphson method:

$$\mathbf{y}_k + \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}^T} \right)_k (\mathbf{x}_{k+1} - \mathbf{x}_k) = \mathbf{y}_\circ$$

Rearrangement gives:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{J}_k^{-1}(\mathbf{y}_k - \mathbf{y}_\circ)$$

where

$$\mathbf{J}_k \hat{=} \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}^T} \right)_k = \begin{pmatrix} \left(\frac{\partial H}{\partial T} \right)_V & \left(\frac{\partial H}{\partial V} \right)_T \\ \left(\frac{\partial p}{\partial T} \right)_V & \left(\frac{\partial p}{\partial V} \right)_T \end{pmatrix}_k$$

so that:

$$\mathbf{J}_k^{-1} = \begin{pmatrix} C_P^{ig} & 0 \\ \frac{NR}{V} & -\frac{NRT}{V^2} \end{pmatrix}_k^{-1} = \frac{-1}{C_P^{ig} NRT} \begin{pmatrix} -\frac{NRT}{V^2} & 0 \\ -\frac{NR}{V} & C_P^{ig} \end{pmatrix}_k = \begin{pmatrix} \frac{1}{C_P^{ig}} & 0 \\ \frac{V}{C_P^{ig} T} & -\frac{V^2}{NRT} \end{pmatrix}_k$$

The remaining algebra is straightforward:

$$\begin{pmatrix} T \\ V \end{pmatrix}_{k+1} = \begin{pmatrix} T \\ V \end{pmatrix}_k - \begin{pmatrix} \frac{1}{C_P^{ig}} & 0 \\ \frac{V}{C_P^{ig} T} & -\frac{V^2}{NRT} \end{pmatrix}_k \left[\begin{pmatrix} C_P^{ig} T \\ \frac{NRT}{V} \end{pmatrix}_k - \begin{pmatrix} H \\ p \end{pmatrix}_\circ \right]$$

Iteration example: $H_\circ = 10^4 \text{ J}$, $p_\circ = 10^6 \text{ Pa}$, $N = 1 \text{ mol}$, $C_P^{ig} = \frac{5}{2}R$, $R = 8.3145 \text{ J mol}^{-1} \text{ K}^{-1}$.

k	$T [\text{K}]$	$V [\text{m}^3]$
0	298.15	0.001
1	481.087257201275	0.00221018092537634
2	481.087257201275	0.00319913692002833
3	481.087257201275	0.00383965458178457
4	481.087257201275	0.00399357233671433
6	481.087257201275	0.00399998967128617
7	481.087257201275	0.0039999999997333
8	481.087257201275	0.004

The Newton–Raphson iteration is a so-called second order method. One characteristic feature is that the number of significant digits will double in each iteration *sufficiently* close to the solution (iteration 3 onward). Verify this behaviour. From the table it is also clear that T converges in one step whilst V requires 8 iterations. Give a reason for this observation⁴. Finally, it should be mentioned that the Newton–Raphson method is sensitive to the starting values. E.g. try to start the iteration at $V = 0.01$ rather than $V = 0.001$. Suggest a possible fix to the algorithm in this case⁵.

⁴: $H(T, V)$ is strictly linear in both T and V

⁵: Unphysical volume update. Step length restriction is necessary.

1.4 Epilogue

I have in this little text sought to establish a fairly rigorous derivation of the energy balance for an idealized plug flow reactor. It is neither highly sophisticated nor does it require advanced mathematics. Still, it is not of a kind that is eagerly agreed upon by the chemical engineering community—be it professors, students or working professionals. Many people find the painstaking calculations of differentials and partial derivatives confusing and of little practical interest, but the latter is definitely wrong. The very fact that ∇T , ∇v and ∇c_i are solution variables of a set of model equation whereas $\partial_T h$, $\partial_v h$ and $\partial_{c_i} h$ are explicit (or sometimes implicit) state functions establishing the coefficient matrix of the model equations is so important that it can hardly be overemphasized.

The culprit in this controversy might be the teaching of $\text{d}y/\text{d}x = y'$ in highschool mathematics. By doing so the students learn that $\text{d}y/\text{d}x$ is synonymous with $y' \triangleq (\partial y / \partial x)$ and that the rest of the story is just syntactic sugar. For one-variable systems I can agree that the difference is subtle, but for many-variable systems it is not. The discussion has much in common with the use of substantial derivatives in fluid mechanics which says: $\text{d}y/\text{d}t = (\partial y / \partial t) + (\partial y / \partial x_1) \text{d}x_1/\text{d}t + (\partial y / \partial x_2) \text{d}x_2/\text{d}t + \dots$. In this case I think it can hardly be misunderstood that $\text{d}y/\text{d}t$ and $(\partial y / \partial t)$ are different mathematical objects—and very different ones as well.

Exercise 8

1 Question: Dynamics 03 - ODE integration

We want to write a integrator for solving ordinary differential equations. The idea is that one has a generic integrator, which one used to solve a user-defined system of ordinary differential equations.

$$\begin{aligned}\dot{\underline{x}} &= \underline{f}(\underline{x}, \underline{\theta}) \\ \underline{x}(t) &:= \int_0^t \dot{\underline{x}} d\tau\end{aligned}$$

There are many numerical methods known that can be utilised to construct a generic integrator module. The chosen structure should be such that the integrator can be re-used in terms of being able to combine it with any user-defined ODE system.

1. Euler method
2. Rung-Kutta method (RK4)

Given the differential equation:

$$\dot{x} = -0.1x; \quad x(0) = 10$$

1. Demonstrate the working code using a step size of 0.1 and 10 integrating from 0-50.
2. Compare solutions with the exact solution, meaning the values calculated from an analytical solution.

2 Question: Topology 05

The subject of the problem is the plant in Figure 1. It is a Liquid/liquid extraction process to separate the caffeine from coffee.

Tasks:

- Sketch the topology of the decaffeinated coffee plant.
- Provide a table which shows the existence of components in the capacities assuming ideal separation.

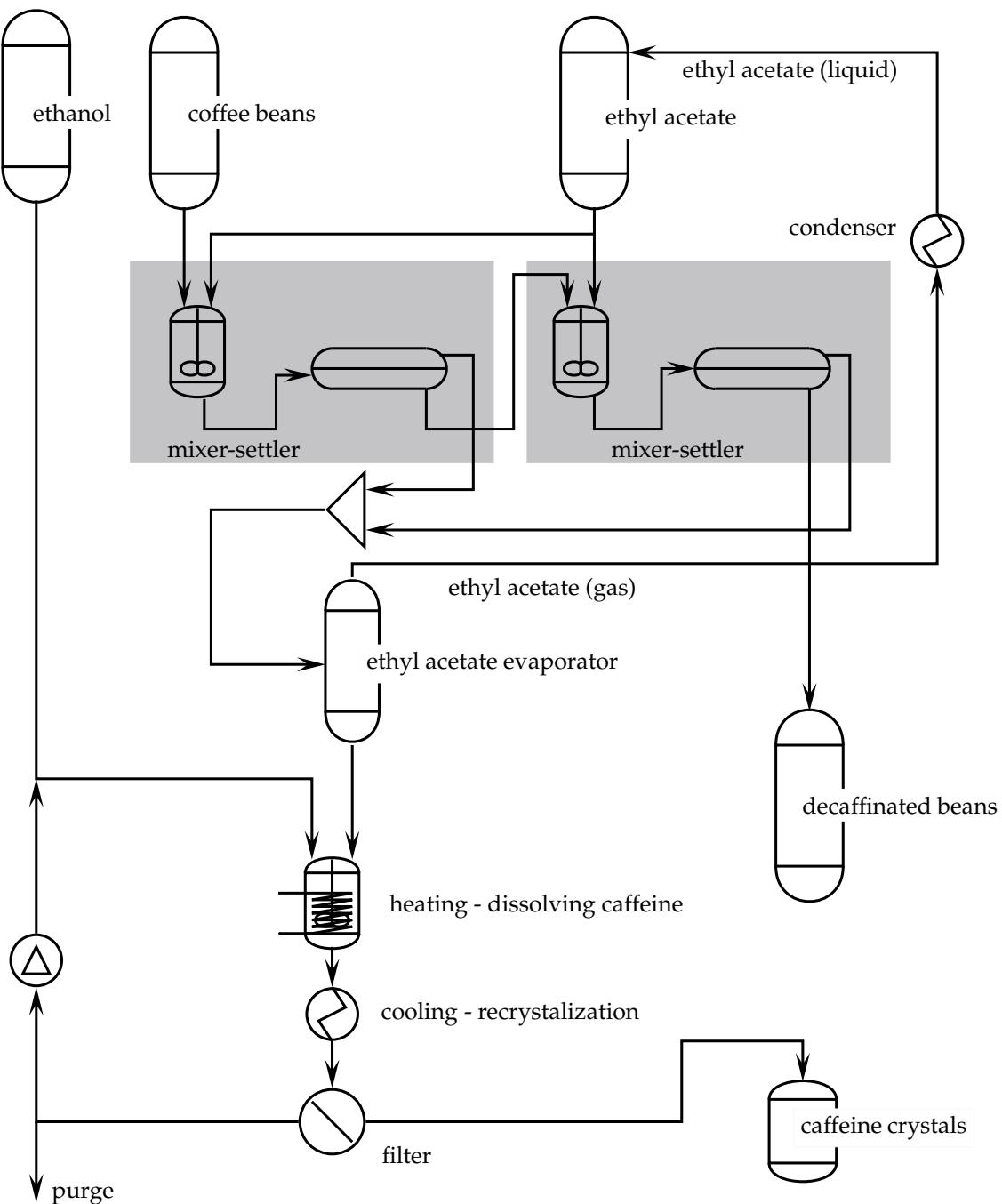


Figure 1: A Caffein Extraction plant

3 Question: Reactions 03

When CO_2 exists in natural gas prior to combustion, its presence has several disadvantages: the heating value is lowered, the gas volume to be handled and transported is increased and the CO_2 being a greenhouse gass is released into the atmosphere. Currently several technologies are pursued for separating CO_2 from natural gas. Here, we consider the process of carbon dioxide adsorption on activated carbon. The process is a fixed bed in a tube, with the fixed bed being the activated carbon in one or the other geometrical form (left in Figure 2). For the purpose of modelling we first assume two sep-

arate phases in a simplified geometry (middle in Figure 2), which we next further simplify to a series of stages in which the gas phase is well mixed and moves from stage to stage, whilst the solid phase is stagnant (right in Figure 2):

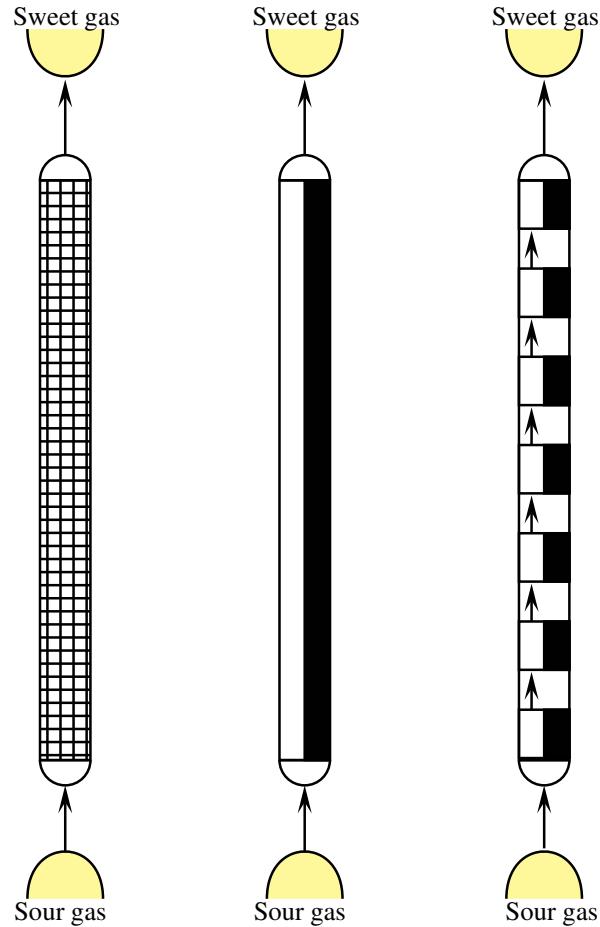


Figure 2: A simple topology of a CO_2 adsorption process

Thus we view the column as a stack of stages each of which we model as shown in Figure 3.

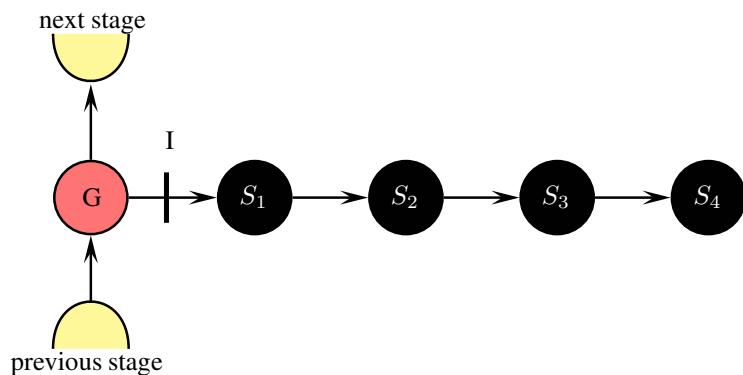
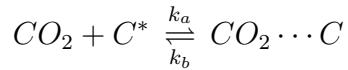


Figure 3: A simple topology of a CO_2 adsorption stage

The G represents the gas phase whilst the solid is compartmentalised in the horizontal

direction, here into 4 separate lumps. The adsorption reaction is



where C^* is the active carbon and the $CO_2 \cdots C$ is the CO_2 adsorbed on the C . The forward reaction is first order in CO_2 concentration and C^* concentration, whilst the backward reaction is first order in $CO_2 \cdots C$ concentration.

We assume that the transport in the solid is governed by a simple transfer law which is proportional to the negative discrete gradient of the composition of the two coupled capacities.

For the transport into the solid between G and S_1 we assume, that the transport in the gas phase is very fast compared to the transport into the solid. Consequently the composition on the surface I is the same as in the gas phase. For the transport into the solid then we need to make a correction due to the fact that we assume a pseudo-homogeneous phase for the solid with the diffusing gas.

$$\hat{n}_{I|S_1,CO_2} := -k_{I|S_1} (\epsilon^{-1} c_{S_1,CO_2} - c_{G,CO_2})$$

For the definition of the composition of the CO_2 in the solid phase and diffusing gas, we use a constant volume. All kinetic data are based on the pseudo phase, with ϵ accounting for the pseudo-phase porosity a measure for the ratio of free space and solid phase material. The individual volume piece would be the volume of the complete solid phase divided by the number of stages and divided by the number of solid lumps per stage.

Assume:

1. isothermal conditions
2. all properties that you require constant
3. all volumes to be constant

Tasks

1. Write the species balance for all lumps, that is G $S_i, i := 1, 2, 3, 4$
2. Add the transfer laws
3. Add the reactions
4. Fill in the relations linking the variables in the transfer law and reaction kinetics with the state.

1 Solution: Dynamics 03

$$\dot{x} = -0.1x$$

Euler method

The euler method is

$$x_{n+1} = x_n + h f(t_n, x_n)$$

First, $f(t_0, x_0)$ should be calculated. We have

$$f(t_0, x_0) = -1$$

Assuming the step size to b $h = 1$, we have,

$$x_1 = x_0 + h f(t_0, x_0) = 10 + (-1) = 9$$

The above steps should be repeated to find x_2, x_3, \dots

$$x_2 = x_1 + h f(t_1, x_1) = 9 + (-0.9) = 8.1$$

$$x_3 = x_2 + h f(t_2, x_2) = 8.1 + (-0.81) = 7.29$$

Rung kutta method

The RK4 method for this problem is given by the following equations:

$$x_{n+1} = x_n + \frac{1}{6} (k_1 + 2 k_2 + 2 k_3 + k_4)$$
$$t_{n+1} = t_n + h$$

where x_{n+1} is the RK4 approximation of $x(t_{n+1})$, and

$$k_1 = h f(t_n, x_n)$$
$$k_2 = h f\left(t_n + \frac{1}{2}h, x_n + \frac{1}{2}k_1\right)$$
$$k_3 = h f\left(t_n + \frac{1}{2}h, x_n + \frac{1}{2}k_2\right)$$
$$k_4 = h f\left(t_n + \frac{1}{2}h, x_n + k_3\right)$$

By substituting the initial value and iterating, the sequence of x is calculated

$$x = [10 \ 9.0484 \ 8.1873 \ 7.4082 \ 6.0653]$$

Analytical solution

By integrating both sides of the equation we will have By substituting the initial value and iterating, the sequence of x is calculated

$$\begin{aligned}\int_{x_0}^x \dot{x} &= \int_0^t -0.1 x \\ \int_{x_0}^x \frac{dx}{x(t)} &= \int_0^t -0.1 dt \\ x(t) &:= e^{-0.1 t} x_0\end{aligned}$$

1.1 Matlab Code

The matlab code consist of a suite of functions:

- Main program
- Integrator, which goes over the time interval in steps
- Euler step: executes one R-K step
- Runge-Kutta step: executes one R-K step
- ODEs

1.1.1 Launcher

```
1 %  
2 % test problem for Runge Kutta 4 and Euler  
3 %  
4 % 2012-10-23 Preisig, H A  
5 %%  
6  
7 t_end = 10;  
8 step = 0.1;  
9 x0 = 10;  
10 par = -0.1;  
11  
12 [t, x] = Integrator(@RK4_Step, @TestModel, t_end, step, x0, par);  
13  
14 figure(1)  
15 plot(t, x, 'b')  
16 hold on  
17  
18 %%  
19 [t, x] = Integrator(@Euler_Step, @TestModel, t_end, step, x0, par);  
20  
21 plot(t, x, 'r')  
22  
23 %%  
24 plot(t, x0*exp(-0.1*t), 'xk')
```

```

26 %%%
27
28 legend('Runge-Kutta', 'Euler', 'Exact')
29 title('Test of Euler and Runge-Kutta integration on dx=-0.1*x')
30 xlabel('t')
31 ylabel('x')
32 hold off

```

1.1.2 Integrator

```

1 %
2 % implements the integration of a system of differential equations
3 % using a specified integration method.
4 %
5 % 2012-10-23 Preisig, H A
6 %%
7
8 function [t, x] = Integrator(INTEGRATOR, ODE, t_end, step, x0, par)
9 %
10 % ODE :: rhs of the ode's as function of t and x
11 % t_end :: end time, starting time is always 0
12 % step :: step size
13 % x0 :: initial conditions
14 % par :: parameters
15 % x :: trajectory
16
17 n = t_end/step;
18 x = zeros(length(x0), n);
19 t=x;
20 x(:,1) = x0;
21
22 for i = 2:n
23     t(i) = t(i-1) + step;
24     x(:, i) = feval(INTEGRATOR, ODE, t(i), x(:, i-1), step, par);
25 end

```

1.1.3 Runge-Kutta step

```

1 %
2 % Runge Kutta 4 step
3 %
4 % 2012-10-23 Preisig, H A
5 %%
6
7
8 function x_next = RK4_Step(ODE, t, x, h, par)
9
10 % t :: current time k'th step
11 % x :: current x so x(k)
12 % x_next :: next x, so x(k+1)
13 % t :: current time
14 % h :: step size
15 % par :: parameters
16
17 k1 = feval(ODE, t, x, par); % slope at the beginning of interval
18 k2 = feval(ODE, t + 0.5*h, x + 0.5*k1, par); % slope at midpoint using k1
19 k3 = feval(ODE, t + 0.5*h, x + 0.5*k2, par); % slope at midpoint using k2
20 k4 = feval(ODE, t + h, x + k3, par); % slope at end
21 x_next = x + (h/6)*(k1 + 2*k2 + 2*k3 + k4); % make step

```

1.1.4 Euler step

```

1  %
2  % Euler step
3  %
4  % 2012-10-23 Preisig, H A
5  %%
6
7
8 function x_next = Euler_Step(ODE, t, x, h, par)
9
10 % t      :: current time k'th step
11 % x      :: current x so x(k)
12 % x_next :: next x, so x(k+1)
13 % t      :: current time
14 % h      :: step size
15 % par    :: parameters
16
17 k1 = feval(ODE, t, x, par);           % slope at the beginning of interval
18 x_next = x + h * k1;                 % make step

```

1.1.5 Test model

```

1 %
2 % test problem dx = -0.1*x
3 %
4 % 2012-10-23 Preisig, H A
5
6
7 %%
8 function dx = TestModel(t, x, par)
9
10 dx = par(1) * x;

```

Figure 1 shows the analytical solution and the approximations from Euler and RK4 methods.

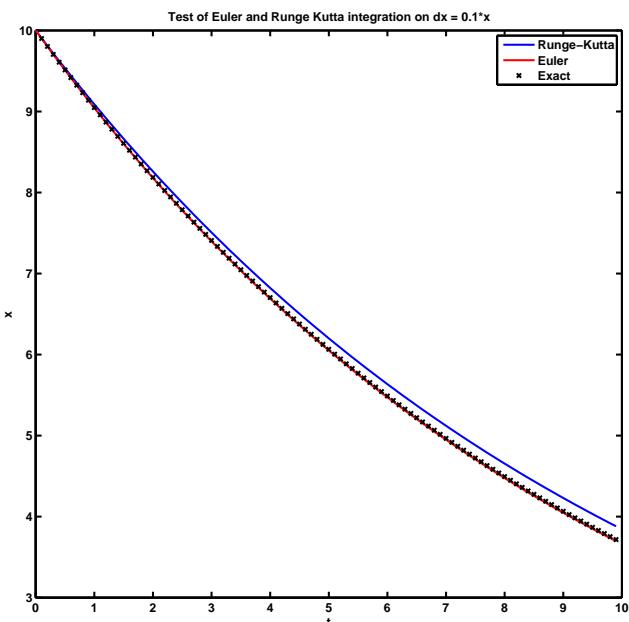


Figure 1: The matlab graphics for the test problem

1.2 Python code

The Python code has two pieces: The first implements a class of integrators, which are plugged into the second module, which contains the whole problem, integrator and differential equations.

Note that the model can be subclassed defining another model as demonstrated in the main program section of the module.

1.2.1 Integrators

```
1  """
2  2012-10-11 created
3  2012-10-13 step returns dt, dxdt and x_new
4
5  @author: Preisig, Heinz A
6  @organization: NTNU, Chemical Engineering
7  """
8
9  class Stepper(object):
10     """
11         Implements different "steppers" for computing ODE integrals.
12     """
13
14     def __init__(self, model, dt):
15         """
16             Keeps the model and the (initial) time step.
17         """
18         self.model = model
19         self.dt = dt
20
21     class Euler(Stepper):
22         """
23             Implements the Euler method.
24         """
25
26         def step(self, t, x):
27             dxdt = self.model.rhs(t, x)
28             x_new = x + dxdt * self.dt;
29             return self.dt, dxdt, x_new
30
31     class RungeKutta(Stepper):
32         """
33             Implements the Runge-Kutta 3:4 method.
34             Pending...
35         """
36
37         def step(self, t, x):
38             h = self.dt
39             k1 = h * self.model.rhs(t, x)
40             k2 = h * self.model.rhs(t + 0.5 * h, x + 0.5 * k1)
41             k3 = h * self.model.rhs(t + 0.5 * h, x + 0.5 * k2)
42             k4 = h * self.model.rhs(t + h, x + k3)
43             x_new = x + (1.0 / 6.0) * (k1 + 2.0 * k2 + 2.0 * k3 + k4)
44             dxdt = (x_new - x) / h
45             return self.dt, dxdt, x_new
```

1.2.2 The model module

```
1 | """
```

```

2 In this module, the term "model" is used for the complete "thing" including the
3 differential equation *and* the integrator.
4
5 2012-10-11 created (HAP).
6 2012-10-13 extended to multiple integrators and demo for re-use (HAP).
7 2012-10-16 replaced an ugly if-elif-else testing of the ode solver method with a
8     first class function object (THW).
9
10 2012-10-11 created
11 2012-10-13 extended to multiple integrators and demo for re-use
12
13 @author: Preisig, Heinz A
14 @organization: NTNU, Chemical Engineering
15 ''
16
17 import Integrator_04 as ODE
18 import math
19 import gnuplot
20
21 class Model:
22     '''
23         The model is the differential equations and the integrator
24     '''
25
26     def __init__(self, x0=0.0, dt=1.0, par=[], odesolver=ODE.Euler):
27         '''
28             The initialization gets the initial conditions, the parameters, and the
29             step size for the time stepper - being an integrator.
30
31             @param x0: initial conditions
32             @param dt: time step
33             @param par: list of parameters
34         '''
35         self.t = [0]                                     # keeps the time
36         self.dxdt = []                                  # the time derivative of the state
37         self.x = [x0]                                    # keeps the state over time
38         self.par = par
39
40         # plug in desired integrator
41         self.integrator = odesolver(self, dt)          # plug in the integrator
42
43     def rhs(self, t, x):
44         '''
45             <USER SPECIFIC>
46
47             Returns the value of the time derivative of the current state and stores
48             the newly calculated derivative.
49         '''
50         dxdt = self.par[0] * x
51         return dxdt
52
53     def integrateTimeInterval(self):
54         '''
55             Integrate over the given time interval; thus updating time *and* state.
56         '''
57         x = self.x[-1]
58         t = self.t[-1]
59         dt, dxdt, x_new = self.integrator.step(t, x)
60         self.dxdt.append(dxdt)
61         self.x.append(x_new)
62         self.t.append(t + dt)
63
64     class MyModel(Model):
65         def rhs(self, t, x):
66             dxdt = self.par[0] * x

```

```

67     return dxdt
68
69 if __name__ == '__main__':
70
71     #prepare for plotting
72     plot = gnuplot.gnuplot(xlabel='t', ylabel='x', xmin=0, xmax=5, ymin=0, ymax=10, \
73                             title='integration two exponential using Euler and R-K, exact(+')
74
75     # instantiate models
76     x0 = 10
77     dt = 0.1
78     par1 = [-2]
79     par2 = [-1]
80
81     # default model with Euler
82     m = Model(x0=x0, dt=dt, par=par1, odesolver=ODE.Euler) # initialise object
83     for i in range(0, 5):                                     # step through the time interval
84         m.integrateTimeInterval()
85
86     plot.addlist(m.x, color='blue')    # add to plot
87     print '\nDefault model results (Euler):', m.x
88
89     # default model with Runge Kutta
90     m = Model(x0=x0, dt=dt, par=par1, odesolver=ODE.RungeKutta) # initialise object
91     for i in range(0, 5):                                     # step through the time interval
92         m.integrateTimeInterval()
93
94     plot.addlist(m.x, color='green')    # add to plot
95     print '\nDefault model results (Runge-Kutta):', m.x
96
97
98     # my model Euler
99     m = MyModel(x0=x0, dt=dt, par=par2)                      # initialise object
100    for i in range(0, 5):                                     # step through the time interval
101        m.integrateTimeInterval()
102
103    plot.addlist(m.x, color='blue')    # add to plot
104    print 'My model results (Euler):', m.x
105
106    # my model Runge Kutta
107    m = MyModel(x0=x0, dt=dt, par=par2, odesolver=ODE.RungeKutta)
# initialise object
108    for i in range(0, 5):                                     # step through the time interval
109        m.integrateTimeInterval()
110
111    plot.addlist(m.x, color='green')    # add to plot
112    print 'My model results (Runge-Kutta):', m.x
113
114
115 # comparison with exact solution
116
117     t = 0
118     x_exact = []
119     for i in range(0, 5):
120         t = i * dt
121         x_exact.append(math.e ** (par1[0] * t) * 10)
122
123     print 'Exact value first model:', x_exact
124     plot.addlist(x_exact, color='red', style='points')      # add to plot
125
126     t = 0
127     x_exact = []
128     for i in range(0, 5):
129         t = i * dt
130         x_exact.append(math.e ** (par2[0] * t) * 10)

```

```
131
132 |     print 'Exact value first model: ', x_exact
133 |     plot.addlist(x_exact, color='red', style='points')      # add to plot
134
135 | #plotting
136 |     plot.plot('model_04')
```

2 Suggested solution: Decaffination plant

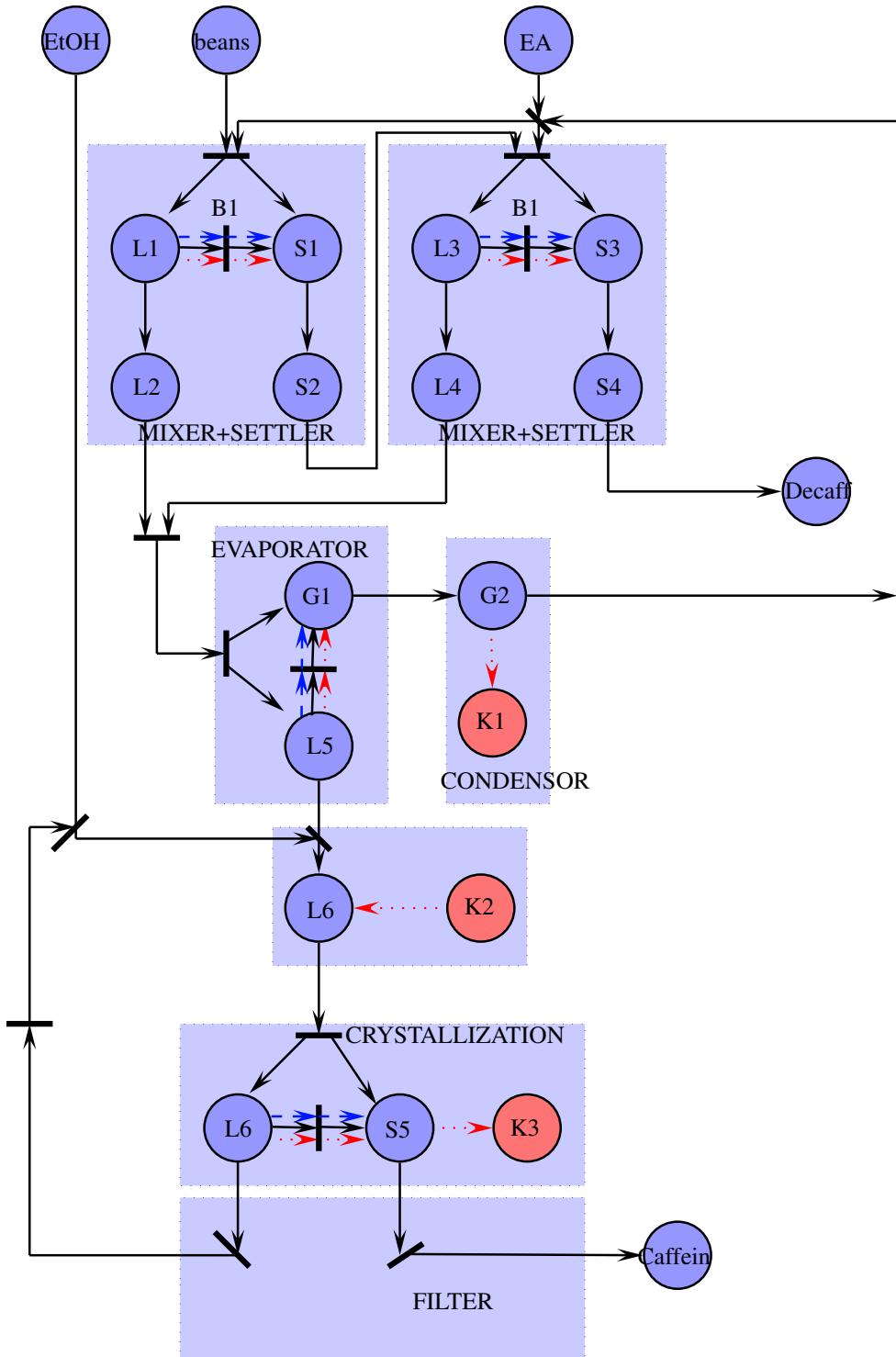


Figure 2: Topology of a Caffeine Extraction plant

3 Solution: Reaction 03 - CO_2 adsorption

1. Writing the balance for each species we will have:

$$\begin{aligned}\dot{\underline{n}}_G &:= \hat{\underline{n}}_f - \hat{\underline{n}}_p - \hat{\underline{n}}_{G|I} \\ \dot{\underline{n}}_i &:= \hat{\underline{n}}_a - \hat{\underline{n}}_b + \tilde{\underline{n}}_i\end{aligned}$$

where,

$$\begin{aligned}i &:= S_1, S_2, S_3, S_4 \\ a|b &\in I|1, 1|2, 2|3, 3|4\end{aligned}$$

2. Transport: The flow in the gas phase is pressure driven from stage to stage:

$$\begin{aligned}\hat{\underline{n}}_g &:= \underline{c}_g \hat{V}_g \quad ; \quad g \in [f, p] \\ \hat{V}_g &:= -k_g \begin{cases} (p_G - p_{G-1}); & g := f \\ (p_{G+1} - p_G); & g := p \end{cases}\end{aligned}$$

The diffusion into the solid, the activated carbon, we model with a transfer law that is linear in the discrete concentration gradient, though with a correction, as we model the solid, which is porous as a pseudo-homogeneous system. Thus

$$\hat{n}_{I|S_1, CO_2} := -k_{I|S_1} (\epsilon^{-1} c_{S_1, CO_2} - c_{G, CO_2})$$

For the transport inside the solid we have:

$$\hat{n}_{S_i|S_{i+1}, CO_2} := -k_{S_i|S_{i+1}} (c_{S_{i+1}, CO_2} - c_{S_i, CO_2})$$

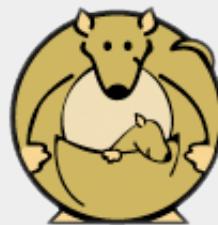
3. The production rate is:

$$\begin{aligned}\tilde{\underline{n}}_i &:= V \underline{N}^T \tilde{\xi} \\ \text{change of extent of reaction} \quad \tilde{\xi} &:= \underline{\underline{K}} \underline{g}(\underline{c}) \\ \text{reaction constant matrix} \quad \underline{\underline{K}} &:= \text{diag} [k_f \quad k_b] \\ \text{reaction constant matrix} \quad \underline{\underline{N}} &:= \begin{bmatrix} \nu_f \\ \nu_b \end{bmatrix} \\ \text{dependency on the concentrations} \quad \begin{bmatrix} g_f(\underline{c}) \\ g_b(\underline{c}) \end{bmatrix} &:= \begin{bmatrix} c_{S_i, CO_2} \quad c_{S_i, C^*} \\ c_{S_i, CO_2 \dots C} \end{bmatrix}\end{aligned}$$

4. For the definition of the composition of the CO_2 in the solid phase and diffusing gas, we use a constant volume. All kinetic data are based on the pseudo phase. The individual volume piece would be the volume of the complete solid phase divided by the number of stages and divided by the number of solid lumps per stage.

$$\begin{aligned}\underline{c}_G &:= V_G^{-1} \underline{n}_G \\ \underline{p}_G &:= V_G^{-1} \underline{n}_G R T \\ T &:: \text{given} \\ V_G &:: \text{given} \\ c_{S_i, CO_2} &:= V_{S_i}^{-1} n_{S_i, CO_2} \\ V_{S_i} &:: \text{given}\end{aligned}$$

The Plug Flow Reactor (TKP4106)



Zooball/Kangaroo

At a computer expo (COMDEX), Bill Gates reportedly compared the computer industry with the auto industry and stated: "If GM had kept up with the technology like the computer industry has, we would all be driving \$25.00 cars that got 1,000 miles to the gallon." In response to Bill's comments, General Motors issued a press release (by Mr. Welch himself) stating: If GM had developed technology like Microsoft, we would all be driving cars with the following characteristics:

- For no reason at all, your car would crash twice a day.
- Every time they repainted the lines on the road, you would have to buy a new car.
- Only one person at a time could use the car, unless you bought Car95 or CarNT, and then added more seats.
- Apple would make a car powered by the sun, reliable, five times as fast, and twice as easy to drive, but would run on only five per cent of the roads.
- The airbag would say 'Are you sure?' before going off.
- Occasionally, for no reason, your car would lock you out and refuse to let you in until you simultaneously lifted the door handle, turned the key, and grabbed the radio antenna.
- You would press the start button to shut off the engine.
- ...

[General Motors vs. Bill Gates](#)

Assignments

1. Download the thermodynamics module [srk_ammonia.py](#).
2. Download the flowsheet module [flowsheet.py](#).
3.
 - a. Download the ammonia reactor module [ammonia_reactor.py](#).
 - b. Beware the integrating namespace [tkp4106.py](#).
 - c. Finish the initialization of $p(V) = p_0$ in Section 2.
 - d. Make sure the equation is solved correctly.
4. Run [ammonia_reactor.py](#) from the command line:

```
python ammonia_reactor.py rk2 explicit 12 1
until you hit an error in the integration method
hpn_vs_tvn_integrator(), confer Section 6 in that file. You may have
a problem getting past feed.get_cfw() in Section 3. That is probably
because you have not implemented tkp4106.molecular_weight()
```

which is referenced in [srk ammonia.py](#). Fix this problem by hard-coding the missing values in-place.

Start reading about [Modelling issues](#) to understand the three physical principles (energy, momentum, mass) that lie behind the Plug-Flow-Reactor model, and also the meaning of linearization.

%Predefined.

HTML text.

5.17.1 Verbatim: “srk_ammonia.py”

```
1  """
2  @summary:      Mock-up thermodynamic class for ammonia reactor calculations.
3  Based on ideal gas as a function of T, V, n i.e. *not* T, p, n.
4  Pressure has therefore to be iterated if necessary. This is part
5  of the training of our students though...
6  @author:       Tore Haug-Warberg
7  @organization: Department of Chemical Engineering, NTNU, Norway
8  @contact:      haugwarb@nt.ntnu.no
9  @license:      GPLv3
10 @requires:     Python 2.3.5 or higher
11 @since:        2011.10.04 (THW)
12 @version:      0.6
13 @todo 1.0:
14 @change:       started (2011.10.04)
15 @note:         Bla-bla.
16 """
17
18 class Model:
19     '''Ideal gas implemented on the form of Helmholtz energy A(T, V, nvec).'''
20     def __init__(self, args):
21
22         # Turn component names into lower case before any comparisons are made.
23         args = [arg.lower() for arg in args]
24
25         # from string import lower                                # alternative conversion
26         # args = map(lower, args)                                 # alternative conversion
27
28         # The model component list is hard-coded. This may change in the future,
29         # but so far we must live with the hack.
30         import tkp4016
31
32         # Molecular weight [10^10 g/mol].
33         mw = lambda str: \
34             [1.0e-10*n/m for (n, m) in [tkp4016.molecular_weight(str)]].pop()
35
36         cfw = [('ammonia', 'NH3', mw('NH3')), \
37                 ('nitrogen', 'N2', mw('N2')), \
38                 ('hydrogen', 'H2', mw('H2')), \
39                 ('argon', 'Ar', mw('Ar')), \
40                 ('methane', 'CH4', mw('CH4'))]
41
42         tmp = [c for (c, f, w) in cfw]
43
44         # Check that given components are in range of model.
45         for arg in args:
46             if not arg in tmp:
47                 raise SyntaxError("unknown component '%s'" % (arg,))
48
49         tmp = [c for (c, f, w) in cfw if c in args]
50
51         # Check that given components are in correct order.
52         if not tmp == args:
53             print 'Warning: component list: %s\n' \
54                 'reordered to: %s' % (args, tmp)
55
56         # Select values from list 'v' being True in list 'b'.
57         compact = lambda b, v: [vi for (bi, vi) in zip(b, v) if bi]
58
59         # Make Boolean flags (True | False) for extraction of data.
60         flags = [c in args for (c, f, w) in cfw]
61         self.cfw = compact(flags, cfw) # extract (component name, formula, mw)s
62         nc = len(self.cfw)           # number of chemical components in mixture
63
64         # Enthalpies of formation and standard entropies from DIPPR (1996) data-
65         # base. Heat capacity parameters from Reid, Poling and Prausnitz (1987)
66         # book. These are the data needed for calculating other state variables.
67         # The units are:
68         #   temperature [kK]
69         #   pressure    [kbar]
```

```

70     #   volume      [dm3]
71     #   mole number [mol]
72     #   energy      [10^5 J]
73     #   mass        [10^10 g]
74     #   time         [s]
75     # The reason for these odd choices is numerical stability.
76     #
77     self.dict = {\
78       'fix_rgash':0.083145119843087,
79       'var_t':0.29815,
80       'var_v':0.001,
81       'var_n':[1.0]*nc,
82       'par_h0':compact(flags, [-.45898, 0.00000, 0.00000, 0.00000, -.74520]),
83       'par_s0':compact(flags, [1.92660, 1.91500, 1.30571, 1.54737, 1.86270]),
84       'par_c1_cp':compact(flags, [0.2731, 0.3115, 0.27140, 0.20786, 0.01925]),
85       'par_c2_cp':compact(flags, [0.2383, -.1357, 0.09274, 0.00000, 0.52130]),
86       'par_c3_cp':compact(flags, [0.1707, 0.2680, -.13810, 0.00000, 0.11970]),
87       'par_c4_cp':compact(flags, [-.1185, -.1168, 0.07645, 0.00000, -.11320])
88     }
89
90     # Run self.__call__() to calculate derived 'state' properties.
91     self()
92
93     def __call__(self, **args):
94       for (k, v) in args.iteritems():
95         self.dict[k] = v                      # store input arguments (if any)
96
97       t = self.dict['var_t']
98       v = self.dict['var_v']
99       n = self.dict['var_n']
100      r = self.dict['fix_rgash']
101
102      if t<0 or v<0 or min(n)<0:
103        return False
104
105      self.dict['state_t'] = t
106      self.dict['state_v'] = v
107      self.dict['state_n'] = n
108
109      ntot = sum(n)
110      mtot = sum([ni*wi for (ni, wi) \
111                  in zip(n, [w for (c, f, w) in self.cfw])])
112
113      self.dict['state_ntot'] = ntot           # total number of moles [mol]
114      self.dict['state_mtot'] = mtot            # total mass [10^10 g]
115
116      nc = len(n)
117      eye = [int(i==j) for i in xrange(0,nc) for j in xrange(0,nc)]
118
119      self.dict['state_t_t'] = 1.0
120      self.dict['state_t_v'] = 0.0
121      self.dict['state_t_n'] = [0.0]*nc
122      self.dict['state_v_t'] = 1.0
123      self.dict['state_v_v'] = 0.0
124      self.dict['state_v_n'] = [0.0]*nc
125      self.dict['state_n_t'] = [0.0]*nc
126      self.dict['state_n_v'] = [0.0]*nc
127      self.dict['state_n_n'] = eye
128
129      t0 = 0.29815                         # reference temperature [kK]
130      p0 = 0.00101325                      # standard state pressure [kbar]
131
132      self.dict['state_p'] = ntot*r*t/v
133      self.dict['state_p_t'] = ntot*r/v
134      self.dict['state_p_v'] = -ntot*r*t/v**2
135      self.dict['state_p_n'] = [r*t/v]*nc
136      self.dict['state_h'] = 0.0
137      self.dict['state_h_t'] = 0.0
138      self.dict['state_h_v'] = 0.0
139      self.dict['state_h_n'] = [0.0]*nc
140      self.dict['state_mu'] = [0.0]*nc
141      self.dict['state_mu0'] = [0.0]*nc

```

```

142
143     import math
144
145     for i in xrange(0, nc):
146         hti = self.dict['par_h0'][i] + \
147             self.dict['par_c1_cp'][i]*(t-t0) + \
148             self.dict['par_c2_cp'][i]**(t**2-t0**2)/2.0 + \
149             self.dict['par_c3_cp'][i]**(t**3-t0**3)/3.0 + \
150             self.dict['par_c4_cp'][i]**(t**4-t0**4)/4.0
151         cpi = self.dict['par_c1_cp'][i] + \
152             self.dict['par_c2_cp'][i]*t + \
153             self.dict['par_c3_cp'][i]*t**2 + \
154             self.dict['par_c4_cp'][i]*t**3
155         sti = self.dict['par_s0'][i] + \
156             self.dict['par_c1_cp'][i]*math.log(t/t0) + \
157             self.dict['par_c2_cp'][i]*(t-t0) + \
158             self.dict['par_c3_cp'][i]**(t**2-t0**2)/2.0 + \
159             self.dict['par_c4_cp'][i]**(t**3-t0**3)/3.0
160         self.dict['state_h'] += hti*n[i]
161         self.dict['state_h_t'] += cpi*n[i]
162         self.dict['state_h_n'][i] = hti
163         self.dict['state_mu'][i] = hti - t*sti + r*t*math.log(n[i]*r*t/v/p0)
164         self.dict['state_mu0'][i] = hti - t*sti
165
166     return True
167
168     def __getitem__(self, key):
169         return self.dict[key]
170
171     def __setitem__(self, key, val):
172         self.dict[key] = val
173         return None
174
175     def __str__():
176         return 'T=%8.6f; p=%8.6f; H=%8.5f; V=%8.6f' % \
177             (self.dict['state_t'],
178              self.dict['state_p'],
179              self.dict['state_h'],
180              self.dict['state_v'])
181
182     def get_cfw(self):
183         return self.cfw

```

5.17.2 Verbatim: “flowsheet.py”

```
1  """
2  @summary: Flowsheet module. UnitParentClass is an 'abstract' class used for
3      implementing features that are common to all unit operations (so far
4      Stream and Reactor). Common features are (in regular Python syntax):::
5
6      obj['variable_name']          # __getitem__('variable_name')
7      obj['variable_name'] = value # __setitem__('variable_name', value)
8      obj()                      # __call__()
9      print obj                  # __str__()
10     obj.component_list()       # [(name, formula), ...]
11     obj.connect(another_obj)    # obj[var_t] = another_obj[var_t], ...
12     obj.functor(name, fun, args) # obj.name(*args) => fun(z, *args)
13
14     The module also contains a collection of functions for calculating the
15     pressure drop, heat exchange, kinetics, Jacobian matrix, etc. of a
16     unit operation object.
17
18     @author:      Tore Haug-Warberg
19     @organization: Department of Chemical Engineering, NTNU, Norway
20     @contact:    haugwarb@nt.ntnu.no
21     @license:    GPLv3
22     @requires:   Python 2.3.5 or higher
23     @since:      2011.10.04 (THW)
24     @version:    0.5
25     @todo 1.0:   Finish methods arrhenius(), tubeandshell()
26     @change:     started (2011.10.04)
27     @note:
28 """
29
30 import srk_ammonia
31 import math
32
33 # Unit operation parent class. It should have been an abstract class (that is a
34 # class without a constructor), but this is not straightforward in Python. Note
35 # that 'UnitParentClass' represents a thermodynamic state object, it is *NOT* a
36 # flow object since there is no concept of time here.
37 class UnitParentClass:
38     '''Base class for unit operation objects.'''
39     def __init__(self, tag, module, component_list):
40         self.model = module.Model(component_list)
41         self.tag = tag
42         self.module = module
43         self.functors = {}
44
45     def __getitem__(self, key):
46         return self.model[key]
47
48     def __setitem__(self, key, val):
49         self.model[key] = val
50         return None
51
52     def __call__(self, **args):
53         return self.model(**args)
54
55     def __str__(self):
56         return ":" + self.tag + "; " + str(self.model)
57
58     def get_cfw(self):
59         return self.model.get_cfw()
60
61     def get_module(self):
62         return self.module
63
64     def connect(self, arg):
65         self.model['var_t'] = arg.model['var_t']
66         self.model['var_v'] = arg.model['var_v']
67         self.model['var_n'] = arg.model['var_n']
68         self.model()
69         for (name,fun) in arg.functors.iteritems():
```

```

70         setattr(self.__class__, name, fun)
71
72     def functor(self, *args):
73         fun = lambda self, x=None: args[1](self, x, *args[2])
74         setattr(self.__class__, args[0], fun)
75         self.functors[args[0]] = fun
76         return self
77
78     def duplicate(self, tag, arg={}):
79         component_list = [name for (name, formula, mw) in self.get_cfw()]
80         module = self.get_module()
81         obj = self.__class__(tag, module, component_list)
82         obj.connect(self)
83         return obj
84
85
86     # Derived process Stream class.
87     class Stream(UnitParentClass):
88         '''Syntactic sugar.'''
89         pass
90
91     # Derived chemical Reactor class.
92     class Reactor(UnitParentClass):
93         '''Syntactic sugar.'''
94         pass
95
96     # Global functions used in reactor simulation. Connect to UnitParentClass object
97     # using so-called 'lambda'-functions, see method 'functor()' in this file.
98     def constantpdrop(obj, z, dp):
99         """
100         Constant pressure drop (dp/dz = constant) along the unit.
101         @param obj: unit operation object
102         @param z: axial position
103         @param dp: pressure drop [kbar] per reactor length
104         @type obj: aUnitParentClass
105         @type z: aFloat
106         @type dp: aFloat
107         @return: aFloat
108         """
109         return dp
110
111     def constantcooling(obj, z, duty):
112         """
113         Constant heat transfer (dQ/dz = constant) along the unit.
114         @param obj: unit operation object
115         @param z: axial position
116         @param duty: heat transfer [1.0e5 J] per reactor length
117         @type obj: aUnitParentClass
118         @type z: aFloat
119         @type duty: aFloat
120         @return: aFloat
121         """
122         return duty
123
124     def tubeandshell(obj, z, ua, t0):
125         """
126         Heat transfer calculation for a 'tube-and-shell' heat exchanger.
127         @param obj: unit operation object
128         @param z: axial position
129         @type obj: aUnitParentClass
130         @type z: aFloat
131         @return: aFloat
132         """
133         return ua*(t0 - obj['state_t'])
134
135     def constanrate(obj, z, nmat, k):
136         """
137         Constant reaction rate (r = constant) along the unit.
138         @param obj: unit operation object
139         @param z: axial position
140         @param nmat: reaction stoichiometry matrix
141         @param k: extent of reactions (one for each column of nmat)

```

```

142     @type obj: aUnitParentClass
143     @type z: aFloat
144     @type nmat: aList [ aList [ aFloat, aFloat, ... ] ]
145     @type k: aList [ aFloat, aFloat, ... ]
146     @return: aList [ aFloat, aFloat, ... ]
147     """
148     return [sum([nui*ki for (nui, ki) in zip(nu, k)]) for nu in nmat]
149
150 def firstorder(obj, z, nmat, keyc, k):
151     """
152     First order kinetics with respect to given 'key' components.
153     @param obj: unit operation object
154     @param z: axial position
155     @param nmat: reaction stoichiometry matrix
156     @param keyc: key components (one for each column of nmat)
157     @param k: rate constants (one for each column of nmat)
158     @type obj: aUnitParentClass
159     @type z: aFloat
160     @type nmat: aList [ aList [ aFloat, aFloat, ... ] ]
161     @type keyc: aList [ anInt, anInt, ... ]
162     @type k: aList [ aFloat, aFloat, ... ]
163     @return: aList [ aFloat, aFloat, ... ]
164     """
165     return [\n
166         sum([nui*obj['state_n'][ci]*ki for (nui, ci, ki) in zip(nu, keyc, k)]) \n
167         for nu in nmat\n
168     ]
169
170 def arrhenius(obj, z, nmat, keyc, k, a, t0):
171     """
172     Arrhenius chemical reaction kinetics.
173     @param obj: unit operation object
174     @param z: axial position
175     @type obj: aUnitParentClass
176     @type z: aFloat
177     @return: aList [ aFloat, aFloat, ... ]
178     """
179     return [\n
180         sum([nui*(math.exp(-a/obj['state_t'])/obj['fix_rgass'])/math.exp(-a/t0/obj['fix_rgass']))*obj['st\n
181         for nu in nmat\n
182     ]
183
184 # Matrix-like thermodynamic state functions. Explicit in temperature, volume and
185 # mole numbers.
186 def hpn_vs_tvn_jacobian(obj, null=None):
187     """
188     Thermodynamic Jacobian of d(H,p,N1,N2,...)/d(T,V,N1,N2,...) calculated as
189     [ [ dH/dT, dH/dV, dH/dN1, ... ], [ dp/dT, ... ], ... ].
190     @param obj: unit operation object
191     @param null: not used
192     @type obj: aUnitParentClass
193     @type null: anObject
194     @return: aList [ aList [ aFloat, aFloat, ... ] ]
195     """
196     nc = len(obj['state_n'])
197     dh = [obj['state_h_t']] + [obj['state_h_v']] + obj['state_h_n']
198     dp = [obj['state_p_t']] + [obj['state_p_v']] + obj['state_p_n']
199     dn = [\n
200         [obj['state_n_t'][i]] +\n
201         [obj['state_n_v'][i]] +\n
202             obj['state_n_n'][i*nc:(i+1)*nc] for i in xrange(0, nc)\n
203     ]
204     return [dh] + [dp] + dn
205
206 def hpn(obj, null=None):
207     """
208     Thermodynamic constraint function [ [H], [p], [N1], [N2], ... ].
209     @param obj: unit operation object
210     @param null: not used
211     @type obj: aUnitParentClass
212     @type null: anObject
213     @return: aList [ [ aFloat ], [ aFloat ], ... ]

```

```

214 """
215     return [[obj['state_h']]]) + \
216         [[obj['state_p']]]) + [[ni] for ni in obj['state_n']]
217
218 # Enthalpy, pressure, composition solver. No fall-back solution for erroneous
219 # thermodynamic calculations (cross your fingers). This is quite easy to program
220 # but it causes a mild code bloat and is left as an exercise for the interested
221 # reader.
222 import tkp4106
223
224 def hpn_vs_tvn_solver(obj, y1, eps, maxiter=50):
225 """
226     Thermodynamic equation solver. Iterates on 'tvn' = (T,V,N1,N2,...) to meet a
227     given specification 'y1' = (H,p,N1,N2,...).
228     @param obj: unit operation object
229     @param y1: [[H],[p],[N1],[N2],...] specification
230     @param eps: convergence criterion (upper bound)
231     @param maxiter: maximum number of iterations (negative value implies a fixed
232                     number of iterations).
233     @type obj: aUnitParentClass
234     @type y1: aList [ aList [ aFloat, aFloat, ... ] ]
235     @type eps: aFloat
236     @type maxiter: anInt
237     @return: aUnitParentClass
238 """
239     converged = False                                # convergence flag
240     norm = 1.0                                         # convergence control variable
241     nc = len(obj['state_n'])                         # number of chemical components in mixture
242     ni = 0                                            # number of iterations
243     while not converged:
244         ni += 1
245         dy = pass                                     # y1 - (h,p,n)
246         dx = tkp4106.solve(obj.jac(), dy)
247         tmp = max([abs(dxi[-1]) for dxi in dx])
248         converged = tmp < eps and tmp >= norm or (ni+maxiter) == 0
249         norm = tmp
250         if maxiter > 0:
251             print "norm=%8.3g;%s;" % (norm, obj)
252         if not converged and ni >= abs(maxiter):
253             raise ArithmeticError("maxIterations(%s) exceeded" % (ni,))
254         obj['var_t']+= pass                            # dt
255         obj['var_v']+= pass                           # dv
256         obj['var_n'] = pass                           # dn_i
257         obj()
258
259     return obj
260
261 # Numerical integration of enthalpy, pressure and composition problems. With or
262 # without chemical reactions.
263 def hpn_vs_tvn_integrator(method, obj, z0, z1, nz):
264 """
265     Thermodynamic integrator using Euler, RK2 or RK4 methods. Both explicit and
266     implicit update schemes are possible. The lambda function 'obj.update()' is
267     supposed to exist and is used to iterate on 'tvn' = (T,V,N1,N2,...) to meet
268     a given specification 'y1' = (H,p,N1,N2,...) in one or more iterations. One
269     iteration means an explicit update. Iteration till full convergence is also
270     possible. This is the implicit update. In calculating the right side of the
271     differential equation three other lambda functions must exist: These are
272     'obj.heatexchange()', 'obj.pressureprofile()' and 'obj.kinetics()' .
273     @author: Stud. Techn. Stig-Erik Nogva
274     @organization: Department of Chemical Engineering, NTNU, Norway
275     @param method: 'euler', 'rk2' or 'rk4'
276     @param obj: unit operation object
277     @param z0: start of integration
278     @param z1: end of integration
279     @param nz: number of integration steps
280     @type method: aString
281     @type obj: aUnitParentClass
282     @type z0: aNumber
283     @type z1: aNumber
284     @type nz: aNumber
285     @return: theUnitParentClass

```

```

286 """
287     objs = []                      # utility list (Runge-Kutta needs intermediate states)
288     dz = float(z1-z0)/nz           # integrator step size
289
290     for z in [z0+k*dz for k in xrange(0, nz)]:
291
292         # Calculate right side of ODE on the dot(y) = y(z) form.
293         yz = [obj.heatexchange(z)] + \
294             [obj.pressureprofile(z)] + obj.kinetics(z)
295
296     if method == 'euler':
297         y1 = pass                      # (h,p,n) + yz*dz
298
299     elif method == 'rk2':
300         while len(objs) < 2:
301             tmp = obj.duplicate('RK2_'+str(len(objs))) # 1 intermediate obj
302             objs.append(tmp)
303
304         for i in range(0, len(objs)):
305             objs[i].connect(obj)      # connect to master object in every step
306
307         # Obtain 1 auxiliary quantity
308         k1 = [yzi*dz for yzi in yz]
309         yk2 = [[yi[-1]+k1i] for (yi, k1i) in zip(objs[0].hpn(), k1)]
310         objs[0].update(yk2)          # iterate on the intermediate state
311
312         yz2 = [objs[0].heatexchange(z+1.0*dz)] + \
313             [objs[0].pressureprofile(z+1.0*dz)] + \
314                 objs[0].kinetics(z+1.0*dz)
315         k2 = [yzi*dz for yzi in yz2]
316         k = [k1i+k2i for (k1i, k2i) in zip(k1, k2)]
317
318         y1 = [[yi[-1]+(1/float(2))*ki] for (yi, ki) in zip(obj.hpn(), k)]
319
320     elif method == 'rk4':
321         while len(objs) < 4:
322             tmp = obj.duplicate('RK4_'+str(len(objs))) # 3 intermediate objs
323             objs.append(tmp)
324
325         for i in range(0, len(objs)):
326             objs[i].connect(obj)      # connect to master object in every step
327
328         # Obtain the 4 auxiliary quantities
329         k1 = [yzi*dz for yzi in yz]
330         yk2 = [[yi[-1]+0.5*k1i] for (yi, k1i) in zip(objs[0].hpn(), k1)]
331         objs[0].update(yk2)          # iterate on intermediate state 1
332
333         yz2 = [objs[0].heatexchange(z+0.5*dz)] + \
334             [objs[0].pressureprofile(z+0.5*dz)] + \
335                 objs[0].kinetics(z+0.5*dz)
336         k2 = [yzi*dz for yzi in yz2]
337         yk3 = [[yi[-1]+0.5*k2i] for (yi, k2i) in zip(objs[1].hpn(), k2)]
338         objs[1].update(yk3)          # iterate on intermediate state 2
339
340         yz3 = [objs[1].heatexchange(z+0.5*dz)] + \
341             [objs[1].pressureprofile(z+0.5*dz)] + \
342                 objs[1].kinetics(z+0.5*dz)
343         k3 = [yzi*dz for yzi in yz3]
344         yk4 = [[yi[-1]+k3i] for (yi, k3i) in zip(objs[2].hpn(), k3)]
345         objs[2].update(yk4)          # iterate on intermediate state 3
346
347         yz4 = [objs[2].heatexchange(z)] + \
348             [objs[2].pressureprofile(z)] + objs[2].kinetics(z)
349         k4 = [yzi*dz for yzi in yz4]
350         k = [k1i+2*k2i+2*k3i+k4i for (k1i, k2i, k3i, k4i) \
351                         in zip(k1, k2, k3, k4)]
352
353         y1 = [[yi[-1]+(1/float(6))*ki] for (yi, ki) in zip(obj.hpn(), k)]
354
355     else:
356         raise NameError('Method ' + method + " " + 'not implemented yet')
357

```

```
358 |         # Note: 'y1' is the final [ [H], [p], [N1], ... ] after the step 'dz' is
359 |         # taken. Lambda function 'obj.update()' is responsible for updating the
360 |         # thermodynamic state accordingly.
361 |         obj.update(y1)
362 |
363 |         print "z=%5.3f; %s;" % (z+dz, obj)
364 |
365 |     return obj
```

5.17.3 Verbatim: “ammonia_reactor.py”

```

1  """
2  @summary:      A simple ammonia reactor calculation illustrating some principles
3  of OOP (Object Oriented Programming) in chemical engineering::
4
5          'feed'           -----      'outlet'
6          ) -----> | ... 'rx' ... | -----> (
7          -----
8
9          The outcome of the study is a converged feed stream and an
10         integrated outlet from the reactor.
11 @author:        Tore Haug-Warberg
12 @organization: Department of Chemical Engineering, NTNU, Norway
13 @contact:       haugwarb@nt.ntnu.no
14 @license:       GPLv3
15 @requires:     Python 2.3.5 or higher
16 @since:        2011.10.04 (THW)
17 @version:      0.6
18 @todo 1.0:
19 @change:       started (2011.10.04)
20 @note:         This module defines the reaction chemistry (kinetics) and heat
21         transport for a minimal setup of an ammonia reactor. Nothing very
22         fancy, but there are 7 things to learn (see item numbering in
23         source code). From the command line run this script as::
24
25         >>> python ammonia_reactor.py 'euler|rk2|rk4' \
26                           'implicit|explicit' \
27                           <nz> <maxiter>
28
29         nz      = number of integration steps.
30         maxiter = maximum number of iterations spent on the thermodynamic
31         state calculations. If maxiter < 0 then exactly abs(maxiter)
32         iterations will be used independent of the residuals norm.
33 """
34
35 import srk_ammonia
36 import flowsheet
37 import tkp4106
38
39 # 1) There are 3 thermodynamic objects in action: 'feed', 'rx' and 'outlet'.
40 # Each object represents one - and only one - thermodynamic state. This means
41 # that 'rx', describing a state that varies in space, has to be integrated over
42 # the length over the reactor. The reactor profiles of temperature, pressure,
43 # etc. are lost in the process of integration, however, because 'rx' can keep
44 # only one (1) state at a time. It is of course possible to keep the profiles
45 # in memory as intermediate thermodynamic state objects, but this could easily
46 # be an overkill because explicit Euler integration requires somewhere in the
47 # range of 10,000 - 100,000 steps in order to reach 6 digits precision - which
48 # would eventually bind a substantial block of memory.
49 syngas = ['ammonia', 'nitrogen', 'hydrogen']
50
51 feed   = flowsheet.Stream('Feed', srk_ammonia, syngas)
52 outlet = flowsheet.Stream('Outlet', srk_ammonia, syngas)
53 rx     = flowsheet.Reactor('Rx', srk_ammonia, syngas)
54
55 # Initialize feed stream.
56 feed['var_t'] = 0.7                                # temperature [kK]
57 feed['var_v'] = 1.0                                # volume [dm3]
58 feed['var_n'] = [0.04, 0.24, 0.72]                 # mole fractions
59 feed()                                              # run thermodynamics code
60 feed['var_n'] = [ni/feed['state_mtot']/1e7 for ni in feed['state_n']] # [mol/kg]
61
62 # Re-initialize (change T and V to show extra flexibility).
63 feed(var_t=0.8, var_v=feed['var_v']/feed['state_mtot']/1e7)
64
65 print "Initial %s" % (feed,)
66
67 # 2) The feed stream has a specified pressure p0 whereas most thermodynamic equ-
68 # ations of state are explicit in volume (and temperature and composition). The
69 # relation p(V) = p0 must therefore be solved iteratively (using Newton's

```

```

70 # method in this case).
71 eps = 1.0e-8                                # convergence criterion
72 p0  = 0.25                                    # synthesis pressure [kbar]
73
74 print "\nNewton-Raphson solution of p(v)=p0:"
75
76 converged = False                            # convergence flag
77 norm = 1.0                                     # convergence control variable
78
79 # Solve p(v) = p0 using Newton's method. The thermodynamics model respond to the
80 # free variable 'var_v' and calculates pressure 'state_p' and pressure
81 # derivative 'state_p_n'.
82 while not converged:
83     dpdv = pass                                # Jacobian (1 x 1)
84     dp = pass                                  # pressure residual (1 x 1)
85     dv = tkp4106.solve(dpdv, dp)[0][-1]        # volume change (scalar)
86     feed['var_v'] += pass                      # update the model
87     converged = abs(dv) < eps and abs(dv) >= norm # continue till norm is steady
88     norm = abs(dv)                             # new norm
89
90     # The model fails if 'var_v' becomes unphysical (negative volume typically).
91     # If this happens we must shorten the iteration step until the model says it
92     # is OK. An exception is raised if the step becomes too small.
93     while not feed():
94         if abs(dv) < eps:
95             raise ArithmeticError("cannot converge p(v)=p0 relation")
96         pass                                      # step back to last successful state
97         pass                                      # reduce the step length
98         pass                                      # try once more
99     print "norm=%8.3g;" % (norm, feed)
100
101 print "\nConverged=%s" % (feed,)
102
103 # 3) Calculate the (atoms x component) matrix and the (components x reactions)
104 # stoichiometry from molecular formulas of the components in the mixture.
105 tmp = [formula for (name, formula, mw) in feed.get_cfw()]
106 amat = tkp4106.atom_matrix(tmp)
107 nmat = tkp4106.null(amat)
108
109 # 4) There is the use of functors in the simulation code. Their meaning is a bit
110 # magic to newbies, but to old-timers they offer a great way of code separation
111 # The key issue is that we can start writing algorithms (an Euler integrator in
112 # this case) requiring a certain functionality (pressure drop, heat exchange
113 # and reaction kinetics), without knowing the exact nature of the underlying
114 # functions. The properties are instead registered in the 'rx' object using so-
115 # called lambda expressions calling the correct function run-time by dereferenc-
116 # ing the function pointer. In effect, the heat exchange, pressure drop and
117 # reaction kinetics can be changed in one place of the code without affecting
118 # the solution algorithm. It yields, in fact, a way of defining the transport
119 # properties externally without changing neither the unit operation class nor
120 # the integration method. The same idiom is also used for defining thermodynamic
121 # state derivatives (the Jacobian). In this case we want to control the exact
122 # meaning of 'y1', 'y2', 'x1', 'x2', etc. in d(y1,y2,...)/d(x1,x2,...).
123 rx.connect(feed)
124
125 # Select a 'key' component for the reaction kinetics. Normalize the correspond-
126 # ing stoichiometric coefficient to -1. Make a shallow copy of matrix row before
127 # doing operations on 'nmat'. The algorithm works for single reactions only.
128 keyc = [name for (name, formula, mw) in rx.get_cfw()].index('nitrogen')
129 piv = list(nmat[keyc])
130 for i in xrange(0, len(nmat)):
131     for j in xrange(0, len(nmat[i])):
132         nmat[i][j] /= -piv[j]
133
134 # Declare transport properties and kinetics for the reactor. Non-linear example.
135 # rx.functor('pressureprofile', flowsheet.constantpdrop, [-0.005])          # dp/dz
136 # rx.functor('heatexchange', flowsheet.tubeandshell, [30.0, 0.28])      # ua*(t-t0)
137 # rx.functor('kinetics', flowsheet.arrhenius, [nmat, [keyc], [4/3.0], 0.1, 0.8])
138
139 # Declare transport properties and kinetics for the reactor. Linear example.
140 rx.functor('pressureprofile', flowsheet.constantpdrop, [0.0])                # dp/dz
141 rx.functor('heatexchange', flowsheet.constantcooling, [-20.0])    # heat [1.0e5 J]

```

```

142 rx.functor('kinetics', flowsheet.firstorder, [nmat, [keyc], [4/3.0]]) # rx rates
143
144 # 5) Interact with the command line reader to get hold of the integrator scheme
145 # and the number of steps required for the integration.
146 import sys
147
148 method, iterator, nz, maxiter = sys.argv[1:]
149 nz, maxiter = int(nz), int(maxiter)
150
151 # Declare a thermodynamic iterator (for use inside the integrator).
152 if iterator == 'implicit':
153     maxiter = abs(maxiter)
154
155 if iterator == 'explicit':
156     maxiter = -abs(maxiter)
157
158 # Declare a thermodynamic function solver and state derivatives for the reactor.
159 rx.functor('update', flowsheet.hpn_vs_tvn_solver, [eps, maxiter]) # state update
160 rx.functor('jac', flowsheet.hpn_vs_tvn_jacobian, []) # Jacobian matrix
161 rx.functor('hpn', flowsheet.hpn, []) # constraint variables
162
163 # 6) Integrate over the reactor using the given integration 'method' and the
164 # given 'iterator' mechanism.
165 print "\n%s.%s.integration_using_%s_steps:" % \
166     (iterator.capitalize(), method.capitalize(), nz)
167
168 flowsheet.hpn_vs_tvn_integrator(method, rx, 0, 1, nz) # integrate from z=0 to z=1
169
170 print "\nIntegrated %s" % (rx,)
171
172 # 7) Calculate the reactor outlet using an analytic solution based on the matrix
173 # exponential of the (constant) ODE coefficient. Let y = (h,p,c) and dot(y)=C*y
174 # Then y(z=1) = expm(C)*y(z=0) where 'expm' is the matrix exponential of C:
175 #
176 #      | 1 Q/p 0 0 0 |
177 #      | 0 1 0 0 0 |
178 # expm = | 0 0 1 nu_0/nu_1(fac - 1) 0 |
179 #      | 0 0 0 fac 0 |
180 #      | 0 0 0 nu_2/nu_1(fac - 1) 1 |
181 #
182 # Here, 'Q' is the heat load, 'p' is the (constant) reactor pressure, 'nu_i' are
183 # stoichiometric coefficients and 'fac' is the resilience factor of the 'key'
184 # component.
185 import math
186
187 outlet.connect(rx) # inherit lambda functions from 'rx'
188 outlet(var_t=feed['var_t'], var_v=feed['var_v'], var_n=feed['var_n']) # re-init
189
190 # Calculate the resilience factor of the 'key' component.
191 fac = math.exp(outlet.kinetics(0)[keyc]/outlet['state_n'][keyc])
192
193 # Calculate the matrix exponential.
194 nc = len(outlet['state_n'])
195 expm = [[float(i==j) for i in xrange(0,nc+2)] for j in xrange(0,nc+2)] # identity
196 expm[0][1] = outlet.heatexchange(0)/outlet['state_p'] # heat transfer
197 expm[2+keyc][2+keyc] = fac # 'key' component resilience
198 for i in [j for j in xrange(0,nc) if j != keyc]:
199     expm[2+i][2+keyc] = nmat[i][-1]/nmat[keyc][-1]*(fac-1.0) # other reactions
200
201 # Calculate the outlet state from y(z=1) = expm(C)*y(z=0).
202 y1 = tkp4106.mprod(expm, outlet.hpn())
203
204 print "\nNewton-Raphson solution of f(h,p,c)=0:"
205
206 flowsheet.hpn_vs_tvn_solver(outlet, y1, eps, 20)
207
208 print "\nConverged %s" % (outlet,)
```

5.17.4 Verbatim: “tkp4106.py”

```
1  """
2  @summary:      Increase local namespace with TKP4106 functionality.
3  @author:       Tore Haug-Warberg
4  @organization: Department of Chemical Engineering, NTNU, Norway
5  @contact:      haugwarb@nt.ntnu.no
6  @license:      GPLv3
7  @requires:     Python 2.3.5 or higher
8  @since:        2012.09.05 (THW)
9  @version:      0.9
10 @todo 1.0:
11 @change:       started (2012.09.05)
12 """
13
14 from molecular_weight import molecular_weight
15 from tridiagmprod import tridiagmprod
16 from atom_matrix import atom_matrix
17 from atoms import atoms
18 from solve import solve
19 from mprod import mprod
20 from rref import rref
21 from null import null
```

5.17.5 ammonia_reactor.py, see also Sec. 5.19.2

First reference occurs in *ammonia_reactor.py*, see Section 5.19.2 on page 335.

5.17.6 `srk_ammonia.py`, see also Sec. 5.17.1

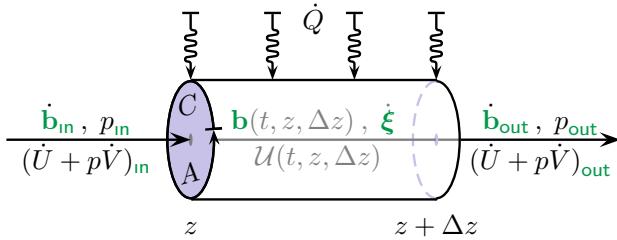
First reference occurs in *srk_ammonia.py*, see Section 5.17.1 on page 291.

Plug Flow Reactor. Part III

Tore Haug-Warberg
Department of Chemical Engineering
NTNU (Norway)

13 November 2011
(completed after 240 hours of writing, programming and testing)

1 Modelling issues



From an academic perspective the title of this text is a little pretentious. It says “Modelling Issues” which means quite a lot to people devoting their professional lives to the several aspects of chemical reactor calculations, while it means next to nothing for a novice

in the field. Let our perspective be something in between—that of an expert novice maybe. On our behalf then, the idealized plug flow reactor is like the one depicted in the figure. The mass and energy balances for steady state (^{s-s}) operation of the reactor were developed in Parts I and II of this paper. In short we found that:

$$\left(\frac{\partial h \text{ [energy mass}^{-1}]}{\partial z \text{ [length]}} \right)^{s-s} = C \text{ [length]} q \text{ [heat mass}^{-1} \text{ area}^{-1}]$$

and

$$\left(\frac{\partial c \text{ [mole mass}^{-1}]}{\partial z \text{ [length]}} \right)^{s-s} = A \text{ [area]} N r \text{ [mole mass}^{-1} \text{ volume}^{-1}]$$

What is missing here is a momentum balance of the reactor. It is needed to resolve the pressure distribution inside the reactor, which of course is of great interest for reactor design and operation, but at the same time it is pulling our wagon too far. The calculations are so involved and require so much input about reactor geometry, transport properties and kinetics that we must do without. Our replacement of the momentum balance is simply:

$$\left(\frac{\partial p \text{ [pressure]}}{\partial z \text{ [length]}} \right)^{s-s} = \nabla p \text{ [pressure]}$$

That is to say we rely on an explicit pressure profile $p(z)$ given at the outset of the simulation (we shall most of the time use $\nabla p = 0$).

Counting the number of equations there is 1 energy balance, 1 pressure profile and C mass balances. That makes $C + 2$ equations which are going to be solved simultaneously in $C + 2$ variables. The big question is: What variables? In practise we cannot choose the solution variables freely but must tackle whatever needs our models impose on us—*i.e.* the models we use to evaluate h , q and \mathbf{r} —and there is much fuzz about which variables are the most versatile.

Chemical engineers traditionally use T , p , x_1 , x_2 , … that is temperature, pressure and mole fractions. There is no theoretical reason for this choice except that these variables are always reported in process flow diagrams. They are also quite natural in the sense that they play a part of our sensation of the physical world.

Thermodynamicists think differently and usually prefer T , v , c_1 , c_2 , … that is temperature, specific volume and specific concentrations. This choice is natural from a theoretical point of view because most equations of state are given as $p(T, v, \mathbf{c})$ models. By iterating directly on the variables as they appear in the equation of state we can formulate very concise and elegant solvers.

Being trained thermodynamicists and having a keen eye on aesthetics we shall stick to the last alternative even though we then have to *solve* for pressure as a function of volume rather than just specifying it. The equations we need to be solve can be condensed into (see Parts I and II for an explanation of the syntax):

$$\begin{aligned} \text{Energy: } & \partial_T h \cdot \nabla T + \partial_v h \cdot \nabla v + \partial_{c_1} h \cdot \nabla c_1 + \partial_{c_2} h \cdot \nabla c_2 + \dots = Cq \\ \text{Momentum: } & \partial_T p \cdot \nabla T + \partial_v p \cdot \nabla v + \partial_{c_1} p \cdot \nabla c_1 + \partial_{c_2} p \cdot \nabla c_2 + \dots = \nabla p \\ \text{Mass (1): } & \nabla c_1 = A \sum_i \mathbf{N}_{1,i} \mathbf{r}_i \\ \text{Mass (2): } & \nabla c_2 = A \sum_i \mathbf{N}_{2,i} \mathbf{r}_i \\ & \vdots \end{aligned}$$

This set of equations is more easily handled using matrix algebra. To minimize the use of extra symbols $\partial_{\mathbf{c}} h$ and $\partial_{\mathbf{c}} p$ are taken to be row vectors while \mathbf{r} is (still) a column vector:

$$\begin{pmatrix} \partial_T h & \partial_v h & \partial_{\mathbf{c}} h \\ \partial_T p & \partial_v p & \partial_{\mathbf{c}} p \\ 0 & 0 & \mathbf{I} \end{pmatrix} \begin{pmatrix} \nabla T \\ \nabla v \\ \nabla \mathbf{c} \end{pmatrix} = \begin{pmatrix} Cq \\ \nabla p \\ A \mathbf{N} \mathbf{r} \end{pmatrix}$$

The equations above illustrate the ambivalence we are facing with regard to p or v being our primary iteration variable. In this case we shall iterate on v to satisfy ∇p given as the gradient of a predefined function $p(z)$. But, since pressure is a non-linear function of v it implies that ∇p shows up on the *right* side while ∇T , ∇v and $\nabla \mathbf{c}$ appear as *solution* variables on the left side. If p had been a primary iteration variable we could have dropped the second row in the equation set, but at the same time we had to handle the $p(v)$ inversion inside the equation of state. This is a questionable approach because

it involves a nested hierarchy of solvers which can cause all kinds of numerical problems. Usually, it is safer to handle all the equations in one solver, at least so when the equations are few in number like in this case. On a very condensed form we can write

$$\mathbf{J}(\mathbf{x})\nabla\mathbf{x} = \mathbf{f}(z, \mathbf{x}) \quad (1)$$

which is the equation system we have to integrate in order to calculate the temperature and concentration profiles of the reactor. Note carefully that $\mathbf{J}(\mathbf{x})$ is a purely thermodynamic state function while $\mathbf{f}(z, \mathbf{x})$ is a function of both the thermodynamic state variables and the space co-ordinate. The mathematical definitions of \mathbf{J} and \mathbf{f} are not known to us at this point—they are what we might call anonymous *lambda*-functions in functional programming—but their semantic meaning is all clear. E.g. their scientific units most conform¹.

The separation of the problem into \mathbf{J} and \mathbf{f} tells us that the transport and kinetic properties q and \mathbf{r} , used in defining \mathbf{f} on the right side, may require thermodynamic information, while the Jacobian \mathbf{J} is independent of the spatial co-ordinate and of the transport properties. Anyhow, the anti-derivative of the reactor model is

$$\mathbf{x}(z) = \mathbf{x}_o + \int_0^z \mathbf{J}(\mathbf{x})^{-1} \mathbf{f}(\zeta, \mathbf{x}) d\zeta ,$$

and the next question is how we can make an integrator for this problem. Basically, there are three options: Analytic, explicit and implicit solutions. We shall have a look at all three cases. Briefly stated there are few analytical solutions of practical interest, but the few that exist are important for: i) our theoretical insight, and ii) serving as test cases for numerical calculations. For the numerical solutions we must be aware that words like “explicit” and “implicit” have two different meanings. The terms do either refer to how the ODE is formulated, or they refer to how the integration is performed. The distinction is quite subtle and the implementation details are bewildering—these are the combinations we shall look at:

- Explicit ODE with explicit Euler integration (forward Euler).
- Implicit ODE with (semi)implicit Euler integration (backward Euler).
- Explicit ODE with explicit Runge–Kutta integration.
- Implicit ODE with explicit Runge–Kutta integration.

From a practical point of view it is easier to implement the explicit *solvers* compared to the implicit ones, but at the same time they are numerically unstable. This is a classic result from numerical mathematics which we should know about, but which is not so important for the PFR we are studying. What we shall see is that the explicit *model* formulation fails to conserve (even explicit) constraints in energy and pressure, while the implicit formulation does this to our satisfaction.

¹It also means that $\mathbf{f}(z, \mathbf{x})$ and $\mathbf{f}(z, \mathbf{x}(\mathbf{y}))$, and $\mathbf{f}(z, \mathbf{y})$, shall refer to the same kind of function in this document. The free variables change, and the function definitions need not be the same, but the function values are always interpreted as the gradient in specific enthalpy, pressure and composition.

1.1 Analytic solutions

Equation 1 is written with the variables $\mathbf{x} \triangleq [T, v, \mathbf{c}]$ in mind but it applies equally well to any other set of thermodynamic state variables yielding an invertible Jacobian \mathbf{J} . In particular we could try to replace \mathbf{x} by $\mathbf{y} \triangleq [h, p, \mathbf{c}]$ which yields a much simpler formulation. Note carefully that Jacobian reduces to $\mathbf{J}(\mathbf{y}) \equiv \mathbf{I}$:

$$\nabla \mathbf{y} = \mathbf{f}(z, \mathbf{y}) \quad (2)$$

Now, if $\mathbf{f}(z, \mathbf{y})$ is written as a linear function in \mathbf{y} we have the classic problem of an ordinary differential equation (ODE) with constant coefficients. The standard formulation of the problem is shown below (matrix \mathbf{C} has nothing to do with the circumference C used in the energy balance):

$$\nabla \mathbf{y} = \mathbf{Cy}$$

For PFRs that experience a constant circumference C , constant cross-sectional area A , constant pressure drop ∇p , constant heat flux q , and constant reaction rates \mathbf{r} or first order kinetics $\mathbf{r}_i \propto c_{j(i)}$, we can spell out four different cases of linear differential equations with *constant coefficients*. To keep the algebra as simple as possible—but not simpler—we shall assume one chemical reaction (i.e. $\dim \mathbf{N} = \dim \mathbf{c} \times 1$) and a dimensionless reactor length in the range $z \in [0, 1]$:

$$\begin{aligned} 1) \left\{ \begin{array}{l} \nabla h = 0 \\ \nabla p = \nabla p \\ \nabla \mathbf{c} = \xi \mathbf{N} \end{array} \right. & \quad 2) \left\{ \begin{array}{l} \nabla h = 0 \\ \nabla p = \nabla p \\ \nabla \mathbf{c} = kc_1 \mathbf{N} \end{array} \right. \\ 3) \left\{ \begin{array}{l} \nabla h = q \\ \nabla p = 0 \\ \nabla \mathbf{c} = \xi \mathbf{N} \end{array} \right. & \quad 4) \left\{ \begin{array}{l} \nabla h = q \\ \nabla p = 0 \\ \nabla \mathbf{c} = kc_1 \mathbf{N} \end{array} \right. \end{aligned}$$

Here, ξ means the overall *extent of reaction*, q means the overall *heat transfer* and kc_1 denotes the *first order* reaction *with respect to* component 1 (an arbitrary choice from our side). A textual interpretation of the four cases follows:

Case	Description
1	Adiabatic, fixed pressure drop, fixed extent of reaction
2	Adiabatic, fixed pressure drop, first order reaction
3	Fixed heat load, isobaric, fixed extent of reaction
4	Fixed heat load, isobaric, first order reaction

Behind the terminology of *constant coefficients* there is an implication that the equations can be recast into matrix expressions. This is advantageous from a theoretical perspective because it renders a generic solution of the problem $\nabla \mathbf{y} = \mathbf{Cy}$ where \mathbf{C} takes one

of the four shapes shown below:

$$\mathbf{C}_1 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ \frac{\nabla p}{h} & 0 & 0 & 0 \\ \frac{\xi\nu_1}{h} & 0 & 0 & 0 \\ \frac{\xi\nu_2}{h} & 0 & 0 & 0 \end{pmatrix} \quad \mathbf{C}_2 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ \frac{\nabla p}{h} & 0 & 0 & 0 \\ 0 & 0 & k\nu_1 & 0 \\ 0 & 0 & k\nu_2 & 0 \end{pmatrix}$$

$$\mathbf{C}_3 = \begin{pmatrix} 0 & \frac{q}{p} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \frac{\xi\nu_1}{p} & 0 & 0 \\ 0 & \frac{\xi\nu_2}{p} & 0 & 0 \end{pmatrix} \quad \mathbf{C}_4 = \begin{pmatrix} 0 & \frac{q}{p} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & k\nu_1 & 0 \\ 0 & 0 & k\nu_2 & 0 \end{pmatrix}$$

Here, we have been assuming a two-component mixture with chemical reaction $\nu_1 A = \nu_2 B$. More components can easily be added without violating the structure of the matrices. The solution(s) can be written

$$\mathbf{y}(z) = e^{z\mathbf{C}}\mathbf{y}(0)$$

where $e^{z\mathbf{C}}$ means the *matrix exponential* of $z\mathbf{C}$. Covering the matrix theory in detail would take us astray from the PFR subject, but it is important to know that what is said next *can* be formalized—if not always as closed analytical formulas—at least in the form of numerical calculations. But, for the \mathbf{C} -matrices mentioned above we can follow the simple approach and find the matrix exponentials by inspection because the matrices have such simple structures. Writing out solutions of mathematical problems without any further details is somewhat arrogant but I think that in this case it implies less confusion—not more confusion—to do it quick and simple. You should verify the results by backsubstituting into the matrix formula using $\mathbf{y}(0) = [h, p, \mathbf{c}]_{z=0}$ though:

$$e^{z\mathbf{C}_1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{z\nabla p}{h} & 1 & 0 & 0 \\ \frac{z\xi\nu_1}{h} & 0 & 1 & 0 \\ \frac{z\xi\nu_2}{h} & 0 & 0 & 1 \end{pmatrix} \quad e^{z\mathbf{C}_2} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{z\nabla p}{h} & 1 & 0 & 0 \\ 0 & 0 & e^{zk\nu_1} & 0 \\ 0 & 0 & \frac{\nu_2}{\nu_1} (e^{zk\nu_1} - 1) & 1 \end{pmatrix}$$

$$e^{z\mathbf{C}_3} = \begin{pmatrix} 1 & \frac{zq}{p} & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \frac{z\xi\nu_1}{p} & 1 & 0 \\ 0 & \frac{z\xi\nu_2}{p} & 0 & 1 \end{pmatrix} \quad e^{z\mathbf{C}_4} = \begin{pmatrix} 1 & \frac{zq}{p} & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & e^{zk\nu_1} & 0 \\ 0 & 0 & \frac{\nu_2}{\nu_1} (e^{zk\nu_1} - 1) & 1 \end{pmatrix}$$

Case 4 is maybe the most interesting for the chemical engineering student since it gives the opportunity to study PFRs with a maximum in temperature along the reactor. The argument is simple: Consider an exothermic first order reaction with constant cooling. A first order reaction means that the reaction rate will decrease monotonically along the reactor. Then, by balancing the heat production in the middle the reactor with the heat

taken away at the same spot it should be clear that excess heat is produced at the inlet and excess cooling is applied at the outlet. The result is a curved temperature profile which of course looks more interesting than a flat one.

1.2 Explicit Euler-integration

Talking about numerical integration the word *explicit* means the differential equations are stated without iterative calculations. So, how can that be arranged for a non-linear problem? The short answer is it cannot, the long answer is we can make piecewise linear approximations to the functions we want to integrate and solve each little sub-problem explicitly. The outcome will not be *the* answer, but merely a numerical approximation to it. There are many things to worry about in such calculations. Numerical accuracy and stability are maybe the most important issues.

We shall not look very deep into the matter but try to understand what happens in a numerical integrator and see how we can formulate the equations in a piecewise manner. Our starting point is Eq. 1:

$$\mathbf{J}(\mathbf{x})\nabla\mathbf{x} = \mathbf{f}(z, \mathbf{x})$$

Inverting \mathbf{J} (yes, we must assume that the Jacobian is invertible—else the problem is thermodynamically inconsistent) yields the explicit formula

$$\nabla\mathbf{x} = \mathbf{J}(\mathbf{x})^{-1}\mathbf{f}(z, \mathbf{x})$$

Then comes the piecewise approximation $\nabla\mathbf{x} \approx (\Delta z)^{-1}\Delta\mathbf{x}$ which is *assumed* to be valid on the range $[z, z + \Delta z]$:

$$\Delta\mathbf{x} = \mathbf{J}(\mathbf{x}_z)^{-1}\mathbf{f}(z, \mathbf{x}_z)\Delta z + \mathcal{O}(\Delta z)^2$$

The truncation error is of second order, that is $\mathcal{O}(\Delta z)^2$, but the integrated answer will not be that accurate because the number of steps taken in the interval is proportional to $(\Delta z)^{-1}$ which means the integration error will be $\mathcal{O}(\Delta z)^2(\Delta z)^{-1} = \mathcal{O}(\Delta z)^1$, that is of first order only. We shall later learn how to implement schemes of higher order, namely the Runge–Kutta integration methods of 2nd and 4th order. From the definition $\Delta\mathbf{x} \hat{=} \mathbf{x}_{z+\Delta z} - \mathbf{x}_z$ we can write the final update formula as:

$$\mathbf{x}_{z+\Delta z}^{\text{e-e}} \hat{=} \mathbf{x}_z + \mathbf{J}(\mathbf{x}_z)^{-1}\mathbf{f}(z, \mathbf{x}_z)\Delta z \quad (3)$$

By applying this formula successively on the integration domain $z \in [0, 1]$ we can calculate the sequence $\mathbf{x}_0, \mathbf{x}_{\Delta z}, \mathbf{x}_{2\Delta z}, \dots$ very easily. Furthermore, it is (almost) evident that \mathbf{x}_{Nz} will converge to the true solution $\mathbf{x}(Nz)$ when $\Delta z \rightarrow 0$ and $N \rightarrow \infty$. But, this requires an infinite number of steps which eventually would take infinite time on a computer. Another problem of the numerical solution is that computers have fixed word lengths. Irrational numbers are *approximated* inside the computer as decimal numbers represented by 16, 32, 64, or 128 bits length. This gives a *round-off* error in (nearly) every multiplication or division that is carried out. There is therefore a trade-off between a smaller Δz to achieve higher accuracy in the updating formula, and a not-so-small Δz to avoid excessive round-off errors (and to reduce the computation time).

1.3 Implicit Euler-integration

Physical theories build on a limited number of conservation laws. For example mass and energy conservation is essentially what lies behind our PFR model. This is the strong point of physics. The weaker part of the theory arises from the lack of appropriate models expressed directly in the conserved properties. This branch of physics belongs to thermodynamics. In our case the conservation laws are made linear in the thermodynamic variables h, p, \mathbf{c} , while in most cases the equation of state serving the calculation of p (and h) is on the form $p(T, v, \mathbf{c})$. Hence, to update the equation of state we need to solve the relationships between T, v and h, p iteratively (the problem is strongly coupled and non-linear). If these relationships are solved at each step taken from z to $z + \Delta z$ the method is said to be implicit. Recall that for the explicit method in Eq. 3 there is no need for an iterative solution because matrix inversion in itself is an explicit method.

Why should we worry about implicit integration then? It sounds complicated and if explicit integration works why bother? The answer is simple, definite and instructive: Explicit integration violates the conservation principle(s) because of the linearization term that is behind Eq. 3. If this feature is considered to be unfortunate we should consider implicit integration. This is because it solves the conservation equations accurately at each step of the integration. It is not to say that the integration is more accurate, it is only consistent. Consistently wrong you might say, but it is not inconsistent.

To write an implicit integrator we need to understand that the conservation laws put constraints on $\mathbf{y} \hat{=} [h, p, \mathbf{c}]$ while the thermodynamics, heat exchange, pressure drop and kinetics models rely on $\mathbf{x} \hat{=} [T, v, \mathbf{c}]$. We must therefore be able to solve the relationship $\mathbf{x}(\mathbf{y})$ by e.g. Newton–Raphson iteration (to obtain second order convergence) in parallel with the integration task. This topic is also known as: Integration on manifolds, geometric integration, or Differential–Algebraic–Equations (DAEs) solving. The starting point is the same as in Eq. 2 except for the implicit relation $\mathbf{x}(\mathbf{y})$ that sits on the right side:

$$\nabla \mathbf{y} = \mathbf{f}(z, \mathbf{x}(\mathbf{y}))$$

Linearization (this time in \mathbf{y}) yields:

$$\Delta \mathbf{y} = \mathbf{f}(z, \mathbf{x}(\mathbf{y}_{z+\Delta z})) \Delta z + \mathcal{O}(\Delta z)^2$$

This is the *fully* implicit formulation of the problem, where “fully” indicates that the right side is evaluated at the next location $z + \Delta z$, i.e. not the current z . Solving this problem with Newton–Raphson iteration is not so easy because it requires derivative information about $\mathbf{f}(z, \mathbf{x}(\mathbf{y}))$. We know very little about the structure of this function and can hardly make anything ready on general terms, but for the relation $\mathbf{x}(\mathbf{y})$ we know a lot. It is a thermodynamic mapping with a fixed structure awaiting only a thermodynamic model to calculate the numbers run-time. We shall therefore restrict ourselves to the following *semi-implicit* formulation of the problem

$$\Delta \mathbf{y} = \mathbf{f}(z, \mathbf{x}(\mathbf{y}_z)) \Delta z + \mathcal{O}(\Delta z)^2$$

where the right side is assumed constant at each position z . This yields the simpler update formula:

$$\mathbf{y}_{z+\Delta z} \doteq \mathbf{y}_z + \mathbf{f}(z, \mathbf{x}(\mathbf{y}_z))\Delta z$$

Even though the formulation above is semi-implicit it is consistent with any conservation principle that yields a constant contribution on the right side (linear profile). The method is therefore referred to as just “implicit” when there is no danger of misunderstanding. Later on we shall see in practise how the method works for a problem with linear enthalpy and pressure profiles. Notwithstanding these merits the semi-implicit method is just an approximation with respect to changes that are *not* subject to conservation. Temperature is one example. So, even when the energy is conserved the temperature profile is not necessarily correct. Incorrect is not the same as inconsistent though.

To solve for $\mathbf{y}_{z+\Delta z}$ we shall alter the values of \mathbf{x} . We must then make some additional calculations denoted as *iterations* $0, 1, \dots, k, k+1$. Because the problem formulation is semi-implicit we need derivatives for \mathbf{y} versus \mathbf{x} but not for $\mathbf{f}(z, \mathbf{x}(\mathbf{y}))$. Linearization of $\mathbf{y}_{z+\Delta z}$ on the left side yields the following approximation:

$$\mathbf{y}_z^k + \mathbf{J}(\mathbf{x}_z^k)\Delta \mathbf{x}^k \approx \mathbf{y}_z^0 + \mathbf{f}(z, \mathbf{x}(\mathbf{y}_z^0))\Delta z$$

By definition $\mathbf{y}_z^0 \equiv \mathbf{y}_z$ and we sincerely hope that $\mathbf{y}_z^\infty \rightarrow \mathbf{y}_{z+\Delta z}$. We cannot prove the last property, but if it is correct the iteration process is said to *converge* locally. The Newton–Raphson procedure may converge or it may diverge. Impossible to say in fact without problem specific information. If it does converge, however, it shows *second order* convergence. In practise this means that the number of *significant* digits will double in each iteration when k is sufficiently large. What *sufficiently* large means is also hard to say, but in normal cases it is typically in the range $k_{\text{crit}} \in [3, 5]$. Solving for $\Delta \mathbf{x}^k$ we get:

$$\Delta \mathbf{x}^k \approx \mathbf{J}(\mathbf{x}_z^k)^{-1} \underbrace{[\mathbf{y}_z^0 + \mathbf{f}(z, \mathbf{x}(\mathbf{y}_z^0))\Delta z - \mathbf{y}_z^k]}_{\mathbf{y}_{z+\Delta z}} \quad (4)$$

Note the underbrace above: $\mathbf{y}_{z+\Delta z}$ comes in as a constant estimate on the right side such that if (i.e. hopefully then) the iteration converges we get $\mathbf{y}_z^k \rightarrow \mathbf{y}_z^\infty \rightarrow \mathbf{y}_{z+\Delta z}$ which makes $\Delta \mathbf{x}^k \rightarrow \mathbf{0}$. Finally, when the update *norm* satisfies $||\Delta \mathbf{x}^k|| \leq \epsilon$ the iteration is stopped. A suitable stop criterion must be set by us—or in practise the programmer. The definition of $\Delta \mathbf{x}^k \doteq \mathbf{x}_z^{k+1} - \mathbf{x}_z^k$ leads to

$$\mathbf{x}_z^{i-e, k+1} \doteq \mathbf{x}_z^k + \mathbf{J}(\mathbf{x}_z^k)^{-1} [\mathbf{y}_{z+\Delta z} - \mathbf{y}_z^k] \quad (5)$$

which is the final update formula for the implicit Euler integration method. But, for the special case $k = 0$ we can identify $\mathbf{y}_{z+\Delta z} - \mathbf{y}_z^k$ on the right side being equal to $\mathbf{y}_{z+\Delta z} - \mathbf{y}_z^0 = \mathbf{f}(z, \mathbf{x}(\mathbf{y}_z^0))\Delta z$, see Eq. 4. This leaves the much simpler formula:

$$\mathbf{x}_z^{i-e, 1} = \mathbf{x}_z^0 + \mathbf{J}(\mathbf{x}_z^0)^{-1} \mathbf{f}(z, \mathbf{x}(\mathbf{y}_z^0))\Delta z$$

Comparing the right side of this formula with the explicit Euler formula in Eq. 3 reveals the following relationship (after noticing that $\mathbf{x}_z^0 \equiv \mathbf{x}_z$ and $\mathbf{y}_z^0 \equiv \mathbf{y}_z$):

$$\mathbf{x}_z^{i-e, 1} \equiv \mathbf{x}_{z+\Delta z}^{e-e}$$

The conclusion is that the first iteration of the implicit Euler scheme is identical to the explicit Euler update (if, and only if, the update is calculated using Newton–Raphson iteration). We can therefore say that the two integration methods are examples of N 'th level explicit Euler schemes. For $N = 1$ we retain the classic Euler integration and for $N \rightarrow \infty$ we get implicit Euler integration, but in many cases it is enough to make only 2 or 3 Newton–Raphson updates in order to reach a sufficiently converged \mathbf{x} -state. Thus, it makes sense to integrate several times trying out 1st, 2nd and 3rd level updates to verify that the solution converges smoothly to a value that is independent of the linearization. What cannot be controlled in this manner is the accuracy of the integration. Usually, higher accuracy means higher order approximation methods like for instance the Runge–Kutta family of non-stiff integrators. To control stiffness as well (that is integrating ODEs showing a wide spread in the eigenvalues) we have to deal with an entirely different approach using variable step length and precondition of the equations. This is way outside the current scope.

1.4 Runge–Kutta integration

The Runge–Kutta methods belong to a family of explicit integrators often considered to be the work horses of numerical integration. The members of this family are characterized by an order parameter n saying that the global integration error is proportional to $(\Delta z)^n$, where n is typically 2, 3, 4 and 5. A Runge–Kutta method of order 1 will then be equivalent to explicit (forward) Euler integration. It can be argued that schemes of even order are better “balanced” than schemes of odd order. The odd-ordered schemes are therefore used for truncation error control, mostly, while the integration itself is carried out with one of the even-ordered schemes.

We shall have a further look at second and fourth order schemes called RK2 and RK4 throughout this text. These are explicit integration schemes, but the methods will be defined such that we can choose to stay on the h, p, \mathbf{c} manifold if we wish. It is then important to know what “on” means. Just like for the explicit and (semi)implicit Euler methods this question does not need be answered once and for all, but can await us specifying (later) the number of iterations we would like to spend on the update of T, v, \mathbf{c} at each step of the integration.

1.5 Calculation example

A good calculation example must serve many needs. Firstly, it should be verifiable. Only this way is it possible to prove (or disprove) that the equations are solved correctly. Secondly, it should be familiar to the reader. An example that comes as a total surprise can hardly serve as an example because the perspective is missing. Thirdly, it should be realistic. An unrealistic example can perhaps be more intriguing but it adds nothing to our physical experience. Forthly, it should contribute new insight. However, to come up with an example that is both verifiable, familiar, realistic and new is not so easy.

The production of ammonia from nitrogen and hydrogen is a classical textbook example. It is the most important of all the industrial reactions and without it we would

have been in the 19th century still. But, it has a very complicated reactor design and we shall not try too hard to be realistic. Uniform cooling, zero pressure drop and first order reaction is the best we can do if we also want to verify the calculation by comparing it with an analytical solution, see also [Section 1.1](#).

The ammonia reaction is exothermic and shows a substantial temperature increase under normal operation. So, by matching the cooling duty with the reaction rate it is possible to obtain a curved temperature profile along the reactor axis. The chemical compositions vary exponentially along the same axis and for the reactor as a whole we can expect a pronounced non-linear behaviour. This puts our solution method on trial. We shall therefore investigate several integration schemes: explicit and implicit Euler, and explicit RK2 and RK4 (Runge–Kutta 2nd and 4th order) with both explicit and implicit function updates.

For the reactor calculation we need of course a set of differential equations, but we also need to fill in with thermodynamic state information. Ideal gas is the simplest non-trivial concept we can use in this case. The gas mixture of ammonia, nitrogen and hydrogen is non-ideal at synthesis conditions, but the physical insight of the problem is not changed very much by this fact. The only artifact we should know about is that the ideal gas enthalpy is independent of pressure whereas the real enthalpy is not (this feature can betray us badly at adiabatic conditions). The thermodynamic relations we are using are listed below:

$$p^{\text{ig}} = \frac{\sum_i c_i RT}{v}$$

$$h^{\text{ig}} = \sum_i c_i (\Delta_f h_i^\circ + \int_{0.29815}^T c_{p,i}^\circ(\tau) d\tau)$$

where $R \triangleq 0.083145 \dots 10^5 \text{ J mol}^{-1} \text{ K}^{-1}$, and where

$$\frac{\Delta_f h_{\text{NH}_3}^\circ}{[10^5 \text{ J mol}^{-1}]} = -0.45898; \quad \Delta_f h_{\text{N}_2}^\circ = \Delta_f h_{\text{H}_2}^\circ = 0$$

and finally:

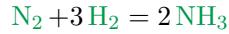
$$\frac{c_{p,\text{NH}_3}^\circ(\tau)}{[10^5 \text{ J mol}^{-1} \text{ K}^{-1}]} = 0.27310 + 0.23830\tau + 0.17070\tau^2 - 0.11850\tau^3$$

$$\frac{c_{p,\text{N}_2}^\circ(\tau)}{[10^5 \text{ J mol}^{-1} \text{ K}^{-1}]} = 0.31150 - 0.13570\tau + 0.26800\tau^2 - 0.11680\tau^3$$

$$\frac{c_{p,\text{H}_2}^\circ(\tau)}{[10^5 \text{ J mol}^{-1} \text{ K}^{-1}]} = 0.27140 + 0.09274\tau - 0.13810\tau^2 + 0.07645\tau^3$$

As explained at the beginning of this chapter the mixture is normalized to one kilogram of material which implies that all enthalpies, volumes and mole numbers are reported as specific quantities in the upcoming tables. This fixes the size of the problem. Everything

is on mass basis. The last statement can be a little bewildering because the reaction stoichiometry is



which is independent of the system size. This equation reflects only the chemical *stoichiometry*, however, and not the total conversion in the system. It is the kinetics model that scales the chemical reaction equation to the size of the system. Now, to integrate through the reactor we need to know the complete intensive state of the gas mixture at the inlet. The initial temperature, pressure and composition (mole fractions) chosen in this case are:

$$\begin{aligned} T_o &= 0.800 \text{ kK} \\ p_o &= 0.250 \text{ kbar} \\ z_o &= [0.04, 0.24, 0.72] [-] \end{aligned}$$

The units of thermodynamics (**kK**, **kbar**, **10^5J** , **dm^3** and **mol**) are maybe curious but they are in fact judiciously selected to increase the numerical stability of the solvers. This issue is hard to explain without the prior knowledge of numerical mathematics and fixed word-length computers and we shall leave it open for the interested reader. Note also that the initial pressure is a dependent variable in this case and that it must be iterated on since the thermodynamic model is explicit in volume—not in pressure. Carrying on we shall assume a uniform cooling profile along the reactor equal to $\nabla h = -20 \text{ } 10^5 \text{J}$, zero pressure drop $\nabla p = 0 \text{ kbar}$, and first order reaction of nitrogen equal to $\nabla c_{\text{N}_2} = -(4/3)c_{\text{N}_2} \text{ mol}$. All gradients are defined per kilogram of material and per reactor length. The outcome is a set of differential equations equivalent to Case 4 in [Section 1.1](#):

$$\nabla \mathbf{y} \hat{=} \nabla \begin{pmatrix} h \\ p \\ c_{\text{NH}_3} \\ c_{\text{N}_2} \\ c_{\text{H}_2} \end{pmatrix} \rightarrow \begin{pmatrix} -20 \\ 0 \\ (8/3)c_{\text{N}_2} \\ -(4/3)c_{\text{N}_2} \\ -(4/1)c_{\text{N}_2} \end{pmatrix}$$

The analytical solution is

$$\mathbf{y}(z) \hat{=} \begin{pmatrix} h \\ p \\ c_{\text{NH}_3} \\ c_{\text{N}_2} \\ c_{\text{H}_2} \end{pmatrix} \rightarrow \begin{pmatrix} h_o - 20z \\ p_o \\ c_{\text{NH}_3}^o - 2(\alpha - 1)c_{\text{N}_2}^o \\ \alpha c_{\text{N}_2}^o \\ c_{\text{H}_2}^o + 3(\alpha - 1)c_{\text{N}_2}^o \end{pmatrix}$$

where $\alpha \triangleq e^{-(4/3)z}$. The enthalpy, pressure and composition profiles are easily calculated from the last formula and by iterating on temperature and volume at each step along the reactor axis (we need in fact only one step to integrate the entire reactor) we can calculate the profiles to our discretion. **E.g.** dividing the reactor into 5 segments yields the following exact answer to our differential equation problem (reported in more familiar units for the ease of reading):

z	$\frac{T}{[\text{K}]}$	$\frac{V}{[\text{dm}^3]}$	$\frac{h}{[\text{MJ}]}$	$\frac{p}{[\text{bar}]}$	$\frac{c_{\text{NH}_3}}{[\text{mol}]}$	$\frac{c_{\text{N}_2}}{[\text{mol}]}$	$\frac{c_{\text{H}_2}}{[\text{mol}]}$
0	800.000	30.0438	1.495255	250.000	4.5168	27.1006	81.3019
0.2	882.267	29.4106	1.095255	250.000	17.2037	20.7571	62.2714
0.4	919.963	27.6941	0.695255	250.000	26.9211	15.8985	47.6954
0.6	921.796	25.4676	0.295255	250.000	34.3638	12.1771	36.5313
0.8	894.927	23.0285	-1.04745	250.000	40.0645	9.3268	27.9804
1	844.596	20.5069	-0.504745	250.000	44.4307	7.1436	21.4309

The numbers printed in blue ink are the variables we want to investigate further using a small assortment of homemade integrators. So, integrating from $z = 0$ to $z = 1$ in 3 steps (numbers being exact to 6 digits are printed in blue) yields:

Method	N	$\frac{T}{[\text{K}]}$	$\frac{V}{[\text{dm}^3]}$	$\frac{h}{[\text{MJ}]}$	$\frac{p}{[\text{bar}]}$
Euler	1	923.156	21.7968	-0.522353	239.498
Euler	3	928.546	21.0031	-0.504745	250.001
RK2	1	828.557	20.4743	-0.507512	248.660
RK2	3	829.427	20.3859	-0.504745	250.000
RK4	1	844.365	20.5106	-0.504997	249.918
RK4	3	844.444	20.5057	-0.504745	250.000
Exact	-	844.596	20.5069	-0.504745	250.000

We see that all the explicit methods fail: Euler-1 fails badly, RK2-1 fails less, while RK4-1 is pretty close—but they all fail. The implicit methods behave differently. Except for Euler-3 they are all correct in their predictions of *enthalpy* and *pressure*. This means the energy and momentum balances are *consistent* with the underlying conservation principles. The temperature and the volume are still off which means the calculations are not correct—only consistent.

By increasing the number of integration steps we may hope to rectify the situation and get truely correct answers. In fact, by integrating from $z = 0$ to $z = 1$ in 12 steps (numbers being exact to 6 digits are still printed in blue) we get:

Method	N	$\frac{T}{[\text{K}]}$	$\frac{V}{[\text{dm}^3]}$	$\frac{h}{[\text{MJ}]}$	$\frac{p}{[\text{bar}]}$
Euler	1	862.454	20.7456	-0.507013	248.550
Euler	3	863.160	20.6421	-0.504745	250.000
RK2	1	843.829	20.5017	-0.504892	249.982
RK2	3	843.875	20.5014	-0.504745	250.000
RK4	1	844.595	20.5069	-0.504746	250.000
RK4	3	844.596	20.5069	-0.504745	250.000
Exact	-	844.596	20.5069	-0.504745	250.000

This time RK4-3 yields correct answers all over the line. The same resolution with RK2-3 and Euler-3 would require 380 and 500,000 steps respectively. Note: The total calculation effort is bigger because one step of RK4-3 requires 4 intermediate steps each

using 3 iterations in Eq. 5. The total number of steps is then $12*4*3 = 144$. For RK2-3 the total number of steps is $360*2*3 = 2160$, and for Euler-3 it is $500,000*1*3 = 1,500,000$. Notwithstanding the extra calculations required to fulfill the RK4 and RK2 steps, the conclusion is that higher order schemes are superior to lower order schemes (of course I should say).

In interesting spin-off from this disussion is that there is no difference between implicit and explicit problem formulations when we talk about numerical accuracy. I.e. explicit Euler and implicit Euler yield the same accuracy as do RK2 with explicit and implicit model formulations and the same for RK4. But then it comes to conservation laws we see the difference. The implicit model formulation always yield correct enthalpies and pressures whereas the explicit formulations do not. For RK4 the difference is in the last digit only, but it is nevertheless present and it is visible.

Exercise 9

1 Question: Topology 06

The subject of this problem is the plant shown in Figure 1. It is a drying process, which is the final stage in producing starch from potatoes. To extract the starch, the potatoes are first peeled by scratching of the outer surface, then crushed. The starch grains are released from the destroyed cells and released into the water. The starch is solid and not solvable in water. The mixture of solids and liquid phase changes as the process proceeds, but from the outside it very much appears as a uniform phase, a pseudo phase, which we describe as a sludge. From the sludge, what is mainly being washed out is sugars and other soluble components. After several stages of washing and filtrating the last filter set is extracting the maximum of water in large drum filters. The wet starch is mixed in a tank to even the water contents and finally dried in a very large tubular dryer in with hot air.

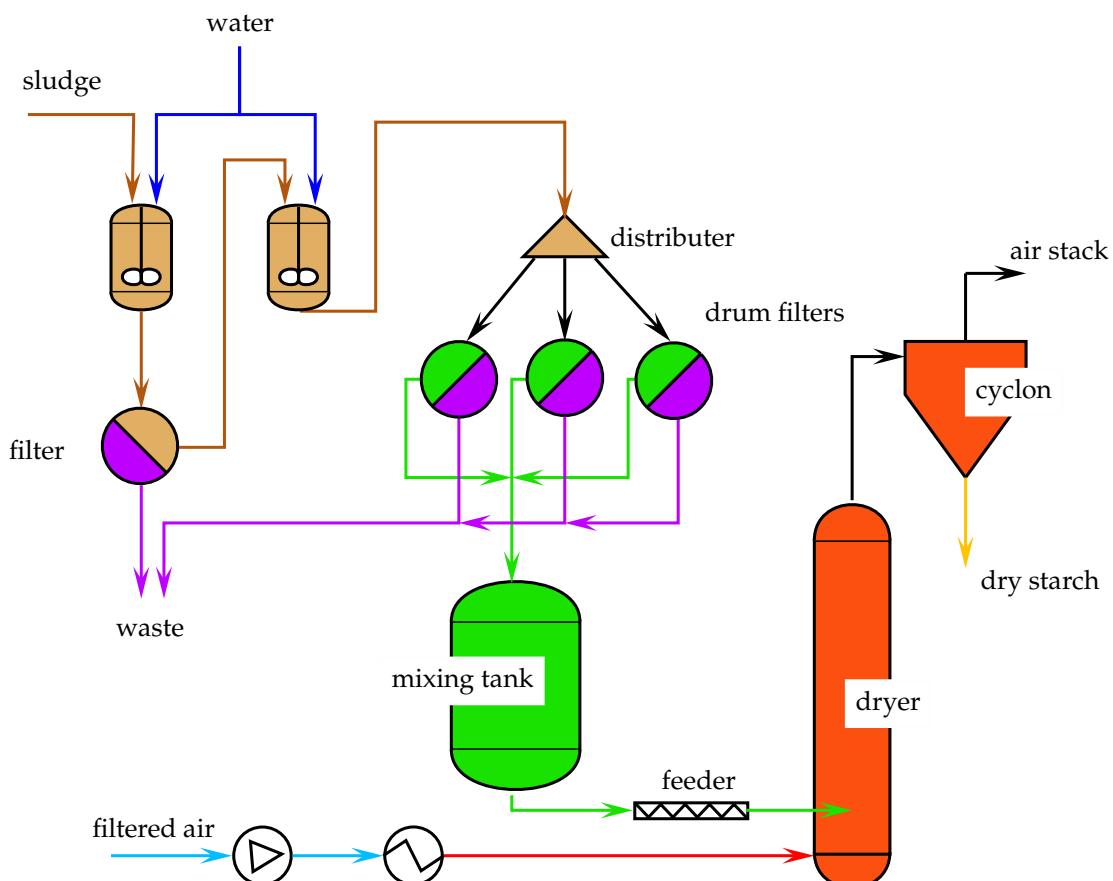


Figure 1: A flowsheet of potato starch plant

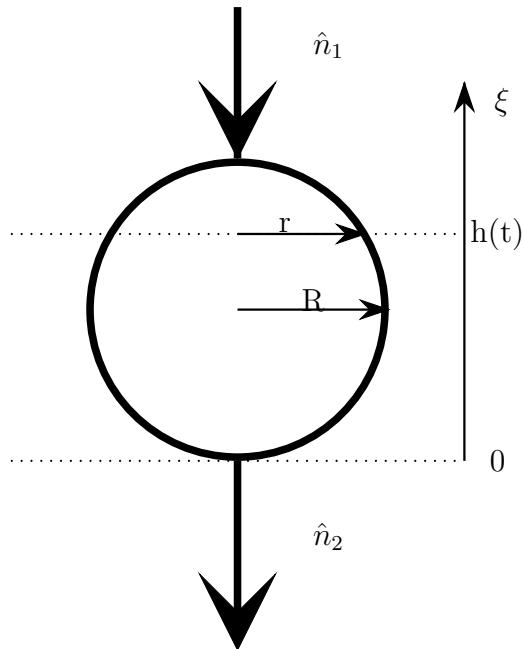
- Sketch the topology of the potato starch plant.

2 Objective

- Be familiar with a systematic approach and get consequently the job done quicker. Things must become a routine.
- Re-write model in the system's theory state-space notation
- The control formulation must clearly stand out
- Exercise linearisation

3 Problem Formulation

Level control in tanks is a standard non-linear problem because the outflow depends on the level in the tank, which is also the quantity being measured. It is common to write the problem in terms of the variable *level* and not the variable *mass*, which in addition introduces the non-linearity associated with the change of diameter with height of the tank. The geometry is given, thus R , the spheres radius is known. In addition, one would have to consider also the change of density, if the composition or the temperature of the fluid in the tank changes. Even if one ignores the change of density and assumes a single component fluid, say water, the combination of the two remaining problems results in surprisingly complicated equations as you will see.



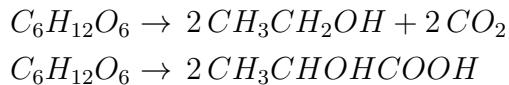
Assume constant density and the inflow being controlled as well as the outflow. The volumetric flow rates at the input is controlled using a fast flow controller, which will provide the desired inflow \hat{V}_1 . The outflow is a function of the fluid level in the tank. We assume the following simple valve equation applies:

$$\hat{V}_2 := c \sqrt{h(t)}, \quad (1)$$

with c being the valve constant, which is what is manipulated and $h(t)$ the level in the tank as a function of time.

Question: Simple model of a fermenter

Yeast fermentation of sugar and starch is one of the main processes for generating ethanol, which today is also called a biofuel. The fermentation is usually done in a tank with mild stirring. The cell is taking up the nutrient and ejects the waste, in this case we look at only ethanol and acidic acid as being the fermentation products. Not knowing what the reaction really is, we use here a simplified version:



The individual yeast cell is the actual reactor. The tank act purely as container for the nutrient solution with the yeast-reactors flowating about, taking up nutrient, here sugar and ejecting the ethanol and the sideproduct acidic acid.

The nutrient is diffusing to the surface of the cell membrane, then through the membrane and finally into the cell bulk, where the reaction is catalyse and the products diffuse in the oposite direction into the tank contents, where they accumulate. Thermal effects are neglected.

- Establish a simple picture of the plant
- Generate a physical topology assuming diffusion for the transfer systems
- Simplify physical topology assuming fast transfer system for the diffusion processes
- Colour it with the species
- Generate the component mass balances for the tank and a represenatitive yeast cell (vector equations)
- Look carefully at the transfer system and suggest at least one appropriate model for that part.
- Establish model for the distributed transfer system
- Establish model for the fast transfer system
- Establish model for reactions
- What additional definitions and transformation do you require ? Add them to the equation system and check for the degree of freedom you have in your set of equations.

1 Suggested solution: Potato starch plant

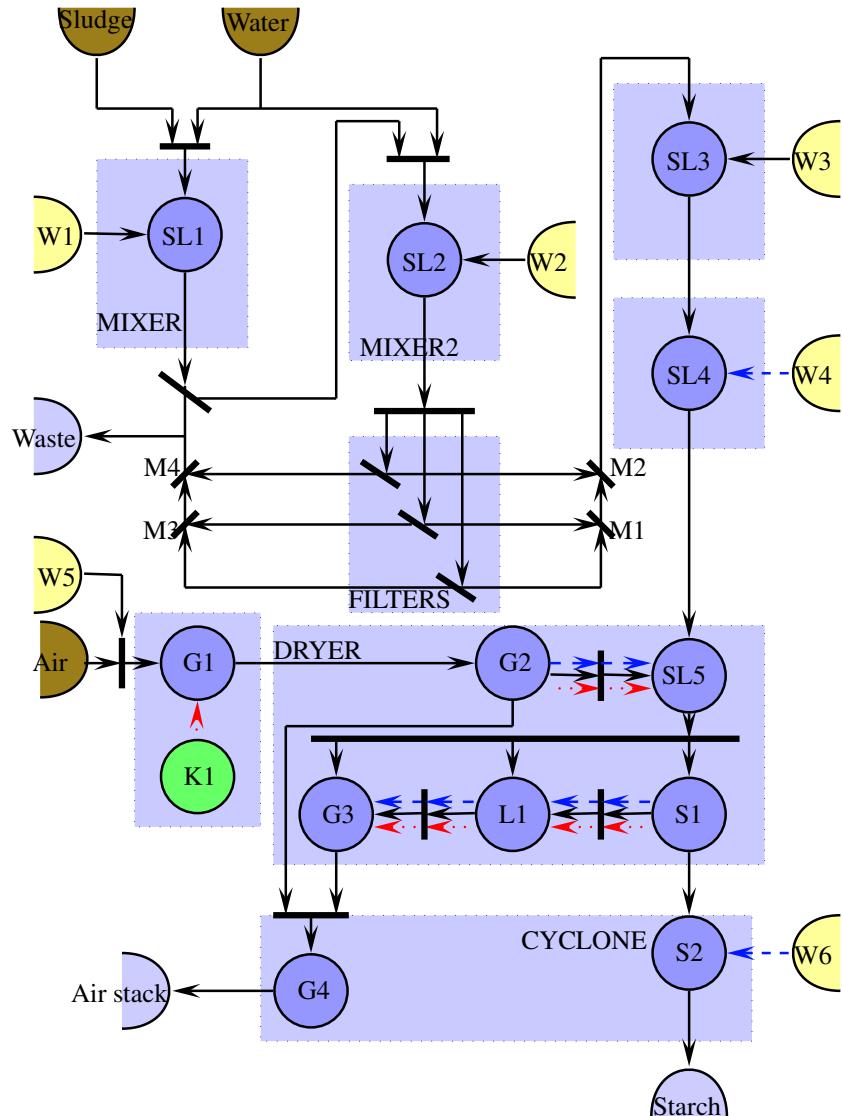


Figure 1: Topology of the potato starch plant

2 Suggested Solution

The dynamic mass balance and the molar flows are then :

$$\frac{d}{dt}n(t) = \hat{n}_1 - \hat{n}_2 \quad (1)$$

$$\hat{n}_m := \rho \hat{V}_m ; \quad m := 1, 2 \quad (2)$$

$$\hat{V}_2 := c \sqrt{h(t)} \quad (3)$$

Note: the density ρ is here given in moles per volume. With these definitions the differ-

ential equation expands to :

$$\frac{d}{dt}n(t) = \rho \hat{V}_1 - \rho c \sqrt{h(t)} \quad (4)$$

where we use the molar density.

The common measurement available is the level in the tank. Thus the objective of the remaining derivation is to recast the dynamics now represented in mass in terms of the level. This a state variable transformation from mass to level is being sought.

We start with defining a relation between the mass and the next-associated geometrical variable, namely the volume:

$$n(t) := \rho V(t) \quad (5)$$

The volume is the integral of the area over the height:

$$V(h(t)) := \int_0^{h(t)} A(\xi) \xi \quad (6)$$

Next the area is linked to the radius:

$$A(r) := \pi r^2 \quad (7)$$

Missing is the relation between radius and height, which can be found using geometrical arguments, specifically Pythagoras. The relation between r and the ξ co-ordinate is:

$$r^2 + (\xi - R)^2 = R^2 \quad (8)$$

The variable transformation is thus using the chain rule:

$$\frac{dn(t)}{dt} := \rho \frac{d}{dt} \left(\int_0^{h(t)} A(\xi) d\xi \right), \quad (9)$$

$$:= \rho \frac{\partial}{\partial h} \left(\int_0^{h(t)} A(\xi) d\xi \right) \frac{dh}{dt}, \quad (10)$$

$$:= \rho A(h(t)) \frac{dh(t)}{dt}, \quad (11)$$

$$:= \rho \pi r^2 \frac{dh(t)}{dt}, \quad (12)$$

$$:= \rho \pi (R^2 - (h - R)^2)^2 \frac{dh(t)}{dt}, \quad (13)$$

$$:= \rho \pi (2 h R - h^2) \frac{dh(t)}{dt}, \quad (14)$$

$$(15)$$

Which results the balance equation :

$$\rho A(h(t)) \frac{d}{dt} h(t) = \rho \hat{V}_1 - \rho c \sqrt{h(t)} \quad (16)$$

Defining the state and the input

$$x(t) := h(t) \quad (17)$$

$$u(t) := \hat{V}_1 \quad (18)$$

gives the model in the desired state-space representation :

$$\rho A(x(t)) \frac{d}{dt} x(t) = \rho u(t) - \rho c \sqrt{x(t)} \quad (19)$$

$$\frac{d}{dt} x(t) = \frac{u(t) - c \sqrt{x(t)}}{\pi x(t) (-x(t) + 2R)} \quad (20)$$

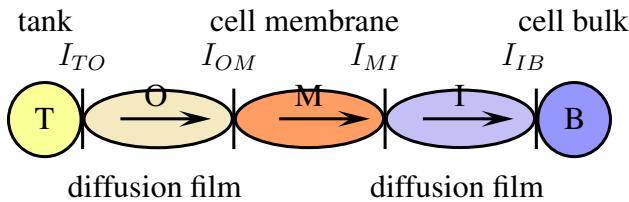
This model is obviously non-linear. Linearisation around a point x_o, u_o yields the two matrices for the locally linearised model :

$$\underline{\underline{A}} := 1/2 \frac{c}{x_o^{3/2} \pi (x_o - 2R)} + \frac{u_o - c \sqrt{x_o}}{\pi x_o^2 (x_o - 2R)} + \frac{u_o - c \sqrt{x_o}}{\pi x_o (x_o - 2R)^2} \quad (21)$$

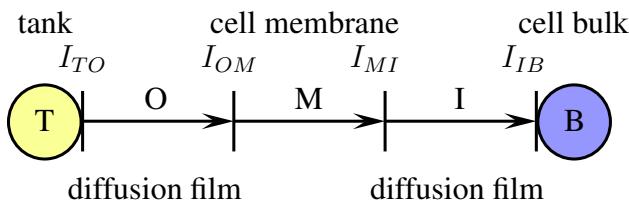
$$\underline{\underline{B}} := -\frac{1}{\pi x_o (x_o - 2R)} \quad (22)$$

3 Suggested Solution: Fermenter

3.1 Simple Topology: distributed transfer



3.2 Simple Topology: event-dynamic transfer



3.3 Simple Topology: event-dynamic transfer - coloured

All part contain all species.

3.4 Model distributed transfer system

Using T for the tank, O for the outer diffusion film, M for the membrane, I for the inner film and B for the bulk, the mass conservation gives

$$\begin{aligned}\dot{\underline{n}}_T &= -\hat{\underline{n}}_{TO-\varepsilon} \\ \underline{0} &= \hat{\underline{n}}_{TO-\varepsilon} - \hat{\underline{n}}_{TO+\varepsilon} \\ \frac{\partial \underline{n}_O}{\partial t} &= -\frac{\partial \hat{\underline{n}}_O}{\partial r} \\ \underline{0} &= \hat{\underline{n}}_{OM-\varepsilon} - \hat{\underline{n}}_{OM+\varepsilon} \\ \frac{\partial \underline{n}_M}{\partial t} &= -\frac{\partial \hat{\underline{n}}_M}{\partial r} \\ \underline{0} &= \hat{\underline{n}}_{MI-\varepsilon} - \hat{\underline{n}}_{MI+\varepsilon} \\ \frac{\partial \underline{n}_I}{\partial t} &= -\frac{\partial \hat{\underline{n}}_I}{\partial r} \\ \underline{0} &= \hat{\underline{n}}_{IB-\varepsilon} - \hat{\underline{n}}_{IB+\varepsilon} \\ \dot{\underline{n}}_B &= \hat{\underline{n}}_{IB-\varepsilon} + \tilde{\underline{n}}_B\end{aligned}$$

Using Fick's first law for the description of the transport

$$\hat{\underline{n}}_r := -\underline{k}_r \frac{\partial \underline{c}}{\partial r}$$

In this case one would have a jump at the interfaces. The alternative is to use the chemical potential as the effort variable, which then must be computed from the component mass vector. In any case, the chemical potential comes into the model, either directly as effort variable or as part of the computation of the jump at the interface.

3.5 Model of fast transfer system

Now the model has only two capacities and three-in-series transfer system. Taking the connection equations out and slightly change the notation, we get:

$$\begin{aligned}\dot{\underline{n}} &= -\hat{\underline{n}}_{T|O} \\ \underline{0} &= \hat{\underline{n}}_{T|O} - \hat{\underline{n}}_{O|I} \\ \underline{0} &= \hat{\underline{n}}_{O|I} - \hat{\underline{n}}_{I|B} \\ \dot{\underline{n}} &= \hat{\underline{n}}_{I|B} + \tilde{\underline{n}}_B\end{aligned}$$

For the transport the same decision has to be taken, only that the driving force is now a difference

$$\hat{\underline{n}}_{a|b} := -\underline{K}_{A|B} (\underline{\mu}_B - \underline{\mu}_A)$$

The additional equations are:

$$\text{chem potential} \quad \underline{\mu} := \underline{\mu}^o + RT \log x$$

base chem potential	$\underline{\mu}^o :: \text{given}$
temperature	$T :: \text{given}$
mole fraction	$\underline{x} := n^{-1} \underline{n}$
total mass	$n := \underline{e}^T \underline{n}$
one vector	$\underline{e} := [1, 1, \dots, 1]^T$

For both lumped systems:

\underline{e} given and \underline{n} from integration knowing the initial conditions $\rightarrow n$

$\rightarrow \underline{x}$

$\rightarrow T$ given and $\underline{\mu}^o$ given R and mole fraction from above $\rightarrow \underline{\mu}$

\rightarrow given $\underline{\mathbf{K}}_{A|B} \rightarrow \hat{\underline{n}}_{a|b}$

\rightarrow mass balances

\rightarrow with initial conditions given, integration closes the loop.

Numerical Integration (TKP4106)



Zooball/Giraffe

"Another Glitch in the Call"

We don't need no indirection
We don't need no flow control
No data typing or declarations
Hey! You! Leave those lists alone!

Chorus:

All in all, it's just a pure-LISP function call.
All in all, it's just a pure-LISP function call.

...

We don't need no...

Assignments

1. Finish the equation solver `hpn_vs_tvn_solver()` in [flowsheet.py](#).
2. Run [ammonia_reactor.py](#) from the command line:

```
python ammonia_reactor.py rk2 explicit 12 1
python ammonia_reactor.py rk2 explicit 12 3
python ammonia_reactor.py rk2 implicit 12 30
python ammonia_reactor.py rk4 explicit 12 1
python ammonia_reactor.py rk4 explicit 12 3
python ammonia_reactor.py rk4 implicit 12 30
```

3. Finish the Euler integration option in method `hpn_vs_tvn_integrator()` in [flowsheet.py](#).
4. Run [ammonia_reactor.py](#) from the command line:

```
python ammonia_reactor.py euler explicit 12 1
python ammonia_reactor.py euler explicit 12 3
python ammonia_reactor.py euler implicit 12 30
```

5. Compare the results you've got.

Continue reading about [Modelling issues](#) with focus on Euler and Runge-Kutta integration.

%Predefined.

HTML text.

5.19.1 Verbatim: “flowsheet.py”

```
1  """
2  @summary: Flowsheet module. UnitParentClass is an 'abstract' class used for
3      implementing features that are common to all unit operations (so far
4      Stream and Reactor). Common features are (in regular Python syntax):::
5
6      obj['variable_name']          # __getitem__('variable_name')
7      obj['variable_name'] = value # __setitem__('variable_name', value)
8      obj()                      # __call__()
9      print obj                  # __str__()
10     obj.component_list()       # [(name, formula), ...]
11     obj.connect(another_obj)   # obj[var_t] = another_obj[var_t], ...
12     obj.functor(name, fun, args) # obj.name(*args) => fun(z, *args)
13
14     The module also contains a collection of functions for calculating the
15     pressure drop, heat exchange, kinetics, Jacobian matrix, etc. of a
16     unit operation object.
17
18     @author:      Tore Haug-Warberg
19     @organization: Department of Chemical Engineering, NTNU, Norway
20     @contact:    haugwarb@nt.ntnu.no
21     @license:    GPLv3
22     @requires:   Python 2.3.5 or higher
23     @since:      2011.10.04 (THW)
24     @version:    0.5
25     @todo 1.0:   Finish methods arrhenius(), tubeandshell()
26     @change:     started (2011.10.04)
27     @note:
28 """
29
30 import srk_ammonia
31 import math
32
33 # Unit operation parent class. It should have been an abstract class (that is a
34 # class without a constructor), but this is not straightforward in Python. Note
35 # that 'UnitParentClass' represents a thermodynamic state object, it is *NOT* a
36 # flow object since there is no concept of time here.
37 class UnitParentClass:
38     '''Base class for unit operation objects.'''
39     def __init__(self, tag, module, component_list):
40         self.model = module.Model(component_list)
41         self.tag = tag
42         self.module = module
43         self.functors = {}
44
45     def __getitem__(self, key):
46         return self.model[key]
47
48     def __setitem__(self, key, val):
49         self.model[key] = val
50         return None
51
52     def __call__(self, **args):
53         return self.model(**args)
54
55     def __str__(self):
56         return " " + self.tag + " ; " + str(self.model)
57
58     def get_cfw(self):
59         return self.model.get_cfw()
60
61     def get_module(self):
62         return self.module
63
64     def connect(self, arg):
65         self.model['var_t'] = arg.model['var_t']
66         self.model['var_v'] = arg.model['var_v']
67         self.model['var_n'] = arg.model['var_n']
68         self.model()
69         for (name,fun) in arg.functors.iteritems():
```

```

70         setattr(self.__class__, name, fun)
71
72     def functor(self, *args):
73         fun = lambda self, x=None: args[1](self, x, *args[2])
74         setattr(self.__class__, args[0], fun)
75         self.functors[args[0]] = fun
76         return self
77
78     def duplicate(self, tag, arg={}):
79         component_list = [name for (name, formula, mw) in self.get_cfw()]
80         module = self.get_module()
81         obj = self.__class__(tag, module, component_list)
82         obj.connect(self)
83         return obj
84
85
86     # Derived process Stream class.
87     class Stream(UnitParentClass):
88         '''Syntactic sugar.'''
89         pass
90
91     # Derived chemical Reactor class.
92     class Reactor(UnitParentClass):
93         '''Syntactic sugar.'''
94         pass
95
96     # Global functions used in reactor simulation. Connect to UnitParentClass object
97     # using so-called 'lambda'-functions, see method 'functor()' in this file.
98     def constantpdrop(obj, z, dp):
99         """
100         Constant pressure drop (dp/dz = constant) along the unit.
101         @param obj: unit operation object
102         @param z: axial position
103         @param dp: pressure drop [kbar] per reactor length
104         @type obj: aUnitParentClass
105         @type z: aFloat
106         @type dp: aFloat
107         @return: aFloat
108         """
109         return dp
110
111     def constantcooling(obj, z, duty):
112         """
113         Constant heat transfer (dQ/dz = constant) along the unit.
114         @param obj: unit operation object
115         @param z: axial position
116         @param duty: heat transfer [1.0e5 J] per reactor length
117         @type obj: aUnitParentClass
118         @type z: aFloat
119         @type duty: aFloat
120         @return: aFloat
121         """
122         return duty
123
124     def tubeandshell(obj, z, ua, t0):
125         """
126         Heat transfer calculation for a 'tube-and-shell' heat exchanger.
127         @param obj: unit operation object
128         @param z: axial position
129         @type obj: aUnitParentClass
130         @type z: aFloat
131         @return: aFloat
132         """
133         return ua*(t0 - obj['state_t'])
134
135     def constanrate(obj, z, nmat, k):
136         """
137         Constant reaction rate (r = constant) along the unit.
138         @param obj: unit operation object
139         @param z: axial position
140         @param nmat: reaction stoichiometry matrix
141         @param k: extent of reactions (one for each column of nmat)

```

```

142     @type obj: aUnitParentClass
143     @type z: aFloat
144     @type nmat: aList [ aList [ aFloat, aFloat, ... ] ]
145     @type k: aList [ aFloat, aFloat, ... ]
146     @return: aList [ aFloat, aFloat, ... ]
147     """
148     return [sum([nui*ki for (nui, ki) in zip(nu, k)]) for nu in nmat]
149
150 def firstorder(obj, z, nmat, keyc, k):
151     """
152     First order kinetics with respect to given 'key' components.
153     @param obj: unit operation object
154     @param z: axial position
155     @param nmat: reaction stoichiometry matrix
156     @param keyc: key components (one for each column of nmat)
157     @param k: rate constants (one for each column of nmat)
158     @type obj: aUnitParentClass
159     @type z: aFloat
160     @type nmat: aList [ aList [ aFloat, aFloat, ... ] ]
161     @type keyc: aList [ anInt, anInt, ... ]
162     @type k: aList [ aFloat, aFloat, ... ]
163     @return: aList [ aFloat, aFloat, ... ]
164     """
165     return [\n
166         sum([nui*obj['state_n'][ci]*ki for (nui, ci, ki) in zip(nu, keyc, k)]) \n
167         for nu in nmat\n
168     ]
169
170 def arrhenius(obj, z, nmat, keyc, k, a, t0):
171     """
172     Arrhenius chemical reaction kinetics.
173     @param obj: unit operation object
174     @param z: axial position
175     @type obj: aUnitParentClass
176     @type z: aFloat
177     @return: aList [ aFloat, aFloat, ... ]
178     """
179     return [\n
180         sum([nui*(math.exp(-a/obj['state_t'])/obj['fix_rgass'])/math.exp(-a/t0/obj['fix_rgass']))*obj['st\n
181         for nu in nmat\n
182     ]
183
184 # Matrix-like thermodynamic state functions. Explicit in temperature, volume and
185 # mole numbers.
186 def hpn_vs_tvn_jacobian(obj, null=None):
187     """
188     Thermodynamic Jacobian of d(H,p,N1,N2,...)/d(T,V,N1,N2,...) calculated as
189     [ [ dH/dT, dH/dV, dH/dN1, ... ], [ dp/dT, ... ], ... ].
190     @param obj: unit operation object
191     @param null: not used
192     @type obj: aUnitParentClass
193     @type null: anObject
194     @return: aList [ aList [ aFloat, aFloat, ... ] ]
195     """
196     nc = len(obj['state_n'])
197     dh = [obj['state_h_t']] + [obj['state_h_v']] + obj['state_h_n']
198     dp = [obj['state_p_t']] + [obj['state_p_v']] + obj['state_p_n']
199     dn = [\n
200         [obj['state_n_t'][i]] +\n
201         [obj['state_n_v'][i]] +\n
202             obj['state_n_n'][i*nc:(i+1)*nc] for i in xrange(0, nc)\n
203     ]
204     return [dh] + [dp] + dn
205
206 def hpn(obj, null=None):
207     """
208     Thermodynamic constraint function [ [H], [p], [N1], [N2], ... ].
209     @param obj: unit operation object
210     @param null: not used
211     @type obj: aUnitParentClass
212     @type null: anObject
213     @return: aList [ [ aFloat ], [ aFloat ], ... ]

```

```

214 """
215     return [[obj['state_h']]]) + \
216         [[obj['state_p']]]) + [[ni] for ni in obj['state_n']]
217
218 # Enthalpy, pressure, composition solver. No fall-back solution for erroneous
219 # thermodynamic calculations (cross your fingers). This is quite easy to program
220 # but it causes a mild code bloat and is left as an exercise for the interested
221 # reader.
222 import tkp4106
223
224 def hpn_vs_tvn_solver(obj, y1, eps, maxiter=50):
225 """
226     Thermodynamic equation solver. Iterates on 'tvn' = (T,V,N1,N2,...) to meet a
227     given specification 'y1' = (H,p,N1,N2,...).
228     @param obj: unit operation object
229     @param y1: [[H],[p],[N1],[N2],...] specification
230     @param eps: convergence criterion (upper bound)
231     @param maxiter: maximum number of iterations (negative value implies a fixed
232                     number of iterations).
233     @type obj: aUnitParentClass
234     @type y1: aList [ aList [ aFloat, aFloat, ... ] ]
235     @type eps: aFloat
236     @type maxiter: anInt
237     @return: aUnitParentClass
238 """
239     converged = False                                # convergence flag
240     norm = 1.0                                         # convergence control variable
241     nc = len(obj['state_n'])                         # number of chemical components in mixture
242     ni = 0                                            # number of iterations
243     while not converged:
244         ni += 1
245         dy = pass                                     # y1 - (h,p,n)
246         dx = tkp4106.solve(obj.jac(), dy)
247         tmp = max([abs(dxi[-1]) for dxi in dx])
248         converged = tmp < eps and tmp >= norm or (ni+maxiter) == 0
249         norm = tmp
250         if maxiter > 0:
251             print "norm=%8.3g;%s;" % (norm, obj)
252         if not converged and ni >= abs(maxiter):
253             raise ArithmeticError("maxIterations(%s) exceeded" % (ni,))
254         obj['var_t']+= pass                            # dt
255         obj['var_v']+= pass                           # dv
256         obj['var_n'] = pass                          # dn_i
257         obj()
258
259     return obj
260
261 # Numerical integration of enthalpy, pressure and composition problems. With or
262 # without chemical reactions.
263 def hpn_vs_tvn_integrator(method, obj, z0, z1, nz):
264 """
265     Thermodynamic integrator using Euler, RK2 or RK4 methods. Both explicit and
266     implicit update schemes are possible. The lambda function 'obj.update()' is
267     supposed to exist and is used to iterate on 'tvn' = (T,V,N1,N2,...) to meet
268     a given specification 'y1' = (H,p,N1,N2,...) in one or more iterations. One
269     iteration means an explicit update. Iteration till full convergence is also
270     possible. This is the implicit update. In calculating the right side of the
271     differential equation three other lambda functions must exist: These are
272     'obj.heatexchange()', 'obj.pressureprofile()' and 'obj.kinetics()' .
273     @author: Stud. Techn. Stig-Erik Nogva
274     @organization: Department of Chemical Engineering, NTNU, Norway
275     @param method: 'euler', 'rk2' or 'rk4'
276     @param obj: unit operation object
277     @param z0: start of integration
278     @param z1: end of integration
279     @param nz: number of integration steps
280     @type method: aString
281     @type obj: aUnitParentClass
282     @type z0: aNumber
283     @type z1: aNumber
284     @type nz: aNumber
285     @return: theUnitParentClass

```

```

286 """
287     objs = []                      # utility list (Runge-Kutta needs intermediate states)
288     dz = float(z1-z0)/nz           # integrator step size
289
290     for z in [z0+k*dz for k in xrange(0, nz)]:
291
292         # Calculate right side of ODE on the dot(y) = y(z) form.
293         yz = [obj.heatexchange(z)] + \
294             [obj.pressureprofile(z)] + obj.kinetics(z)
295
296     if method == 'euler':
297         y1 = pass                      # (h,p,n) + yz*dz
298
299     elif method == 'rk2':
300         while len(objs) < 2:
301             tmp = obj.duplicate('RK2_'+str(len(objs))) # 1 intermediate obj
302             objs.append(tmp)
303
304         for i in range(0, len(objs)):
305             objs[i].connect(obj)      # connect to master object in every step
306
307         # Obtain 1 auxiliary quantity
308         k1 = [yzi*dz for yzi in yz]
309         yk2 = [[yi[-1]+k1i] for (yi, k1i) in zip(objs[0].hpn(), k1)]
310         objs[0].update(yk2)          # iterate on the intermediate state
311
312         yz2 = [objs[0].heatexchange(z+1.0*dz)] + \
313             [objs[0].pressureprofile(z+1.0*dz)] + \
314                 objs[0].kinetics(z+1.0*dz)
315         k2 = [yzi*dz for yzi in yz2]
316         k = [k1i+k2i for (k1i, k2i) in zip(k1, k2)]
317
318         y1 = [[yi[-1]+(1/float(2))*ki] for (yi, ki) in zip(obj.hpn(), k)]
319
320     elif method == 'rk4':
321         while len(objs) < 4:
322             tmp = obj.duplicate('RK4_'+str(len(objs))) # 3 intermediate objs
323             objs.append(tmp)
324
325         for i in range(0, len(objs)):
326             objs[i].connect(obj)      # connect to master object in every step
327
328         # Obtain the 4 auxiliary quantities
329         k1 = [yzi*dz for yzi in yz]
330         yk2 = [[yi[-1]+0.5*k1i] for (yi, k1i) in zip(objs[0].hpn(), k1)]
331         objs[0].update(yk2)          # iterate on intermediate state 1
332
333         yz2 = [objs[0].heatexchange(z+0.5*dz)] + \
334             [objs[0].pressureprofile(z+0.5*dz)] + \
335                 objs[0].kinetics(z+0.5*dz)
336         k2 = [yzi*dz for yzi in yz2]
337         yk3 = [[yi[-1]+0.5*k2i] for (yi, k2i) in zip(objs[1].hpn(), k2)]
338         objs[1].update(yk3)          # iterate on intermediate state 2
339
340         yz3 = [objs[1].heatexchange(z+0.5*dz)] + \
341             [objs[1].pressureprofile(z+0.5*dz)] + \
342                 objs[1].kinetics(z+0.5*dz)
343         k3 = [yzi*dz for yzi in yz3]
344         yk4 = [[yi[-1]+k3i] for (yi, k3i) in zip(objs[2].hpn(), k3)]
345         objs[2].update(yk4)          # iterate on intermediate state 3
346
347         yz4 = [objs[2].heatexchange(z)] + \
348             [objs[2].pressureprofile(z)] + objs[2].kinetics(z)
349         k4 = [yzi*dz for yzi in yz4]
350         k = [k1i+2*k2i+2*k3i+k4i for (k1i, k2i, k3i, k4i) \
351                         in zip(k1, k2, k3, k4)]
352
353         y1 = [[yi[-1]+(1/float(6))*ki] for (yi, ki) in zip(obj.hpn(), k)]
354
355     else:
356         raise NameError('Method ' + method + " " + 'not implemented yet')
357

```

```
358 # Note: 'y1' is the final [ [H], [p], [N1], ... ] after the step 'dz' is
359 # taken. Lambda function 'obj.update()' is responsible for updating the
360 # thermodynamic state accordingly.
361 obj.update(y1)
362
363 print "z=%5.3f; %s;" % (z+dz, obj)
364
365 return obj
```

5.19.2 Verbatim: “ammonia_reactor.py”

```

1  """
2  @summary:      A simple ammonia reactor calculation illustrating some principles
3  of OOP (Object Oriented Programming) in chemical engineering::
4
5          'feed'           -----      'outlet'
6          ) -----> | ... 'rx' ... | -----> (
7          -----
8
9          The outcome of the study is a converged feed stream and an
10         integrated outlet from the reactor.
11 @author:        Tore Haug-Warberg
12 @organization: Department of Chemical Engineering, NTNU, Norway
13 @contact:       haugwarb@nt.ntnu.no
14 @license:       GPLv3
15 @requires:     Python 2.3.5 or higher
16 @since:        2011.10.04 (THW)
17 @version:      0.6
18 @todo 1.0:      None
19 @change:       started (2011.10.04)
20 @note:         This module defines the reaction chemistry (kinetics) and heat
21         transport for a minimal setup of an ammonia reactor. Nothing very
22         fancy, but there are 7 things to learn (see item numbering in
23         source code). From the command line run this script as::
24
25         >>> python ammonia_reactor.py 'euler|rk2|rk4' \
26                           'implicit|explicit' \
27                           <nz> <maxiter>
28
29         nz      = number of integration steps.
30         maxiter = maximum number of iterations spent on the thermodynamic
31         state calculations. If maxiter < 0 then exactly abs(maxiter)
32         iterations will be used independent of the residuals norm.
33 """
34
35 import srk_ammonia
36 import flowsheet
37 import tkp4106
38
39 # 1) There are 3 thermodynamic objects in action: 'feed', 'rx' and 'outlet'.
40 # Each object represents one - and only one - thermodynamic state. This means
41 # that 'rx', describing a state that varies in space, has to be integrated over
42 # the length over the reactor. The reactor profiles of temperature, pressure,
43 # etc. are lost in the process of integration, however, because 'rx' can keep
44 # only one (1) state at a time. It is of course possible to keep the profiles
45 # in memory as intermediate thermodynamic state objects, but this could easily
46 # be an overkill because explicit Euler integration requires somewhere in the
47 # range of 10,000 - 100,000 steps in order to reach 6 digits precision - which
48 # would eventually bind a substantial block of memory.
49 syngas = ['ammonia', 'nitrogen', 'hydrogen']
50
51 feed   = flowsheet.Stream('Feed', srk_ammonia, syngas)
52 outlet = flowsheet.Stream('Outlet', srk_ammonia, syngas)
53 rx     = flowsheet.Reactor('Rx', srk_ammonia, syngas)
54
55 # Initialize feed stream.
56 feed['var_t'] = 0.7                                # temperature [kK]
57 feed['var_v'] = 1.0                                # volume [dm3]
58 feed['var_n'] = [0.04, 0.24, 0.72]                 # mole fractions
59 feed()                                              # run thermodynamics code
60 feed['var_n'] = [ni/feed['state_mtot']/1e7 for ni in feed['state_n']] # [mol/kg]
61
62 # Re-initialize (change T and V to show extra flexibility).
63 feed(var_t=0.8, var_v=feed['var_v']/feed['state_mtot']/1e7)
64
65 print "Initial %s" % (feed,)
66
67 # 2) The feed stream has a specified pressure p0 whereas most thermodynamic equ-
68 # ations of state are explicit in volume (and temperature and composition). The
69 # relation p(V) = p0 must therefore be solved iteratively (using Newton's

```

```

70 # method in this case).
71 eps = 1.0e-8 # convergence criterion
72 p0 = 0.25 # synthesis pressure [kbar]
73
74 print "\nNewton-Raphson solution of p(v)=p0:"
75
76 converged = False # convergence flag
77 norm = 1.0 # convergence control variable
78
79 # Solve p(v) = p0 using Newton's method. The thermodynamics model respond to the
80 # free variable 'var_v' and calculates pressure 'state_p' and pressure
81 # derivative 'state_p_n'.
82 while not converged:
83     dpdv = pass # Jacobian (1 x 1)
84     dp = pass # pressure residual (1 x 1)
85     dv = tkp4106.solve(dpdv, dp)[0][-1] # volume change (scalar)
86     feed['var_v'] += pass # update the model
87     converged = abs(dv) < eps and abs(dv) >= norm # continue till norm is steady
88     norm = abs(dv) # new norm
89
90     # The model fails if 'var_v' becomes unphysical (negative volume typically).
91     # If this happens we must shorten the iteration step until the model says it
92     # is OK. An exception is raised if the step becomes too small.
93     while not feed():
94         if abs(dv) < eps:
95             raise ArithmeticError("cannot converge p(v)=p0 relation")
96         pass # step back to last successful state
97         pass # reduce the step length
98         pass # try once more
99     print "norm=%8.3g;" % (norm, feed)
100
101 print "\nConverged=%s" % (feed,)
102
103 # 3) Calculate the (atoms x component) matrix and the (components x reactions)
104 # stoichiometry from molecular formulas of the components in the mixture.
105 tmp = [formula for (name, formula, mw) in feed.get_cfw()]
106 amat = tkp4106.atom_matrix(tmp)
107 nmat = tkp4106.null(amat)
108
109 # 4) There is the use of functors in the simulation code. Their meaning is a bit
110 # magic to newbies, but to old-timers they offer a great way of code separation
111 # The key issue is that we can start writing algorithms (an Euler integrator in
112 # this case) requiring a certain functionality (pressure drop, heat exchange
113 # and reaction kinetics), without knowing the exact nature of the underlying
114 # functions. The properties are instead registered in the 'rx' object using so-
115 # called lambda expressions calling the correct function run-time by dereferenc-
116 # ing the function pointer. In effect, the heat exchange, pressure drop and
117 # reaction kinetics can be changed in one place of the code without affecting
118 # the solution algorithm. It yields, in fact, a way of defining the transport
119 # properties externally without changing neither the unit operation class nor
120 # the integration method. The same idiom is also used for defining thermodynamic
121 # state derivatives (the Jacobian). In this case we want to control the exact
122 # meaning of 'y1', 'y2', 'x1', 'x2', etc. in d(y1,y2,...)/d(x1,x2,...).
123 rx.connect(feed)
124
125 # Select a 'key' component for the reaction kinetics. Normalize the correspond-
126 # ing stoichiometric coefficient to -1. Make a shallow copy of matrix row before
127 # doing operations on 'nmat'. The algorithm works for single reactions only.
128 keyc = [name for (name, formula, mw) in rx.get_cfw()].index('nitrogen')
129 piv = list(nmat[keyc])
130 for i in xrange(0, len(nmat)):
131     for j in xrange(0, len(nmat[i])):
132         nmat[i][j] /= -piv[j]
133
134 # Declare transport properties and kinetics for the reactor. Non-linear example.
135 # rx.functor('pressureprofile', flowsheet.constantpdrop, [-0.005]) # dp/dz
136 # rx.functor('heatexchange', flowsheet.tubeandshell, [30.0, 0.28]) # ua*(t-t0)
137 # rx.functor('kinetics', flowsheet.arrhenius, [nmat, [keyc], [4/3.0], 0.1, 0.8])
138
139 # Declare transport properties and kinetics for the reactor. Linear example.
140 rx.functor('pressureprofile', flowsheet.constantpdrop, [0.0]) # dp/dz
141 rx.functor('heatexchange', flowsheet.constantcooling, [-20.0]) # heat [1.0e5 J]

```

```

142 rx.functor('kinetics', flowsheet.firstorder, [nmat, [keyc], [4/3.0]]) # rx rates
143
144 # 5) Interact with the command line reader to get hold of the integrator scheme
145 # and the number of steps required for the integration.
146 import sys
147
148 method, iterator, nz, maxiter = sys.argv[1:]
149 nz, maxiter = int(nz), int(maxiter)
150
151 # Declare a thermodynamic iterator (for use inside the integrator).
152 if iterator == 'implicit':
153     maxiter = abs(maxiter)
154
155 if iterator == 'explicit':
156     maxiter = -abs(maxiter)
157
158 # Declare a thermodynamic function solver and state derivatives for the reactor.
159 rx.functor('update', flowsheet.hpn_vs_tvn_solver, [eps, maxiter]) # state update
160 rx.functor('jac', flowsheet.hpn_vs_tvn_jacobian, []) # Jacobian matrix
161 rx.functor('hpn', flowsheet.hpn, []) # constraint variables
162
163 # 6) Integrate over the reactor using the given integration 'method' and the
164 # given 'iterator' mechanism.
165 print "\n%s.%s.integration_using_%s.steps:" % \
166     (iterator.capitalize(), method.capitalize(), nz)
167
168 flowsheet.hpn_vs_tvn_integrator(method, rx, 0, 1, nz) # integrate from z=0 to z=1
169
170 print "\nIntegrated %s" % (rx,)
171
172 # 7) Calculate the reactor outlet using an analytic solution based on the matrix
173 # exponential of the (constant) ODE coefficient. Let  $y = (h, p, c)$  and  $\dot{y} = C \cdot y$ 
174 # Then  $y(z=1) = \exp(C) \cdot y(z=0)$  where 'expm' is the matrix exponential of C:
175 #
176 #      | 1 Q/p 0 0 0 |
177 #      | 0 1 0 0 0 |
178 # expm = | 0 0 1 nu_0/nu_1(fac - 1) 0 |
179 #      | 0 0 0 fac 0 |
180 #      | 0 0 0 nu_2/nu_1(fac - 1) 1 |
181 #
182 # Here, 'Q' is the heat load, 'p' is the (constant) reactor pressure, 'nu_i' are
183 # stoichiometric coefficients and 'fac' is the resilience factor of the 'key'
184 # component.
185 import math
186
187 outlet.connect(rx) # inherit lambda functions from 'rx'
188 outlet(var_t=feed['var_t'], var_v=feed['var_v'], var_n=feed['var_n']) # re-init
189
190 # Calculate the resilience factor of the 'key' component.
191 fac = math.exp(outlet.kinetics(0)[keyc]/outlet['state_n'][keyc])
192
193 # Calculate the matrix exponential.
194 nc = len(outlet['state_n'])
195 expm = [[float(i==j) for i in xrange(0,nc+2)] for j in xrange(0,nc+2)] # identity
196 expm[0][1] = outlet.heatexchange(0)/outlet['state_p'] # heat transfer
197 expm[2+keyc][2+keyc] = fac # 'key' component resilience
198 for i in [j for j in xrange(0,nc) if j != keyc]:
199     expm[2+i][2+keyc] = nmat[i][-1]/nmat[keyc][-1]*(fac-1.0) # other reactions
200
201 # Calculate the outlet state from  $y(z=1) = \exp(C) \cdot y(z=0)$ .
202 y1 = tkp4106.mprod(expm, outlet.hpn())
203
204 print "\nNewton-Raphson solution of f(h,p,c)=0:"
205
206 flowsheet.hpn_vs_tvn_solver(outlet, y1, eps, 20)
207
208 print "\nConverged %s" % (outlet,)

```

5.19.3 flowsheet.py, see also Sec. 5.19.1

First reference occurs in *flowsheet.py*, see Section 5.19.1 on page 329.

5.19.4 ammonia_reactor.py, see also Sec. 5.19.2

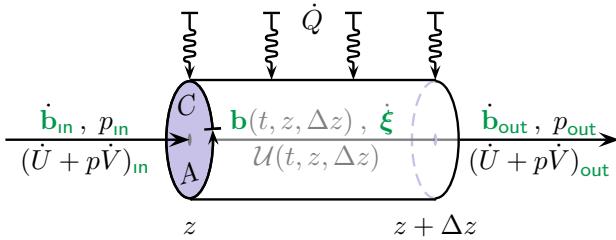
First reference occurs in *ammonia_reactor.py*, see Section 5.19.2 on page 335.

Plug Flow Reactor. Part III

Tore Haug-Warberg
Department of Chemical Engineering
NTNU (Norway)

13 November 2011
(completed after 240 hours of writing, programming and testing)

1 Modelling issues



From an academic perspective the title of this text is a little pretentious. It says “Modelling Issues” which means quite a lot to people devoting their professional lives to the several aspects of chemical reactor calculations, while it means next to nothing for a novice

in the field. Let our perspective be something in between—that of an expert novice maybe. On our behalf then, the idealized plug flow reactor is like the one depicted in the figure. The mass and energy balances for steady state (^{s-s}) operation of the reactor were developed in Parts I and II of this paper. In short we found that:

$$\left(\frac{\partial h \text{ [energy mass}^{-1}]}{\partial z \text{ [length]}} \right)^{\text{s-s}} = C \text{ [length]} q \text{ [heat mass}^{-1} \text{ area}^{-1}]$$

and

$$\left(\frac{\partial c \text{ [mole mass}^{-1}]}{\partial z \text{ [length]}} \right)^{\text{s-s}} = A \text{ [area]} N r \text{ [mole mass}^{-1} \text{ volume}^{-1}]$$

What is missing here is a momentum balance of the reactor. It is needed to resolve the pressure distribution inside the reactor, which of course is of great interest for reactor design and operation, but at the same time it is pulling our wagon too far. The calculations are so involved and require so much input about reactor geometry, transport properties and kinetics that we must do without. Our replacement of the momentum balance is simply:

$$\left(\frac{\partial p \text{ [pressure]}}{\partial z \text{ [length]}} \right)^{\text{s-s}} = \nabla p \text{ [pressure]}$$

That is to say we rely on an explicit pressure profile $p(z)$ given at the outset of the simulation (we shall most of the time use $\nabla p = 0$).

Counting the number of equations there is 1 energy balance, 1 pressure profile and C mass balances. That makes $C + 2$ equations which are going to be solved simultaneously in $C + 2$ variables. The big question is: What variables? In practise we cannot choose the solution variables freely but must tackle whatever needs our models impose on us—*i.e.* the models we use to evaluate h , q and \mathbf{r} —and there is much fuzz about which variables are the most versatile.

Chemical engineers traditionally use T , p , x_1 , x_2 , … that is temperature, pressure and mole fractions. There is no theoretical reason for this choice except that these variables are always reported in process flow diagrams. They are also quite natural in the sense that they play a part of our sensation of the physical world.

Thermodynamicists think differently and usually prefer T , v , c_1 , c_2 , … that is temperature, specific volume and specific concentrations. This choice is natural from a theoretical point of view because most equations of state are given as $p(T, v, \mathbf{c})$ models. By iterating directly on the variables as they appear in the equation of state we can formulate very concise and elegant solvers.

Being trained thermodynamicists and having a keen eye on aesthetics we shall stick to the last alternative even though we then have to *solve* for pressure as a function of volume rather than just specifying it. The equations we need to be solve can be condensed into (see Parts I and II for an explanation of the syntax):

$$\begin{aligned} \text{Energy: } & \partial_T h \cdot \nabla T + \partial_v h \cdot \nabla v + \partial_{c_1} h \cdot \nabla c_1 + \partial_{c_2} h \cdot \nabla c_2 + \dots = Cq \\ \text{Momentum: } & \partial_T p \cdot \nabla T + \partial_v p \cdot \nabla v + \partial_{c_1} p \cdot \nabla c_1 + \partial_{c_2} p \cdot \nabla c_2 + \dots = \nabla p \\ \text{Mass (1): } & \nabla c_1 = A \sum_i \mathbf{N}_{1,i} \mathbf{r}_i \\ \text{Mass (2): } & \nabla c_2 = A \sum_i \mathbf{N}_{2,i} \mathbf{r}_i \\ & \vdots \end{aligned}$$

This set of equations is more easily handled using matrix algebra. To minimize the use of extra symbols $\partial_{\mathbf{c}} h$ and $\partial_{\mathbf{c}} p$ are taken to be row vectors while \mathbf{r} is (still) a column vector:

$$\begin{pmatrix} \partial_T h & \partial_v h & \partial_{\mathbf{c}} h \\ \partial_T p & \partial_v p & \partial_{\mathbf{c}} p \\ 0 & 0 & \mathbf{I} \end{pmatrix} \begin{pmatrix} \nabla T \\ \nabla v \\ \nabla \mathbf{c} \end{pmatrix} = \begin{pmatrix} Cq \\ \nabla p \\ A \mathbf{N} \mathbf{r} \end{pmatrix}$$

The equations above illustrate the ambivalence we are facing with regard to p or v being our primary iteration variable. In this case we shall iterate on v to satisfy ∇p given as the gradient of a predefined function $p(z)$. But, since pressure is a non-linear function of v it implies that ∇p shows up on the *right* side while ∇T , ∇v and $\nabla \mathbf{c}$ appear as *solution* variables on the left side. If p had been a primary iteration variable we could have dropped the second row in the equation set, but at the same time we had to handle the $p(v)$ inversion inside the equation of state. This is a questionable approach because

it involves a nested hierarchy of solvers which can cause all kinds of numerical problems. Usually, it is safer to handle all the equations in one solver, at least so when the equations are few in number like in this case. On a very condensed form we can write

$$\mathbf{J}(\mathbf{x})\nabla\mathbf{x} = \mathbf{f}(z, \mathbf{x}) \quad (1)$$

which is the equation system we have to integrate in order to calculate the temperature and concentration profiles of the reactor. Note carefully that $\mathbf{J}(\mathbf{x})$ is a purely thermodynamic state function while $\mathbf{f}(z, \mathbf{x})$ is a function of both the thermodynamic state variables and the space co-ordinate. The mathematical definitions of \mathbf{J} and \mathbf{f} are not known to us at this point—they are what we might call anonymous *lambda*-functions in functional programming—but their semantic meaning is all clear. E.g. their scientific units most conform¹.

The separation of the problem into \mathbf{J} and \mathbf{f} tells us that the transport and kinetic properties q and \mathbf{r} , used in defining \mathbf{f} on the right side, may require thermodynamic information, while the Jacobian \mathbf{J} is independent of the spatial co-ordinate and of the transport properties. Anyhow, the anti-derivative of the reactor model is

$$\mathbf{x}(z) = \mathbf{x}_o + \int_0^z \mathbf{J}(\mathbf{x})^{-1} \mathbf{f}(\zeta, \mathbf{x}) d\zeta ,$$

and the next question is how we can make an integrator for this problem. Basically, there are three options: Analytic, explicit and implicit solutions. We shall have a look at all three cases. Briefly stated there are few analytical solutions of practical interest, but the few that exist are important for: i) our theoretical insight, and ii) serving as test cases for numerical calculations. For the numerical solutions we must be aware that words like “explicit” and “implicit” have two different meanings. The terms do either refer to how the ODE is formulated, or they refer to how the integration is performed. The distinction is quite subtle and the implementation details are bewildering—these are the combinations we shall look at:

- Explicit ODE with explicit Euler integration (forward Euler).
- Implicit ODE with (semi)implicit Euler integration (backward Euler).
- Explicit ODE with explicit Runge–Kutta integration.
- Implicit ODE with explicit Runge–Kutta integration.

From a practical point of view it is easier to implement the explicit *solvers* compared to the implicit ones, but at the same time they are numerically unstable. This is a classic result from numerical mathematics which we should know about, but which is not so important for the PFR we are studying. What we shall see is that the explicit *model* formulation fails to conserve (even explicit) constraints in energy and pressure, while the implicit formulation does this to our satisfaction.

¹It also means that $\mathbf{f}(z, \mathbf{x})$ and $\mathbf{f}(z, \mathbf{x}(\mathbf{y}))$, and $\mathbf{f}(z, \mathbf{y})$, shall refer to the same kind of function in this document. The free variables change, and the function definitions need not be the same, but the function values are always interpreted as the gradient in specific enthalpy, pressure and composition.

1.1 Analytic solutions

Equation 1 is written with the variables $\mathbf{x} \triangleq [T, v, \mathbf{c}]$ in mind but it applies equally well to any other set of thermodynamic state variables yielding an invertible Jacobian \mathbf{J} . In particular we could try to replace \mathbf{x} by $\mathbf{y} \triangleq [h, p, \mathbf{c}]$ which yields a much simpler formulation. Note carefully that Jacobian reduces to $\mathbf{J}(\mathbf{y}) \equiv \mathbf{I}$:

$$\nabla \mathbf{y} = \mathbf{f}(z, \mathbf{y}) \quad (2)$$

Now, if $\mathbf{f}(z, \mathbf{y})$ is written as a linear function in \mathbf{y} we have the classic problem of an ordinary differential equation (ODE) with constant coefficients. The standard formulation of the problem is shown below (matrix \mathbf{C} has nothing to do with the circumference C used in the energy balance):

$$\nabla \mathbf{y} = \mathbf{Cy}$$

For PFRs that experience a constant circumference C , constant cross-sectional area A , constant pressure drop ∇p , constant heat flux q , and constant reaction rates \mathbf{r} or first order kinetics $\mathbf{r}_i \propto c_{j(i)}$, we can spell out four different cases of linear differential equations with *constant coefficients*. To keep the algebra as simple as possible—but not simpler—we shall assume one chemical reaction (i.e. $\dim \mathbf{N} = \dim \mathbf{c} \times 1$) and a dimensionless reactor length in the range $z \in [0, 1]$:

$$\begin{aligned} 1) \left\{ \begin{array}{l} \nabla h = 0 \\ \nabla p = \nabla p \\ \nabla \mathbf{c} = \xi \mathbf{N} \end{array} \right. & \quad 2) \left\{ \begin{array}{l} \nabla h = 0 \\ \nabla p = \nabla p \\ \nabla \mathbf{c} = kc_1 \mathbf{N} \end{array} \right. \\ 3) \left\{ \begin{array}{l} \nabla h = q \\ \nabla p = 0 \\ \nabla \mathbf{c} = \xi \mathbf{N} \end{array} \right. & \quad 4) \left\{ \begin{array}{l} \nabla h = q \\ \nabla p = 0 \\ \nabla \mathbf{c} = kc_1 \mathbf{N} \end{array} \right. \end{aligned}$$

Here, ξ means the overall *extent of reaction*, q means the overall *heat transfer* and kc_1 denotes the *first order* reaction *with respect to* component 1 (an arbitrary choice from our side). A textual interpretation of the four cases follows:

Case	Description
1	Adiabatic, fixed pressure drop, fixed extent of reaction
2	Adiabatic, fixed pressure drop, first order reaction
3	Fixed heat load, isobaric, fixed extent of reaction
4	Fixed heat load, isobaric, first order reaction

Behind the terminology of *constant coefficients* there is an implication that the equations can be recast into matrix expressions. This is advantageous from a theoretical perspective because it renders a generic solution of the problem $\nabla \mathbf{y} = \mathbf{Cy}$ where \mathbf{C} takes one

of the four shapes shown below:

$$\mathbf{C}_1 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ \frac{\nabla p}{h} & 0 & 0 & 0 \\ \frac{\xi\nu_1}{h} & 0 & 0 & 0 \\ \frac{\xi\nu_2}{h} & 0 & 0 & 0 \end{pmatrix} \quad \mathbf{C}_2 = \begin{pmatrix} 0 & 0 & 0 & 0 \\ \frac{\nabla p}{h} & 0 & 0 & 0 \\ 0 & 0 & k\nu_1 & 0 \\ 0 & 0 & k\nu_2 & 0 \end{pmatrix}$$

$$\mathbf{C}_3 = \begin{pmatrix} 0 & \frac{q}{p} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & \frac{\xi\nu_1}{p} & 0 & 0 \\ 0 & \frac{\xi\nu_2}{p} & 0 & 0 \end{pmatrix} \quad \mathbf{C}_4 = \begin{pmatrix} 0 & \frac{q}{p} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & k\nu_1 & 0 \\ 0 & 0 & k\nu_2 & 0 \end{pmatrix}$$

Here, we have been assuming a two-component mixture with chemical reaction $\nu_1 A = \nu_2 B$. More components can easily be added without violating the structure of the matrices. The solution(s) can be written

$$\mathbf{y}(z) = e^{z\mathbf{C}}\mathbf{y}(0)$$

where $e^{z\mathbf{C}}$ means the *matrix exponential* of $z\mathbf{C}$. Covering the matrix theory in detail would take us astray from the PFR subject, but it is important to know that what is said next *can* be formalized—if not always as closed analytical formulas—at least in the form of numerical calculations. But, for the \mathbf{C} -matrices mentioned above we can follow the simple approach and find the matrix exponentials by inspection because the matrices have such simple structures. Writing out solutions of mathematical problems without any further details is somewhat arrogant but I think that in this case it implies less confusion—not more confusion—to do it quick and simple. You should verify the results by backsubstituting into the matrix formula using $\mathbf{y}(0) = [h, p, \mathbf{c}]_{z=0}$ though:

$$e^{z\mathbf{C}_1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{z\nabla p}{h} & 1 & 0 & 0 \\ \frac{z\xi\nu_1}{h} & 0 & 1 & 0 \\ \frac{z\xi\nu_2}{h} & 0 & 0 & 1 \end{pmatrix} \quad e^{z\mathbf{C}_2} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ \frac{z\nabla p}{h} & 1 & 0 & 0 \\ 0 & 0 & e^{zk\nu_1} & 0 \\ 0 & 0 & \frac{\nu_2}{\nu_1} (e^{zk\nu_1} - 1) & 1 \end{pmatrix}$$

$$e^{z\mathbf{C}_3} = \begin{pmatrix} 1 & \frac{zq}{p} & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & \frac{z\xi\nu_1}{p} & 1 & 0 \\ 0 & \frac{z\xi\nu_2}{p} & 0 & 1 \end{pmatrix} \quad e^{z\mathbf{C}_4} = \begin{pmatrix} 1 & \frac{zq}{p} & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & e^{zk\nu_1} & 0 \\ 0 & 0 & \frac{\nu_2}{\nu_1} (e^{zk\nu_1} - 1) & 1 \end{pmatrix}$$

Case 4 is maybe the most interesting for the chemical engineering student since it gives the opportunity to study PFRs with a maximum in temperature along the reactor. The argument is simple: Consider an exothermic first order reaction with constant cooling. A first order reaction means that the reaction rate will decrease monotonically along the reactor. Then, by balancing the heat production in the middle the reactor with the heat

taken away at the same spot it should be clear that excess heat is produced at the inlet and excess cooling is applied at the outlet. The result is a curved temperature profile which of course looks more interesting than a flat one.

1.2 Explicit Euler-integration

Talking about numerical integration the word *explicit* means the differential equations are stated without iterative calculations. So, how can that be arranged for a non-linear problem? The short answer is it cannot, the long answer is we can make piecewise linear approximations to the functions we want to integrate and solve each little sub-problem explicitly. The outcome will not be *the* answer, but merely a numerical approximation to it. There are many things to worry about in such calculations. Numerical accuracy and stability are maybe the most important issues.

We shall not look very deep into the matter but try to understand what happens in a numerical integrator and see how we can formulate the equations in a piecewise manner. Our starting point is Eq. 1:

$$\mathbf{J}(\mathbf{x})\nabla\mathbf{x} = \mathbf{f}(z, \mathbf{x})$$

Inverting \mathbf{J} (yes, we must assume that the Jacobian is invertible—else the problem is thermodynamically inconsistent) yields the explicit formula

$$\nabla\mathbf{x} = \mathbf{J}(\mathbf{x})^{-1}\mathbf{f}(z, \mathbf{x})$$

Then comes the piecewise approximation $\nabla\mathbf{x} \approx (\Delta z)^{-1}\Delta\mathbf{x}$ which is *assumed* to be valid on the range $[z, z + \Delta z]$:

$$\Delta\mathbf{x} = \mathbf{J}(\mathbf{x}_z)^{-1}\mathbf{f}(z, \mathbf{x}_z)\Delta z + \mathcal{O}(\Delta z)^2$$

The truncation error is of second order, that is $\mathcal{O}(\Delta z)^2$, but the integrated answer will not be that accurate because the number of steps taken in the interval is proportional to $(\Delta z)^{-1}$ which means the integration error will be $\mathcal{O}(\Delta z)^2(\Delta z)^{-1} = \mathcal{O}(\Delta z)^1$, that is of first order only. We shall later learn how to implement schemes of higher order, namely the Runge–Kutta integration methods of 2nd and 4th order. From the definition $\Delta\mathbf{x} \hat{=} \mathbf{x}_{z+\Delta z} - \mathbf{x}_z$ we can write the final update formula as:

$$\mathbf{x}_{z+\Delta z}^{\text{e-e}} \hat{=} \mathbf{x}_z + \mathbf{J}(\mathbf{x}_z)^{-1}\mathbf{f}(z, \mathbf{x}_z)\Delta z \quad (3)$$

By applying this formula successively on the integration domain $z \in [0, 1]$ we can calculate the sequence $\mathbf{x}_0, \mathbf{x}_{\Delta z}, \mathbf{x}_{2\Delta z}, \dots$ very easily. Furthermore, it is (almost) evident that \mathbf{x}_{Nz} will converge to the true solution $\mathbf{x}(Nz)$ when $\Delta z \rightarrow 0$ and $N \rightarrow \infty$. But, this requires an infinite number of steps which eventually would take infinite time on a computer. Another problem of the numerical solution is that computers have fixed word lengths. Irrational numbers are *approximated* inside the computer as decimal numbers represented by 16, 32, 64, or 128 bits length. This gives a *round-off* error in (nearly) every multiplication or division that is carried out. There is therefore a trade-off between a smaller Δz to achieve higher accuracy in the updating formula, and a not-so-small Δz to avoid excessive round-off errors (and to reduce the computation time).

1.3 Implicit Euler-integration

Physical theories build on a limited number of conservation laws. For example mass and energy conservation is essentially what lies behind our PFR model. This is the strong point of physics. The weaker part of the theory arises from the lack of appropriate models expressed directly in the conserved properties. This branch of physics belongs to thermodynamics. In our case the conservation laws are made linear in the thermodynamic variables h, p, \mathbf{c} , while in most cases the equation of state serving the calculation of p (and h) is on the form $p(T, v, \mathbf{c})$. Hence, to update the equation of state we need to solve the relationships between T, v and h, p iteratively (the problem is strongly coupled and non-linear). If these relationships are solved at each step taken from z to $z + \Delta z$ the method is said to be implicit. Recall that for the explicit method in Eq. 3 there is no need for an iterative solution because matrix inversion in itself is an explicit method.

Why should we worry about implicit integration then? It sounds complicated and if explicit integration works why bother? The answer is simple, definite and instructive: Explicit integration violates the conservation principle(s) because of the linearization term that is behind Eq. 3. If this feature is considered to be unfortunate we should consider implicit integration. This is because it solves the conservation equations accurately at each step of the integration. It is not to say that the integration is more accurate, it is only consistent. Consistently wrong you might say, but it is not inconsistent.

To write an implicit integrator we need to understand that the conservation laws put constraints on $\mathbf{y} \hat{=} [h, p, \mathbf{c}]$ while the thermodynamics, heat exchange, pressure drop and kinetics models rely on $\mathbf{x} \hat{=} [T, v, \mathbf{c}]$. We must therefore be able to solve the relationship $\mathbf{x}(\mathbf{y})$ by e.g. Newton–Raphson iteration (to obtain second order convergence) in parallel with the integration task. This topic is also known as: Integration on manifolds, geometric integration, or Differential–Algebraic–Equations (DAEs) solving. The starting point is the same as in Eq. 2 except for the implicit relation $\mathbf{x}(\mathbf{y})$ that sits on the right side:

$$\nabla \mathbf{y} = \mathbf{f}(z, \mathbf{x}(\mathbf{y}))$$

Linearization (this time in \mathbf{y}) yields:

$$\Delta \mathbf{y} = \mathbf{f}(z, \mathbf{x}(\mathbf{y}_{z+\Delta z})) \Delta z + \mathcal{O}(\Delta z)^2$$

This is the *fully* implicit formulation of the problem, where “fully” indicates that the right side is evaluated at the next location $z + \Delta z$, i.e. not the current z . Solving this problem with Newton–Raphson iteration is not so easy because it requires derivative information about $\mathbf{f}(z, \mathbf{x}(\mathbf{y}))$. We know very little about the structure of this function and can hardly make anything ready on general terms, but for the relation $\mathbf{x}(\mathbf{y})$ we know a lot. It is a thermodynamic mapping with a fixed structure awaiting only a thermodynamic model to calculate the numbers run-time. We shall therefore restrict ourselves to the following *semi-implicit* formulation of the problem

$$\Delta \mathbf{y} = \mathbf{f}(z, \mathbf{x}(\mathbf{y}_z)) \Delta z + \mathcal{O}(\Delta z)^2$$

where the right side is assumed constant at each position z . This yields the simpler update formula:

$$\mathbf{y}_{z+\Delta z} \doteq \mathbf{y}_z + \mathbf{f}(z, \mathbf{x}(\mathbf{y}_z))\Delta z$$

Even though the formulation above is semi-implicit it is consistent with any conservation principle that yields a constant contribution on the right side (linear profile). The method is therefore referred to as just “implicit” when there is no danger of misunderstanding. Later on we shall see in practise how the method works for a problem with linear enthalpy and pressure profiles. Notwithstanding these merits the semi-implicit method is just an approximation with respect to changes that are *not* subject to conservation. Temperature is one example. So, even when the energy is conserved the temperature profile is not necessarily correct. Incorrect is not the same as inconsistent though.

To solve for $\mathbf{y}_{z+\Delta z}$ we shall alter the values of \mathbf{x} . We must then make some additional calculations denoted as *iterations* $0, 1, \dots, k, k+1$. Because the problem formulation is semi-implicit we need derivatives for \mathbf{y} versus \mathbf{x} but not for $\mathbf{f}(z, \mathbf{x}(\mathbf{y}))$. Linearization of $\mathbf{y}_{z+\Delta z}$ on the left side yields the following approximation:

$$\mathbf{y}_z^k + \mathbf{J}(\mathbf{x}_z^k)\Delta \mathbf{x}^k \approx \mathbf{y}_z^0 + \mathbf{f}(z, \mathbf{x}(\mathbf{y}_z^0))\Delta z$$

By definition $\mathbf{y}_z^0 \equiv \mathbf{y}_z$ and we sincerely hope that $\mathbf{y}_z^\infty \rightarrow \mathbf{y}_{z+\Delta z}$. We cannot prove the last property, but if it is correct the iteration process is said to *converge* locally. The Newton–Raphson procedure may converge or it may diverge. Impossible to say in fact without problem specific information. If it does converge, however, it shows *second order* convergence. In practise this means that the number of *significant* digits will double in each iteration when k is sufficiently large. What *sufficiently* large means is also hard to say, but in normal cases it is typically in the range $k_{\text{crit}} \in [3, 5]$. Solving for $\Delta \mathbf{x}^k$ we get:

$$\Delta \mathbf{x}^k \approx \mathbf{J}(\mathbf{x}_z^k)^{-1} \underbrace{[\mathbf{y}_z^0 + \mathbf{f}(z, \mathbf{x}(\mathbf{y}_z^0))\Delta z - \mathbf{y}_z^k]}_{\mathbf{y}_{z+\Delta z}} \quad (4)$$

Note the underbrace above: $\mathbf{y}_{z+\Delta z}$ comes in as a constant estimate on the right side such that if (i.e. hopefully then) the iteration converges we get $\mathbf{y}_z^k \rightarrow \mathbf{y}_z^\infty \rightarrow \mathbf{y}_{z+\Delta z}$ which makes $\Delta \mathbf{x}^k \rightarrow \mathbf{0}$. Finally, when the update *norm* satisfies $||\Delta \mathbf{x}^k|| \leq \epsilon$ the iteration is stopped. A suitable stop criterion must be set by us—or in practise the programmer. The definition of $\Delta \mathbf{x}^k \doteq \mathbf{x}_z^{k+1} - \mathbf{x}_z^k$ leads to

$$\mathbf{x}_z^{i-e, k+1} \doteq \mathbf{x}_z^k + \mathbf{J}(\mathbf{x}_z^k)^{-1} [\mathbf{y}_{z+\Delta z} - \mathbf{y}_z^k] \quad (5)$$

which is the final update formula for the implicit Euler integration method. But, for the special case $k = 0$ we can identify $\mathbf{y}_{z+\Delta z} - \mathbf{y}_z^k$ on the right side being equal to $\mathbf{y}_{z+\Delta z} - \mathbf{y}_z^0 = \mathbf{f}(z, \mathbf{x}(\mathbf{y}_z^0))\Delta z$, see Eq. 4. This leaves the much simpler formula:

$$\mathbf{x}_z^{i-e, 1} = \mathbf{x}_z^0 + \mathbf{J}(\mathbf{x}_z^0)^{-1} \mathbf{f}(z, \mathbf{x}(\mathbf{y}_z^0))\Delta z$$

Comparing the right side of this formula with the explicit Euler formula in Eq. 3 reveals the following relationship (after noticing that $\mathbf{x}_z^0 \equiv \mathbf{x}_z$ and $\mathbf{y}_z^0 \equiv \mathbf{y}_z$):

$$\mathbf{x}_z^{i-e, 1} \equiv \mathbf{x}_{z+\Delta z}^{e-e}$$

The conclusion is that the first iteration of the implicit Euler scheme is identical to the explicit Euler update (if, and only if, the update is calculated using Newton–Raphson iteration). We can therefore say that the two integration methods are examples of N 'th level explicit Euler schemes. For $N = 1$ we retain the classic Euler integration and for $N \rightarrow \infty$ we get implicit Euler integration, but in many cases it is enough to make only 2 or 3 Newton–Raphson updates in order to reach a sufficiently converged \mathbf{x} -state. Thus, it makes sense to integrate several times trying out 1st, 2nd and 3rd level updates to verify that the solution converges smoothly to a value that is independent of the linearization. What cannot be controlled in this manner is the accuracy of the integration. Usually, higher accuracy means higher order approximation methods like for instance the Runge–Kutta family of non-stiff integrators. To control stiffness as well (that is integrating ODEs showing a wide spread in the eigenvalues) we have to deal with an entirely different approach using variable step length and precondition of the equations. This is way outside the current scope.

1.4 Runge–Kutta integration

The Runge–Kutta methods belong to a family of explicit integrators often considered to be the work horses of numerical integration. The members of this family are characterized by an order parameter n saying that the global integration error is proportional to $(\Delta z)^n$, where n is typically 2, 3, 4 and 5. A Runge–Kutta method of order 1 will then be equivalent to explicit (forward) Euler integration. It can be argued that schemes of even order are better “balanced” than schemes of odd order. The odd-ordered schemes are therefore used for truncation error control, mostly, while the integration itself is carried out with one of the even-ordered schemes.

We shall have a further look at second and fourth order schemes called RK2 and RK4 throughout this text. These are explicit integration schemes, but the methods will be defined such that we can choose to stay on the h, p, \mathbf{c} manifold if we wish. It is then important to know what “on” means. Just like for the explicit and (semi)implicit Euler methods this question does not need be answered once and for all, but can await us specifying (later) the number of iterations we would like to spend on the update of T, v, \mathbf{c} at each step of the integration.

1.5 Calculation example

A good calculation example must serve many needs. Firstly, it should be verifiable. Only this way is it possible to prove (or disprove) that the equations are solved correctly. Secondly, it should be familiar to the reader. An example that comes as a total surprise can hardly serve as an example because the perspective is missing. Thirdly, it should be realistic. An unrealistic example can perhaps be more intriguing but it adds nothing to our physical experience. Forthly, it should contribute new insight. However, to come up with an example that is both verifiable, familiar, realistic and new is not so easy.

The production of ammonia from nitrogen and hydrogen is a classical textbook example. It is the most important of all the industrial reactions and without it we would

have been in the 19th century still. But, it has a very complicated reactor design and we shall not try too hard to be realistic. Uniform cooling, zero pressure drop and first order reaction is the best we can do if we also want to verify the calculation by comparing it with an analytical solution, see also [Section 1.1](#).

The ammonia reaction is exothermic and shows a substantial temperature increase under normal operation. So, by matching the cooling duty with the reaction rate it is possible to obtain a curved temperature profile along the reactor axis. The chemical compositions vary exponentially along the same axis and for the reactor as a whole we can expect a pronounced non-linear behaviour. This puts our solution method on trial. We shall therefore investigate several integration schemes: explicit and implicit Euler, and explicit RK2 and RK4 (Runge–Kutta 2nd and 4th order) with both explicit and implicit function updates.

For the reactor calculation we need of course a set of differential equations, but we also need to fill in with thermodynamic state information. Ideal gas is the simplest non-trivial concept we can use in this case. The gas mixture of ammonia, nitrogen and hydrogen is non-ideal at synthesis conditions, but the physical insight of the problem is not changed very much by this fact. The only artifact we should know about is that the ideal gas enthalpy is independent of pressure whereas the real enthalpy is not (this feature can betray us badly at adiabatic conditions). The thermodynamic relations we are using are listed below:

$$p^{\text{ig}} = \frac{\sum_i c_i RT}{v}$$

$$h^{\text{ig}} = \sum_i c_i (\Delta_f h_i^\circ + \int_{0.29815}^T c_{p,i}^\circ(\tau) d\tau)$$

where $R \triangleq 0.083145 \dots 10^5 \text{ J mol}^{-1} \text{ K}^{-1}$, and where

$$\frac{\Delta_f h_{\text{NH}_3}^\circ}{[10^5 \text{ J mol}^{-1}]} = -0.45898; \quad \Delta_f h_{\text{N}_2}^\circ = \Delta_f h_{\text{H}_2}^\circ = 0$$

and finally:

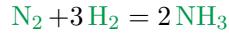
$$\frac{c_{p,\text{NH}_3}^\circ(\tau)}{[10^5 \text{ J mol}^{-1} \text{ K}^{-1}]} = 0.27310 + 0.23830\tau + 0.17070\tau^2 - 0.11850\tau^3$$

$$\frac{c_{p,\text{N}_2}^\circ(\tau)}{[10^5 \text{ J mol}^{-1} \text{ K}^{-1}]} = 0.31150 - 0.13570\tau + 0.26800\tau^2 - 0.11680\tau^3$$

$$\frac{c_{p,\text{H}_2}^\circ(\tau)}{[10^5 \text{ J mol}^{-1} \text{ K}^{-1}]} = 0.27140 + 0.09274\tau - 0.13810\tau^2 + 0.07645\tau^3$$

As explained at the beginning of this chapter the mixture is normalized to one kilogram of material which implies that all enthalpies, volumes and mole numbers are reported as specific quantities in the upcoming tables. This fixes the size of the problem. Everything

is on mass basis. The last statement can be a little bewildering because the reaction stoichiometry is



which is independent of the system size. This equation reflects only the chemical *stoichiometry*, however, and not the total conversion in the system. It is the kinetics model that scales the chemical reaction equation to the size of the system. Now, to integrate through the reactor we need to know the complete intensive state of the gas mixture at the inlet. The initial temperature, pressure and composition (mole fractions) chosen in this case are:

$$\begin{aligned} T_o &= 0.800 \text{ kK} \\ p_o &= 0.250 \text{ kbar} \\ z_o &= [0.04, 0.24, 0.72] [-] \end{aligned}$$

The units of thermodynamics (**kK**, **kbar**, **10^5J** , **dm^3** and **mol**) are maybe curious but they are in fact judiciously selected to increase the numerical stability of the solvers. This issue is hard to explain without the prior knowledge of numerical mathematics and fixed word-length computers and we shall leave it open for the interested reader. Note also that the initial pressure is a dependent variable in this case and that it must be iterated on since the thermodynamic model is explicit in volume—not in pressure. Carrying on we shall assume a uniform cooling profile along the reactor equal to $\nabla h = -20 \text{ } 10^5 \text{J}$, zero pressure drop $\nabla p = 0 \text{ kbar}$, and first order reaction of nitrogen equal to $\nabla c_{\text{N}_2} = -(4/3)c_{\text{N}_2} \text{ mol}$. All gradients are defined per kilogram of material and per reactor length. The outcome is a set of differential equations equivalent to Case 4 in [Section 1.1](#):

$$\nabla \mathbf{y} \hat{=} \nabla \begin{pmatrix} h \\ p \\ c_{\text{NH}_3} \\ c_{\text{N}_2} \\ c_{\text{H}_2} \end{pmatrix} \rightarrow \begin{pmatrix} -20 \\ 0 \\ (8/3)c_{\text{N}_2} \\ -(4/3)c_{\text{N}_2} \\ -(4/1)c_{\text{N}_2} \end{pmatrix}$$

The analytical solution is

$$\mathbf{y}(z) \hat{=} \begin{pmatrix} h \\ p \\ c_{\text{NH}_3} \\ c_{\text{N}_2} \\ c_{\text{H}_2} \end{pmatrix} \rightarrow \begin{pmatrix} h_o - 20z \\ p_o \\ c_{\text{NH}_3}^o - 2(\alpha - 1)c_{\text{N}_2}^o \\ \alpha c_{\text{N}_2}^o \\ c_{\text{H}_2}^o + 3(\alpha - 1)c_{\text{N}_2}^o \end{pmatrix}$$

where $\alpha \triangleq e^{-(4/3)z}$. The enthalpy, pressure and composition profiles are easily calculated from the last formula and by iterating on temperature and volume at each step along the reactor axis (we need in fact only one step to integrate the entire reactor) we can calculate the profiles to our discretion. **E.g.** dividing the reactor into 5 segments yields the following exact answer to our differential equation problem (reported in more familiar units for the ease of reading):

z	$\frac{T}{[\text{K}]}$	$\frac{V}{[\text{dm}^3]}$	$\frac{h}{[\text{MJ}]}$	$\frac{p}{[\text{bar}]}$	$\frac{c_{\text{NH}_3}}{[\text{mol}]}$	$\frac{c_{\text{N}_2}}{[\text{mol}]}$	$\frac{c_{\text{H}_2}}{[\text{mol}]}$
0	800.000	30.0438	1.495255	250.000	4.5168	27.1006	81.3019
0.2	882.267	29.4106	1.095255	250.000	17.2037	20.7571	62.2714
0.4	919.963	27.6941	0.695255	250.000	26.9211	15.8985	47.6954
0.6	921.796	25.4676	0.295255	250.000	34.3638	12.1771	36.5313
0.8	894.927	23.0285	-1.104745	250.000	40.0645	9.3268	27.9804
1	844.596	20.5069	-0.504745	250.000	44.4307	7.1436	21.4309

The numbers printed in blue ink are the variables we want to investigate further using a small assortment of homemade integrators. So, integrating from $z = 0$ to $z = 1$ in 3 steps (numbers being exact to 6 digits are printed in blue) yields:

Method	N	$\frac{T}{[\text{K}]}$	$\frac{V}{[\text{dm}^3]}$	$\frac{h}{[\text{MJ}]}$	$\frac{p}{[\text{bar}]}$
Euler	1	923.156	21.7968	-0.522353	239.498
Euler	3	928.546	21.0031	-0.504745	250.001
RK2	1	828.557	20.4743	-0.507512	248.660
RK2	3	829.427	20.3859	-0.504745	250.000
RK4	1	844.365	20.5106	-0.504997	249.918
RK4	3	844.444	20.5057	-0.504745	250.000
Exact	-	844.596	20.5069	-0.504745	250.000

We see that all the explicit methods fail: Euler-1 fails badly, RK2-1 fails less, while RK4-1 is pretty close—but they all fail. The implicit methods behave differently. Except for Euler-3 they are all correct in their predictions of *enthalpy* and *pressure*. This means the energy and momentum balances are *consistent* with the underlying conservation principles. The temperature and the volume are still off which means the calculations are not correct—only consistent.

By increasing the number of integration steps we may hope to rectify the situation and get truely correct answers. In fact, by integrating from $z = 0$ to $z = 1$ in 12 steps (numbers being exact to 6 digits are still printed in blue) we get:

Method	N	$\frac{T}{[\text{K}]}$	$\frac{V}{[\text{dm}^3]}$	$\frac{h}{[\text{MJ}]}$	$\frac{p}{[\text{bar}]}$
Euler	1	862.454	20.7456	-0.507013	248.550
Euler	3	863.160	20.6421	-0.504745	250.000
RK2	1	843.829	20.5017	-0.504892	249.982
RK2	3	843.875	20.5014	-0.504745	250.000
RK4	1	844.595	20.5069	-0.504746	250.000
RK4	3	844.596	20.5069	-0.504745	250.000
Exact	-	844.596	20.5069	-0.504745	250.000

This time RK4-3 yields correct answers all over the line. The same resolution with RK2-3 and Euler-3 would require 380 and 500,000 steps respectively. Note: The total calculation effort is bigger because one step of RK4-3 requires 4 intermediate steps each

using 3 iterations in Eq. 5. The total number of steps is then $12*4*3 = 144$. For RK2-3 the total number of steps is $360*2*3 = 2160$, and for Euler-3 it is $500,000*1*3 = 1,500,000$. Notwithstanding the extra calculations required to fulfill the RK4 and RK2 steps, the conclusion is that higher order schemes are superior to lower order schemes (of course I should say).

In interesting spin-off from this discussion is that there is no difference between implicit and explicit problem formulations when we talk about numerical accuracy. I.e. explicit Euler and implicit Euler yield the same accuracy as do RK2 with explicit and implicit model formulations and the same for RK4. But then it comes to conservation laws we see the difference. The implicit model formulation always yield correct enthalpies and pressures whereas the explicit formulations do not. For RK4 the difference is in the last digit only, but it is nevertheless present and it is visible.

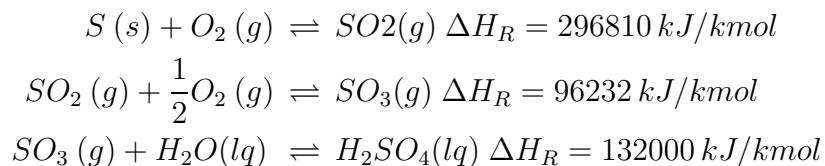
Exercise 10

Preisig, H A

Chemical Engineering, NTNU

1 Question: Topology 07

The simplified flowsheet of the industrial sulfuric acid production process consists of a sulfur burner, multi-pass converter, heat exchangers and absorbers as shown in Figure 1. The main steps in the process consist of burning sulfur (S) in air to form sulfur dioxide (SO_2), converting SO_2 to sulfur trioxide (SO_3) using oxygen (O_2) from air, and absorbing SO_3 in water (H_2O) or a diluted solution of sulfuric acid (H_2SO_4) to form a concentrated solution of acid (> 96%).



Filtered ambient air is drawn through a high efficiency drying tower by the main compressor to remove moisture. The compressed dry air enters a refractory-lined furnace where molten sulfur is burned to produce SO_2 . The hot SO_2 combustion gas is then cooled in a steam boiler to the proper temperature to promote conversion to SO_3 in the conversion step. A multi-bed catalytic adiabatic reactor is used as the SO_2 oxidation reaction is limited by the chemical equilibrium. Note that O_2 oxidizes SO_2 to SO_3 with a catalyst. The catalyst used here is vanadium oxide (V_2O_5) mixed with an alkali metal sulfate. This mixture is supported on small silica beads.

The overall process is designed to give a conversion of sulfur dioxide to sulfuric acid of over 99.7%. Several conversion steps, addition of fresh air and inter-stage cooling are necessary as the reaction is reversible and exothermal. SO_2 conversion is further improved and tail gas emissions are reduced through an intermediate SO_3 absorption step (Abs1). This absorption step takes place after the fourth bed of catalyst and changes the gas composition, thus shifting the equilibrium curve to higher conversions. The absorption of SO_3 is finalized in the second absorber (Abs2). For heat-integration reasons, two feed-effluent heat exchangers (FEHE) are used.

- Sketch the topology of the sulfuric acid plant.

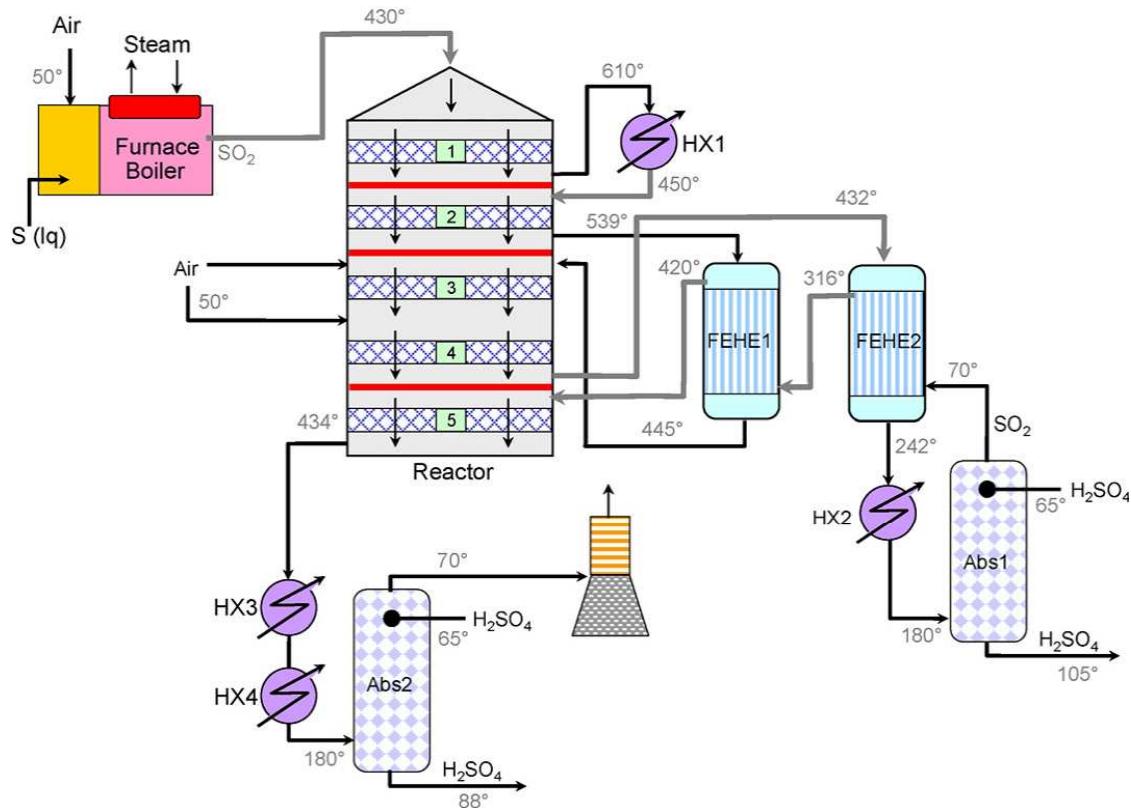


Figure 1: A flowsheet of Acid Sulphuric production plant

2 Question: Modelling a mass transfer

It is often of interest to have a visual indicator for the level of fluid in a tank. For this purpose one attaches a vertical see-through tube to the tank which is connected to a terminal at the bottom of the tank. Often, the connecting pipe is of small diameter. Sometimes it has a valve built in.

It is of interest to explore what the level measurement shows as the tank is operating in a dynamic mode, as the level may change relatively rapidly in the tank due to adding and removing fluid through the feed or the outflow terminal of the tank. The tank has one controlled feed and one output to a variable user. The feed is controlled using an on/off controller as it is common in the toilet flush box using the level glass as a measure for the level.

2.1 Additional information

The volumetric flow rate from the tank through the tube into the level glass is driven by the difference in static pressures at the bottom of the two vessels. The following relation is a good candidate for expressing this rate:

$$\hat{V}_{T|L} := c_{T|L} \operatorname{sign}(p_T - p_L) \sqrt{\frac{|p_T - p_L|}{\rho}} \quad (1)$$

where c represents the valve constant including the resistance effects of the pipe, the system T is for the tank and L for the level glass.

2.2 What to do

2.2.1 Construct Model

It is the level in the tank, which the level glass is indicating. Thus it is the dynamics of the two levels, the one in the tank and the one in the level glass, that are of interest.

Establish a mathematical model strictly in steps and simulate thereafter:

- Sketch the process.
- Draw the physical topology of the tank that is able to describe the process that occurs with the valve being in a position that is different from closed.
- Overlay the control structure.
- Choose the fundamental extensive quantities describing this process.
- Write the balance equations.
- Introduce a state variable transformation form the conserved extensive quantity, mass, to level, thus use the level in the tank and the level in the level glass as the secondary state variables.
- Define the input variables.
- Transform the differential equations to show the dynamics of the level changes.

2.2.2 Cast Model into Systems Notation

The model is to be cast into the systems notation by defining

- state vector \underline{x}
- input vector \underline{u}
- output vector \underline{y}
- parameters θ

2.2.3 Linearise Model

Use the model in the systems notation to obtain a linearised version.

3 Question: Dynamics 04 - ODE integration

We want to write a integrator for solving initial value problems for models that consist of sets differential-algebraic equations (DAE) of a specific class. We want a generic tool for the integration, whereby it shall be possible to implement different methods.

The class of DAEs are of the form:

$$\begin{aligned}\dot{\underline{x}} &= \underline{f}(\underline{z}, \theta_x, t) \\ \underline{z} &:= \underline{g}(\underline{x}, t)\end{aligned}$$

Obviously we could substitute the second algebraic equation into the first one yielding a simple ordinary differential equation. So why using this form? In our applications, namely the modelling of physical-chemical-biological processes, first equation are the conservation equations, which in fact are linear, whilst the second equation is the collection of transport, kinetic, material models and geometry relations that complete the model.

For the start we want to implement two very commonly used methods, namely

1. Euler method
2. Rung-Kutta method (RK4)

To test the scheme, we use a simple linear set of differential equations:

$$\begin{aligned}\dot{\underline{x}} &= \begin{bmatrix} e_1 & a_1 & 0 & 0 \\ -a_1 & e_1 & 0 & 0 \\ c_1 & 0 & e_2 & a_2 \\ 0 & 0 & -a_2 & e_2 \end{bmatrix} \underline{x} \\ x(0) &:= [10; 10; 10; 10] \\ e_1 &:= -0.01 \\ e_2 &:= -0.01 \\ a_1 &:= 1 \\ a_2 &:= 2 \\ c_1 &:= 1\end{aligned}$$

These are two coupled oscillators, so should produce Lissajou figures. Plot for example x_2 vs x_4 , but first plot all states vs time.

1. Generate a class matrix, which extends the class list with the operators +, - and *, whereby the * the normal matrix products. Since vectors are matrix with only one column, this class can be used to do all necessary matrix operations.
2. Realise the Euler and the Runge-Kutta algorithm using the matrix class
3. Produce the solutions for the above system using a sampling time of 0.1 and 1000 points.

4. Plot x_i , $i := 1, \dots, 4$ vs time
5. Phase plot x_2 vs x_4 to visualise the Lissajou figure.
6. Change parameter a_2 to for example 3 or 4 and plot the Lissajou figure.
7. Compare with the analytical solution.

3.1 Hints to the design

Think what objects and operations you need :

- Matrix, a two dimensional objects (A,B)
- Vector, a one dimensional object, which though can be seen as a matrix with one dimension being 1. (v)
- Scalar (a)

Given the sample objects above in brackets and the two algorithms, the following operations may be required:

1. $A + B$
2. $A - B$
3. $A * B$
4. $v * A$ same as $A * B$
5. $A * v$ same as $A * B$
6. $v^T * v$ same as $A * B$
7. $A * a$
8. A/a
9. $a * A$

Thus a matrix class would have to do:

- Matrix transposition
- Matrix addition and subtraction,
- Matrix product (scalar product being a special case)
- Matrix product with scalar
- Scalar times matrix

1 Suggested solution: Sulfuric acid production plant

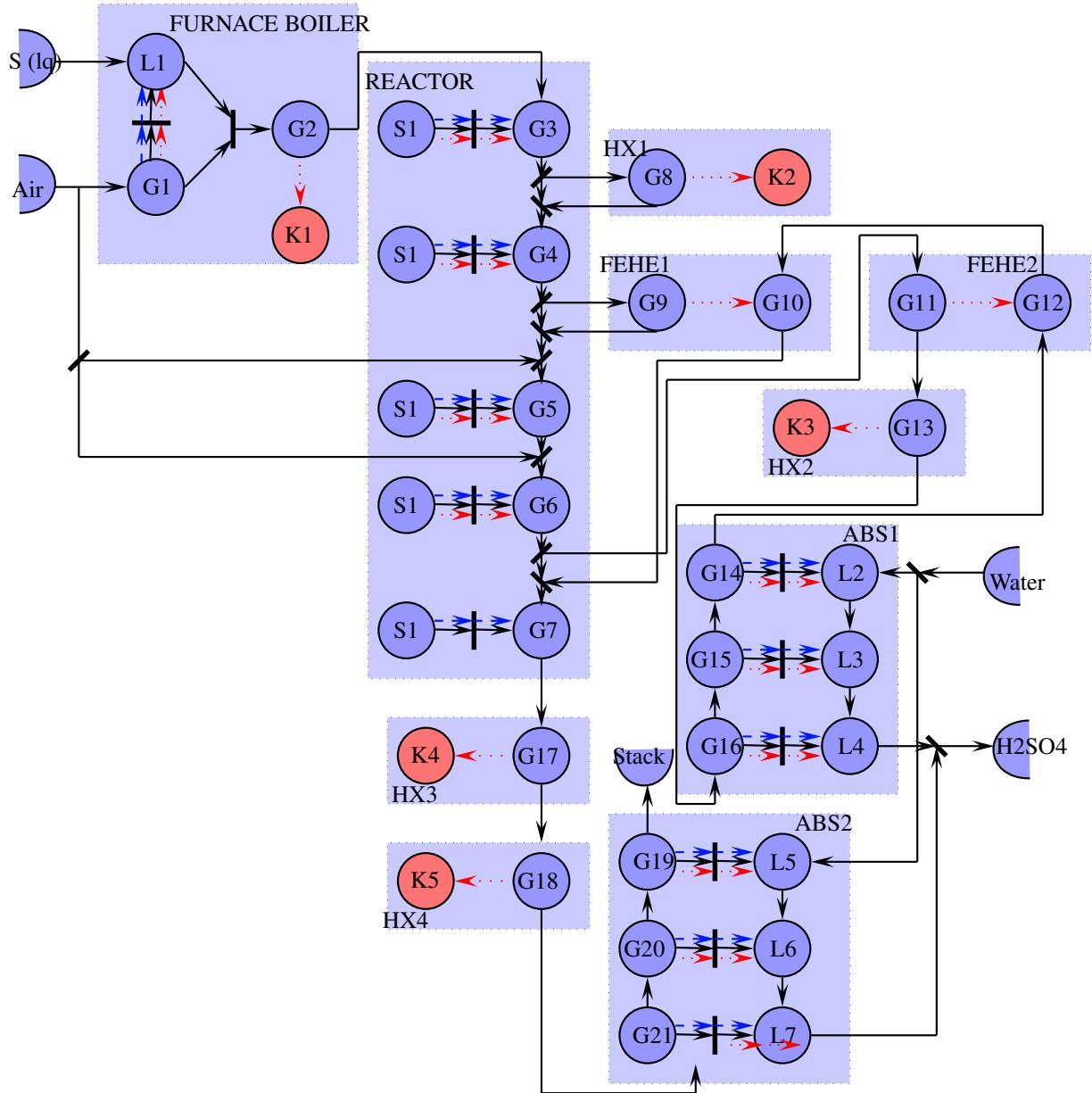


Figure 1: Topology of the Sulfuric acid production plant

2 Solution: Modelling a tank with a level glass indicator

3 Model

3.1 Step 1 : Sketch Process and Establish Physical Topology

The process of the level glass attached to a tank is shown in Figure 3.1 together with a suggested physical topology.

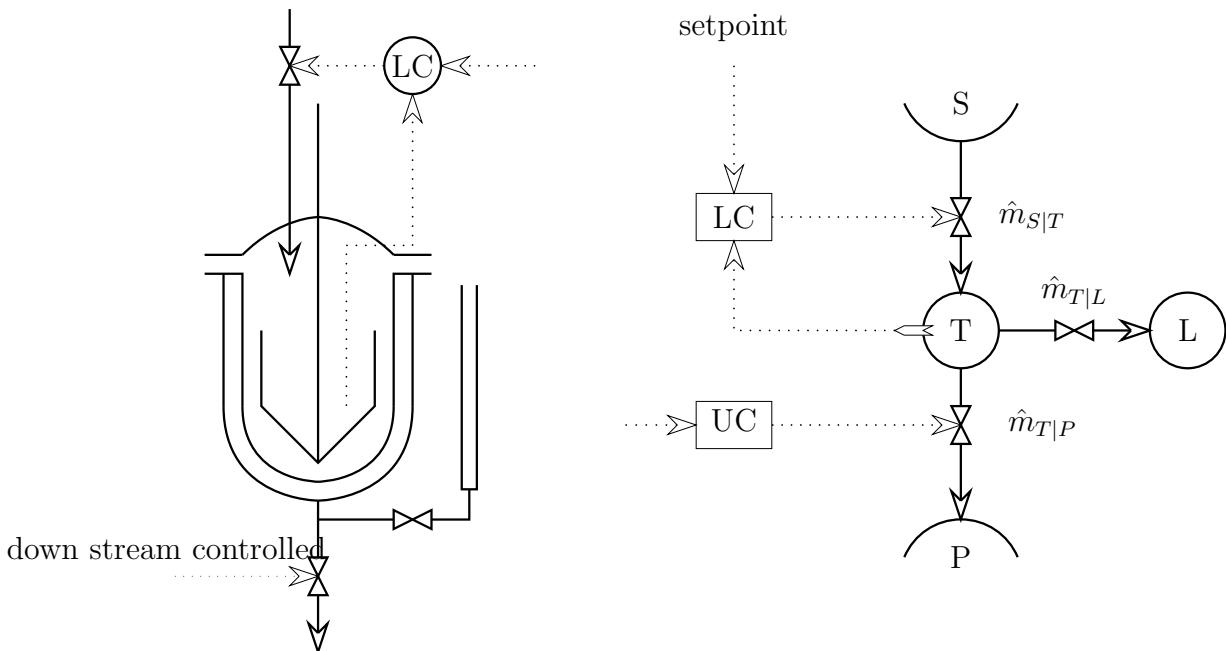


Figure 2: *Process structure and its abstraction*

3.2 Step 2 : Primary Model

Since it is the objective to evaluate how well the level in the level glass reflects the level in the tank, only the total mass balance of the tank and the level glass are required, assuming no significant thermal and concentration effects are predicted.

$$\frac{dm_T}{dt} = \hat{m}_{S|T} - \hat{m}_{T|P} - \hat{m}_{T|L}, \quad (1)$$

$$\frac{dm_L}{dt} = \hat{m}_{T|L}. \quad (2)$$

The final goal is to write a model that accounts for the evolution of the levels of liquid in the two vessels. For this we try to express the variables appearing in the balance equations in terms of the levels l_T and l_L .

3.3 Step 3 : Define Flows and Add Assumptions

First we express the mass flows in the balance equations. We choose to express the mass flow rates in terms of volumetric flow rates. For an arbitrary mass stream m , the mass flow rate is

$$\hat{m}_m := \rho_m \hat{V}_m, \quad (3)$$

where ρ_m is the density of the mass stream, and \hat{V}_m is the respective volumetric stream.

Assuming that the mass stream flows between the elementary systems A and B , that is $m := A|B$, the density of the mass stream will be the density of the substance in the system from which the stream originates:

$$\rho_{A|B} := \begin{cases} \rho_A & \text{if } \hat{V}_{A|B} > 0, \\ \rho_B & \text{if } \hat{V}_{A|B} < 0. \end{cases} \quad (4)$$

This can also be written using the sign (\cdot) function as

$$\rho_{A|B} := \frac{1}{2} \left(\left(1 + \text{sign}(\hat{V}_{A|B}) \right) \rho_A + \left(1 - \text{sign}(\hat{V}_{A|B}) \right) \rho_B \right). \quad (5)$$

However in the case of our problem we shall assume that the density of the water is constant in all the parts of the process, that is, for each mass transfer we have

$$\rho_A \equiv \rho_B =: \rho, \quad (6)$$

and consequently

$$\hat{m}_{A|B} := \rho \hat{V}_{A|B}. \quad (7)$$

The volumetric flow rate from the tube to the level glass is driven by the difference in static pressures at the bottom of the two vessels. The following relation is a good candidate for expressing this rate:

$$\hat{V}_{T|L} := c_{T|L} \sqrt{\frac{|p_T - p_L|}{\rho}} \text{sign}(p_T - p_L), \quad (8)$$

where $c_{T|L}$ represents the valve constant including the resistance effects of the pipe. As we are interested in a model that accounts for the levels, it is useful to use the expression

$$p_\Sigma := \rho g l_\Sigma; \quad \Sigma := T, L \quad (9)$$

to obtain

$$\hat{V}_{T|L} := a \sqrt{g |l_T - l_L|} \text{sign}(l_T - l_L). \quad (10)$$

$$a := c_{T|L} \sqrt{g} \quad (11)$$

With these the balance equations (1) and (2) become

$$\frac{dm_T}{dt} := \rho \hat{V}_{S|T} - \rho \hat{V}_{T|P} - \rho a \sqrt{|l_T - l_L|} \text{sign}(l_T - l_L), \quad (12)$$

$$\frac{dm_L}{dt} := \rho a \sqrt{|l_T - l_L|} \text{sign}(l_T - l_L). \quad (13)$$

3.4 Step 4: State Variable Transformation

Now we express the extensive variables in terms of the variables of interest (the user variables) that are in our case the two levels. If we make no specific assumptions about the shapes of the two vessels we can write

$$m_T := \rho V_T(l_T), \quad (14)$$

$$m_L := \rho V_L(l_L). \quad (15)$$

In the case that the vessels are cylindrical:

$$V_T(l_T) := A_T l_T \quad (16)$$

$$V_L(l_L) := A_L l_L. \quad (17)$$

In general, taking the time derivatives in these relations we have

$$\begin{bmatrix} \frac{dm_T}{dt} \\ \frac{dm_L}{dt} \end{bmatrix} := \begin{bmatrix} \frac{\partial m_T}{\partial l_T} & \frac{\partial m_T}{\partial l_L} \\ \frac{\partial m_L}{\partial l_T} & \frac{\partial m_L}{\partial l_L} \end{bmatrix} \begin{bmatrix} \frac{dl_T}{dt} \\ \frac{dl_L}{dt} \end{bmatrix} \quad (18)$$

$$:= \begin{bmatrix} \rho \frac{\partial V_T}{\partial l_T} & 0 \\ 0 & \rho \frac{\partial V_L}{\partial l_L} \end{bmatrix} \begin{bmatrix} \frac{dl_T}{dt} \\ \frac{dl_L}{dt} \end{bmatrix}. \quad (19)$$

Using these in (12) and (13) we obtain:

$$\begin{bmatrix} \frac{dl_T}{dt} \\ \frac{dl_L}{dt} \end{bmatrix} = \begin{bmatrix} \left(\frac{\partial V_T}{\partial l_T} \right)^{-1} \left(\hat{V}_{S|T} - \hat{V}_{T|P} - a \sqrt{|l_T - l_L|} \operatorname{sign}(l_T - l_L) \right) \\ \left(\frac{\partial V_L}{\partial l_L} \right)^{-1} a \sqrt{|l_T - l_L|} \operatorname{sign}(l_T - l_L) \end{bmatrix}. \quad (20)$$

This is a model in input-state form if we choose the state vector to be

$$\underline{\mathbf{x}} := \begin{bmatrix} l_T \\ l_L \end{bmatrix} \quad (21)$$

and the input to be

$$\underline{\mathbf{u}} := \begin{bmatrix} \hat{V}_{S|T} \\ \hat{V}_{T|P} \end{bmatrix} \quad (22)$$

Assuming that the vessels are cylindrical, the system (20) can be written as

$$\frac{d\underline{\mathbf{x}}}{dt} = \begin{bmatrix} \frac{1}{A_T} \left(u_1 - u_2 - a \sqrt{|x_1 - x_2|} \operatorname{sign}(x_1 - x_2) \right) \\ \frac{1}{A_L} a \sqrt{|x_1 - x_2|} \operatorname{sign}(x_1 - x_2) \end{bmatrix} \quad (23)$$

The quantities A_T , A_L , c , ρ , g play the role of parameters of our model.

The equations is linear in the inputs:

$$\frac{d\underline{\mathbf{x}}}{dt} := \underline{\underline{\mathbf{S}}} \begin{bmatrix} -1 \\ 1 \end{bmatrix} a \tilde{x}(\underline{\mathbf{x}}) + \underline{\underline{\mathbf{S}}} \underline{\underline{\mathbf{F}}} \underline{\mathbf{u}}. \quad (24)$$

Which isolates the nonlinearity into the function:

$$\tilde{x}(\underline{\mathbf{x}}) := \sqrt{|x_1 - x_2|} \operatorname{sign}(x_1 - x_2). \quad (25)$$

and defines the two matrices:

$$\underline{\underline{\mathbf{S}}} := \begin{bmatrix} 1/A_T & 0 \\ 0 & 1/A_L \end{bmatrix}, \quad (26)$$

$$\underline{\underline{\mathbf{F}}} := \begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix}. \quad (27)$$

3.5 Step 5: Linearisation

Given the general nonlinear system:

$$\dot{\underline{\mathbf{x}}} = \underline{\mathbf{f}}(\underline{\mathbf{x}}, \underline{\mathbf{u}}) \quad (28)$$

$$\underline{\mathbf{y}} := \underline{\mathbf{g}}(\underline{\mathbf{x}}, \underline{\mathbf{u}}) \quad (29)$$

linearisation is done about an operating point defined by the state $\underline{\mathbf{x}}^0$. For this operating point one computes the steady state relation between the state $\underline{\mathbf{x}}^0$ and the input $\underline{\mathbf{u}}^0$ by solving the equation:

$$\underline{\mathbf{0}} = \underline{\mathbf{f}}(\underline{\mathbf{x}}_0, \underline{\mathbf{u}}_0) \quad (30)$$

for $\underline{\mathbf{u}}^0$. Note that this procedure implies that the first set of equations can be solved explicitly for the input variables.

The linearisation is then done about the point defined by x_0, u_0, y_0 by expanding the two nonlinear equations in a Taylor series, thus:

$$\underline{\mathbf{f}}(\underline{\mathbf{x}}, \underline{\mathbf{u}}) = \underline{\mathbf{f}}(\underline{\mathbf{x}}_0, \underline{\mathbf{u}}_0) + \frac{\partial \underline{\mathbf{f}}(\underline{\mathbf{x}}, \underline{\mathbf{u}})}{\partial \underline{\mathbf{x}}^T} \Big|_{\underline{\mathbf{x}}_0, \underline{\mathbf{u}}_0} (\underline{\mathbf{x}} - \underline{\mathbf{x}}_0) + \frac{\partial \underline{\mathbf{f}}(\underline{\mathbf{x}}, \underline{\mathbf{u}})}{\partial \underline{\mathbf{u}}^T} \Big|_{\underline{\mathbf{x}}_0, \underline{\mathbf{u}}_0} (\underline{\mathbf{u}} - \underline{\mathbf{u}}_0) + ||O^2|| \quad (31)$$

$$\underline{\mathbf{g}}(\underline{\mathbf{x}}, \underline{\mathbf{u}}) = \underline{\mathbf{g}}(\underline{\mathbf{x}}_0, \underline{\mathbf{u}}_0) + \frac{\partial \underline{\mathbf{g}}(\underline{\mathbf{x}}, \underline{\mathbf{u}})}{\partial \underline{\mathbf{x}}^T} \Big|_{\underline{\mathbf{x}}_0, \underline{\mathbf{u}}_0} (\underline{\mathbf{x}} - \underline{\mathbf{x}}_0) + \frac{\partial \underline{\mathbf{g}}(\underline{\mathbf{x}}, \underline{\mathbf{u}})}{\partial \underline{\mathbf{u}}^T} \Big|_{\underline{\mathbf{x}}_0, \underline{\mathbf{u}}_0} (\underline{\mathbf{u}} - \underline{\mathbf{u}}_0) + ||O^2|| \quad (32)$$

The output at steady-state is simply calculated from the non-linear measurement equation

$$\underline{\mathbf{y}}_0 := \underline{\mathbf{g}}(\underline{\mathbf{x}}_0, \underline{\mathbf{u}}_0) \quad (33)$$

Finally by defining the deviation variables and Jacobian matrices:

$$\Delta_{\underline{\mathbf{x}}} := \underline{\mathbf{x}} - \underline{\mathbf{x}}_0 \quad (34)$$

$$\Delta_{\underline{\mathbf{u}}} := \underline{\mathbf{u}} - \underline{\mathbf{u}}_0 \quad (35)$$

$$\Delta_{\underline{\mathbf{y}}} := \underline{\mathbf{y}} - \underline{\mathbf{y}}_0 \quad (36)$$

$$\underline{\underline{\mathbf{A}}} := \frac{\partial \underline{\mathbf{f}}(\underline{\mathbf{x}}, \underline{\mathbf{u}})}{\partial \underline{\mathbf{x}}^T} \Big|_{\underline{\mathbf{x}}_0, \underline{\mathbf{u}}_0} \quad (37)$$

$$\underline{\underline{\mathbf{B}}} := \frac{\partial \underline{\mathbf{f}}(\underline{\mathbf{x}}, \underline{\mathbf{u}})}{\partial \underline{\mathbf{u}}^T} \Big|_{\underline{\mathbf{x}}_0, \underline{\mathbf{u}}_0} \quad (38)$$

$$\underline{\underline{\mathbf{C}}} := \frac{\partial \underline{\mathbf{g}}(\underline{\mathbf{x}}, \underline{\mathbf{u}})}{\partial \underline{\mathbf{x}}^T} \Big|_{\underline{\mathbf{x}}_0, \underline{\mathbf{u}}_0} \quad (39)$$

$$\underline{\underline{\mathbf{D}}} := \frac{\partial \underline{\mathbf{g}}(\underline{\mathbf{x}}, \underline{\mathbf{u}})}{\partial \underline{\mathbf{u}}^T} \Big|_{\underline{\mathbf{x}}_0, \underline{\mathbf{u}}_0} \quad (40)$$

and truncating after the linear term, the equations reduce to the familiar linear form :

$$\Delta \dot{\underline{\mathbf{x}}}(t) = \underline{\underline{\mathbf{A}}} \Delta_{\underline{\mathbf{x}}}(t) + \underline{\underline{\mathbf{B}}} \Delta_{\underline{\mathbf{u}}}(t) \quad (41)$$

$$\Delta \underline{\mathbf{y}}(t) = \underline{\underline{\mathbf{C}}} \Delta_{\underline{\mathbf{x}}}(t) + \underline{\underline{\mathbf{D}}} \Delta_{\underline{\mathbf{u}}}(t) \quad (42)$$

This model describes the approximate behaviour about a chosen stationary point in the state domain. The state is here the deviation variables. The system matrices $\{\underline{\underline{\mathbf{A}}}, \underline{\underline{\mathbf{B}}}, \underline{\underline{\mathbf{C}}}, \underline{\underline{\mathbf{D}}}\}$ are Jacobians of the original representation.

Thus for our system we have only the Jacobian with respect to the vector $\underline{\mathbf{x}}$ to compute:

$$\underline{\underline{\mathbf{J}}}_x := \underline{\underline{\mathbf{A}}}, \quad (43)$$

$$:= \underline{\underline{\mathbf{S}}} \begin{bmatrix} -1 \\ 1 \end{bmatrix} \frac{\partial \hat{x}(\underline{\mathbf{x}})}{\partial \underline{\mathbf{x}}^T}. \quad (44)$$

Which in turn reduces to computing the partial derivative:

$$\frac{\partial \hat{x}(\underline{\mathbf{x}})}{\partial \underline{\mathbf{x}}^T} := \frac{\partial}{\partial \underline{\mathbf{x}}} |y(\underline{\mathbf{x}})|^{1/2} \text{sign}(y(\underline{\mathbf{x}})), \quad (45)$$

$$:= \frac{\partial \hat{x}(\underline{\mathbf{x}})}{\partial y} \frac{\partial y(\underline{\mathbf{x}})}{\partial \underline{\mathbf{x}}^T}. \quad (46)$$

With $y(\underline{\mathbf{x}}) := x_1 - x_2$. The differential with respect to y :

$$\frac{\partial \hat{x}(\underline{\mathbf{x}})}{\partial y} := \frac{\partial \text{sign}(y)}{\partial y} |y|^{1/2} + \text{sign}(y) \frac{\partial |y|^{1/2}}{\partial y}, \quad (47)$$

$$:= \text{sign}(y) \frac{\partial |y|^{1/2}}{\partial |y|} \frac{\partial |y|}{\partial y}, \quad (48)$$

$$:= \text{sign}(y) \frac{1}{2} |y|^{-1/2} \frac{\partial}{\partial y} (\text{sign}(y) y), \quad (49)$$

$$:= \text{sign}(y) \frac{1}{2} |y|^{-1/2} \text{sign}(y), \quad (50)$$

$$:= \frac{1}{2} |y|^{-1/2}. \quad (51)$$

The last bit is to differentiate y :

$$\frac{\partial y(\underline{\mathbf{x}})}{\partial \underline{\mathbf{x}}} := \frac{\partial}{\partial \underline{\mathbf{x}}}(x_1 - x_2), \quad (52)$$

$$:= \begin{bmatrix} 1 & -1 \end{bmatrix}. \quad (53)$$

So now we can assemble the Jacobian:

$$\underline{\underline{\mathbf{A}}} := \underline{\underline{\mathbf{S}}} \begin{bmatrix} -1 \\ 1 \end{bmatrix} \frac{a}{2} |y|^{-1/2} \begin{bmatrix} 1 & -1 \end{bmatrix}, \quad (54)$$

$$:= \underline{\underline{\mathbf{S}}} \begin{bmatrix} -1 & 1 \\ 1 & -1 \end{bmatrix} \frac{a}{2} |x_1^o - x_2^o|^{-1/2}. \quad (55)$$

And

$$\underline{\underline{\mathbf{B}}} := \underline{\underline{\mathbf{S}}} \underline{\underline{\mathbf{F}}}. \quad (56)$$

4 Simulation

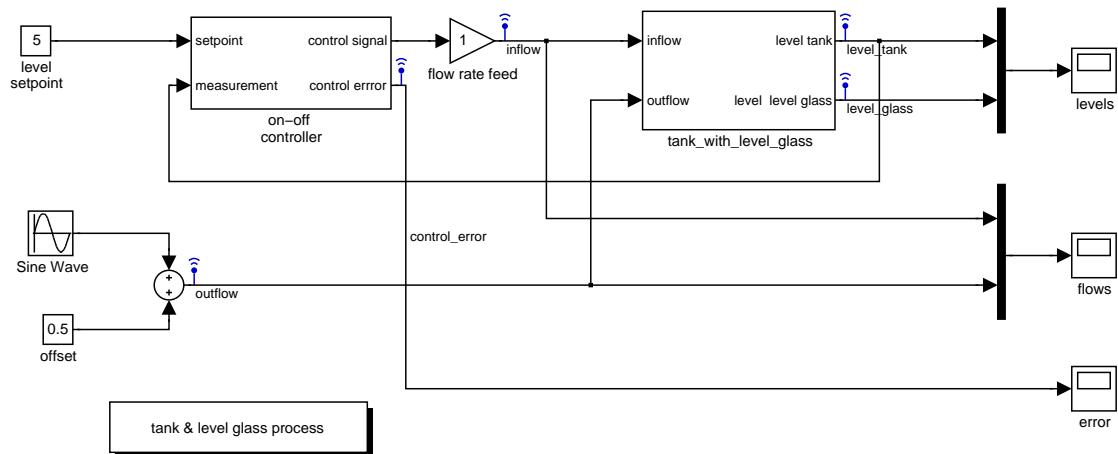


Figure 3: Simulink block diagram of the on/off controlled tank with level glass

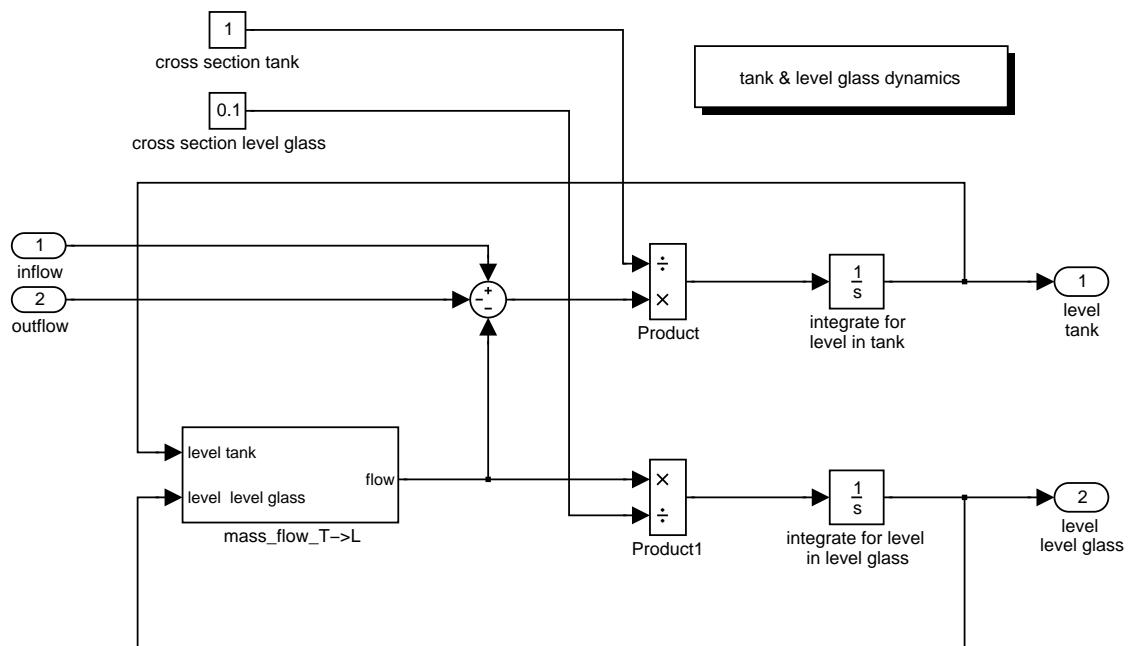


Figure 4: Simulink block diagram of the tank with level glass model

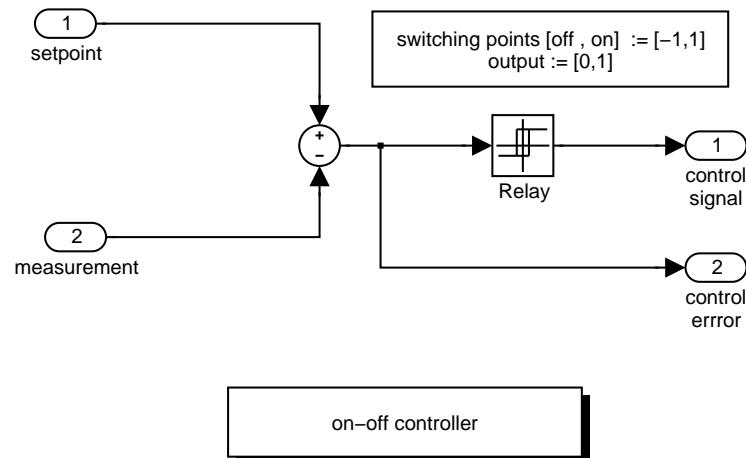


Figure 5: *Simulink block diagram of the on-off controller*

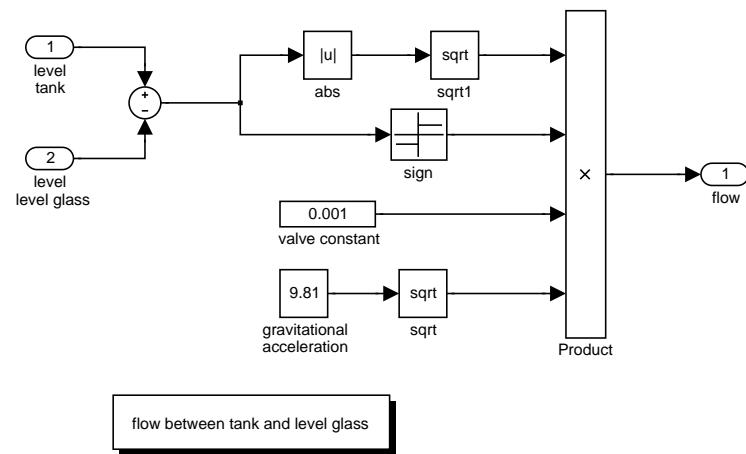


Figure 6: *Simulink block diagram of the flow model between tank and level glass*

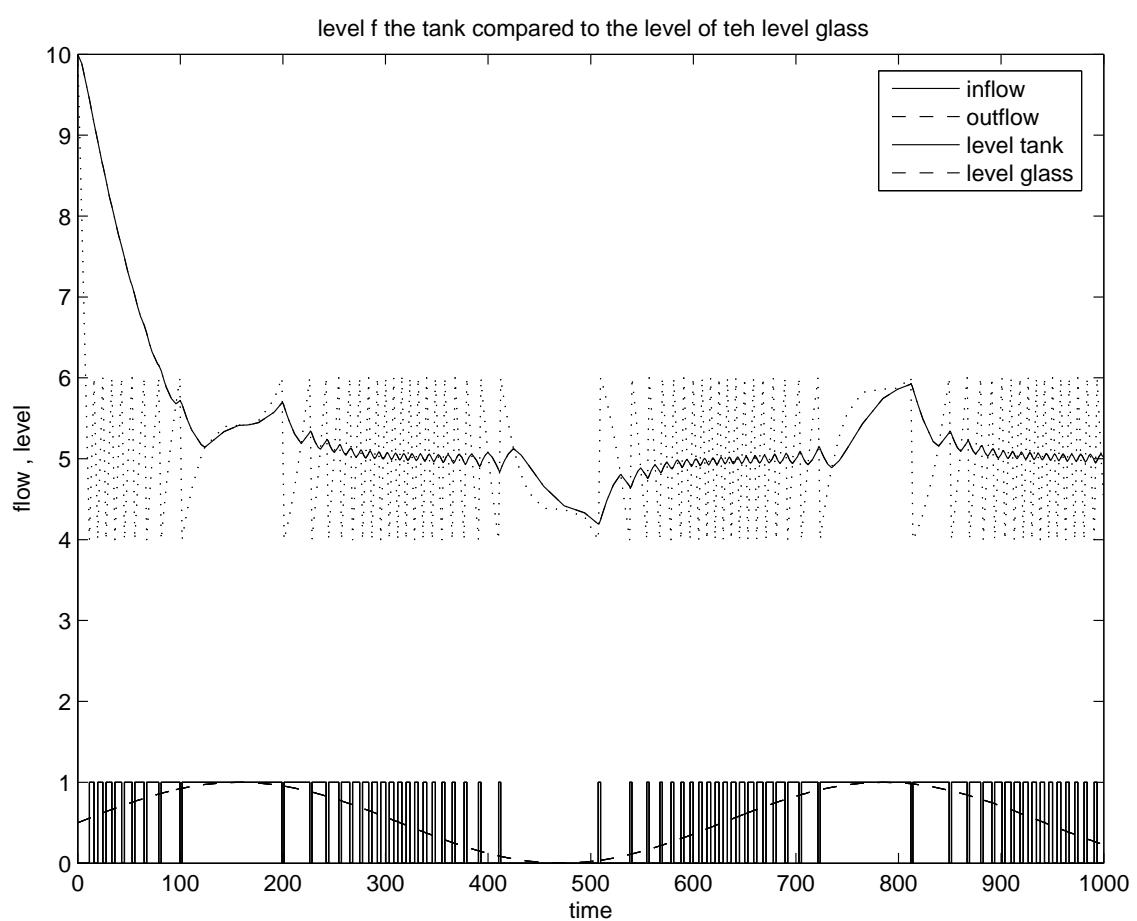


Figure 7: *Level in the tank and the level glass and the normed flows*

5 Solution: Dynamics 04 ODE Systems with Euler and Runge-Kutta

Euler method

The Euler method is

$$\underline{\mathbf{x}}_{n+1} = \underline{\mathbf{x}}_n + h \underline{\mathbf{f}}(t_n, \underline{\mathbf{x}}_n)$$

The main issue is to recognise the different objects and the operations required by the algorithm.

The state $\underline{\mathbf{x}}$ is a vector, thus can be represented as a corresponding matrix. The h is a scalar and the function $\underline{\mathbf{f}}()$ is a vector of the same dimension as the state. The operation required are thus matrix addition and scalar times matrix.

Rung kutta method

The RK4 method for this problem is given by the following equations:

$$\begin{aligned}\underline{\mathbf{x}}_{n+1} &= \underline{\mathbf{x}}_n + \frac{1}{6} (\underline{\mathbf{k}}_1 + 2\underline{\mathbf{k}}_2 + 2\underline{\mathbf{k}}_3 + \underline{\mathbf{k}}_4) \\ t_{n+1} &= t_n + h\end{aligned}$$

where x_{n+1} is the RK4 approximation of $x(t_{n+1})$, and

$$\begin{aligned}\underline{\mathbf{k}}_1 &= h \underline{\mathbf{f}}(t_n, \underline{\mathbf{x}}_n) \\ \underline{\mathbf{k}}_2 &= h \underline{\mathbf{f}}\left(t_n + \frac{1}{2}h, \underline{\mathbf{x}}_n + \frac{1}{2}\underline{\mathbf{k}}_1\right) \\ \underline{\mathbf{k}}_3 &= h \underline{\mathbf{f}}\left(t_n + \frac{1}{2}h, \underline{\mathbf{x}}_n + \frac{1}{2}\underline{\mathbf{k}}_2\right) \\ \underline{\mathbf{k}}_4 &= h \underline{\mathbf{f}}\left(t_n + \frac{1}{2}h, \underline{\mathbf{x}}_n + \underline{\mathbf{k}}_3\right)\end{aligned}$$

The state and the time step is as about. So the same objects are required and also the same operations.

5.1 Python code

The Python code has two pieces: The first implements a class of integrators, which are plugged into the second module, which contains the whole problem, integrator and differential equations.

Note that the model can be subclassed defining another model as demonstrated in the main program section of the module.

5.1.1 Integrators

```
1  """
2  2012-10-11 created
3  2012-10-13 step returns dt, dxdt and x_new
4  2012-10-21 extend to matrix operations
5
6
7 @author: Preisig, Heinz A
8 @organization: NTNU, Chemical Engineering
9 """
10 from matrix import Scalar
11
12 class Stepper(object):
13     """
14     Implements different "steppers" for computing ODE integrals.
15     """
16
17     def __init__(self, model, dt):
18         """
19         Keeps the model and the (initial) time step.
20         """
21         self.model = model
22         self.dt = dt
23
24     class Euler(Stepper):
25         """
26         Implements the Euler method.
27         """
28
29         def step(self, t, x):
30             dxdt = self.model.rhs(t, x)
31             x_new = x + dxdt * self.dt;
32             return self.dt, dxdt, x_new
33
34     class RungeKutta(Stepper):
35         """
36         Implements the Runge-Kutta 3:4 method.
37         Pending...
38         """
39
40         def step(self, t, x):
41             h = self.dt
42             k1 = h * self.model.rhs(t, x)
43             k2 = h * self.model.rhs(t + Scalar(0.5) * h, x + Scalar(0.5) * k1)
44             k3 = h * self.model.rhs(t + Scalar(0.5) * h, x + Scalar(0.5) * k2)
45             k4 = h * self.model.rhs(t + h, x + k3)
46             x_new = x + Scalar(1.0 / 6.0) * (k1 + Scalar(2.0) * k2 +
47                                              Scalar(2.0) * k3 + k4)
48             dxdt = (x_new - x) / h
49             return self.dt, dxdt, x_new
```

5.1.2 The model module

```
1  """
2  In this module, the term "model" is used for the complete "thing" including the
3  differential equation *and* the integrator.
4
5  2012-10-11 created (HAP).
6  2012-10-13 extended to multiple integrators and demo for re-use (HAP).
7  2012-10-16 replaced an ugly if-elif-else testing of the ode solver method with a
8      first class function object (THW).
9  """
```

```

10 2012-10-11 created
11 2012-10-13 extended to multiple integrators and demo for re-use
12
13 @author: Preisig, Heinz A
14 @organization: NTNU, Chemical Engineering
15 ''
16
17 import Integrator_vector_01 as ODE
18 from matrix import Matrix, Scalar
19 import math
20 from gnuplot import gnuplot
21 from string import replace
22
23 class Model:
24     '''
25         The model is the differential equations and the integrator
26     '''
27
28     def __init__(self, x0=Matrix([[0]]), dt=1.0, par=[], odesolver=ODE.Euler):
29         '''
30             The initialization gets the initial conditions, the parameters, and the
31             step size for the time stepper - being an integrator.
32             Note that the resulting state is stored as a list of vectors, whereby vectors are
33                 matrices of the dimension n,1.
34
35             @param x0: initial conditions
36             @type x0: Matrix
37             @param dt: time step
38             @type dt: float or integer
39             @param par: list of parameters
40             @type par: list of Scalar
41         '''
42         self.t = [0]                                     # keeps the time
43         self.dxdt = Matrix([[]])                         # the time derivative of the state
44         self.x = []                                     # keeps the state over time
45         self.x.append(x0)
46         self.par = par
47
48         # plug in desired integrator
49         self.integrator = odesolver(self, dt)          # plug in the integrator
50
51     def rhs(self, t, x):
52         '''
53             <USER SPECIFIC>
54             Can be redefined through subclassing
55
56             Returns the value of the time derivative of the current state and stores
57             the newly calculated derivative.
58         '''
59         dxdt = self.par[0] * x
60         return dxdt
61
62     def integrateTimeInterval(self):
63         '''
64             Integrate over the given time interval; thus updating time and state.
65             The state is kept as list of one-column matrices.
66         '''
67         x = self.x[-1]
68         t = self.t[-1]
69         dt, dxdt, x_new = self.integrator.step(t, x)
70         self.dxdt.extend(dxdt)
71         self.x.append(x_new)
72         self.t.append(t + dt)
73
74     def getCSV(self):

```

```

75     s = ''
76     for i in range(len(self.t)):
77         s_t = self.t[i].__str__()
78         _x = str(self.x[i])
79         _x = replace(_x, '[', ', ', ', ')
80         _x = replace(_x, ']', ', ', ', ')
81         s += '%s,%s\n' % (s_t, _x)
82     return s
83
84 class MyModel(Model):
85     """
86     Demonstrates defining a new complete model replacing the right-hand-side of the ODEs
87     """
88     def rhs(self, t, x):
89         dxdt = self.par[0] * x
90         return dxdt
91
92 class TwoDModel(Model):
93     """
94     Demonstrates defining a new complete model replacing the right-hand-side of the ODEs
95     but now a two-dimensional system of equations
96     """
97     def rhs(self, t, x):
98         dxdt = self.par[0] * x
99         return dxdt
100
101
102 # =====
103 if __name__ == '__main__':
104
105     # instantiate models
106     txtfile = open('01.dat', 'w')          # open data file for plotting
107     x0 = Matrix([[10.0]])
108     dt = Scalar(0.1)
109     par1 = [Scalar(-2)]
110     par2 = [Scalar(-1)]
111     n_samples = 5
112
113     # default model with Euler
114     m = Model(x0=x0, dt=dt, par=par1, odesolver=ODE.Euler) # initialise
115     for i in range(0, n_samples):                      # step through the time interval
116         m.integrateTimeInterval()
117         txtfile.write(m.getCSV())
118     txtfile.close()
119     print '\nDefault model results (Euler):\n', m.getCSV()
120
121     # default model with Runge Kutta
122     txtfile = open('02.dat', 'w')          # open data file for plotting
123     m = Model(x0=x0, dt=dt, par=par1, odesolver=ODE.RungeKutta) # initialise
124     for i in range(0, n_samples):          # step through the time interval
125         m.integrateTimeInterval()
126         txtfile.write(m.getCSV())
127     txtfile.close()
128     print '\nDefault model results (Runge-Kutta):\n', m.getCSV()
129
130
131     # my model Euler
132     txtfile = open('03.dat', 'w')          # open data file for plotting
133     m = MyModel(x0=x0, dt=dt, par=par2, odesolver=ODE.Euler) # initialise
134     for i in range(0, n_samples):          # step through the time interval
135         m.integrateTimeInterval()
136         txtfile.write(m.getCSV())
137     txtfile.close()
138     print 'My model results (Euler):\n', m.getCSV()
139

```

```

140
141 # my model Runge Kutta
142 txtfile = open('04.dat', 'w')      # open data file for plotting
143 m = MyModel(x0=x0, dt=dt, par=par2, odesolver=ODE.RungeKutta) # initialise
144 for i in range(0, n_samples):      # step through the time interval
145     m.integrateTimeInterval()
146     txtfile.write(m.getCSV())
147     txtfile.close()
148 print 'My_model_results_(Runge_Kutta):' + '\n', m.getCSV()
149
150
151 # comparison with exact solution
152 txtfile = open('05.dat', 'w')      # open data file for plotting
153 t = 0
154 x_exact = []
155 for i in range(0, n_samples):
156     t = i * dt
157     x_exact.append(math.e ** (par1[0] * t) * 10)
158     txtfile.write('%s %s\n' % (t, x_exact[-1]))
159 txtfile.close()
160 print 'Exact_value_first_model:' + '\n', m.getCSV()
161
162 txtfile = open('06.dat', 'w')      # open data file for plotting
163 t = 0
164 x_exact = []
165 for i in range(0, n_samples):
166     t = i * dt
167     x_exact.append(math.e ** (par2[0] * t) * 10)
168     txtfile.write('%s %s\n' % (t, x_exact[-1]))
169 txtfile.close()
170 print 'Exact_value_first_model:' + '\n', m.getCSV()
171
172 # 2-D problem
173 txtfile = open('07.dat', 'w')
174 t = 0.0
175 x0 = Matrix([[10.0], [10.0]])
176 # print x0.dim()
177 par3 = [Matrix([[-5.0, 0.0], [0.0, -0.10]])]
178 m = TwoDModel(x0=x0, dt=dt, par=par3, odesolver=ODE.Euler)
179 for i in range(0, n_samples):      # step through the time interval
180     m.integrateTimeInterval()
181     txtfile.write(m.getCSV())
182     txtfile.close()
183 print 'My_model_results_(Runge_Kutta):' + '\n', m.getCSV()
184
185
186 txtfile = open('08.dat', 'w')
187 t = 0.0
188 x0 = Matrix([[10.0], [10.0]])
189 m = TwoDModel(x0=x0, dt=dt, par=par3, odesolver=ODE.RungeKutta)
190 for i in range(0, n_samples):      # step through the time interval
191     m.integrateTimeInterval()
192     txtfile.write(m.getCSV())
193     txtfile.close()
194 print 'My_model_results_(Runge_Kutta):' + '\n', m.getCSV()
195
196
197 # =====
198 #plotting
199 plot = gnuplot(output='integratorplot',
200                  xlabel='t', ylabel='x',
201                  xmin=0, xmax=1, ymin=0, ymax=10,
202                  title='integration two exponentials using Euler and R-K')
203
204
```

```

205 plot.add('01.dat', x=1, y=2, width=2, color='red', type=1,
206         title='model_1:Euler')
207 plot.add('02.dat', x=1, y=2, width=2, color='blue', type=1,
208         title='model_1:R-K')
209 plot.add('03.dat', x=1, y=2, width=2, color='red', type=2,
210         title='model_2:Euler')
211 plot.add('04.dat', x=1, y=2, width=2, color='blue', type=2,
212         title='model_2:R-K')
213 plot.add('05.dat', x=1, y=2, width=2, color='green', type=1,
214         title='model_1:exact', style='points')
215 plot.add('06.dat', x=1, y=2, width=2, color='green', type=2,
216         title='model_2:exact', style='points')
217 plot.add('07.dat', x=1, y=2, width=2, color='red', type=3,
218         title='model_3.1:Euler')
219 plot.add('07.dat', x=1, y=2, width=2, color='blue', type=3,
220         title='model_3.2:Euler')#, style='linespoints')
221 plot.add('08.dat', x=1, y=2, width=2, color='red', type=4,
222         title='model_3.1:R-K')#, style='linespoints')
223 plot.add('08.dat', x=1, y=3, width=2, color='blue', type=4,
224         title='model_3.2:R-K')
225 # plot.save('test.dat')
226 plot.plot('integratorplot')

```

5.1.3 The matrix class

```

1 """
2 @summary: A simple class for matrixes being implemented as list of lists latter being ro
3
4 @author: Preisig, Heinz A
5 @organization: NTNU, Chemical Engineering
6
7 @since: 2012-10-21
8 @license: GPLv3
9 @requires: Python 2.7.1 or higher
10 @version: 1.0
11 """
12
13 class Matrix(list):
14     """
15     Implements an object matrix as a list of lists
16     This implies that the matrix is a list of row vectors
17     """
18
19     def __init__(self, m):
20         """
21         generates a matrix from a list of lists
22         @param m: list of lists
23         @type m: list
24         """
25
26         list.__init__(self, m)
27         _n, _m = self.dim()
28
29     def dim(self):
30         """
31         dimension of the 2-D object
32         If the matrix is not regular, an exception is raised.
33         @return: the two dimensions as tuple
34         """
35         m = len(self)
36         n = len(self[0])
37
38         for i in self:
39             nn = len(i)

```

```

40         if nn != n:
41             print('incompatible dimensions', m, n, nn)
42             raise Exception('incompatible dimensions')
43
44     return m, n
45
46     def getRow(self, i):
47         return Matrix(self[i])
48
49     def getColumn(self, j):
50         r = [self[i][j] for i in range(len(self))]
51         return Matrix([r]).transpose()
52
53     def __add__(self, other):
54         if other.__class__ != Matrix:
55             raise Exception('other is wrong class')
56
57         if self.dim() != other.dim():
58             raise Exception('incompatible dimensions')
59
60     else:
61         C = []
62         for i in range(len(self)):
63             row = []
64             for j in range(len(self[0])):
65                 row.append(self[i][j] + other[i][j])
66         C.append(row)
67
68     return Matrix(C)
69
70     def __sub__(self, other):
71         if other.__class__ != Matrix:
72             raise Exception('other is wrong class')
73
74         if self.dim() != other.dim():
75             raise Exception('incompatible dimensions')
76
77     else:
78         C = []
79         for i in range(len(self)):
80             row = []
81             for j in range(len(self[0])):
82                 row.append(self[i][j] - other[i][j])
83         C.append(row)
84
85     return Matrix(C)
86
87     def __mul__(self, other):
88         """
89             Two cases must be considered, namely
90             1. matrix * matrix
91             2. matrix * scalar (float, int)
92             @param other: second operand
93             @type other: Matrix | Scalar | float|int
94         """
95         if isinstance(other, float) or isinstance(other, int):
96             mA, nA = self.dim()
97             C = []
98             for i in range(mA):
99                 C.append([])
100                for j in range(nA):
101                    C[i].append(other * self[i][j])
102
103     return Matrix(C)
104     else:
105         mA, nA = self.dim()

```

```

105     m_B, n_B = other.dim()
106     if n_A != m_B:
107         print 'incompatible dimensions %s != %s' % (n_A, m_B)
108         raise Exception('incompatible dimensions')
109     C = []
110     for i in range(m_A):
111         C.append([])
112         for k in range(n_B):
113             acc = 0
114             for j in range(n_A):
115                 acc += self[i][j] * other[j][k]
116             C[i].append(acc)
117     return Matrix(C)
118
119     def __div__(self, other):
120         """
121             Two cases must be considered, namely
122             1. matrix * matrix
123             2. matrix * scalar (float, int)
124             @param other: second operand
125             @type other: Matrix | Scalar | float|int
126         """
127         if isinstance(other, float) or isinstance(other, int):
128             m_A, n_A = self.dim()
129             C = []
130             for i in range(m_A):
131                 C.append([])
132                 for j in range(n_A):
133                     C[i].append(self[i][j] / other)
134             return Matrix(C)
135         else:
136             return float.__div__(self, other)
137
138     def transpose(self):
139         """
140             transposed of the matrix
141             @return: new matrix object
142         """
143         C = []
144         m, n = self.dim()
145         for i in range(n):
146             C.append([])
147             for j in range(m):
148                 C[i].append(self[j][i])
149         return Matrix(C)
150
151     class Scalar(float):
152         """
153             Defines a scalar for the purpose of defining the scalar [op] Matrix operation
154         """
155
156
157         def __mul__(self, other):
158             """
159                 Multiply scalar with matrix
160                 @param other: a matrix
161                 @type other:Matrix | float
162             """
163             if isinstance(other, Matrix):
164                 m_A, n_A = other.dim()
165                 C = []
166                 for i in range(m_A):
167                     C.append([])
168                     for j in range(n_A):

```

```

170             C[i].append(self * other[i][j])
171     return Matrix(C)
172 else:
173     return Scalar(float.__mul__(self, other))
174
175
176
177 if __name__ == '__main__':
178     print 'module_matrix_test:\n\n'
179     A = Matrix([[1, 2], [3, 4]])
180     B = Matrix([[1, 4], [3, 4]])
181     v = Matrix([[10], [20]])
182     a = Scalar(2)
183
184
185     print 'dim(A)', A.dim()
186     print 'A+B', A + B
187     print 'A+A', A + A
188     print 'A-A', A - A
189     print 'A*A', A * A
190     print 'A*v', A * v
191     print 'A.transpose()', A.transpose()
192     print 'v.transpose()', v.transpose()
193     print 'v.transpose()*A', v.transpose() * A
194     print 'a*A', a * A
195     print 'A*a', A * a
196     print 'A/a', A / a
197     print 'A.getColumn(1)', A.getColumn(1)
198     print 'A.getColumn(-1)', A.getColumn(-1)

```

5.1.4 The oscillator

```

1 """
2 Created on Oct 29, 2012
3
4 @author: Preisig, Heinz A
5 @organization: NTNU, Chemical Engineering
6
7
8 from model_05 import Model
9 from matrix import Matrix, Scalar
10 from Integrator_vector_01 import Euler, RungeKutta
11 from gnuplot import gnuplot
12
13 class Oscillator(Model):
14     """
15     Two coupled oscillators
16     """
17     def rhs(self, t, x):
18         dxdt = self.par * x
19         return dxdt
20
21
22
23 if __name__ == '__main__':
24     x0 = Matrix([[10], [10], [10], [10]])
25     dt = Scalar(0.01)
26     n_samples = 10000
27     r = -0.01
28     a1 = 1
29     a2 = 3
30     par = Matrix([[r, a1, 0, 0],
31                  [-a1, r, 0, 0],
32                  [1, 0, r, a2],
33                  [0, 1, -a2, r]])

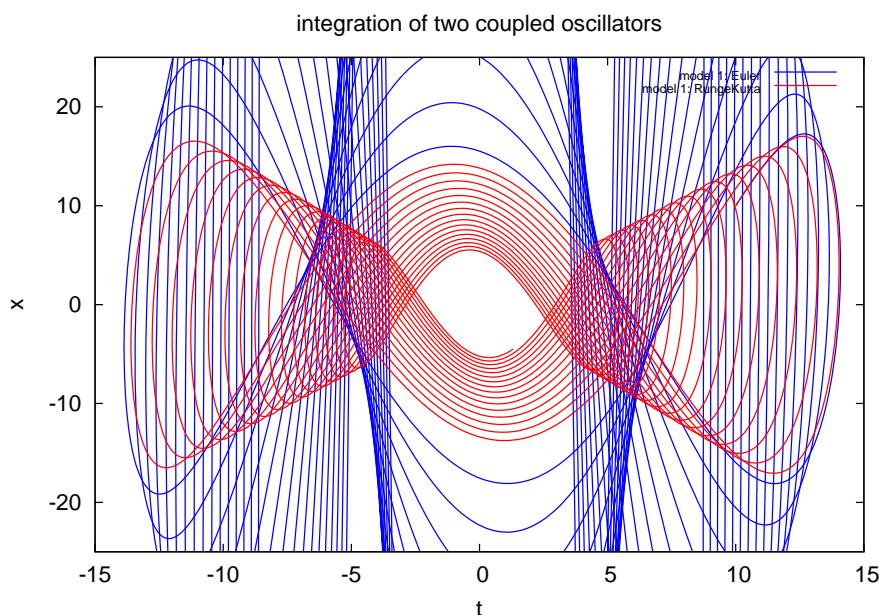
```

```

33      [0, 0, -a2, r] ,
34    ])
35
36 m = Oscillator(x0=x0, dt=dt, par=par, odesolver=Euler)
37 for i in range(0, n_samples):                      # step through the time interval
38     m.integrateTimeInterval()
39
40
41 txtfile = open('oscillator_Euler.dat', 'w')
42 txtfile.write(m.getCSV())
43 txtfile.close()
44
45 m = Oscillator(x0=x0, dt=dt, par=par, odesolver=RungeKutta)
46 for i in range(0, n_samples):                      # step through the time interval
47     m.integrateTimeInterval()
48
49
50 txtfile = open('oscillator_RungeKutta.dat', 'w')
51 txtfile.write(m.getCSV())
52 txtfile.close()
53
54
55 x_r = 15
56 y_r = 25
57
58 plot = gnuplot(output='oscillator',
59                  xlabel='t', ylabel='x',
60                  xmin=-x_r, xmax=x_r, ymin=-y_r, ymax=y_r, \
61                  title='integration of two coupled oscillators')
62
63
64 plot.add('oscillator_Euler.dat', x=2, y=4, width=2, color='blue',
65           type=1, title='model 1:Euler')
66 plot.add('oscillator_RungeKutta.dat', x=2, y=4, width=2, color='red',
67           type=1, title='model 1:RungeKutta')
68
69 plot.plot('oscillator')

```

5.1.5 Results



5.1.6 Matlab code

The following Matlab code gives the solution of the discrete oscillator, in contrast to the Python code.

```
1 % A 2-d example of a dynamic system
2 %
3 % 2012-10-21 Preisig, H A
4 %%
5
6 r = -0.01;
7 a = 1;
8 A11 = [ r , a; -a , r ];
9 a = 3;
10 A22 = [ r , a; -a , r ];
11 A12 = zeros(2,2);
12 A21 = zeros(2,2);
13 A21(1,1) = 1;
14
15 A = [A11, A12; A21, A22]
16
17 k = length(A(1,:));
18 dt = 0.1;
19 Phi = expm(A*dt);
20
21 n = 1000;
22 x = zeros(k,n);
23 x(:,1) = 10 * ones(k,1);
24
25 for i=2:n
26     x(:, i) = Phi*x(:, i-1);
27 end
28
29 figure(1); plot([1:n],x)
30 figure(2); plot(x(2,:),x(4,:))
```

Unittesting (TKP4106)



Zooball/Cow

Comparative Religion

- Taoism: Shit happens.
- Confucianism: Confucius say, "Shit happens."
- Hinduism: This shit has happened before.
- Protestantism: Let shit happen to someone else.
- Seventh Day Adventism: No shit shall happen on Saturdays.
- Jehovah's Witnesses: May we have a moment to show you some of our shit?
- Creationism: God made all shit.
- Hare Krishna: Shit happens, rama rama.
- Rastafarianism: Let's smoke this shit!
- Satanism: SNEPPAH TIHS.
- Stoicism: This shit is good for me.
- Nihilism: No shit.
- Vikingism: Shit stinks.
- ...

The Origin of Faeces

Assignments

1. Blabla

HTML text number 1.

%Predefined number 1.

HTML text number 2.

%Predefined number 2.

HTML text number 3.

Exercise 11

Maryam Ghadrdan and Preisig, H A

Chemical Engineering, NTNU

1 Question: Topology 08 Compartmental model of human

1.1 Blood flow in human

The blood flow of humans are sometimes modelled as a network of units connected by the main network of blood vessels. In turn, the units are modelled as networks of compartments. Units include heart, lungs, brain, arms, legs, intestines, liver, kidneys.

Generate a pictorial representation of the body in the form of a (hierarchical) graph representing the body's blood circulation system. Think about the function of the individual "units" and provide information on the main functionalities to the extent you can find it.

1.2 Interaction substrate / blood

Suggest a topology that describes the food intake, digestion and the transfer of components like sugar and water into the blood.

2 Question: Pipe with friction

We want to model the flow of the fluid with dynamic viscosity μ in an inclined tube (Figure 1).

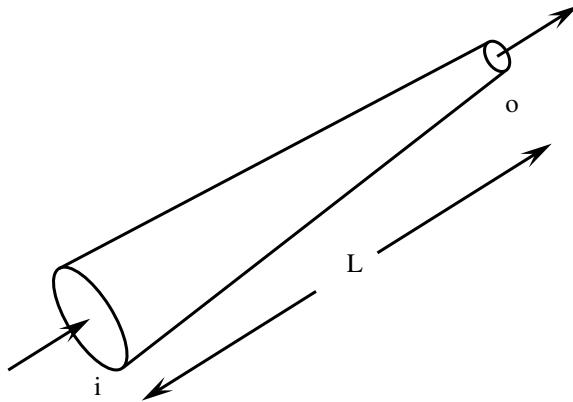


Figure 1: A schematic of an inclined tube

The objective is to write the volumetric flow rate as a function of height, pressure drop. The diameter changes linearly with the length. Density is constant so the fluid is incompressible.

$$\hat{V} = \hat{V}(p_i, p_o, h_i, h_o, r_o, r_i, L)$$

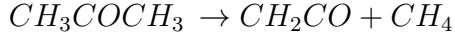
2.1 Tasks

- Establish the global mass balance and the mechanical energy balance for the inclined tube. Note the diameter changes from inlet to outlet.
- Establish the expansion of each term (kinetic energy, potential energy etc.). Note that the main issue is with the friction term - the diameter changes along the length of the pipe. The "expansion" must be done iteratively until you express everything as a function of the state and parameters. So you satisfy the degree of freedom
- For this time, do substitute the equations for two cases:
 - Laminar flow
 - Very high velocities, where you can assume that the friction factor is constant.

3 Question: Dynamics 05

In the literature we find the following description of the process:

Vapour phase cracking of acetone is described by the following endothermic reaction:



The reaction takes place in a jacketed tubular reactor. Pure acetone enters the reactor at a temperature of $T_0 = 1035\text{ K}$ and pressure of $P_0 = 162\text{ kPa}$, and the temperature of the external gas in the heat exchanger is constant at $T_a = 1150\text{ K}$. Other data are as follows:

Volumetric flow rate:

$$\nu_0 = 0.002 \frac{m^3}{s}$$

Volume of the reactor:

$$V_R = 1\text{ m}^3$$

Overall heat transfer coefficient:

$$U = 110 \frac{W}{m^3 \cdot K}$$

Heat transfer area:

$$a = 150 \frac{m^2}{m^3}$$

Reaction constant:

$$k = 3.58 \exp \left[34222 \left(\frac{1}{1035} - \frac{1}{T} \right) \right] \text{ s}^{-1}$$

Heat of reaction:

$$\Delta H_R = 80770 + 6.8 (T - 298) - 5.75 \times 10^{-3} (T^2 - 298^2) - 1.27 \times 10^{-6} (T^3 - 298^3) \frac{J}{mol}$$

Heat capacity of acetone:

$$C_{P_A} = 26.63 + 0.1830 T - 45.86 \times 10^{-6} T^2 \frac{J}{mol \cdot K}$$

Heat capacity of ketene:

$$C_{P_B} = 20.04 + 0.0945 T - 30.95 \times 10^{-6} T^2 \frac{J}{mol \cdot K}$$

Heat capacity of methane:

$$C_{P_C} = 13.39 + 0.0770 T - 18.71 \times 10^{-6} T^2 \frac{J}{mol \cdot K}$$

Determine the temperature profile of the gas along the length of the reactor. Assume constant pressure throughout the reactor.

In order to calculate the temperature profile in the reactor, we have to solve the material balance and energy balance equations simultaneously.

Mole balance:

$$\frac{dX}{dV} = \frac{-r_A}{F_{A_0}}$$

Energy balance:

$$\frac{dT}{dV} = \frac{U a (T_a - T) + r_A \Delta H_R}{F_{A_0} (C_{P_A} + X \Delta C_P)}$$

where X is the conversion of acetone, V is the volume of the reactor, $F_{A_0} = C_{A_0} \nu_0$ is the molar flow rate of acetone at the inlet, T is the temperature of the reactor, $\Delta C_P = C_{P_B} + C_{P_C} - C_{P_A}$, and C_{A_0} is the concentration of acetone vapour at the inlet. The reaction rate is given as

$$-r_A = k C_{A_0} \frac{1 - X}{1 + X} \frac{T_0}{T}$$

3.1 Task

We have made a point that we want to have component mass and energy as our state. The model above describes the process in the intensive variables mole fraction and temperature. We want to have it in our variables in order to control the error in the conserved quantities and also for other reasons, like not having to substitute, increase readability and documentation value of the model in algebraic and coded form.

In order to do so, we have to think in terms of a moving co-ordinate system, that is: think of sitting in a boat on the plug flow. The plug flow implies no axial mixing thus ideal radial mixing in the tubular reactor. So taking this view, we can further assume that we deal with a volume element around the boat that has the same intensive properties. Thus we think of the tubular reactor like a moving batch reactor: a lump of material comes in and travels down the pipe. This takes some time during which it exchanges heat with the jacket and during which it undergoes a reaction under the conditions as they change during the trip.

Thus we formulate the problem as a "travelling" batch reactor.

- Establish a model for the volume section you have in mind travelling down the pipe, say it has volume $V_s := V_R/100$
- As usual, write the model in generic form, thus with generic flow and reaction terms, which you expand thereafter recursively until each of them is a function of the state variables, parameters and given conditions, like the state at the entrance and the temperature in the jacket, to mention two. Make it a point to change the notation.
- Put together a code that for the right-hand-side of your balance equations by inverting the sequence of computations as established above.
- Solve the equations with your Euler and Runge-Kutta procedure.
- Plot results.

1 Suggested solution: Blood circulation

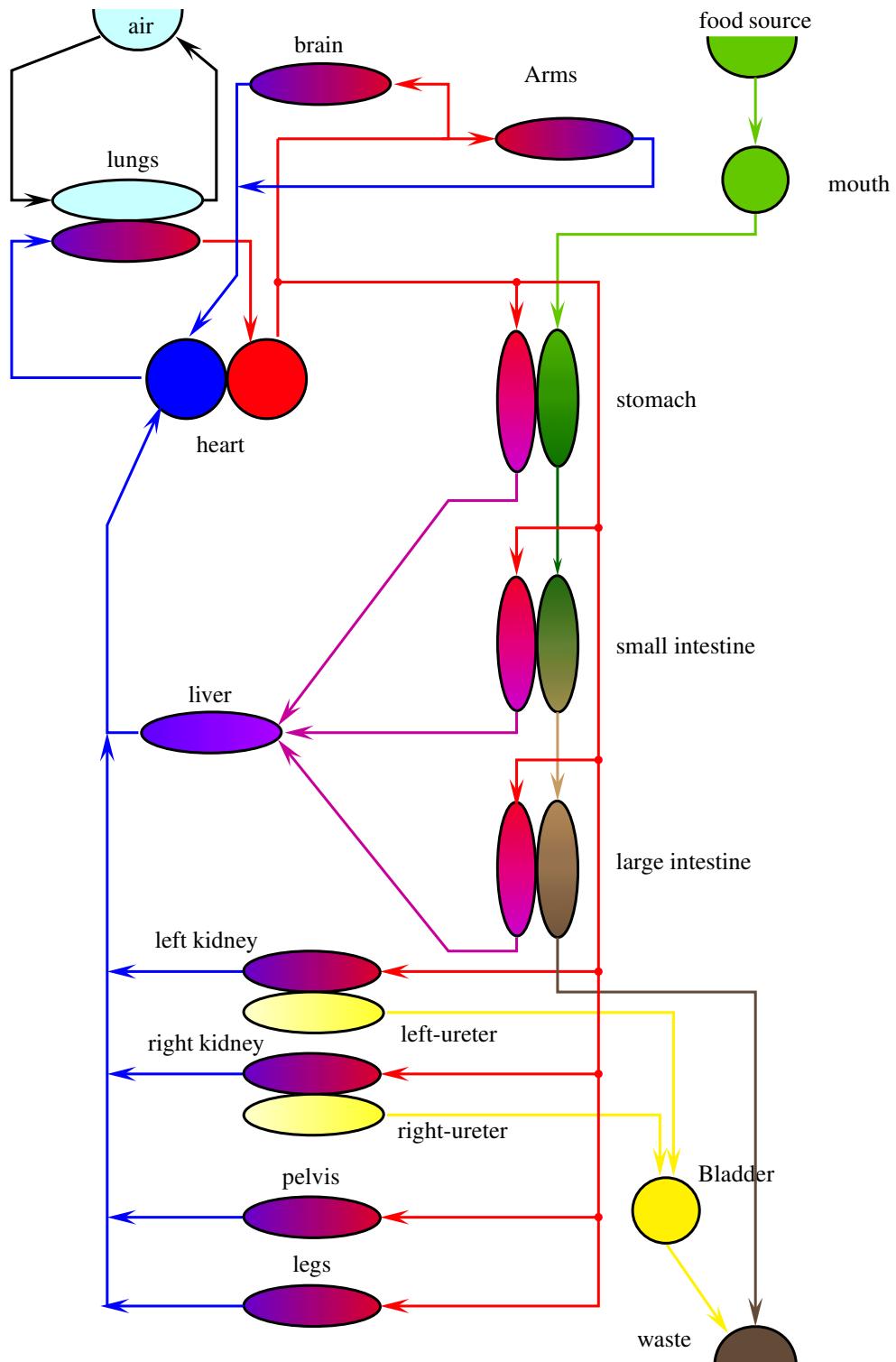


Figure 1: Topology of the blood circulation in a human

2 Solution: Pipe with friction

Model

The state space appropriate for the description of PCB systems is defined by the component mass, total energy, and linear momentum in the three co-ordinates. This state may be reduced if we make assumptions about the momentum balance. In the context of fluid systems, the momentum balances describe the fluid flow. Making the assumptions of

- incompressible fluid
- constant density
- iso-entropic, iso-thermal process, thus no energy conversion (friction into internal energy, for example)
- steady flow

So for a pipe that has a diameter changing with the length $r(x) := r(0) + a * x$ and has a certain roughness ϵ :

$$\begin{aligned}
 \text{internal energy} & 0 = \hat{U}_i - \hat{U}_o \\
 \text{kinetic energy} & + \frac{\hat{m} v_i^2}{2} - \frac{\hat{m} v_o^2}{2} \\
 \text{potential energy} & + \hat{m} g h_i - \hat{m} g h_o \\
 \text{volume work} & + p_i \hat{V} - p_o \hat{V} \\
 \text{friction work} & - \hat{w}^f
 \end{aligned}$$

The iso-thermal condition makes the internal energy of the entering and the leaving fluid equal. The interesting term is the friction. It is to be integrated over the length and friction is empirically related to velocity through for example Darcy-Weisbach relation, which gives the pressure drop due to friction being:

$$\Delta p^f := f \frac{L}{D} \frac{\rho v^2}{2}$$

which gives for the friction term:

$$\hat{w}^f := \int_0^L f(Re(v(x))) \frac{1}{2r(x)} \frac{\rho v(x)^2}{2} \hat{V} dx$$

with the velocity and volumetric flow rate being:

$$\begin{aligned}
 v &:= A^{-1} \hat{V} \\
 \hat{V} &:= \rho^{-1} \hat{m} \\
 v &:= A^{-1} \rho^{-1} \hat{m}
 \end{aligned}$$

and considering that the volumetric flow rate is constant we get:

$$\hat{w}^f := \frac{\rho \hat{V}}{4} \int_0^L f(Re(v(x))) \frac{v(x)^2}{r(x)} dx$$

The difficult part is in the friction factor, as it is only given as a function in the extreme domains, namely for laminar flows and for high Reynold numbers, one can find at least an approximation through interpolation.

For laminar flow we have:

$$f = \frac{64 \mu}{\rho v(x) 2 r(x)} = \frac{32 \mu}{\rho v(x) r(x)}$$

substituting f in the last term will give:

$$\begin{aligned} \hat{w}^f &:= \frac{\rho \hat{V}}{4} \int_0^L \frac{32 \mu}{\rho v(x) r(x)} \frac{v(x)^2}{r(x)} dx \\ &:= 8 \mu \hat{V} \int_0^L \frac{v(x)}{r^2(x)} dx \\ &:= 8 \mu \hat{V} \int_0^L \frac{\hat{V}}{A r^2(x)} dx \\ &:= \frac{8}{\pi} \mu \hat{V}^2 \int_0^L \frac{1}{r^4(x)} dx \\ &:= \frac{8}{\pi} \mu \hat{V}^2 \theta(r_i, r_o, L) \end{aligned}$$

where $\theta(r_i, r_o, L)$ is the respective integral. Substituting the change of the diameter being:

$$r(x) = r_i - \frac{r_i - r_o}{L} x$$

The integral to be solved is:

$$\int_0^L (r_i - \frac{r_i - r_o}{L} x)^{-4} dx$$

In the integral tables one finds:

$$\int_0^L (z + a)^n dz := \frac{(z + a)^{n+1}}{n+1} \Big|_0^L$$

yielding:

$$\frac{(r_i^2 + r_i r_o + r_o^2) L}{3 r_i^3 r_o^3}$$

Substitution leads to:

kinetic energy	$0 = + \frac{\hat{m} v_i^2}{2} - \frac{\hat{m} v_o^2}{2}$
potential energy	$+ \hat{m} g h_i - \hat{m} g h_o$
volume work	$+ p_i \hat{V} - p_o \hat{V}$
friction work	$- \hat{w}^f$

So,

$$0 = \frac{1}{2 \pi^2 \rho^2} (r_i^{-4} - r_o^{-4}) \hat{m}$$

$$\begin{aligned}
& + g (h_i - h_o) \hat{m} \\
& + (p_i - p_o) \rho^{-1} \hat{m} \\
& - \frac{8 \mu}{3 \pi \rho^2} \frac{(r_i^2 + r_i r_o + r_o^2) L}{r_i^3 r_o^3} \hat{m}^2
\end{aligned}$$

Now the equation is solved in terms of \hat{m} and can be converted to volumetric flow via $\hat{V} = \rho^{-1} \hat{m}$.

3 Solution: Dynamics 05 Aceton cracking

3.1 Model equations

equations	res	vars	given
$\underline{n}_G := \int_0^t \dot{\underline{n}}_G dt + \underline{n}_G^o$	\underline{n}_G	$\dot{\underline{n}}_G$	\underline{n}_G^o
$H_G := \int_0^t \dot{H}_G dt + H_G^o$	H_G	\dot{H}_G	\underline{H}_G^o
$\dot{\underline{n}}_G = \tilde{\underline{n}}_G$	$\dot{\underline{n}}_G$	$\tilde{\underline{n}}_G$	
$\dot{H}_G = \hat{q}_{G C}$	\dot{H}_G	$\hat{q}_{G C}$	
$\hat{q}_{G C} := -k_{G C} (T_C - T_G)$	$\hat{q}_{G C}$	T_G	$T_C, k_{G C}$
$\tilde{\underline{n}}_G := \underline{\underline{N}}^T V_G k c_{G,A}$	$\tilde{\underline{n}}_G$	$c_{G,A}, k, V_G$	$\underline{\underline{N}}$
$k := a \exp \{b (c^{-1} - T_G^{-1})\}$	k	T_G	a, b, c
$c_{G,A} := [1, 0] \underline{c}_G$	$c_{G,A}$	\underline{c}_G	
$\underline{c}_G := V_G^{-1} \underline{n}_G$	\underline{c}_G	\underline{n}_G, V_G	
$V_G := [1, \dots, 1] \underline{n}_G R T_G / p_G$	V_G	\underline{n}_G, T_G	R, p_G
$H_G := \int_{T_0}^{T_G} \underline{n}_G^T \underline{\gamma}(\tau) d\tau$	T_G	$H_G, \underline{\gamma}, V_G$	T_0
$\underline{\gamma} := \underline{a}_0 + \underline{a}_1 T_G + \underline{a}_2 T_G^2$	$\underline{\gamma}$	T_G	$\underline{a}_0, \underline{a}_1, \underline{a}_2$

The last equations need some massaging:

$$\begin{aligned}
 H_G &:= \int_{T_0}^{T_G} \underline{n}_G^T \underline{\gamma}(\tau) d\tau \\
 &:= \int_{T_0}^{T_G} \underline{n}_G^T (\underline{a}_0 + \underline{a}_1 \tau + \underline{a}_2 \tau^2) d\tau \\
 &:= \underline{n}_G^T (\underline{a}_0 + \underline{a}_1/2 \tau + \underline{a}_2/3 \tau^2) \Big|_{T_0}^{T_G} \\
 &:= \underline{n}_G^T (\underline{a}_0 + \underline{a}_1/2 T_G + \underline{a}_2/3 T_G^2) T_G - \underline{n}_G^T (\underline{a}_0 + \underline{a}_1/2 T_o + \underline{a}_2/3 T_o^2) T_o
 \end{aligned}$$

We define

$$\begin{aligned}
 H_o &:= \underline{n}_G^T (\underline{a}_0 + \underline{a}_1/2 T_o + \underline{a}_2/3 T_o^2) T_o \\
 P(T_G) &:= \underline{n}_G^T (\underline{a}_0 + \underline{a}_1/2 T_G + \underline{a}_2/3 T_G^2) T_G
 \end{aligned}$$

then the feasible root T_G is to be evaluated from:

$$0 := P(T_G) - (H_G + H_o) := r(T_G)$$

The task is then to find the roots for the equation $r(T_G) = 0$

3.2 Newton-Raphson iteration

Since the Jacobian is known:

$$J(T_G) = \underline{\mathbf{n}}_G^T (\underline{\mathbf{a}}_0 + \underline{\mathbf{a}}_1 T_o + \underline{\mathbf{a}}_2 T_o^2)$$

the Newton-Raphson step is:

$$T_G(k+1) := T_G(k) - J^{-1} r(T_G(k))$$

The

3.2.1 Symbols

$\underline{\mathbf{n}}_G$	vector of component masses in moles
H_G	enthalpy
$\dot{\underline{\mathbf{n}}}_G$	time derivatives vector of component masses in moles
\dot{H}_G	time derivative of enthalpy
$\hat{q}_{G C}$	heat flow reacting volume to cooler
$\tilde{\underline{\mathbf{n}}}_G$	production rate of component mass in moles
k	reaction "constant"
$g(\underline{\mathbf{c}}_G, T_G)$	kinetic relation
$c_{G,A}$	concentration of species A in G
$c_{G,B}$	concentration of species B in G
$\underline{\mathbf{c}}_G$	vector of concentration of species in G
T_G	temperature in G
$\underline{\gamma}$	vector of specific heat capacity of pure species in G at constant pressure
V_G	volume of the reacting phase
T_o	initial temperature (temperature at the entrance)
T_C	temperature in the cooler (uniform)
$\underline{\mathbf{N}}$	stoichiometric matrix (reactions vs species)
a_i	vectors with the respective coefficients of the polynomial approximations for the specific heat capacity
a, b, c	kinetic constants

3.3 Runge Kutta procedure

We have the set of ordinary differential equations as below:

$$\begin{aligned}\frac{dy_1}{dt} &= f_1(t, y_1, y_2, \dots, y_n) \\ \frac{dy_2}{dt} &= f_2(t, y_1, y_2, \dots, y_n) \\ &\vdots \\ \frac{dy_n}{dt} &= f_n(t, y_1, y_2, \dots, y_n)\end{aligned}$$

Rung kutta method for simultaneous equations

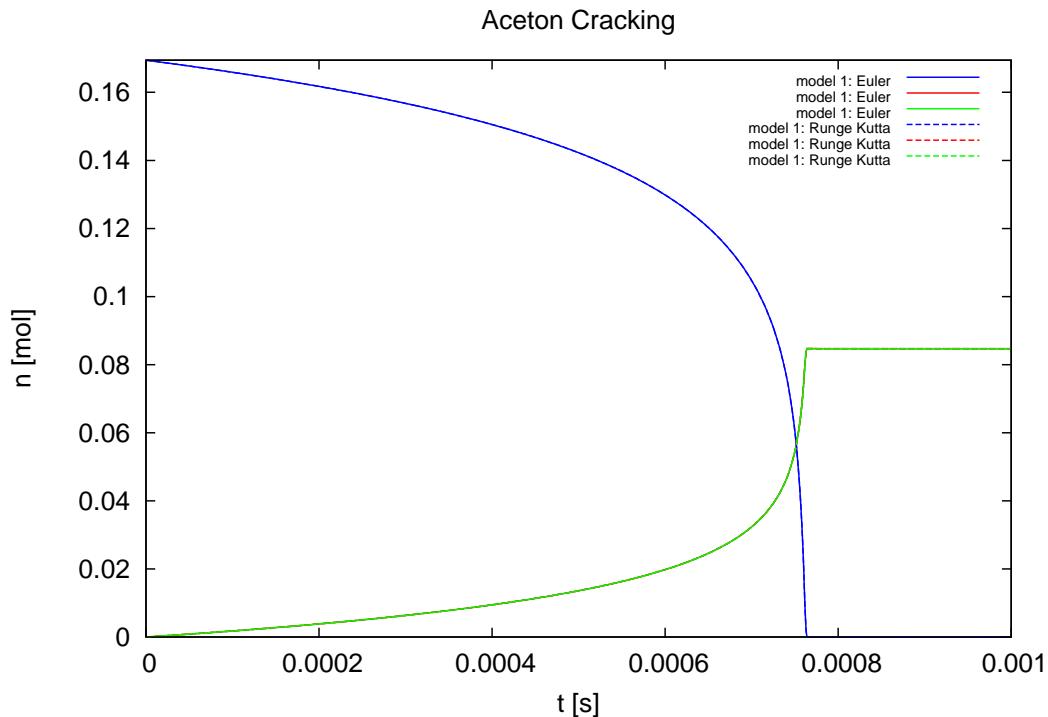
The RK4 method for this problem is given by the following equations:

$$\begin{aligned}y_{i+1,j} &= y_{i,j} + \frac{1}{6} (k_{1,j} + 2 k_{2,j} + 2 k_{3,j} + k_{4,j}) \\ t_{i+1} &= t_i + h\end{aligned}$$

where $j = 1, 2, \dots, n$ and

$$\begin{aligned}k_{1,j} &= h f_j(t_i, y_{i,1}, y_{i,2}, \dots, y_{i,n}) \\ k_{2,j} &= h f_j \left(t_i + \frac{1}{2} h, y_{i,1} + \frac{1}{2} k_{1,1}, y_{i,2} + \frac{1}{2} k_{1,2}, \dots, y_{i,n} + \frac{1}{2} k_{1,n}, \right) \\ k_{3,j} &= h f_j \left(t_i + \frac{1}{2} h, y_{i,1} + \frac{1}{2} k_{2,1}, y_{i,2} + \frac{1}{2} k_{2,2}, \dots, y_{i,n} + \frac{1}{2} k_{2,n}, \right) \\ k_{4,j} &= h f_j \left(t_i + h, y_{i,1} + k_{3,1}, y_{i,2} + \frac{1}{2} k_{3,2}, \dots, y_{i,n} + k_{3,n}, \right)\end{aligned}$$

Plot of the component masses



3.3.1 The model module

```

1  """
2  In this module, the term "model" is used for the complete "thing" including the
3  differential equation *and* the integrator.
4
5  2012-10-11 created (HAP).
6  2012-10-13 extended to multiple integrators and demo for re-use (HAP).
7  2012-10-16 replaced an ugly if-elif-else testing of the ode solver method with a
8      first class function object (THW).
9
10 2012-10-11 created
11 2012-10-13 extended to multiple integrators and demo for re-use
12 2012-11-14 extended to output also the secondary states put into self.y by the
13      the user-defined model, that is, if it is there.
14
15 @author: Preisig, Heinz A
16 @organization: NTNU, Chemical Engineering
17 """
18
19 import Integrator_vector_01 as ODE
20 from matrix import Matrix, Scalar
21 import math
22 from gnuplot import gnuplot
23 from string import replace
24
25 class Model:
26     """
27         The model is the differential equations and the integrator
28     """
29
30     def __init__(self, x0=Matrix([[0]]), dt=1.0, par=[], odesolver=ODE.Euler):
31         """

```

```

32     The initialization gets the initial conditions, the parameters, and the
33     step size for the time stepper - being an integrator.
34     Note that the resulting state is stored as a list of vectors, whereby vectors are
35     matrices of the dimension n,1.
36
37     @param x0: initial conditions
38     @type x0: Matrix
39     @param dt: time step
40     @type dt: float or integer
41     @param par: list of parameters
42     @type par: list of Scalar
43     ''
44     self.t = [0] # keeps the time
45     self.dxdt = Matrix([[]]) # the time derivative of the state
46     self.par = par
47     self.initialConditions(x0)
48     self.initialisation()
49
50     # plug in desired integrator
51     self.integrator = odesolver(self, dt) # plug in the integrator
52
53     def initialConditions(self, x0):
54         self.x = [] # keeps the state over time
55         self.x.append(x0)
56
57     def initialisation(self):
58         ''
59         user initialisation section
60         set up:
61         - secondary states
62         - initial conditions / states
63         - flows
64         - production
65
66         ''
67         self.y = {}
68         return []
69
70     def rhs(self, t, x):
71         ''
72         <USER SPECIFIC>
73         Can be redefined through subclassing
74
75         Returns the value of the time derivative of the current state and stores
76         the newly calculated derivative.
77         ''
78         dxdt = self.par[0] * x
79         return dxdt
80
81     def integrateTimeInterval(self):
82         ''
83         Integrate over the given time interval; thus updating time and state.
84         The state is kept as list of one-column matrices.
85         ''
86         x = self.x[-1]
87         t = self.t[-1]
88         dt, dxdt, x_new = self.integrator.step(t, x)
89         self.dxdt.extend(dxdt)
90         self.x.append(x_new)
91         self.t.append(t + dt)
92
93     def getCSV(self):
94         s = ''
95         for i in range(len(self.t)):
96             s += self.t[i].__str__()

```

```

97     _x = str(self.x[i])
98     _x = replace(_x, '[', ', ', ', ')
99     _x = replace(_x, ',', ', ', ', ')
100    try:
101        s_y = ''
102        for i in self.y:
103            _y = str(self.y[i])
104            _y = replace(_y, '[', ', ', ', ')
105            _y = replace(_y, ',', ', ', ', ')
106            s_y += '%s,%s' % _y
107            print s_y
108    except: pass
109    s += '%s,%s\n' % (s_t, _x, s_y)
110

```

3.3.2 The expanded cracking reactor module

```

1  """
2  Acetone cracking reactor simulation
3  The model assumes a travelling batch reactor.
4
5
6  Created on Oct 29, 2012
7
8  @author: Preisig, Heinz A
9  @organization: NTNU, Chemical Engineering
10
11
12 from model_06 import Model
13 from matrix import Matrix, Scalar
14 from Integrator_vector_01 import Euler, RungeKutta
15 from gnuplotHAP import gnuplot
16 from copy import copy
17 from math import exp
18
19
20 class AcetonReactor(Model):
21     """
22         Acetone cracking reactor
23     """
24     def initialisation(self):
25         """
26             secondary states to be called at the beginning to populate
27             the initial dictionary for the secondary state
28         """
29         self.y = {}
30         T = self.par['T_o']
31         n_aceton = self.par['p_feed'] * self.par['V_Go'] / (self.par['R'] * T)
32
33
34         n = Matrix([[n_aceton], [0], [0]])
35         Ho = n.transpose() * (self.par['a0'] + self.par['a1'] * T / 2 +
36                               self.par['a2'] * T / 3 * T) * T
37         self.y['Ho'] = copy(Ho)
38         self.y['T_G'] = T
39
40         # set up initial conditions
41         x0 = Matrix([[n_aceton], [0], [0], [0]])
42         self.initialConditions(x0)
43
44         # setup dictionaries for the flows as a convinience
45         self.prod_n = {}
46         self.flow_q = {}

```

```

48     def rhs(self, t, x):
49         """
50             The balances for the reactor are the objective
51             @param t: time
52             @type t: float
53             @param x: state
54             @type x: Matrix
55         """
56         self.stateTrans(x);
57         self.reaction();
58         self.heatFlow();
59
60         dxdt = self.prod_n['G'];
61         dHdt = Matrix([[self.flow_q['G|C']]])
62         dxdt.extend(dHdt)
63         # print 'LHS:', dxdt
64         return dxdt
65
66     def stateTrans(self, x):
67         """
68             State variable transformation box providing the mapping between
69             the state and the secondary state
70             @param x: matrix (vector) of the primary state
71             @type x: Matrix (n x 1)
72         """
73         n = self.getCompMass()
74         # H = self.getEnthalpy()
75         self.root()
76         e = Matrix([[1, 1, 1]])
77         # print e.dim(), n.dim()
78         T_G = e * n * self.par['R'] * self.y['T_G'] / self.par['p']
79         self.y['V_G'] = Scalar(T_G)
80         self.y['c_G'] = n / T_G
81         pass
82
83     def reaction(self):
84         """
85             Provides the conversion using the information from the primary
86             and secondary state
87         """
88         k = self.par['a'] * exp(self.par['b'] *
89                               (1 / self.par['c'] - 1 / self.y['T_G']))
90         c_A = self.y['c_G'][0][0]
91         a = self.par['stoich'].transpose() * self.y['V_G']
92         b = a * k * c_A
93         self.prod_n['G'] = b
94         pass
95
96     def heatFlow(self):
97         """
98             heat flow, here only one
99         """
100        q_GC = -self.par['k_G|C'] * (self.par['T_C'] - self.y['T_G'])
101        self.flow_q['G|C'] = q_GC
102
103    def getCompMass(self):
104        """
105            Convenience function to get the component mass
106        """
107        return Matrix(self.x[-1][0:3])
108
109    def getEnthalpy(self):
110        """
111        """

```

```

113     #     print 'getH', self.x[-1][3]
114     return Scalar(self.x[-1][3][0])
115
116 def root(self):
117     n = self.getCompMass()
118     try:
119         H = self.getEnthalpy()
120     except:
121         print len(self.x)
122         Ho = self.y['Ho']
123         T_ = self.y['T_G']
124         T = T_ + 1
125         while abs(T - T_) > 0.1:
126             T_ = T
127             P = n.transpose() * (self.par['a0'] + self.par['a1'] * T_ / 2
128                                  + self.par['a2'] * T_ / 3 * T_)
129             r = -H - Ho + P
130             J = n.transpose() * (self.par['a0'] + self.par['a1'] * T_
131                                  + self.par['a2'] * T_ * T_)
132             T = T_ - r / J
133             self.y['T_G'] = T
134
135 if __name__ == '__main__':
136
137     # parameters :: dict
138     # conditions :: dict
139     # geometry   :: dict
140     # initial conditions :: dict
141
142
143     n_slices = 100
144
145     par = {}
146     par['R'] = Scalar(8.314)
147     par['stoich'] = Matrix([[-2, 1, 1]])
148     par['T_C'] = Scalar(280) # K
149     par['T_o'] = Scalar(1150) # K
150     par['V_Go'] = Scalar(1.0 / n_slices) # m3
151     par['a'] = Scalar(3.58)
152     par['b'] = Scalar(34222)
153     par['c'] = Scalar(1035)
154     par['k_G|C'] = Scalar(150 * 110.0 / n_slices)
155     par['a0'] = Matrix([[26.63], [20.04], [13.39]])
156     par['a1'] = Matrix([[0.1830], [0.0945], [0.0770]])
157     par['a2'] = Matrix([[0.00004586], [0.00003095], [0.00001871]])
158     par['p_feed'] = 162000 # Pa
159     par['p'] = par['p_feed']
160
161
162     dt = Scalar(0.000001)
163     n_samples = 1000
164
165
166
167     m = AcetonReactor(dt=dt, par=par, odesolver=Euler)
168     for i in range(0, n_samples): # step through the time interval
169         m.integrateTimeInterval()
170
171
172     txtfile = open('AcetonCracking_Euler.dat', 'w')
173     txtfile.write(m.getCSV())
174     txtfile.close()
175
176     m = AcetonReactor(dt=dt, par=par, odesolver=RungeKutta)

```

```

178     for i in range(0, n_samples): # step through the time interval
179         m.integrateTimeInterval()
180
181
182     txtfile = open('AcetonCracking_RungeKutta.dat', 'w')
183     txtfile.write(m.getCSV())
184     txtfile.close()
185
186
187     x_r = n_samples * dt
188     y_r = n_aceton = par['p_feed'] * par['V_Go'] / (par['R'] * par['T_o'])
189
190     plot = gnuplot(output='AcetonCracking',
191                     xlabel='t [s]', ylabel='n [mol]',
192                     xmin=0, xmax=x_r, ymin=0, ymax=y_r, \
193                     title='Aceton-Cracking')
194
195
196     plot.add('AcetonCracking_Euler.dat', x=1, y=2, width=2, color='blue',
197             type=1, title='model 1:Euler')
198     plot.add('AcetonCracking_Euler.dat', x=1, y=3, width=2, color='red',
199             type=1, title='model 1:Euler')
200     plot.add('AcetonCracking_Euler.dat', x=1, y=4, width=2, color='green',
201             type=1, title='model 1:Euler')
202     plot.add('AcetonCracking_RungeKutta.dat', x=1, y=2, width=2, color='blue',
203             type=2, title='model 1:Runge-Kutta')
204     plot.add('AcetonCracking_RungeKutta.dat', x=1, y=3, width=2, color='red',
205             type=2, title='model 1:Runge-Kutta')
206     plot.add('AcetonCracking_RungeKutta.dat', x=1, y=4, width=2, color='green',
207             type=2, title='model 1:Runge-Kutta')
208
209     plot.plot('AcetonCracking')

```

The Final Touch (TKP4106)



Zooball/Monkey

From a Real Programmer's diary:

- There is always a second bug.
- If it's possible to make a mistake, I've already made it.
- If it's possible to forget something, I've already forgotten it.
- If it's possible to postpone a task, I've already postponed it.
- If there is a simple solution to a problem it's most likely wrong.
- Anything that walks and quacks like a duck is probably something else.
- Make a clever design and you'll end up shooting yourself in the foot.
- Never trust someone else's code and especially not your own.
- Things take time — at least three times more than you expect.
- Every rule is a rule, but no rule is absolute.

Bjørn Tore Løvfall and Tore Haug-Warberg (2004 - 2008)

Assignments

1. Install **GNUpot** and **GhostScript** on your computer.
2. Download the plot files [graph.gp](#) and [graph.dat](#).
3. Plot the file content(s) from the command line. In a UNIX-style environment the commands are:

```
gnuplot graph.gp  
ps2pdf graph.ps  
open graph.pdf
```

The output shall be like this: [graph.pdf](#)

4. Modify your version of [ammonia reactor.py](#) to make it produce some decent **GNUpot** output. Make a template similar to [graph.gp](#) for plotting the calculated results.
5. Have Great Fun with the tools you've got!

Now, that you have finalized the (quite advanced) Plug Flow Reactor model, it is due time to sit back and think a little: What is a model? What is important for the model and what is not? This little paper on [Modelling perspectives \(Norwegian\)](#) talks about such things. In our model we have e.g. neglected the momentum

balance which means we have ignored sound waves in the system. Now, how important is that? In fact, would you be able to remove that restriction?

%Predefined.

HTML text.

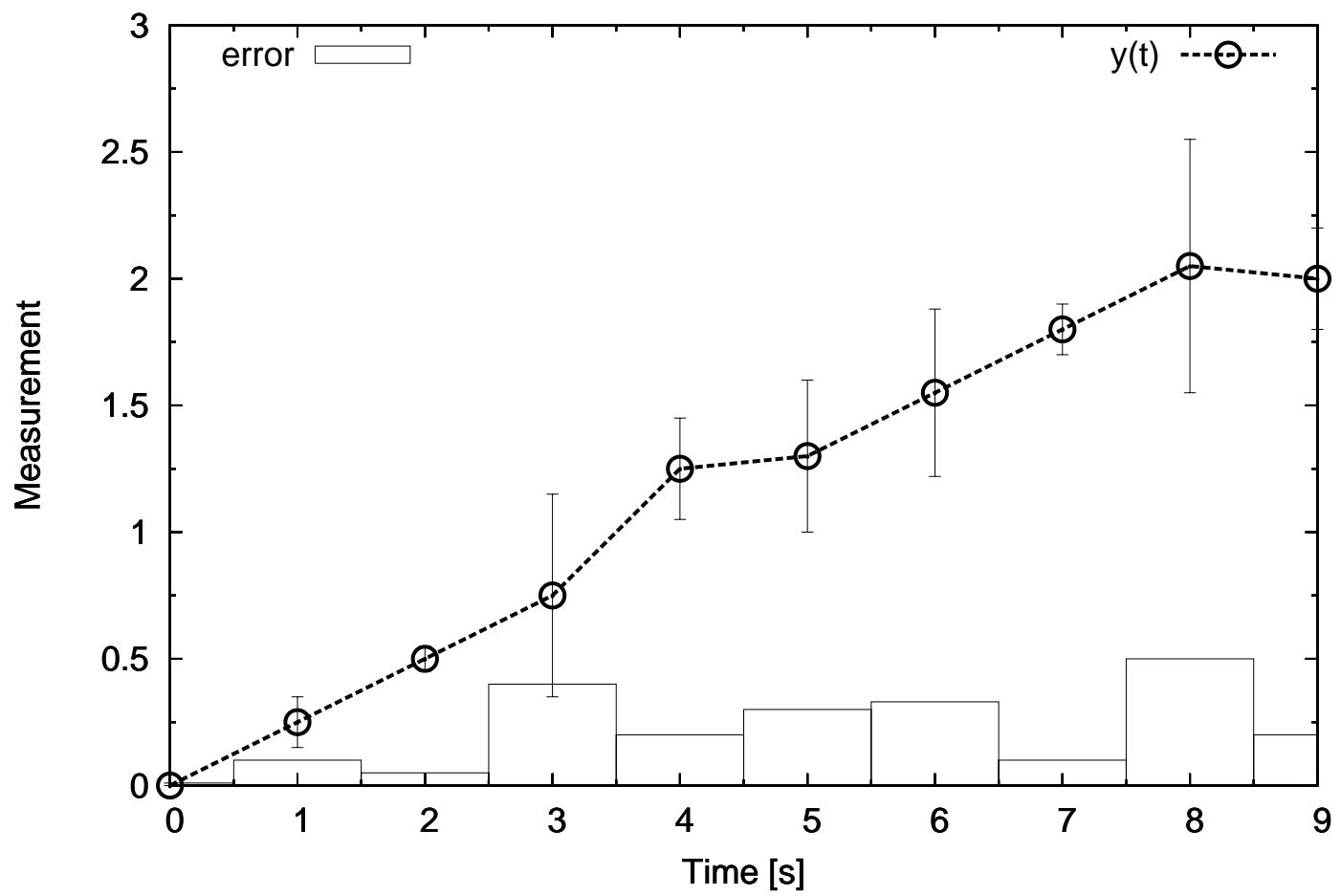
5.23.1 Verbatim: “graph.gp”

```
1  #!/sw/bin/gnuplot -persist
2  #
3  # Test script plotting a y(t) graph with error bars and
4  # separate boxes showing the error level. Data are loaded
5  # loaded from file "graph.dat" and dumped to "graph.ps".
6  #
7  set terminal postscript \
8      landscape noenhanced monochrome \
9      dashed defaultplex "Helvetica" 18
10 set output 'graph.ps'
11
12 set title 'Testing out GNUplot'
13 set xlabel 'Time [s]'
14 set ylabel 'Measurement'
15
16 set xrange [0:9]
17 set yrange [0:3]
18 set mxtics 2
19 set mytics 2
20
21 set style line 1 \
22     linetype 2 linewidth 4 pointsize 2 pointtype 6
23 set style line 2 \
24     linetype 1 linewidth 1 pointsize 0
25
26 set multiplot
27 set style data boxes
28 set key left
29
30 plot "graph.dat" using 1:3 \
31     title "error" linestyle 2
32
33 set style data lines
34 set key right
35
36 plot "graph.dat" using 1:2 \
37     title "y(t)" with linespoints linestyle 1
38
39 plot "graph.dat" using 1:2:3 \
40     notitle with yerrorbars linestyle 2
41
```

5.23.2 Verbatim: “graph.dat”

```
1 # graph.dat
2 #
3 # gnuplot ignores lines that start with #
4 #
5 # t y error-in-y
6 #
7 0 0      0.01
8 1 0.25   0.1
9 2 0.5    0.05
10 3 0.75   0.4
11 4 1.25   0.2
12 5 1.30   0.3
13 6 1.55   0.33
14 7 1.80   0.1
15 8 2.05   0.5
16 9 2.0    0.2
```

Testing out GNUplot



5.23.4 ammonia_reactor.py, see also Sec. 5.19.2

First reference occurs in *ammonia_reactor.py*, see Section 5.19.2 on page 335.

5.23.5 graph.gp, see also Sec. 5.23.1

First reference occurs in *graph.gp*, see Section 5.23.1 on page 399.



ANDREAS WAHL POPULÆRVITENSKAP NINA JENSEN MILJØ OG KLIMA ØYVIND STRØMSEN NETTETS MØRKE SIDER SUNNIVA GYLVER VERDIER TROND HAUKEDAL MOTIVASJON

Nye perspektiver

Jeg elsker modeller. Ikke supermodeller, de setter sjeldent viktige ting i perspektiv. Men det gjør fysiske modeller.

Fysiske modeller er en forenkling, forkorting eller forstørring av virkeligheten. La oss ta atmosfæren vår som eksempel. Den er 100 kilometer tykk. Det er en grense vi mennesker har funnet på, for det finnes ingen skarp overgang mellom atmosfære og det ytre rom der opp. Noen har tenkt at 100 kilometer er lett å huske.

Men hvor tykt er 100 kilometer? Tallet gir liten mening før det settes i sammenheng med størrelser vi kjenner fra før. Dersom det fantes en motorvei rett til værs ville én times biltur være nok til å reise til verdensrommet.

Men det er ikke godt nok for meg, så bli med inn i modellenes verden:

Forestill deg at du kunne krympe planeten vår så den ble på størrelse med en vannmelon, eller en typisk globus. Hvor tykk tror du atmosfæren ville være da? Hvor tykt ville luftlaget rundt vannmelonen være?

Svaret er én millimeter! Pow! DER fikk størrelsen mening. Atmosfæren rundt kloden vår er som et par ark avisepapir pakket rundt en vannmelon.

Krøll avisepapiret sammen til en ball og du vil innse at all luften vi har rundt kloden vår ikke er større enn en lime ved siden av vannmelonen. I den lille, grønne limen finnes alle skyer, all vind, alle verdens fly og all luft mennesker, dyr og planter trenger for å leve.

Hva så med tid? La oss krympe vår planet 4,5 milliarder år lange historie ned til en størrelse vi alle kjenner til: Et kalenderår.

Første januar dannes jorden og resten av solsystemet. Månedene som følger er temmelig ugjestmilde. Jeg ville holdt meg innendørs. Frem til juni har ikke planeten annet enn encellede skapninger som bakterier, alger og amøber å by på. Alle lever i vann, gjerne rundt varme kilder.

Først i august begynner det å skje ting. Da er havene fulle av skapninger. De mest klaustrofobe kaller seg amfibier og flykter opp på land. Først i oktober dukker dyr med hode opp, mens de første på fire bein ikke er å se før 1. desember.

Andre jul dag dør dinosaurene ut, og solen går opp nyttårsaften uten en eneste homosapiens å se. Først sent på kvelden på årets aller siste dag dukker de første menneskene opp.

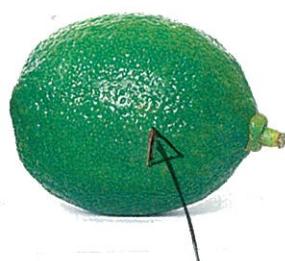
Rundt klokken 23.55 oppstår den moderne sivilisa-

sjon, snarlig etterfulgt av prostitusjon. Egyptere, babylonere, grekere og romere bruker så et minutt hver til å bygge sine turistattraksjoner. Slaget på Stiklestad finner sted fem sekunder før midnatt. Norske aviser blir første gang utgitt mindre enn ett sekund før midnatt.

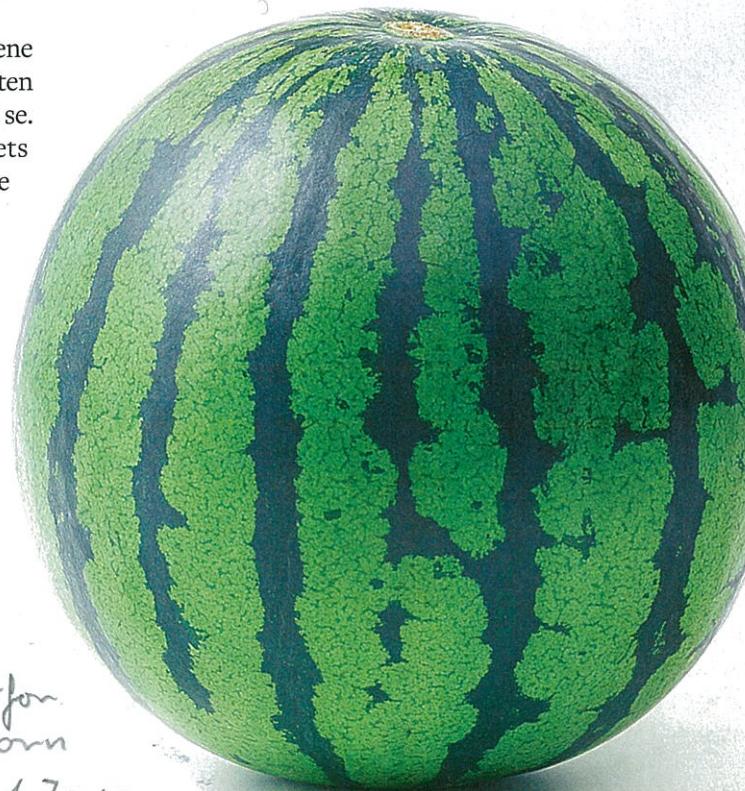
I det samme, siste sekundet har vi mennesker klart å rokke ved balansen i den florlette og avisepapirynne atmosfæren som omgir vannmelonen vår.

Modeller setter ting i perspektiv.

PS. Vi fysikere elsker også å runde av. Tallene og størrelsene i denne teksten er grove tilnærminger.



En lime er altfor stor - et risikorn med diameter 1.7 mm er mer korrekt.



Exercise 11

Preisig, H A and Maryam Ghadrdan

Chemical Engineering, NTNU

1 Question: Topology 09- Solar reactor

1.1 Excerpt from the paper: reactor description

Begin citation¹:

The Solar Reactor. The reactor works in a beam down configuration [22] namely the concentrated sun light is entering the reactor at the top. It is a two-cavity reactor allowing for indirect heating of the reactants through a separation wall between the upper cavity accepting the concentrated irradiation and the lower cavity with the reactants. General design guidelines for two cavity reactors are provided in [23]. Based on the experiences with the earlier designs [14,21] we built a batch reactor with a flat separation wall as shown in Fig. 1. The upper cavity and thus the upper side of the separation wall is subjected to concentrated solar irradiation entering the reactor through a quartz window. The window is protected against condensable gases and particles by the separation wall and an additional inert gas flow. To increase the input solar flux density a secondary concentrator entrance diameter = 80 mm, exit diameter= 65 mm is mounted above the aperture. Furthermore a "45 deg mirror" is used for redirecting the horizontal beam of PSIs solar furnace into a vertical beam entering the secondary concentrator compare Figs. 1 and 2. In the lower cavity a fixed bed of a ZnO-carbon mixture is heated indirectly from the top by radiation emitted by the lower side of the hot separation wall. Both cavities are thermally well insulated to reduce heat losses due to thermal conduction. The walls of the lower cavity are lined by SiC plates for reducing diffusion of zinc vapor into the insulation material. Due to the chemical reaction the fixed bed is shrinking and the gaseous products are leaving the reactor via an outlet pipe made of SiC.

End citation.

Begin citation²:

The solar reactor for the carbothermal reduction of ZnO is shown schematically in [Fig. 3](#) *Figure 1*. It consists of two cavities in series, of which the upper one is functioning as the solar absorber and the lower one as the reaction chamber containing a ZnO/C packed bed [27,49]. The net reaction, represented by $ZnO + C = Zn(g) + CO(g)$, proceeds endothermically at reasonable rates at above 1300 K. Thus, this reactor belongs to the indirect-irradiation, batch-operation, $s + sg$ category. A 5 kW reactor prototype was experimentally investigated at PSIs high-flux solar furnace [56] in the 400 – 1600 K range. The kinetic rate law expression and the activation energy E_a of 201.5 kJ/mol are taken from Refs. [27,49]. The identified parameters and their final values are summarized in Table 2. Heat transfer to the reactor wall is dominated by thermal radiation. At

¹S. Kräupl, U. Frommherz, C. Wieckert; Solar Carbothermic Reduction of ZnO in a Two-Cavity Reactor: Laboratory Experiments for a Reactor Scale-Up; Transactions of the ASME; Vol. 128, February 2006

²Jörg Petrasch, Philippe Osch, Aldo Steinfeld; Dynamics and control of solar thermochemical reactors; Chemical Engineering Journal 145 (2009) 362370

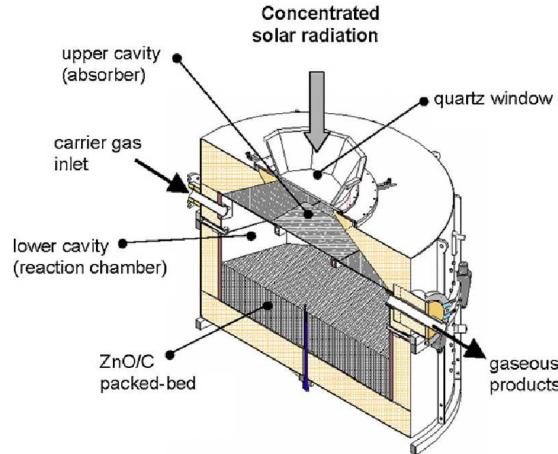


Figure 1: Solar reactor for the reduction of ZnO

$T_R > 1500 K$, the rate of radiative heat transfer is much faster than that of conductive heat transfer through the insulation, as indicated by linearization of radiative heat transfer rate: $\sigma T_{R0}^3 \gg 2 k_{iso}/d_{iso}$. Note that reactor wall and insulation are modeled as a single reservoir. Thus, heat transfer to the insulation is rate-controlled by conduction. The mean insulation thickness is $d = 0.08m$, the surface area of the upper and lower cavities are $A_u = 0.05 m^2$ and $A_l = 0.08 m^2$, and the thermal conductivity of the porous insulation is $k = 0.3 W/mK$. Assuming steady-state conduction heat transfer in a 1-D plane layer, the UA obtained is in the range $0.40.6 W/K$. As expected, the identified values 0.77 and $0.9 W/K$ for UA_l and UA_u , respectively, are higher than the steady state-based estimates since the temperature profile at the wall/insulation is steeper immediately after an external temperature change than in the steady state. Hence, heat transfer rates are higher during transients than in steady state.

End citation.

The authors provide a single-lump model, thus a first-order behaviour. We shall not look at the numerical details. Rather we shall concentrate to generate a topology, which potentially fits the observed behaviour better than the published model does. Indications of the misfit are in the text itself. One of the stated observations are lifted out by showing it in bold face.

1.2 Tasks:

- Sketch your view of the topology of the system in terms of lumped systems
- Provide a complete description of the plant. Set up a table with the first column being the equations, the second the variable that you solve the equation for, the third what variable need to be known, the last what is given (parameter or conditions == state-dependent information of the environment, assumptions about a state of the system etc.).

2 Question: Dynamics 06 - simple absorption process

We have a look at a simple process in which a reaction takes place on a solid.

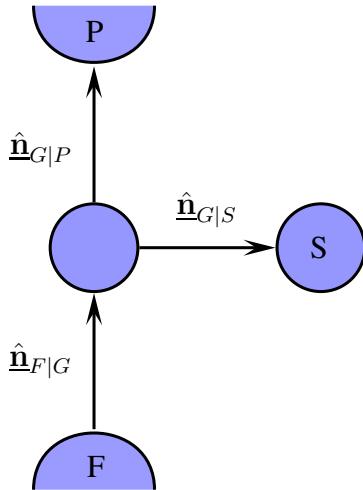


Figure 2: A simple absorption process

2.1 Model

Assume that

- Mass transfer in and out of the reactor is a convective flow that is driven (linearly) by pressure difference
- Mass transfer between gas phase and solid is a diffusion flow also driven (linearly) partial pressure difference.
- The pressure in the solid is the one in the open space in the solid, which is assumed ideally mixed, as can be seen from the topology.
- The diffusion/reaction in the solid can be modelled as a homogeneous reaction system.

2.2 Tasks

1. Provide a complete description of the plant. Set up a table with the first column being the equations, the second the variable that you solve the equation for, the third what variable need to be known, the last what is given (parameter or conditions == state-dependent information of the environment, assumptions about a state of the system, like constant temperature, constant pressure etc.). You may assume constant temperature.
2. Code the equations into your program retaining as much of the structure as possible, thus have a module with the balances, modules for each type of transfer, the reaction, and the necessary state variable transformations.

3. Implement different reactions:

- $2A \rightarrow B$ $A, B ::$ gases
- $A + S \rightleftharpoons AS$ $A ::$ gas, S solid active positions.

4. Make the above to one stage in a column, thus pile up stages and simulate again. Use the stage as a module, define what affects the stage from the immediate environment and use this to build the tower of stages.

5. On the parameter: choose your own !

Start with the balances and the integral of the differential quantities. Then expand each term until things are represented by parameters, states from the integration and things you know, like universal constants, temperature (assuming isothermal), state (conditions) of the environment.

1 Suggested solution: Thermochemical solar reactor

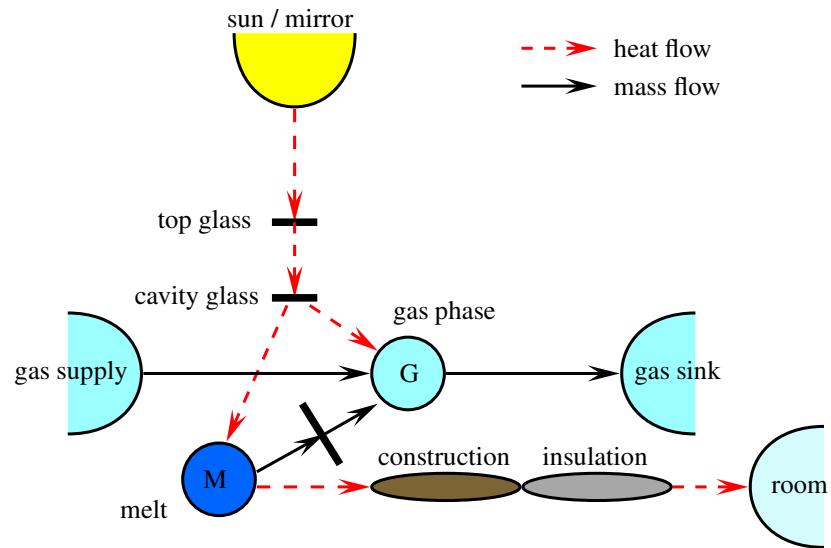


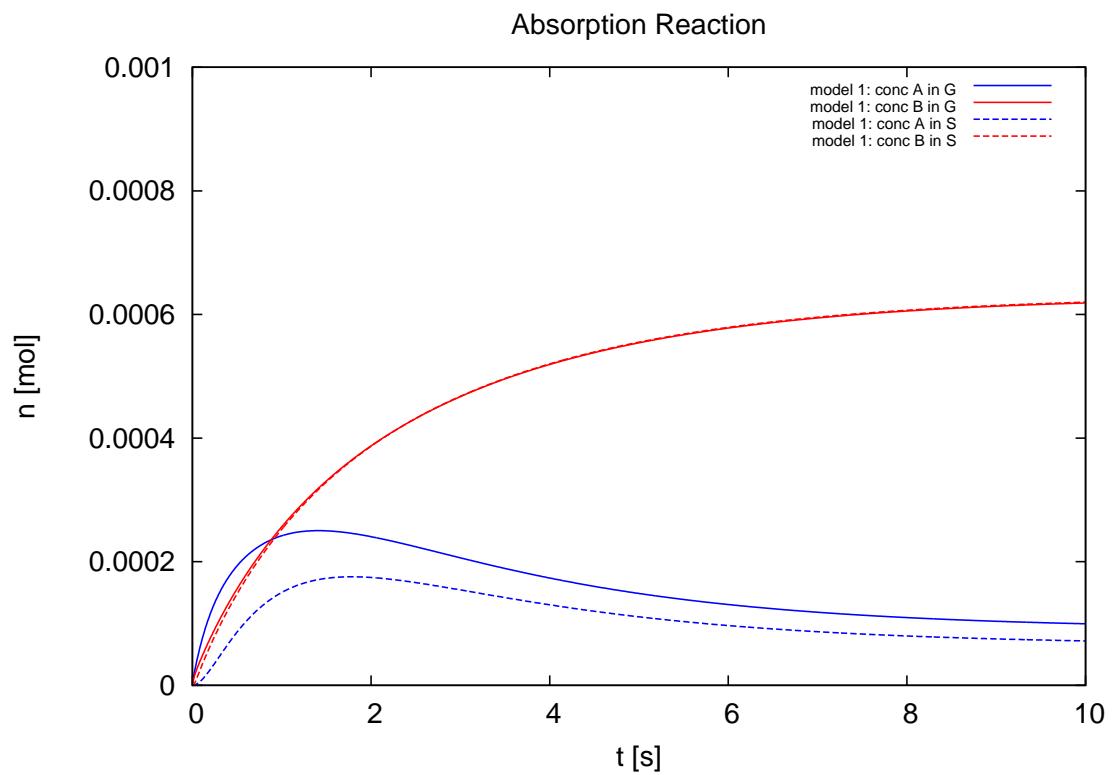
Figure 1: Topology of solar thermo-chemical reactor

2 Solution: Dynamics 06 - simple absorption process

The model equations are:

equations	res	vars	given
$\underline{n}_G := \int_0^t \dot{\underline{n}}_G dt + \underline{n}_G^o$	\underline{n}_G	$\dot{\underline{n}}_G$	\underline{n}_G^o
$\underline{n}_S := \int_0^t \dot{\underline{n}}_S dt + \underline{n}_S^o$	\underline{n}_S	$\dot{\underline{n}}_S$	\underline{n}_S^o
$\dot{\underline{n}}_G = \hat{\underline{n}}_{F G} - \hat{\underline{n}}_{G_P} - \hat{\underline{n}}_{G S}$	$\dot{\underline{n}}_G$	$\hat{\underline{n}}_{F G}, \hat{\underline{n}}_{G_P}, \hat{\underline{n}}_{G S}$	
$\dot{\underline{n}}_S = \hat{\underline{n}}_{G S} + \tilde{\underline{n}}_S$	$\dot{\underline{n}}_S$	$\hat{\underline{n}}_{G S}, \tilde{\underline{n}}_S$	
$\hat{\underline{n}}_{F G} := \underline{c}_F \hat{V}_{F G}$	$\hat{\underline{n}}_{F G}$	$\underline{c}_F, \hat{V}_{F G}$	
$\hat{\underline{n}}_{G P} := \underline{c}_G \hat{V}_{G P}$	$\hat{\underline{n}}_{G P}$	$\underline{c}_G, \hat{V}_{G P}$	
$\hat{\underline{n}}_{G S} := -k_{G S} (\underline{p}_S - \underline{p}_G)$	$\hat{\underline{n}}_{G S}$	$\underline{p}_S, \underline{p}_G$	$k_{G S}$
$\hat{V}_{F G} := -k_{F G} (p_G - p_F)$	$\hat{V}_{F G}$	p_G, p_F	$k_{F G}$
$\hat{V}_{G P} := -\underline{\underline{K}}_{G P} (p_P - p_G)$	$\hat{V}_{G P}$	p_G	$\underline{\underline{K}}_{G P}, p_P$
$\tilde{\underline{n}}_S := \underline{\underline{N}}^T V_S k_G g(\underline{p}_S)$	$\tilde{\underline{n}}_S$	$g(\underline{p}_S)$	$\underline{\underline{N}}^T, V_S, k_G$
$g(\underline{p}_S) := [1, 0] \underline{p}_S$	$g(\underline{p}_S)$	\underline{p}_S	
$p_G := [1, \dots, 1] \underline{p}_G$	p_G	\underline{p}_G	
$p_S := [1, \dots, 1] \underline{p}_S$	p_S	\underline{p}_S	
$p_F := [1, \dots, 1] \underline{p}_F$	p_F		\underline{p}_F
$\underline{p}_G := V_G^{-1} \underline{n}_G R T$	\underline{p}_G	\underline{n}_G	V_G, R, T
$\underline{p}_S := V_S^{-1} \underline{n}_S R T$	\underline{p}_S	\underline{n}_S	V_S, R, T
$\underline{c}_F := \underline{p}_F / (R T)$	\underline{c}_F	\underline{p}_F	R, T
$\underline{c}_G := \underline{p}_G / (R T)$	\underline{c}_G	\underline{p}_G	R, T

2.1 Plot



2.2 Python code

2.2.1 The model module

```
1  ''
2  In this module, the term "model" is used for the complete "thing" including the
3  differential equation *and* the integrator.
4
5  2012-10-11 created (HAP).
6  2012-10-13 extended to multiple integrators and demo for re-use (HAP).
7  2012-10-16 replaced an ugly if-elif-else testing of the ode solver method with a
8      first class function object (THW).
9
10 2012-10-11 created
11 2012-10-13 extended to multiple integrators and demo for re-use
12 2012-11-14 extended to output also the secondary states put into self.y by the
13      the user-defined model, that is, if it is there.
14
15 @author: Preisig, Heinz A
16 @organization: NTNU, Chemical Engineering
17 ''
18
19 import Integrator_vector_01 as ODE
20 from matrix import Matrix, Scalar
21 import math
22 from gnuplot import gnuplot
23 from string import replace
24
25 class Model:
26     ''
27     The model is the differential equations and the integrator
28     ''
29
30     def __init__(self, x0=Matrix([[0]]), dt=1.0, par=[], odesolver=ODE.Euler):
31         ''
32         The initialization gets the initial conditions, the parameters, and the
33         step size for the time stepper - being an integrator.
34         Note that the resulting state is stored as a list of vectors, whereby vectors are
35             matrices of the dimension n,1.
36
37         @param x0: initial conditions
38         @type x0: Matrix
39         @param dt: time step
40         @type dt: float or integer
41         @param par: list of parameters
42         @type par: list of Scalar
43         ''
44         self.t = [0] # keeps the time
45         self.dxdt = Matrix([[]]) # the time derivative of the state
46         self.par = par
47         self.initialConditions(x0)
48         self.initialisation()
49
50         # plug in desired integrator
51         self.integrator = odesolver(self, dt) # plug in the integrator
52
53     def initialConditions(self, x0):
54         self.x = [] # keeps the state over time
55         self.x.append(x0)
56
57     def initialisation(self):
58         ''
59         user initialisation section
60         set up:
```

```

61     - secondary states
62     - initial conditions / states
63     - flows
64     - production
65
66     ''
67     self.y = {}
68     return []
69
70 def rhs(self, t, x):
71     ''
72     <USER SPECIFIC>
73     Can be redefined through subclassing
74
75     Returns the value of the time derivative of the current state and stores
76     the newly calculated derivative.
77     ''
78     dxdt = self.par[0] * x
79     return dxdt
80
81 def integrateTimeInterval(self):
82     ''
83     Integrate over the given time interval; thus updating time and state.
84     The state is kept as list of one-column matrices.
85     ''
86     x = self.x[-1]
87     t = self.t[-1]
88     dt, dxdt, x_new = self.integrator.step(t, x)
89     self.dxdt.extend(dxdt)
90     self.x.append(x_new)
91     self.t.append(t + dt)
92
93 def getCSV(self):
94     s = ''
95     for i in range(len(self.t)):
96         s_t = self.t[i].__str__()
97         _x = str(self.x[i])
98         _x = replace(_x, '[', ', ')
99         _x = replace(_x, ']', ', ')
100        try:
101            s_y = ''
102            for i in self.y:
103                _y = str(self.y[i])
104                _y = replace(_y, '[', ', ')
105                _y = replace(_y, ']', ', ')
106                s_y += '%s,%s\n' % (s_t, _x, _y)
107                print s_y
108        except: pass
109    s += '%s,%s,%s\n' % (s_t, _x, s_y)
110
111

```

2.2.2 The expanded cracking reactor module

```

1   ''
2   Acetone cracking reactor simulation
3   The model assumes a travelling batch reactor.
4
5
6   Created on Oct 29, 2012
7
8   @author: Preisig, Heinz A
9   @organization: NTNU, Chemical Engineering
10
11

```

```

12 from model_06 import Model
13 from matrix import Matrix, Scalar
14 from Integrator_vector_01 import Euler, RungeKutta
15 from gnuplotHAP import gnuplot
16
17
18 class AbsorptionReactor(Model):
19     """
20         Acetone cracking reactor
21     """
22     def initialisation(self):
23         """
24             secondary states to be called at the beginning to populate
25             the initial dictionary for the secondary state
26         """
27         # parameters
28         # # conductivities
29         # ## volume flow
30         self.par = {}
31         self.par['k_FG'] = Scalar(1)
32         self.par['k_GP'] = Scalar(0.1)
33         # ## diffusional flow
34         self.par['K_GS'] = Matrix([[0.001, 0], [0, 0.01]])
35         # # kinetic
36         # ## stoichiometry
37         self.par['N'] = Matrix([[-2, 1]])
38         self.par['k'] = Scalar(0.5)
39         self.par['V_G'] = Scalar(1)
40         self.par['V_S'] = Scalar(1)
41         self.par['R'] = Scalar(8.314) # J/(K mol)
42
43         # secondary states
44         self.y = {}
45         # # P_x :: total pressure in x
46         self.y['P_P'] = Scalar(0);
47         # # p_x :: partial pressure in x
48         self.y['p_F'] = Matrix([[1], [1]])
49         # # temperature in the plant
50         self.y['T'] = Scalar(300); # K
51
52         # flows
53         self.flows = {}
54
55         # production
56         self.prod = {}
57
58         # set up initial conditions
59         self.par['R*T'] = self.par['R'] * self.y['T']
60         x0 = Matrix([[0.01], [0], [0], [0.00001]])
61         x0 = x0 / self.par['R*T']
62         self.initialConditions(x0)
63
64
65     def rhs(self, t, x):
66         """
67             The balances for the reactor are the objective
68             @param t: time
69             @type t: float
70             @param x: state
71             @type x: Matrix
72         """
73         self.stateTrans(x);
74         self.reaction();
75         self.flow();
76

```

```

77
78 dnGdt = self.flows['n_FG'] - self.flows['n_GP'] - self.flows['n_GS']
79 dnSdt = self.flows['n_GS'] + self.prod['S']
80 dxdt = dnGdt
81 dxdt.extend(dnSdt)
82 return dxdt
83
84 def stateTrans(self, x):
85     """
86         State variable transformation box providing the mapping between
87         the state and the secondary state
88         @param x: matrix (vector) of the primary state
89         @type x: Matrix (n x 1)
90         """
91     n_G = self.getCompMass('G')
92     #     print 'G', n_G
93     n_S = self.getCompMass('S')
94     #     print 'S', n_G
95     #     R = self.par['R']
96     V_G = self.par['V_G']
97     V_S = self.par['V_S']
98     #     T = self.y['T']
99     p_F = self.y['p_F']
100
101    a = self.par['R*T']
102    b = a / V_G
103    self.y['p_G'] = b * n_G;
104    self.y['p_S'] = b * n_S;
105
106    self.y['c_F'] = p_F / a;
107    self.y['c_G'] = n_G / V_G;
108    self.y['c_S'] = n_S / V_S;
109
110    one = Matrix([[1, 1]])
111    self.y['x_G'] = n_G / (one * n_G);
112    self.y['x_S'] = n_S / (one * n_S);
113
114    self.y['P_G'] = one * self.y['p_G'];
115    self.y['P_S'] = one * self.y['p_S'];
116    self.y['P_F'] = one * p_F;
117
118 def reaction(self):
119     """
120         Provides the conversion using the information from the primary
121         and secondary state
122         """
123
124     n_A = self.getCompMass('S')[0][0]
125     self.prod['S'] = self.par['N'].transpose() * self.par['V_S'] * \
126             self.par['k'] * n_A
127
128 def flow(self):
129     """
130         all flows
131         """
132
133         # convective mass flow
134         self.flows['V_FG'] = -self.par['k_FG'] * (self.y['P_G'] - self.y['P_F'])
135         self.flows['V_GP'] = -self.par['k_GP'] * (self.y['P_P'] - self.y['P_G'])
136         self.flows['n_FG'] = self.y['c_F'] * self.flows['V_FG']
137         self.flows['n_GP'] = self.y['c_G'] * self.flows['V_GP']
138
139         # diff mass flow
140         neg = Scalar(-1) # that is necessary because of the operation definitions
141                         # int * matrix is not defined
142         self.flows['n_GS'] = neg * self.par['K_GS'] * (self.y['p_S'] - self.y['p_G'])

```

```

142
143     def getCompMass(self, system):
144         """
145             Convenience function to get the component mass
146         """
147         if system == 'G':
148             return Matrix(self.x[-1][:2])
149         if system == 'S':
150             return Matrix(self.x[-1][2:])
151
152
153     if __name__ == '__main__':
154
155
156         dt = Scalar(0.01)
157         n1 = 20
158         n_samples = 1000
159
160
161         m = AbsorptionReactor(dt=dt, odesolver=RungeKutta)
162         for i in range(0, n1): # step through the time interval
163             m.integrateTimeInterval()
164
165         # dt = Scalar(0.005)
166         m.setTimeStep(dt)
167         for i in range(n1, n_samples):
168             m.integrateTimeInterval()
169
170         txtfile = open('AbsorptionReactor.dat', 'w')
171         txtfile.write(m.getCSV())
172         txtfile.close()
173
174
175         x_r = dt * n_samples
176         y_r = 0.001
177
178         plot = gnuplot(output='AbsorptionReactor',
179                         xlabel='t [s]', ylabel='n [mol]',
180                         xmin=0, xmax=x_r, ymin=0, ymax=y_r, \
181                         title='Absorption Reaction')
182
183         plot.add('AbsorptionReactor.dat', x=1, y=2, width=2, color='blue',
184                  type=1, title='model_1: conc_A_in_G')
185         plot.add('AbsorptionReactor.dat', x=1, y=3, width=2, color='red',
186                  type=1, title='model_1: conc_B_in_G')
187         plot.add('AbsorptionReactor.dat', x=1, y=4, width=2, color='blue',
188                  type=2, title='model_1: conc_A_in_S')
189         plot.add('AbsorptionReactor.dat', x=1, y=5, width=2, color='red',
190                  type=2, title='model_1: conc_B_in_S')
191
192         plot.plot('AbsorptionReactor')
193         print 'end'

```

2.2.3 Integrator module

```

1 """
2 2012-10-11 created
3 2012-10-13 step returns dt, dxdt and x_new
4 2012-10-21 extend to matrix operations
5
6
7 @author: Preisig, Heinz A
8 @organization: NTNU, Chemical Engineering
9 """

```

```

10  from matrix import Scalar
11
12  class Stepper(object):
13      """
14          Implements different "steppers" for computing ODE integrals.
15      """
16
17  def __init__(self, model, dt):
18      """
19          Keeps the model and the (initial) time step.
20      """
21      self.model = model
22      self.dt = dt
23
24  class Euler(Stepper):
25      """
26          Implements the Euler method.
27      """
28
29  def step(self, t, x):
30      dxdt = self.model.rhs(t, x)
31      x_new = x + dxdt * self.dt;
32      return self.dt, dxdt, x_new
33
34  class RungeKutta(Stepper):
35      """
36          Implements the Runge-Kutta 3:4 method.
37          Pending...
38      """
39
40  def step(self, t, x):
41      h = self.dt
42      k1 = h * self.model.rhs(t, x)
43      k2 = h * self.model.rhs(t + Scalar(0.5) * h, x + Scalar(0.5) * k1)
44      k3 = h * self.model.rhs(t + Scalar(0.5) * h, x + Scalar(0.5) * k2)
45      k4 = h * self.model.rhs(t + h, x + k3)
46      x_new = x + Scalar(1.0 / 6.0) * (k1 + Scalar(2.0) * k2 +
47                                         Scalar(2.0) * k3 + k4)
48      dxdt = (x_new - x) / h
49      return self.dt, dxdt, x_new

```