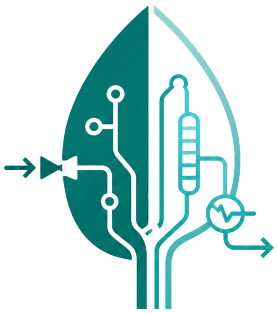


Choupo

The Open Source Chemical Process Simulator

User Guide

Version 2607

30 June 2026

Authors: Vítor Geraldes.

Typeset in LaTeX.

Acknowledgment. Choupo was conceived and architected by Vítor Geraldes. Substantial parts of the initial implementation, documentation, and tutorial corpus were produced with assistance from Anthropic Claude Code using Claude Opus/Fable models. This guide is published under human curation, review, and editorial responsibility by Vítor Geraldes.

License

Rights.

e. Effective Technological Measures means those measures that, in the absence of proper authority, may not be circumvented under laws fulfilling obligations under Article 11 of the WIPO Copyright Treaty adopted on December 20, 1996, and/or similar international agreements.

f. Exceptions and Limitations means fair use, fair dealing, and/or any other exception or limitation to Copyright and Similar Rights that applies to Your use of the Licensed Material.

g. License Elements means the license attributes listed in the name of a Creative Commons Public License. The License Elements of this Public License are Attribution and ShareAlike.

h. Licensed Material means the artistic or literary work, database, or other material to which the Licensor applied this Public License.

i. Licensed Rights means the rights granted to You subject to the terms and conditions of this Public License, which are limited to all Copyright and Similar Rights that apply to Your use of the Licensed Material and that the Licensor has authority to license.

j. Licensor means the individual(s) or entity(ies) granting rights under this Public License.

k. Share means to provide material to the public by any means or process that requires permission under the Licensed Rights, such as reproduction, public display, public performance, distribution, dissemination, communication, or importation, and to make material available to the public including in ways that members of the public may access the material from a place and at a time individually chosen by them.

l. Sui Generis Database Rights means rights other than copyright resulting from Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, as amended and/or succeeded, as well as other essentially equivalent rights anywhere in the world.

m. You means the individual or entity exercising the Licensed Rights under this Public License. Your has a corresponding meaning.

Section 2 -- Scope.

a. License grant.

1. Subject to the terms and conditions of this Public License, the Licensor hereby grants You a worldwide, royalty-free, non-sublicensable, non-exclusive, irrevocable license to exercise the Licensed Rights in the Licensed Material to:

a. reproduce and Share the Licensed Material, in whole or in part; and

b. produce, reproduce, and Share Adapted Material.

2. Exceptions and Limitations. For the avoidance of doubt, where Exceptions and Limitations apply to Your use, this Public License does not apply, and You do not need to comply with its terms and conditions.

3. Term. The term of this Public License is specified in Section 6(a).

4. Media and formats; technical modifications allowed. The Licensor authorizes You to exercise the Licensed Rights in all media and formats whether now known or hereafter created, and to make technical modifications necessary to do so. The Licensor waives and/or agrees not to assert any right or authority to forbid You from making technical modifications necessary to exercise the Licensed Rights, including technical modifications necessary to circumvent Effective Technological Measures. For purposes of this Public License, simply making modifications authorized by this Section 2(a) (4) never produces Adapted Material.

5. Downstream recipients.

a. Offer from the Licensor -- Licensed Material. Every recipient of the Licensed Material automatically receives an offer from the Licensor to exercise the Licensed Rights under the terms and conditions of this Public License.

b. Additional offer from the Licensor -- Adapted Material. Every recipient of Adapted Material from You automatically receives an offer from the Licensor to exercise the Licensed Rights in the Adapted Material under the conditions of the Adapter's License You apply.

c. No downstream restrictions. You may not offer or impose any additional or different terms or conditions on, or apply any Effective Technological Measures to, the Licensed Material if doing so restricts exercise of the Licensed Rights by any recipient of the Licensed Material.

6. No endorsement. Nothing in this Public License constitutes or
   may be construed as permission to assert or imply that You
   are, or that Your use of the Licensed Material is, connected
   with, or sponsored, endorsed, or granted official status by,
   the Licensor or others designated to receive attribution as
   provided in Section 3(a)(1)(A)(i).

b. Other rights.

1. Moral rights, such as the right of integrity, are not
   licensed under this Public License, nor are publicity,
   privacy, and/or other similar personality rights; however, to
   the extent possible, the Licensor waives and/or agrees not to
   assert any such rights held by the Licensor to the limited
   extent necessary to allow You to exercise the Licensed
   Rights, but not otherwise.

2. Patent and trademark rights are not licensed under this
   Public License.

3. To the extent possible, the Licensor waives any right to
   collect royalties from You for the exercise of the Licensed
   Rights, whether directly or through a collecting society
   under any voluntary or waivable statutory or compulsory
   licensing scheme. In all other cases the Licensor expressly
   reserves any right to collect such royalties.

Section 3 -- License Conditions.

Your exercise of the Licensed Rights is expressly made subject to the
following conditions.

a. Attribution.

1. If You Share the Licensed Material (including in modified
   form), You must:

   a. retain the following if it is supplied by the Licensor
      with the Licensed Material:

      i. identification of the creator(s) of the Licensed
         Material and any others designated to receive
         attribution, in any reasonable manner requested by
         the Licensor (including by pseudonym if
         designated);

      ii. a copyright notice;

      iii. a notice that refers to this Public License;

      iv. a notice that refers to the disclaimer of
          warranties;

      v. a URI or hyperlink to the Licensed Material to the
         extent reasonably practicable;

   b. indicate if You modified the Licensed Material and
      retain an indication of any previous modifications; and

   c. indicate the Licensed Material is licensed under this
      Public License, and include the text of, or the URI or
      hyperlink to, this Public License.

2. You may satisfy the conditions in Section 3(a)(1) in any
   reasonable manner based on the medium, means, and context in
   which You Share the Licensed Material. For example, it may be
   reasonable to satisfy the conditions by providing a URI or
   hyperlink to a resource that includes the required
   information.

3. If requested by the Licensor, You must remove any of the
   information required by Section 3(a)(1)(A) to the extent
   reasonably practicable.

b. ShareAlike.

   In addition to the conditions in Section 3(a), if You Share
   Adapted Material You produce, the following conditions also apply.

1. The Adapter's License You apply must be a Creative Commons
   license with the same License Elements, this version or
   later, or a BY-SA Compatible License.

2. You must include the text of, or the URI or hyperlink to, the
   Adapter's License You apply. You may satisfy this condition
   in any reasonable manner based on the medium, means, and
   context in which You Share Adapted Material.

3. You may not offer or impose any additional or different terms
   or conditions on, or apply any Effective Technological
   Measures to, Adapted Material that restrict exercise of the
   rights granted under the Adapter's License You apply.

Section 4 -- Sui Generis Database Rights.

Where the Licensed Rights include Sui Generis Database Rights that
apply to Your use of the Licensed Material:

Trademarks

Choupo is a trademark of TalentGround Lda.

Anthropic, Claude, and Claude Code are trademarks or registered trademarks of Anthropic, PBC.

ParaView is a trademark of Kitware Inc.

Contents

Preface — read this first

Engineers used to text-driven, file-first tools will feel at home. Newcomers will feel curious, then liberated. Choupo is an *educational* process simulator: every Newton iteration, every K-value, every Wegstein step is visible both in the source and at runtime. If something surprises you, the answer is always one `grep` away.

This guide walks you from a cold install to running, reading, modifying, and building your own cases: first the case anatomy (the plain-text dictionaries and the per-stream state files), then the thermodynamic foundation, then every family of unit operations with a minimal working dict, then the outer layers — sweeps, design specifications, optimisation, sizing and costing — and finally the time-dependent binaries, the property workbench, and the browser front-end. Each section opens with a *What you'll learn* note; boxes mark crystal-clear takeaways, common pitfalls, and small experiments you should run as you read.

Four binaries cover four problem classes, and one dispatcher (`runCase`) picks the right one for each case:

Binary	Problem class	Mathematics
`choupoSolve`	steady-state flowsheets	$F(x) = 0$ root-finding; Newton-on-tears recycles
`choupoBatch`	batch vessels + recipes	$\mathrm{d}Y/\mathrm{d}t = f(Y)$, RK4 or stiff Rosenbrock
`choupoCtrl`	dynamic continuous + control	$\mathrm{d}Y/\mathrm{d}t = f(Y, u, t)$ with controllers writing u
`choupoProps`	property evaluation	points, scans, consistency tests, LM fits

1 Quick start

What you'll learn. How to build the simulator once, set up your shell environment, and run your first tutorial in under five minutes.

Choupo needs only a C++17 compiler (`g++`) and GNU `make` — no external libraries. Build once, source the shell environment once, then run any tutorial:

```
cd /path/to/Choupo
make all                                          # release build
source etc/bashrc                                 # adds bin/ + project root to PATH

listCases                                         # available tutorials
runCase tutorials/steady/flash/flash01_benzene_toluene     # run one
less tutorials/steady/flash/flash01_benzene_toluene/log.choupoSolve
```

Tutorials live under `tutorials/<category>/<family>/<name>/`, where the category is one of `steady/` (`choupoSolve`), `batch/` (`choupoBatch`), `ctrl/` (`choupoCtrl`), `props/` (`choupoProps`), `electrochem/`, or `plant/` (multi-sector cases). `runCase` reads each case's `controlDict.application` and dispatches to the right binary automatically — you never pick the executable by hand. After your first build a symlink per binary (e.g. `./choupoSolve`) points to the matching `build/<platform>/` executable.

1.1 Permanent shell setup

Append this line to `~/.bashrc` so every new terminal has the environment ready:

```
source /path/to/Choupo/etc/bashrc
```

A fresh terminal then greets you with:

```
Choupo environment loaded.
  CHOUPO_HOME      = /path/to/Choupo
  CHOUPO_WORKSPACE = /path/to/Choupo/cases   (your cases; newCase writes here)
  Commands available  : choupoSolve, runCase, runApplication, newCase,
                        listCases, cleanCase, Choupo, cdTut, cdWork
```

1.2 Build modes

```
make all                # default --- release, -O2
make MODE=debug all     # -O0 -g, parallel tree under build/<platform>Debug
make print              # show resolved build variables
make clean              # remove current mode's objects
make distclean          # remove the whole build/ tree (and the WASM build)
```

Both modes coexist; the `choupoSolve` symlink points to the *last* binary built.

Key idea. You never recompile to switch cases. Build once, source once, then all you do is edit text dictionaries and re-run.

Try it now. Run `runCase tutorials/steady/flash/flash01_benzene_toluene` and then `less` the `log.choupoSolve` it writes. Notice the per-iteration trace under "Rachford-Rice Newton": that is the simulator showing its work.

2 The shell helpers

> *What you'll learn.* A handful of short commands cover every interaction you'll have with the simulator from the shell.

After sourcing `etc/bashrc` you have:

Command	Purpose
`choupoSolve <case>`	Run a case directly; write everything to stdout.
`runCase [<case>]`	Run + redirect to `log.<binary>` in the case dir; dispatches by `controlDict.application`. Defaults to the current dir.
`runApplication <binary> <args>`	Run any binary, redirect to `log.<binary>` in the cwd.
`listCases`	Tabulate all tutorials with one-line descriptions.
`newCase <name> [note]`	Scaffold a fresh case in `$CHOUPO_WORKSPACE`.
`cleanCase [<case>]`	Remove `log.*` and generated outputs (never touches source).
`Choupo`	`cd` to the project root.
`cdTut`	`cd` to `tutorials/`.
`cdWork`	`cd` to your case workspace.

Two equivalent workflows:

```
# From anywhere (use $CHOUPO_HOME):
runCase $CHOUPO_HOME/tutorials/steady/flash/flash01_benzene_toluene

# From inside the case dir:
cd tutorials/steady/flash/flash01_benzene_toluene
runCase                                  # writes log.choupoSolve here
```

Three further tools appear later in this guide: `bin/choupo-lint` (validate a case without running it, §3.6), `bin/choupo-init0` (materialise the initial stream state, §3.5) and `bin/choupo-import` (seal a case's property data so it runs self-contained, §4.10). `bin/runTests` runs the full regression over every bundled tutorial.

3 Anatomy of a case directory

What you'll learn. Every case is a folder of plain-text files with a fixed division of labour: `system/` says how the problem is organised, `constant/` carries the models and data, and — in a steady case — the `0/` directory carries the complete initial state of every stream, one file per stream, OpenFOAM-style. The solver writes its answer to `converged/`. You will also meet the notation all of them are written in — the *dictionary* — and the reasoning behind it: why a process is *declared* rather than scripted, and why each fact gets exactly one home (§3.1).

The tree below is a *steady* (`choupoSolve`) case — the most common kind, and the one whose anatomy the rest of this section dissects. The other binaries reuse the same `system/` + `constant/` skeleton but not the stream-state directories: a batch or control case has no flowsheet streams to seed — its vessels carry `initial {}` charges and the run writes `trajectory.csv` (§9, §10) — and a props case replaces `flowsheetDict` with a `propsDict` altogether (§11).

```
caseName/
|-- caseName.cho             GUI marker (openable entity) --- empty, or
|                              GUI layout metadata; the solver never reads it
|-- system/
|   |-- controlDict          REQUIRED  application, verbosity, reports{}
|   |-- flowsheetDict        REQUIRED  TOPOLOGY ONLY: units + connections
|   |-- solverDict           optional  per-unit-op solver options
|   |-- outerDict            optional  sweep / designSpec / optimisation
|   `-- postDict             optional  sizing + costing chain
|-- constant/
|   |-- thermoPhysPropDict   REQUIRED  the thermophysical system
|   |-- reactions            optional  named-reaction library
|   |-- crystallisation      optional  crystallisation-kinetics library
|   |-- dryingKinetics       optional  drying-curve library
|   |-- components/          optional  case-local component overlays
|   `-- propertyManifest     optional  sealed-case seal: the dependency closure
|                                      + a sha256 per record (self-contained cases)
|-- 0/                       REQUIRED (steady only)  initial stream state,
|   |-- feed   liquid   vapor                        ONE FILE PER STREAM
`-- (written by the run)
    |-- converged/           the steady solution, same one-file-per-stream shape
    |-- reports/             CSV + .ods reports selected in controlDict
    `-- log.choupoSolve      written by runCase
```

Key idea. A standalone unit operation is just a `flowsheetDict` of length 1. There is no special "single-unit" mode — one consistent case format for everything, from one flash drum to a multi-sector plant.

3.1 The dictionary: one notation for the whole simulator

Every file named above — `controlDict`, `flowsheetDict`, `thermoPhysPropDict`, each stream's state file, the reaction library — is written in the same small notation, the *dictionary*. It is worth meeting the whole language at once, because there is very little of it: five constructs of grammar, and — further down — two keywords that mean more than they say.

```
// a line comment; /* a block comment */ also works

keyword       value;                  // (1) an entry: one keyword, one value
                                      //     value = word, number, or "string"
block                                 // (2) a sub-dictionary: a named group
{
    keyword   value;
    nested    { keyword value; }      //     nesting is unlimited
}
names         ( alpha  beta  gamma ); // (3) a list of words or numbers
things                                // (4) a list of sub-dictionaries
(
```

```
    { name a;  type flash; }          //     order is significant here
    { name b;  type mixer; }
);
```

That is the entire grammar: entries, sub-dictionaries, lists, lists-of-sub-dictionaries, comments. A number may carry its unit (`P 1 bar;`) — §3.7 covers that. There is no schema to register and no include mechanism; exactly two keywords do more than name a value, and you can read a case for a long time before meeting either:

```
variables { A  17.0 m2; }        // name a value once...
// ... then spend it wherever it belongs:
units ( { name effect1;  operation { area $A; } }
        { name effect2;  operation { area $A; } } );

inherits  "../../../constant";   // (in a thermoPhysPropDict) build on the
                                 // system declared at that folder
```

`variables` exists so that one physical quantity shared by several units — the heat-transfer area common to three evaporator effects, say — is written once and referenced, rather than copied three times and then edited twice. It is the same one-fact-one-home rule as the file layout, applied inside a file. `inherits` lets a sector's thermophysical system build on its plant's instead of repeating it (§4).

Why plain text, and why this notation. The format is deliberate, not incidental:

- **A process change is a reviewable diff.** Cases live in version control beside the code. "What changed between the run that worked and the one that didn't?" is answered by `git diff`, not by opening two projects side by side.
- **Comments are first-class.** In a teaching tool a case file is half specification and half explanation — the tutorials carry their reasoning in the dict itself, next to the number it justifies. A format without native comments (JSON) cannot do that; a general-purpose one (YAML, TOML) would force `P 1 bar;` into either a bare string or a nested object. The notation is domain-native: a chemical engineer's quantity, with its unit, on one line.
- **Nothing is hidden.** There is no binary project file, no cache that must agree with the text, no export step between what you edit and what the solver reads. The files in the folder are the complete input — which is the same glass-box promise the solver makes about its iterations.
- **It is writable by hand and by machine.** The tutorials were authored in a text editor; increasingly, cases are drafted with an assistant's help. Both produce the same artefact, and `bin/choupo-lint` (§3.6) checks it before you spend a solve.

A dictionary declares; it does not instruct. This is the deeper idea, and it is what makes the format a *representation* of a process rather than a script that builds one. A dict states what exists and what it is: *there is a flash drum called `flash01`, fed by `feed`, held at 370 K and 1 bar.* It never says what to do, in what order, or when. You will not find a "run" statement, a loop, or an `if` anywhere in a case — and when you want a hundred runs, you do not write a loop either: you declare a sweep (§7), which is one more dictionary.

The payoff is that a case has no single privileged reader. The same declaration is consumed by the solver, by the browser GUI that draws the flowsheet, by the report objects that write the balances, and by `choupo-lint` which reads it and refuses to run it. A script would have exactly one meaning — whatever happens when you execute it. A declaration has as many useful readings as you have tools.

One question, one file. The division of labour across `system/`, `constant/` and `0/` follows a single rule: each fact has exactly one home.

The question	Its one home
What is connected to what?	`system/flowsheetDict` — topology, no numbers
What are the stream numbers?	`0/<stream>` — one file per stream
What are the substances and models?	`constant/thermoPhysPropDict` (+ libraries)
How is this run organised?	`system/controlDict` (+ the optional dicts)

The rule is enforced, not merely recommended. If a flow rate could be written both in `flowsheetDict` and in `0/`, the two could disagree — and some code, somewhere, would have to pick a winner quietly. So a `streams {}` block inside any `flowsheetDict` is refused with an error that names the file and points at `0/`, and a graph with more

(or fewer) streams than state files is fatal *before* the first iteration. You are never one silent override away from simulating something other than what you wrote.

> **Key idea.** Because the state lives in files rather than in the topology, `converged/` comes out in exactly the same shape as `0/`. A solution can therefore seed the next run, a sector can be lifted out of a plant and run on its own, and every number the engine touched is a file you can open, diff and version — the same rule for the data as for the code.

Inheritance: a dict you do not write is inherited. A case that omits `controlDict`, `constant/thermoPhysPropDict` or a reaction library inherits it from the folder above (up to six levels). This is what makes the fractal layout work: a sector of a plant carries only what is genuinely its own — typically its `flowsheetDict` and, when its chemistry differs, its own thermophysical system — and still runs as a case in its own right, borrowing the rest from the plant around it. The one dict that is *never* inherited is `flowsheetDict`: it *is* the node, so a folder without one is not a case. Inheritance is announced in the log, never silent — each unit prints whether its thermodynamics were inherited or locally overridden.

3.2 `controlDict`

The essential keys:

```
application   choupoSolve;      // | choupoBatch | choupoCtrl | choupoProps
description   "Isothermal flash benzene/toluene at 370 K, 1 bar (Raoult)";
verbosity     3;                // 0 silent / 1 +warnings / 2 +summary
                                // 3 info (Newton iterations visible) / 4 debug
reports
{
    streamTable    { }          // see the Reports section
    massBalance    { }
    energyBalance  { }
}
```

For pedagogical use you want `verbosity 3` — that prints every Newton iteration, every Wegstein step, every K-value refresh. Set it lower only when an outer driver sweeps the simulator hundreds of times. Time-dependent cases (`choupoBatch` / `choupoCtrl`) add time controls — see §9.

3.3 `flowsheetDict` — topology only

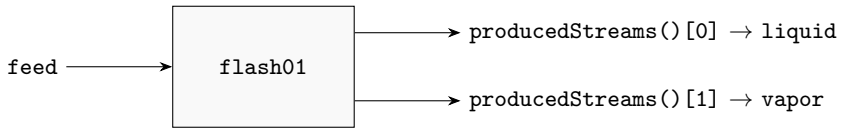
The `flowsheetDict` declares *what is connected to what* — units, their wiring, and (for recycles) the tear declaration. It carries **no stream numbers**: state lives in `0/` (§3.4). The first tutorial in full:

```
// stream state lives on disk in 0/ (per-stream files);
// flowsheetDict is TOPOLOGY only

units
(
    {
        name        flash01;
        type        isothermalFlash;
        in          feed;
        outputs     ( liquid  vapor );

        operation   { T 370.0 K;  P 1.0 bar; }
    }
);
```

Every unit entry follows the same slot order: `name` and `type` (required), an optional `model` (the sub-model selector, e.g. `simultaneous` on a column), then `in` (one inlet) or `inputs (...)` (several), `outputs (...)`, the `operation {...}` block of numeric knobs, and finally optional `reaction` / `crystallisation` / `dryingCurve` references and an optional per-unit `thermo {}` override (§4.9).

Identifying streams: inputs by name, outputs by *position*. This is the one rule that trips up every new author, so learn it once. A unit's `inputs` and its `outputs` are matched to physical streams by *different*

mechanisms. An **input** is wired *by name*: you write `in feed;` (or `inputs ( hot cold );`) and the unit reads each stream by the name you gave it (a `heatExchanger` names its tube-side inlet with `tubeStream`; a multi-feed column maps each named feed to a stage in `operation.feeds`). An **output**, by contrast, is matched *by position*. Every unit produces its streams in a *fixed internal order*, and your `outputs` list does nothing more than *rename that ordered list*: `outputs[k]` becomes the name of the k-th stream the unit emits.



`outputs ( liquid vapor )` renames by index — the ORDER matters

The engine does literally `s = produced[k]; s.name = outputs[k]`. So the order of your `outputs` list must follow the unit's emission order (`isothermalFlash`: liquid, vapour; `crystalliser`: crystals, mother liquor; `cyclone`: clean gas, captured solids; `distillationColumn`: distillate, bottoms, then side draws by ascending stage — §5.6.2). Getting the order wrong sticks the label on the *wrong* physical stream and the engine does *not* warn — the counts still match. The only guard is arity: declaring more names than the unit produces is a loud error. To learn a unit's order, read its `producedStreams()` in the source (glass-box) or its entry in the unit-operation catalogue (§5).

Recycles: tear streams. A recycle is an ordinary stream whose producer sits downstream of its consumer. The engine *detects* the cycle itself — what you declare is the *cut*: which stream the recycle iteration tears. Declare it (and, optionally, the recycle solver knobs) in `solverDict`:

```
tearStreams        ( recycle );
recycleSolver       Newton;      // Newton on the tears (default); or Wegstein
recycleTol          1e-5;
recycleMaxIter      120;
```

The tear's initial guess is its `0/` file, like any other stream (§3.4). `recycleSolver Newton` builds a finite-difference Jacobian over the tear variables (F, z_i, T) and converges quadratically; `Wegstein` keeps the classical fixed-point accelerator — run both on `tutorials/steady/flowsheets/process03_recycle` and compare the iteration counts.

The sequential-plan contract. The solver executes the units *in the order you declare them* — it never reorders. Before anything runs, the plan is validated: every material input must be a domain inlet, an output of an *earlier* unit, or a declared tear; and every declared tear must point backwards and close a real cycle. A cycle with no declared tear refuses, naming the cycle and the `tearStreams` line to paste; an out-of-order acyclic flowsheet refuses with a valid declaration order to paste; a tear that cuts nothing real (forward, off-cycle, misspelled) refuses too. When the plan is valid, the cut is announced in the log:

```
[plan] material recycle: tear 'recycle' cuts  mixer01 -> reactor ->
       separator -> split01 --recycle--> mixer01
```

Why so strict? Because every stream is pre-seeded from `0/`, a unit that reads a stream before its producer has run would silently consume the *initial guess* instead of the computed value — a wrong answer with no error. The validation makes that impossible. For the same honesty, a recycle that exhausts its iterations without converging now exits with status 1 and writes no `converged/` — the directory's name is a promise.

Sectors — the fractal flowsheet. A big plant is organised as *sectors* (composite sub-flowsheets), each a folder with its own `system/flowsheetDict` and, when it needs one, its own `constant/thermoPhysPropDict` (a sector may run a different thermodynamic world — e.g. an electrolyte model in a brine section and NRTL in a solvent-recovery section). The root dict then assembles members and named connections:

```
sectors ( BRINE  EXTRACTION  CARBONATION );    // UPPERCASE folders
// or:   units ( cryst1 cryst2 recovery );     // dignified unit folders

connections
{
    liRichBrine { from BRINE/liRichBrine;  to EXTRACTION/liRichBrine; }
    brineFeed   { to BRINE/brineFeed; }        // to-only   = plant inlet
    tails       { from EXTRACTION/raffinate; } // from-only = plant outlet
```

```
}
```

The connection *key* is the stream's one stable identity (and its `0/` filename); `from`/`to` are producer/consumer ports. A stream's *role* is inferred from the shape of its edge — a `to`-only edge is an inlet, `from`+`to` is internal, `from`-only is an outlet. There is no boundary mini-language to maintain. The whole tree is flattened into one solver problem (`plant.sector.unit` names); the hierarchy is for authoring and namespacing, never a solver-within-a-solver. See `tutorials/plant/` for worked multi-sector cases.

3.4 The `0/` directory — one file per stream

In a steady case a stream's state lives in its own file on disk — never inside `flowsheetDict`. The `0/` directory holds the **complete initial state**: for an *inlet* the file is your boundary specification; for an *internal* or *outlet* stream it is an initial estimate the solver overwrites. (This contract belongs to the steady domain — `choupoSolve`; the time-dependent and props binaries have no stream registry to seed.) The file for the flash tutorial's feed:

```
// 0/feed
componentMolarFlows
{
    benzene    40 kmol/h;
    toluene    60 kmol/h;
}

T    370 K;
P    1.0 bar;
```

Material — pick exactly one form. Four equivalent ways to state what flows; a second form in the same file is a fatal conflict:

```
componentMolarFlows { water 86 kmol/h;  NaCl 12 kmol/h; }            // (a)
componentMassFlows  { water 1549.3 kg/h;  NaCl 701.3 kg/h; }         // (b)
molarFlow 100 kmol/h;   moleFractions { water 0.86;  NaCl 0.14; }    // (c)
massFlow 2250.6 kg/h;   massFractions { water 0.6884; NaCl 0.3116; } // (d)
```

Forms (a)/(b) are the natural process-design description ("mix 40 kmol/h of A with 60 kmol/h of B"); (c)/(d) are the textbook total + composition. Mass converts through each component's MW. Fractions must close: if $|\sum_i x_i - 1| > 10^{-6}$ the reader stops with a fatal error naming the stream and the actual sum — there is no silent renormalisation of a state file.

Thermodynamic closure — T and P. Give `T` and `P`; the phase split is then a *result* (Duhem's theorem: a stream of known composition is fixed by exactly two state variables). Do not write a decorative `vaporFraction 0;` or `1;` — the engine recovers the phase itself. Two honest pins exist for the cases (T, P, z) cannot close:

- `vaporFraction 0.30;` — a genuine *two-phase split* on the saturation manifold, where T and P are not independent and the split is a real degree of freedom;
- `phase gas;` (or `phase liquid;`) — a *phase-intent* pin for a feed the cheap screen cannot classify (e.g. steam in a hot gas mixture: all vapour, but T is below water's T_c so "permanent gas" cannot be proved). Hot air or combustion gas above every component's T_c needs *nothing* — the engine's T_c screen recovers $vf = 1$ on its own.

Multiphase decomposition. A slurry or partly-solid stream adds a `phases {}` block that decomposes the overall material and must sum back to it exactly (validated on read); a particle-size distribution rides in a `psd {}` block. A utility stream may carry `category <word>;` so the utilities report can aggregate consumption by header (§6).

The completeness contract. N streams in the flowsheet graph $=$ N files in `0/`. A missing or an orphan file is **fatal** for `choupoSolve` — there is no partial state. The run announces the check:

```
[state] seeded 3 stream(s) from 0/ via the canonical manifest
[state] 0/ complete: 3 graph stream IDs == 3 state files
```

converged/ — the answer, in the same shape. After a converged solve the engine writes the full solution as converged/<stream> files — same grammar, one file per stream, now carrying the solved numbers (and the resolved phase). The log confirms: [state] wrote converged/. A drilled-in sector case is seeded from its parent's converged/ state, so a sub-case is reproducible from disk alone.

3.5 Materialising 0/: bin/choupo-init0

You author only the *domain inlets* (and a seed for each declared tear); the tool writes every remaining stream file by explicit propagation through the topology:

```
bin/choupo-init0 <caseDir>           # writes the missing 0/ files
bin/choupo-init0 <caseDir> --force   # regenerates internal/outlet estimates
```

The rules are strict and announced: an authored inlet file is never touched; an unseeded recycle is a hard error naming the file you must author (feed propagation is only honest on an acyclic graph); and an incomplete 0/ handed to choupoSolve stays fatal — this tool is the sanctioned way out. The generated files are estimates the solver overwrites; they exist so that the initial state is explicit, inspectable, and on disk — never an invisible in-memory guess.

Key idea. flowsheetDict = topology. 0/ = numbers. converged/ = the answer. Every stream's state is a file you can open, diff, and version-control — the same glass-box rule that governs the solver's iterations governs its data.

3.6 Validating without solving: bin/choupo-lint

The cheap step of the authoring loop — write, *lint*, fix, run. The tool takes the very same road the solver takes — it reads the dicts, assembles the thermophysical package (every component must resolve), flattens the topology (every connection must cable), enforces the 0/ completeness contract and resolves the tears — and then stops, just before anything would solve or be written to disk:

```
bin/choupo-lint <caseDir>     # read-only; nothing is solved, nothing written
```

On top of the load-path checks it reports what a run only discovers mid-execution: an unknown unit type (with the list of available types), a duplicate unit name, two units producing the same stream. Findings are collected and reported as *one* list — the tool describes the whole state of the case, not just the first defect it meets. Exit status: 0 the case will run as declared; 1 findings; 2 a load-phase refusal (whose own message names the rule and the fix).

It also prints the stream *roles* inferred from the topology — no producer → inlet; producer and consumer → internal; producer only → outlet; declared tears flagged — so you can confirm the graph says what you think it says before spending a solve. Roles are never declared in Choupo; the topology is the single source of truth, and the lint report is where you read it.

3.7 Units on every scalar

Choupo uses canonical SI internally: Pa, kmol/s, K, m, m^2, m^3, J, kg. You almost never want to type those raw, so any scalar in any dict may carry a unit suffix. The parser converts the value to SI on the fly and remembers its physical dimension — a typo in the unit is caught with a clear error before the solver runs.

Three accepted forms.

```
P    1     bar;                    // (a) named-unit shortcut
A_w  [-1  2  1  0  0]  1.5e-12;    // (b) bracket form:
                                   //     [M L T Theta N] exponents + SI value
R    0.5;                          // (c) raw SI (no declaration)
```

Form (a) is what you will type 95 % of the time. Form (b) is the escape hatch for quantities with no short named unit (membrane permeabilities, exotic transport coefficients). Form (c) says "trust me, the number is in canonical SI" — the parser cannot check anything, so a typo here passes silently.

Named units shipped by Choupo. The most useful aliases (the full registry of ∼90 units lives in src/core/Units.cpp):

Quantity	Canonical SI	Named-unit aliases
pressure	Pa	`Pa, kPa, MPa, bar, atm, psi, mmHg, torr`
molar flow	kmol/s	`kmol/s, kmol/h, mol/s, mol/h`
mass flow	kg/s	`kg/s, kg/h, g/s, g/h, t/h`
volumetric flow	m^3/s	`m3/s, m3/h, L/s, L/h, L/min`
temperature	K	`K, degC, C, degF, F`
length	m	`m, mm, cm, km, in, ft`
area / volume	m^2 / m^3	`m2, cm2, mm2;  m3, L, mL`
time	s	`s, min, h, day`
mass	kg	`kg, g, t, ton, tonne`
energy / power	J / W	`J, kJ, MJ;  W, kW, MW_power`
heat-transfer coeff.	$W/(m^2K)$	`W/m2/K, W/(m2.K), kW/(m2.K)`
velocity	m/s	`m/s, cm/s, mm/s`
membrane A_w	m/(s·Pa)	`m/s/Pa, m/s/bar`
viscosity	Pa·s	`Pa.s, cP`
concentration	$kmol/m^3$	`kmol/m3, mol/m3, mol/L, M`
molality	mol/kg	`mol/kg, mmol/kg`
density	kg/m^3	`kg/m3, g/cm3, g/mL, mg/L, ppm`
molar energy	J/mol	`J/mol, kJ/mol, J/kmol, kJ/kmol`
dimensionless	—	`-, %`

Note the deliberate quirks: power is `MW_power` (a bare `MW` would collide with the mega-prefix logic of `MPa`), and `degC`/`degF` are affine — use `K` for temperature *differences*.

What you see on a bad unit. Asking for a pressure but typing `P 1 kg;` produces:

```
ERROR: Dictionary 'feed' ():
  parameter 'P' expected dimensions [1 -1 -2 0 0] (Pa)
  but the dict declared [1 0 0 0 0] (kg).  Check the unit
  suffix on this entry.
```

The same check guards every parameter the unit operations read (`T`, `V_R`, `area`, `W_shaft`, `P_perm`, ...).

Output preferences. By default Choupo prints flows in kmol/h, pressures in bar, temperatures in K — the chem-eng convention. To override, drop a `units` block in `controlDict`; an optional `precision` block tunes decimal digits per quantity:

```
units      { pressure atm;  flow kmol/s;  temperature degC; }
precision  { pressure 3;  temperature 2;  flow 4;  composition 4; }
```

The internals stay in SI; only the printed output changes.

3.8 Optional dicts

- `solverDict` — solver defaults per unit-op type, overridable by each unit: e.g. `isothermalFlash { tolerance 1e-8; maxOuterIter 50; }`.
- `outerDict` — sweeps, design specs, optimisation (§7).
- `postDict` — sizing + costing chain (§8).
- `constant/reactions` — the named-reaction library referenced by reactors: `reaction esterification;` or `reactions ( r1 r2 );` (§5.3).

4 The thermodynamic foundation: `constant/thermoPhysPropDict`

> *What you'll learn.* How a case declares its components and selects the models that turn (T, P, z) into K-values, enthalpies, and transport properties — and how the run log announces every choice it makes.

Every case carries a `constant/thermoPhysPropDict` declaring its whole thermophysical system: the components, the equilibrium *formulation* (which K-value structure the VLE runs on), and the model in each phase slot. The minimum viable file:

```
recordType    thermophysicalPropertySystem;
schemaVersion 2;

components       ( benzene  toluene );

equilibrium
{
    formulation gammaPhi;

    liquid
    {
        activityModel ideal;        // gamma_i = 1  (Raoult)
        standardState pureLiquid;
    }

    vapour
    {
        fugacityModel idealGas;     // phi_i = 1
    }
}
```

Everything is declared — the formulation, the standard state, the vapour route — even when the choice is the obvious one; a shape the builder does not recognise gets a named refusal, never a guess.

Components are resolved *by name*, with an explicit evidence order: **case-local overlay > standard > local**. There are exactly two data homes, and the distinction is deliberate:

- `data/standards/` — *curated and public.* The committee-managed reference set (194 species); shipped in the repository, every value carrying primary provenance.
- `data/local/` — *yours and private.* A *gitignored* working tier for data you import, license or estimate. The repository ships it **empty** (a `README` and a `.gitkeep`); the engine reads it when present. A value resolved from it is announced `[local] UNVERIFIED` and carries its source, licence, import digest and method — it is never presented as equivalent to a standard.

Data is thus either curated-and-public or yours-and-private: there is no public "proposed" middle tier (retired 2026-07). A case-local file in `<case>/constant/components/` overlays either home and always wins (§15).

You populate `data/local/` yourself — for example with the reproducible ChemSep 8.3 importer (`bin/curate/chemsep_to_choupo.py`), which writes there. This is why the public repository redistributes *no* third-party property values: for coverage beyond the curated standards it ships Choupo's own open group-contribution estimates under `data/groupEstimative/` (clearly flagged as estimates), and you import any measured or third-party data into your private `data/local/`. Imported literature constants and predictive closures stay distinguishable in the provenance — an imported boiling point is not silently upgraded because an Ambrose–Walton closure makes the component flashable, and missing heat capacities or formation data stay missing (the gap report says what the model still needs).

4.1 Activity models (liquid γ)

model	Notes
ideal	$\gamma_i = 1$ — the Raoult limit. Hydrocarbons, dilute systems.
NRTL	Renon–Prausnitz. Handles LLE. Needs binary pairs.
Wilson	Cheaper than NRTL but structurally *cannot* represent a liquid split.
UNIQUAC	Abrams–Prausnitz; the model many published kinetic studies regressed against.
UNIFAC	*Predictive* — γ from molecular groups, no fitted binaries. The group decomposition lives in each component's .dat.

NRTL/Wilson/UNIQUAC pairs are supplied inline or resolved by name from the pair catalogues. Resolution is **inline/case-local > standard > local > ideal default** (the same two-tier model as components: the public curated data/standards/parameters/<model>/ bank, then your private data/local/ bank, which the repository ships empty). A pair absent from every tier is the announced ideal default ($\tau = 0$), never a silent one:

```
// inside equilibrium.liquid:
activityModel
{
    model NRTL;
    binaryParameters                        // the dict OWNS the numbers,
    {                                       //   or points at a catalogue
        ethanol-water                       //   file via 'source "...";'
        {
            i ethanol;  j water;
            a_ij -0.8009;  b_ij  246.18;   // tau_ij = a_ij + b_ij/T
            a_ji  3.4578;  b_ji -586.0809;
            alpha 0.30;
        }
    }
}
```

A local pair never shadows a standard pair. Staged ChemSep pairs live in the private data/local/parameters/ tier (not redistributed); they have passed identity, unit-conversion and provenance audits, but have not all been independently validated against experimental data; the log therefore marks their use [local] UNVERIFIED. A pair with no parameters anywhere does not abort the run — it defaults to ideal ($\tau = 0$) so you can build the foundation pair-by-pair — but it is never silent: each ideal-defaulted pair is announced with a [thermo] log line, and the GUI colours the pair matrix by where each pair resolved.

4.2 Equation of state (vapour φ — and, for a cubic, the liquid too)

model	Notes
idealGas	$Z = 1$, $\varphi_i = 1$. Adequate at low pressure.
SRK	Soave–Redlich–Kwong cubic; van der Waals one-fluid mixing; optional binaryInteractions ({ i CO2; j CH4; kij 0.0973; }).
PR	Peng–Robinson cubic; better near-critical behaviour (CO_2).

The cubics read each component's (T_c, P_c, ω) from the catalogue and deliver Z, residual H/S, and *both* roots — so a cubic can also serve as the *liquid* model in a φ–φ package (below).

4.3 Per-component correlations

Vapour pressure defaults to Antoine ($\log_{10} P^{\text{sat}} = A - B/(T + C)$), stored per component; a corresponding-states alternative (vaporPressure { model AmbroseWalton; }) covers estimated components. Ideal-gas and liquid heat capacities are polynomials in T; species used at flame temperatures carry two-range NASA-7 forms (model NASA7). Each component's standardThermochemistry { dHf_298; s_298; phase; } block pins the formation datum — the phase keyword (gas / liquid / solid) declares on which rung of the enthalpy ladder the value is tabulated, and every energy balance that crosses a reactor requires it (§5.3).

4.4 Multiple liquid phases (LLE / VLLE)

For liquid–liquid work, select the gammaGamma formulation and declare the liquid phases explicitly:

```
components ( water  nButanol );
```

```
equilibrium
{
    formulation gammaGamma;         // 2+ liquid phases, one gamma surface each

    liquidPhases
    (
        { name aqueous;  activityModel { model NRTL; binaryParameters {...} } }
        { name organic;  activityModel { model NRTL; binaryParameters {...} } }
    );

    vapour { fugacityModel idealGas; }
}
```

The flash then searches for the liquid split by direct Gibbs-energy minimisation (§5.2) — a single-liquid `gammaPhi` system builds only one liquid and will not split.

4.5 Transport properties

Units that need viscosity, conductivity, diffusivity, or surface tension (pipes, dryers, tray hydraulics, exchanger rating) read them from a `transport` block of selectable sub-models:

```
transport
{
    vapour
    {
        viscosity           { model Chung; }      // gas mu, from Tc/Pc/omega/MW
        thermalConductivity { model Eucken; }
        diffusivity         { model Fuller; }     // needs diffusionVolume
    }
    liquid
    {
        viscosity           { model Vogel; }      // or Andrade
        thermalConductivity { model SatoRiedel; } // or Latini
        diffusivity         { model WilkeChang; } // or Scheibel
    }
    interface
    {
        surfaceTension { model BrockBird; }
    }
}
```

When a unit needs a property the package does not supply, the solver stops with a clear error naming the missing model — it never invents a value.

4.6 Dissolved gases: Henry's law

A gas solute (CO_2, NH_3, O_2, H_2S, SO_2, … in water) takes the infinite-dilution Henry reference. WHO dissolves in WHAT is declared structurally, by the `diluteSolution` formulation's `solvent {}` and `solutes {}` groups (next subsection); the solutes' `binaryParameters` declare the Henry-pair files (`data/standards/parameters/Henry/<solute>-<solvent>.dat`, van 't Hoff temperature dependence). The absorber/stripper and the dilute-solution flash world run on this. Each engaged pair is announced with its constants and source (`[henry]` log lines).

4.7 The `formulation` keyword and the four VLE worlds

The `equilibrium.formulation` keyword selects the whole K-value structure the VLE runs on:

`formulation`	World	K-value
`gammaPhi`	γ–φ	$K_i = \gamma_i P_i^{\text{sat}}/(\varphi_i P)$
`diluteSolution`	dilute solution	solvent on Raoult; each solute on the Henry form $y\varphi P = x\gamma^* H(T)\, e^{v_\infty (P-P_s)/RT}$
`phiPhi`	φ–φ	$K_i = \varphi_i^L/\varphi_i^V$ — the same cubic's two roots
`electrolyteGammaPhi`	aqueous electrolyte	ionic activity + osmotic coefficients on the aqueous-ion reference

The dilute-solution world, from `flash08_co2_water_package`:

```
equilibrium
{
    formulation diluteSolution;

    liquid
    {
        solvent { component water;    standardState pureLiquid; }
        solutes
        {
            components    ( CO2 );
            standardState infiniteDilution;
            solutionModel henryDilute;
            binaryParameters
            {
                CO2-water { source "data/standards/parameters/Henry/CO2-water.dat"; }
            }
        }
    }

    vapour { fugacityModel idealGas; }
}
```

Declared parameter files are *verified at assembly* — a declared-but-missing pair refuses loudly, naming the entry to add — and the run announces the assembled world in its `[builder]` lines. In `phiPhi` both phase slots route through the ONE declared `equationOfState` (`fugacityRoute equationOfState; root liquid|vapour;`); two different cubics across the phases are refused: one Gibbs surface per phase, never two. Reference tutorials: `flash08_co2_water_package` (Henry) and `flash09_n2ch4_stryjek` (φ–φ with a declared k_{ij}).

4.8 Pure fluids: IAPWS-IF97 steam

For water-dominated power cycles the package can route a pure component through the hand-rolled IAPWS-IF97 steam kernel:

```
pureFluids { water { method IF97; } }
```

The Rankine tutorial `tutorials/steady/power/rankine02_water` closes its first law to 10^{-12} on IF97 — and its ideal-gas twin `rankine01_water` is kept on purpose, to show what an ideal-gas water *cannot* do (there is no two-phase dome to cross).

4.9 Per-unit `thermo {}` override and model boundaries

A unit may carry a `thermo {...}` block that *replaces* the global models for that unit only (components stay global):

```
{ name turbine;  type turbine;  in feed;  outputs ( expanded );
  operation { W_shaft -3 kW;  eta 0.8; }
  thermo    { equationOfState { model SRK; } } }    // real gas here only
```

A stream crossing from model X into model Y is re-interpreted at the boundary: (T, P, z) are held and each unit recomputes its own enthalpy, so H *steps* at the boundary — visibly, in the printed enthalpy. **H is the conserved truth; T is the model-dependent readout.** The solver never silently nudges T to paper over the step; an opt-in model-boundary audit prints $\Delta H = H_{\text{down}} - H_{\text{up}}$ into the first-law ledger. Default to one consistent global model; reach for the override only when the physics demands it. The run log is loud either way — every unit prints `thermo: inherited (global package)` or `LOCAL override -- ...`.

4.10 Self-contained cases

A case can be sealed so it runs with *no* installation catalogue:

```
bin/choupo-import <case>     # materialise the dependency closure into
                             # constant/ (components/, parameters/, ...)
                             # + the sha256 propertyManifest
```

The importer copies every record the case needs into the case's own `constant/` tree and writes `constant/propertyManifest` — the record-ownership registry (one sha256 per imported file; `sealed true;` forbids the runtime every catalogue fallback). A file the manifest does not claim is yours and untouchable. The definitive test is moving the case out of the repo and watching it still run. Every shipped tutorial is sealed this way; research cases and anything you archive with a thesis should be too.

5 Built-in unit operations

What you'll learn. The complete catalogue of steady-state unit operations — what each `type` does, which `model` sub-selectors exist, and what the `operation` block must contain. Aliases (`flash`, `column`, `FUG`, `REquil`, `boiler`, `condenser`, `MHeatX`, `membraneSW`, `extract`) name the same classes. `UnitOperation::availableTypes()` prints the live registry.

5.1 Quick reference

`type`	**What it does**	**Key spec**
Flash and saturation		
`isothermalFlash` *(alias `flash`)*	VL / LL / VLLE flash at fixed (T, P); duty Q is a result	`T`, `P` optional (inherit feed); `phaseSet VL|LL|VLLE`
`adiabaticFlash`	$H_{\text{out}} = H_{\text{in}}$; solves T	`P`
`bubbleT` / `dewT`	Saturation temperature at fixed P	`P`
Reactors (§5.3)		
`conversionReactor`	Specified per-pass conversion; stoichiometric	`conversion` (or `conversions (...)`)
`equilibriumReactor` *(alias `REquil`)*	Named reactions driven to simultaneous $K_p(T)$ equilibrium	`reactions (...)`; `T`
`gibbsReactor`	Equilibrium by Gibbs minimisation over species	`elements`, `species`; models `elementPotential` / `reactiveFlash` / `directMin`
`cstr`	Stirred tank; kinetics; 1 or N reactions	`V_R`; `thermalMode`
`pfr`	Plug flow, RK4 along volume; 1 or N reactions	`V_R`, `nSteps`; `thermalMode`
Heat transfer (§5.4)		
`heater`	1-stream duty device; T_{out} is a result	`Q`
`phaseChanger` *(aliases `boiler`, `condenser`)*	Crosses the saturation dome; Q a result	exactly one of `Q`, `outletT`, `outletQuality`, `outletState`, `superheat`, `subcool`; or `model geometry`
`heatExchanger`	2-stream ε-NTU rating	`area`, `U`; or `model geometry` / `model design`
`multiStreamHX` *(alias `MHeatX`)*	N hot + M cold in one shell; balance + internal pinch	per-stream `outlet {...T...}` targets
`evaporator`	Single-effect; steam chest is a real stream	`area`, `U`
Rotating equipment (§5.5)		
`compressor` / `turbine`	$W_{\text{shaft}} + \eta$; P_{out}, T_{out} results	`W_shaft`, `eta`
`pump`	Incompressible; closed form	`eta` + one of `P_out` / `dP` / `W_shaft`
`electricLoad`	Sink for shaft work via an energy wire	`energyInputs`, `eta`

type	What it does	Key spec
Columns and staged separations (§5.6)		
`distillationColumn` *(alias `column`)*	Rigorous MESH; models `WangHenke` / `simultaneous`; multi-feed, side draws, Murphree, hydraulics, reactive	`nStages`, `feedStage`, `refluxRatio` or `distillateRate`, `P`
`shortcutColumn` *(alias `FUG`)*	Fenske–Underwood–Gilliland design	keys, recoveries, `refluxFactor`, `P`
`absorber` / `stripper`	Counter-current staged gas–liquid contact (Henry)	`stages`
`extractor` *(alias `extract`)*	Counter-current staged LL extraction (Gibbs-split stages)	`stages`
Membranes and sorption separations (§5.7)		
`spiralWoundModule` *(alias `membraneSW`)*	NF/RO element or train (solution-diffusion + film model)	`membrane`, `area`, `P_perm`
`ionExchanger`	Cation-exchange softener (Gaines–Thomas)	`resin`, `CEC`
`electrodialysisStack`	N cell pairs; Faraday transfer, Nernst + ohmic voltage	`N_cellpairs`, `current`, geometry
`psa`	Pressure-swing adsorption (equilibrium-theory shortcut)	adsorbent isotherms, `bedCapacity`, `eta`
Solids (§5.8)		
`cyclone`	PSD-aware gas–solid classifier; 5 models (`Lapple`, …, `Muschelknautz`)	`bodyDiameter`, …
`bagFilter`	Cake filtration; ΔP headline	`filterArea`, …
`gasSolidSplitter`	Ideal spec'd split (PSD unchanged)	`solidsRecovery`
`pneumaticConveyor`	Dilute-phase conveying line; ΔP + saltation check	`geometry { D L dz }`
`crystalliser`	Models `equilibrium` / `MSMPR` / `FVM`; magma or cake+liquor outputs	`operatingTemperature`; kinetics for MSMPR
`sprayDryer` / `solidDryer` / `evaporativeDryer`	Gas–solid drying (§5.9.1)	atomiser, chamber; sorption isotherm
Flow primitives (§5.10)		
`mixer`	$N \to 1$ adiabatic merge (or declared-T)	—
`splitter`	$1 \to N$ tee at spec'd fractions	`fractions (...)`
`valve`	Isenthalpic throttle	`P`
`pipe`	Darcy–Weisbach line; ΔP a result; two-phase aware	`geometry { D L roughness dz }`

Each unit also accepts top-level solver-tolerance overrides (same keys as `solverDict`) and the per-unit `thermo {}` override (§4.9).

5.2 Flash and saturation

The isothermal flash is the pedagogical core: Newton on the Rachford-Rice equation in V/F, wrapped in a Wegstein outer loop that refreshes $\gamma_i(x)$. A flash is a *knob-less separator* — with an empty `operation {}` it inherits T and P from the feed and just separates; pin either or both to hold the drum state:

```
operation { T 370 K;  P 1 bar;  phaseSet VL; }   // VL (default), LL, or VLLE
```

Degrees of freedom — Q is a result. A flash is fixed by exactly two numbers (Duhem's theorem), here T and P. The duty is therefore a *result, never a spec*: a flash *gives* a Q (read the `Q_kW` KPI), it never *takes* one. To impose a known heat and then separate, chain `heater(Q)` → `flash`; for a target outlet temperature, wrap the heater in a DesignSpec on `$Q` (§7.2). The `adiabaticFlash` is the one duty that needs no external device: $Q = 0$, spec `P`, and the outer Newton finds T.

LL and VLLE. For symmetric γ-models a liquid–liquid flash solved on the K-value fixed point collapses onto the trivial $x_\alpha = x_\beta$ saddle — so Choupo's LL and VLLE paths minimise the Gibbs energy of mixing *directly* (multi-start Nelder–Mead on the phase-split simplex). Run `tutorials/steady/absorption/extraction01_waterButanol` (LL) and `tutorials/steady/thermoTest/vlle03_audit_artificial` (a constructed three-phase global minimum) to watch it work. `bubbleT` and `dewT` are the 1-D saturation solves ($\sum K_i x_i = 1$ and $\sum y_i / K_i = 1$).

5.3 The reactor family

> *What you'll learn.* Six reactors, one ladder of knowledge: specify the outcome (`conversionReactor`), let thermodynamics decide (`equilibriumReactor`, `gibbsReactor`), or let kinetics decide (`cstr`, `pfr` — and, in time, `batchReactor`, §9). Pick the rung that matches what you actually know.

All reactors share two contracts. First, **one reaction grammar**: stoichiometry and kinetics live once, in `constant/reactions`, and a unit references them — `reaction <name>;` for one, `reactions ( r1 r2 );` for a network — never repeated inside the unit. Second, **one enthalpy datum**: the heat of reaction is computed from each species' `standardThermochemistry` block (the elements datum), $\Delta H_{\mathrm{rxn}}(T) = \sum_i \nu_i h_i(T)$, in every reactor, steady or dynamic. There is no `dH_rxn` input to a steady reactor — a reactions-dict `dH_rxn` key is an announced override honoured only for toy components that deliberately lack formation data.

5.3.1 `conversionReactor` — the outcome is a spec

You state the per-pass conversion of the limiting reactant; the outlet follows stoichiometry. No kinetics, no residence time — the honest choice for a gas-phase reactor whose conversion is set by the catalyst, and the right reactor when a *recycle* is the point of the flowsheet (it keeps a finite per-pass conversion).

```
operation { conversion 0.75;  T 895 K; }     // single reaction
reaction  toluene_hda;                        // stoichiometry + limitingReactant
```

With several reactions, each extent is a spec — selectivity becomes a declared property of the catalyst:

```
operation
{
    T 650 K;
    conversions
    (
        { reaction main;  conversion 0.60; }    // fraction of the FEED limiting reactant
        { reaction side;  extent 0.04; }        // or a direct extent
    );
}
reactions ( main side );
```

Tutorials: `conversion01_hda`, `conversion02_parallel_selectivity`.

5.3.2 `equilibriumReactor` (alias `REquil`) — named reactions to equilibrium

You give a *set of reactions*; each is driven to simultaneous chemical equilibrium via its $K_p(T)$, computed from the species' formation data ($K = e^{-\Delta G^\circ/RT}$) — no kinetics, no residence time. The coupled system $K_{p,j} = \prod_i (p_i/P^\circ)^{\nu_{ij}}$ over all extents is solved in log form by the multivariate Newton. This is the reforming / shift / synthesis workhorse — and the tool to use when you want to restrict *which* reactions equilibrate (something the Gibbs reactor, which sees only species, cannot do).

```
{ name reformer;  type equilibriumReactor;
  in feed;  outputs ( effluent );
  operation { T 1000 K; }
  reactions ( steamReforming  waterGasShift ); }   // stoichiometry only; no kinetics
```

KPIs: `Kp_<name>`, `extent_<name>_kmol_h`, `conversion_<name>`, `Q_kW`. Tutorial: `tutorials/steady/reactors/equil01_reforming` (steam-methane reforming and shift together).

5.3.3 `gibbsReactor` — equilibrium without naming reactions

Direct Gibbs-energy minimisation over a declared species pool with atom-balance constraints — no reactions at all. Three selectable models: `elementPotential` (default; Lagrange multipliers), `reactiveFlash` (multi-condensable + NRTL), and `directMin` (multi-start Nelder–Mead on the extent null space).

```
operation
{
```

```
    T 800 K;   P 1 bar;
    elements ( C H O );
    species
    (
        { name CH4; atoms ( 1 4 0 ); }
        { name H2O; atoms ( 0 2 1 ); }
        { name CO;  atoms ( 1 0 1 ); }
        { name CO2; atoms ( 1 0 2 ); }
        { name H2;  atoms ( 0 2 0 ); }
    );
}
```

An optional `approachTemperature <dT>;` evaluates the equilibrium at $T + \Delta T$ — the classical way to model an incomplete approach for reversible systems without kinetics. For the adiabatic flame problem, wrap the reactor in a DesignSpec that varies `$T` until $H_{\text{out}} = H_{\text{in}}$ (`gibbs05_adiabatic_flame`: lean CH_4/air → 1492 K).

5.3.4 `cstr` and `pfr` — kinetics

The kinetic pair. The CSTR solves the design equation(s) $\xi_j = r_j(\boldsymbol{\xi})\, V_R$ — Newton-1D for one reaction, the multivariate Newton for a network; the PFR marches $\mathrm{d}F_i/\mathrm{d}V = \sum_j \nu_{ij} r_j$ with a manual RK4. Both compute concentrations on the liquid molar volume — for a gas-phase reactor prefer `conversionReactor` or `equilibriumReactor`.

```
{ name reactor;  type cstr;   in feed;  outputs ( out );
  operation { V_R 0.004 m3; }
  reaction  esterification; }              // or:  reactions ( r1 r2 );
```

A reaction entry in `constant/reactions` carries stoichiometry (with per-species orders) and an Arrhenius `kinetics` block; `reversible true;` adds the reverse leg by detailed balance ($k_r = k_f / K_{eq}(T)$). Multi-reaction networks are what make *selectivity* modellable — and the CSTR/PFR contrast visible: for the series network A→B→C at equal τ, `pfr03_series_selectivity` keeps 79 % of the intermediate where `cstr03_series_selectivity` keeps 53 % — back-mixing eats the intermediate, a result you *see*, not memorise.

Non-isothermal operation and multiplicity. Both take `thermalMode isothermal` (default) · `adiabatic` · `heatExchange { UA; T_coolant; }` (CSTR) / `heatExchange { U; areaPerVolume; T_coolant; }` (PFR):

```
operation { V_R 0.004 m3;  thermalMode adiabatic;  T_guess 350; }
```

Because the reaction heat already lives inside H on the elements datum, an adiabatic reactor is simply $H_{\text{out}} = H_{\text{in}}$ — no source term. The CSTR eliminates the extents first (at fixed T the design equations are monotone), leaving one equation $\varphi(T) = H_{\text{out}}(T) - H_{\text{in}} - Q_{\text{ext}}(T)$ — the textbook heat-generated vs heat-removed balance, which can have **three roots**. Choupo does not hide that: it *scans* the bracket, prints every steady state it finds, and reports the one nearest `T_guess`. The PFR marches total enthalpy beside the species and recovers $T(V)$ by inversion — a hot spot is something you see in the axial profile.

> **Try it now.** Run `tutorials/steady/reactors/cstr05_multiplicity`: three steady states at 300.4 / 348.4 / 391.3 K, all printed, the `steadyStates` KPI counting them. Change `T_guess` and the same reactor answers differently — that is the physics of ignition and extinction, not a numerical defect. Then run `cstr04_adiabatic` and `pfr04_adiabatic` on the same esterification: +82 K vs +83 K — the difference is the back-mixing the CSTR pays for.

5.3.5 Catalytic kinetics: writing an LHHW rate law

> *What you'll learn.* How to write a Langmuir–Hinshelwood–Hougen–Watson rate law in `constant/reactions`, what `basis activity` obliges you to, and the one experiment that actually demonstrates product inhibition (it is not the one you would try first).

The same block is read by `cstr`, `pfr`, `batchReactor` and `dynamicCSTR`: one `RateLaw`, four reactors, no possibility of them disagreeing. The Theory Guide § "The rate law" derives where the denominator comes from; here is how to write it.

```
esterification_meoac
{
    limitingReactant   methanol;

    stoichiometry
    (
        { component aceticAcid;     nu -1;  order 1; }   // forward orders
        { component methanol;       nu -1;  order 1; }
        { component methylAcetate;  nu  1; }             // orderRev defaults to nu
        { component water;          nu  1; }
    );

    kinetics
    {
        type       LHHW;
        basis      activity;        // a_i = gamma_i x_i    (else ‘concentration‘)

        A          8.497e6;         // k_f0   [mol/(g_cat.s)]
        Ea         6.047e4;         // J/mol
        A_rev      6.127e5;         // the REGRESSED reverse pair ...
        Ea_rev     6.373e4;         // ... or ‘reversible true;‘ for detailed balance

        adsorption
        {
            exponent    2;          // sites the rate-determining step occupies
            vacantSite  0;          // 1 (default) gives the classical ‘1 + sum K c‘
            species
            (
                { component aceticAcid;     K0 0.05245454; }   // K_i = K0 exp(B/T)
                { component methanol;       K0 0.17601898; }   // B defaults to 0
                { component methylAcetate;  K0 0.05602127; }
                { component water;          K0 0.29086872; }
            );
        }
    }
}
```

Read it against the rate-law equation in the Theory Guide: `exponent` is p, `vacantSite` is v, `K0`/`B` are $K_i(T)$, and the `order` / `orderRev` entries are α_i / β_i. Omit the `adsorption` block entirely and the denominator collapses to 1: you are back to `type Arrhenius`, the plain power law, byte for byte.

Two things the engine refuses. You may not ask for `reversible true` *and* an `A_rev`/`Ea_rev` pair — pick the one you can defend. And you may not ask for `reversible true` on `basis activity`: detailed balance runs through K_c, a concentration-basis equilibrium constant, and pretending otherwise would put the equilibrium in the wrong place. Give the regressed reverse pair instead.

`basis activity` obliges you to an activity model. The rate constants of a published LHHW law were regressed *with* a particular γ-model. Pöpken's were regressed on UNIQUAC with the binary parameters of their own Table 3 — which are in the standard catalogue, cited — so `cstr07_lhhw_methylAcetate` declares `activityModel { model UNIQUAC; }` and nothing else will do. Swap in UNIFAC and k_1 stops being their k_1.

Rate constants per gram of catalyst. Add `catalystLoading <kg/m^3>;` to `operation`. The reactor prints it back and converts; with concentrations in kmol/m^3 the loading *is* the factor. Omit it and the constants are taken as already volumetric.

> **Common pitfall.** The obvious way to see product inhibition — delete the `adsorption` block, keep k_f, compare — **shows nothing**. The reaction gets *faster*, because the adsorbed activities are small and dividing by their square is a multiplication. The constants were fitted for that denominator.
> The honest experiment: hold every reactant fraction fixed and *charge the product.* In `batch07_lhhw_methylAcetate`, starting from `methanol 0.4; aceticAcid 0.4; water 0.2;` instead of ...

`water 0.0;` halves the initial rate, 58.3 → 29.0 mmol/(L s). The water is not a reactant. It is sitting on the sites.

Try it now. Run `cstr07_lhhw_methylAcetate`: $X = 49\,\%$, kinetically limited. Raise `V_R` to 1 m^3 and it climbs to 76 % and stops. Then run `batch07_lhhw_methylAcetate`, which integrates the *same* law in time: it reaches 76.6 %. Nothing in either dict asserts that number. It is where the rate law's numerator vanishes — the activity equilibrium $K_a = 25.50$ that Pöpken's independent Eq. 17 recovers from the same eight constants. Two reactors agreeing is how you check you typed the parameters right.

5.4 Heat transfer

`heater` — the unit that *takes* a Q. Single-stream, sensible-only; `operation { Q 100 kW; }` (negative cools) and T_{out} is the result. "I need $T_{out} = 360$ K" is a DesignSpec on `$Q` (§7.2) — the outer loop is visible, the duty is never back-computed in secret.

`phaseChanger` (aliases `boiler`, `condenser`) — crossing the dome. Where a heater must stay off the saturation line, the phase changer lands its outlet *anywhere* on the (T, vf) surface — subcooled, inside the dome, or superheated — wrapping the flash kernel and the dome-aware enthalpy $h = (1 - vf)h_f + vf\,h_g$. P is held from the inlet (or set); then pick *exactly one* outlet coordinate (a degrees-of-freedom guard rejects zero or two):

```
operation
{
    // pick exactly ONE:
    Q             -630 kW;            // duty given -> T, vf results
    outletT        368 K;             // -> Q result
    outletQuality  0.4;               // partial condenser -> Q result
    outletState    saturatedVapour;   // or saturatedLiquid
    superheat      25 K;              // T = Tsat(P) + dT
    subcool        10 K;              // T = Tsat(P) - dT
}
```

The labels *boiler* / *partial condenser* / … are emergent from the converged vf and the sign of Q — not a model switch. With `model geometry;` the duty *emerges* from real heat-transfer physics instead: a `coolant {}` block selects Nusselt (1916) laminar-film condensation on a tube bundle; a `boiling {}` + `heatingMedium {}` pair selects Rohsenow nucleate pool boiling with the Zuber critical-heat-flux ceiling — a design above CHF is hard-refused (burnout), and the Rohsenow surface constant `Csf` has *no default* and demands a `citation` (a ±100 % number must state its provenance). Tutorials: `phasechange01_partial_condenser`, `condenser01_film_nusselt`, `reboiler_water_copper`, `rankine02_water`.

`heatExchanger` — rating, geometry, design. The two-stream exchanger has three faces of one model:

- **default (`model epsNTU`)** — give `area` and `U`; duty and outlet temperatures are results (ε-NTU, `flow counter|co`).
- `model geometry` — give the tube-bundle geometry (`tubeID/OD/Length`, `nTubes`, `passes`, `shellID`, `baffleSpacing`, `tubePitch`, `wallK` or `wallMaterial`) and the per-side correlations (`Gnielinski` tube-side, `Kern` shell-side by default) *compute* U and the area — the glass-box answer to "where does U come from?". Both-side pressure drops are reported too.
- `model design` — give the duty target (an outlet temperature) and the tube choices; the engine *sizes* the bundle (bisects the tube count so the delivered ε-NTU duty meets the target) and then rates its own design.

The two-part tutorial pair `hxWorkflow1_design_from_duty` → `hxWorkflow2_rate_designed` is the standard industrial sequence: converge the balance with a duty target, design the exchanger, then rate the designed geometry (and ask the off-design question the design step cannot answer).

`multiStreamHX` (alias `MHeatX`). N hot + M cold streams in one adiabatic shell. Every inlet carries its own outlet-T target; the unit verifies the energy balance (a non-zero imbalance is *reported*, never absorbed into an invented outlet) and builds the hot/cold composite curves to report the internal pinch ΔT_{min} — a temperature cross is a hard warning. Tutorial: `mheatx01_two_hot_one_cold`.

`evaporator`. Single-effect with a *real* steam chest: `inputs ( feed steam )`, hardware `area` + `U`; the boiling temperature, vapour split, and operating pressure are all results. Multi-effect trains are just chained evaporators — see `evaporator02_triple_effect_sugar` and the DesignSpec that sizes them (§7.2).

5.5 Rotating equipment and electric loads

The credo: rotating hardware is specified by *shaft power and efficiency*, and the pressure it achieves is a result:

```
operation { W_shaft  10 kW;   eta 0.75; }     // compressor: W > 0
operation { W_shaft  -3 kW;   eta 0.80; }     // turbine:    W < 0
```

The compressor/turbine run an outer Newton on P_{out} matching the isentropic enthalpy change scaled by η. The `pump` is incompressible and closed-form, so it accepts `eta` plus exactly *one* of `P_out` / `dP` / `W_shaft` — specifying the header it feeds is a local one-shot inversion, not a DesignSpec. A let-down is a `valve`, not a pump.

Energy wires. Shaft work travels on typed scalar wires declared on the *consumer*:

```
{ name T1;  type turbine;  in steam;  outputs ( exhaust );
  operation { W_shaft -1500 kW;  eta 0.85; }
  energyOutputs ( { name shaft;  kind work;  expression "-W_shaft";  unit W; } ) }

{ name G1;  type electricLoad;
  operation { eta 0.97; }
  energyInputs ( { from T1.shaft;  kind work;  target W_shaft; } ) }
```

Several inputs to the same target *sum* — the gas-turbine shaft-split, where one generator nets the turbine's gross work against the compressor's load (`tutorials/steady/power/combined02_brayton_rankine_shaft`). Typed ports are checked: a `work` wire cannot feed a `heat` input. Heat duties follow a different rule — see §5.11.

5.6 Distillation, absorption, extraction

5.6.1 The rigorous column

`distillationColumn` solves the full MESH stage equations with two selectable numerical strategies (the `model` slot):

- `WangHenke` (default) — the sequential bubble-point method: a tridiagonal (Thomas) component-balance sweep at frozen K, then a bubble-point T update per stage. Cheap per pass, inspectable, but O(100–500) outer iterations, and it can step *through* an azeotrope non-physically. Use it for ideal / wide-boiling systems.
- `simultaneous` (aliases `MESH`, `NaphtaliSandholm`) — Newton on the whole block at once with Armijo backtracking. Quadratic (≈5–6 iterations) and stable through azeotropes — the required choice for non-ideal systems (NRTL ethanol/water: `column03_azeotrope_mesh`).

```
{ name column01;  type distillationColumn;      // model WangHenke is the default
  in feed;  outputs ( distillate  bottoms );
  operation
  {
      nStages          12;
      feedStage         6;
      refluxRatio      2.5;                     // XOR with distillateRate
      distillateRate   50 kmol/h;
      P                1.01325 bar;
  } }
```

A column carries two intrinsic heat ports — the reboiler (heating, at the bottoms T) and the condenser (cooling, at the top T), reported as `Q_reboiler_kW` / `Q_condenser_kW` and sized to plant utilities by temperature level (§5.11).

5.6.2 Multiple feeds, side draws, reactive stages

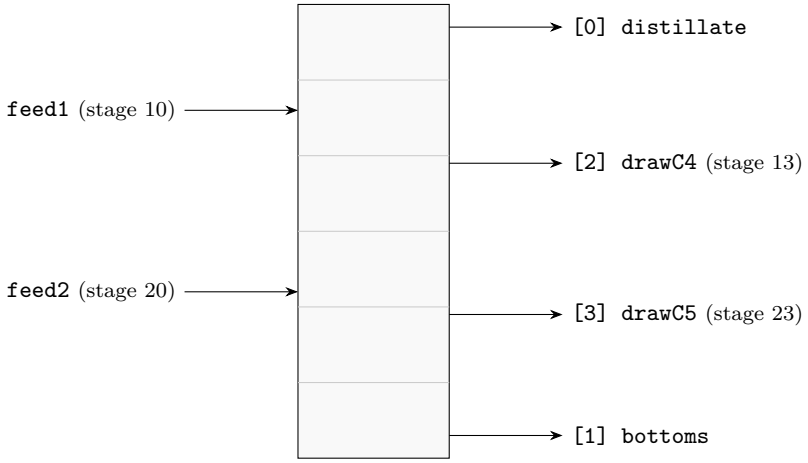
> *What you'll learn.* How the `simultaneous` column handles *several feeds*, *side draws*, and a *reaction on catalytic stages* — with a worked, paper-validated reactive-distillation case.

Feeds are flowsheet streams. A multi-feed column declares its feeds as *streams* (`inputs ( feed acidFeed )`); `operation.feeds` only maps each stream to a stage. This is deliberate (*information follows the streams*): a feed hidden inside the operation block would be a side channel the plant-boundary mass and energy balance never sees — and the first law would not close.

Side draws, and the order of `outputs`. A side draw is an *output* — a liquid or vapour bleed from an intermediate stage, its rate set in `operation.sideDraws`. Because outputs bind *by position* (§3.3), the column's emission order is fixed and your `outputs` list must match it:

$$\underbrace{\texttt{distillate}}_{[0]},\ \underbrace{\texttt{bottoms}}_{[1]},\ \underbrace{\text{side draws by } \textit{ascending stage}}_{[2],[3],\ldots}$$

So a four-product column that draws liquid at stage 13 then 23 is wired `outputs ( distillate bottoms drawC4 drawC5 )`: `drawC4` (position 2) is the stage-13 draw, `drawC5` (position 3) the stage-23 draw. Swap the two names and the label lands on the wrong stream — the engine does *not* warn.



feeds in by name*; products out* by position

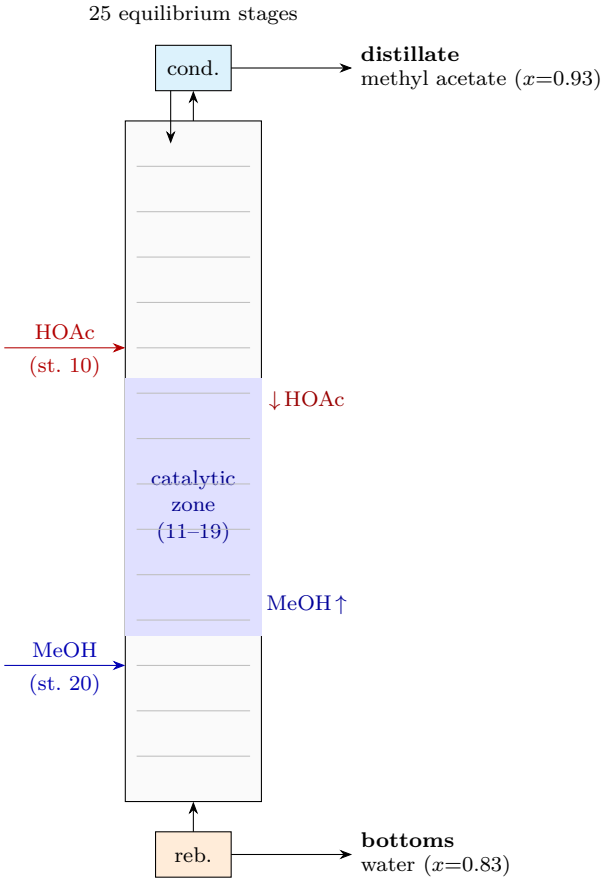
```
operation
{
    nStages 15;  refluxRatio 3.0;  distillateRate 42 kmol/h;  P 1.013 bar;
    feeds      ( { stream feed;  stage 5;  quality 1.0; }
                 { stream feed2; stage 11; quality 1.0; } );
    sideDraws  ( { stage 8;  phase liquid;  rate 15 kmol/h; } );
}
```

A reaction on the stages. A `reaction {}` block turns the column reactive (`simultaneous` only; restricted to mole-conserving $\sum \nu = 0$ reactions so the flow profile is untouched). The extent on each named catalytic stage is closed either by chemical `equilibrium` (activity-based $K_a(T)$, van 't Hoff) or by `kinetics` (a rate law $\times$ catalyst mass).

Worked case — methyl acetate by reactive distillation. The classic industrial esterification:

$$\mathrm{MeOH} + \mathrm{HOAc} \rightleftharpoons \mathrm{MeOAc} + \mathrm{H_2O}.$$

Acetic acid is fed *above* the catalytic zone and methanol *below*, so they flow counter-currently through it; the light product rises to the distillate, water sinks to the bottoms, and removing both products pulls the equilibrium forward — conversion far above a single equilibrium reactor.



The dict says exactly this — two input streams mapped to stages (their state authored in `0/feed` and `0/acidFeed`), plus the reaction:

```
units ( {
    name rdColumn;  type distillationColumn;  model simultaneous;
    inputs  ( feed  acidFeed );              // both feeds are flowsheet streams
    outputs ( distillate  bottoms );
    operation {
        nStages 25;  refluxRatio 2.1;  distillateRate 30.1 mol/h;  P 1.017 bar;
        feeds ( { stream feed;     stage 20; quality 1.0; }   // methanol below the zone
                { stream acidFeed; stage 10; quality 1.0; } );// acetic acid above it
    }
    reaction {
        stoichiometry ( { component methanol;      nu -1; } { component aceticAcid; nu -1; }
                        { component methylAcetate; nu  1; } { component water;      nu  1; } );
        reactiveStages ( 11 12 13 14 15 16 17 18 19 );
        equilibrium { Ka298 38.7; dHrxn -5670; }   // activity-based K_a(T), van't Hoff
    }
} );
```

Running it (`tutorials/steady/distillation/column05_reactive_methylacetate`) converges by homotopy (solve the column non-reactive, then switch the reaction on) and gives **96 % methanol conversion**, a methyl-acetate-rich distillate and a water-rich bottoms — the *equilibrium ceiling*. The measured value (Pöpken *et al.*, *Ind. Eng. Chem. Res.* 40 (2001) 1566) is 87.6 %: the real column is slightly kinetically limited, so the equilibrium model correctly brackets it from above (replace `equilibrium` with a `kinetics` block to model the finite rate). Because both feeds are flowsheet streams, the plant-boundary energy balance closes exactly.

5.6.3 Real trays: Murphree efficiency

Add `murphreeEfficiency 0.70;` to `operation` (`WangHenke` only — `simultaneous` refuses the keyword rather than silently returning ideal stages) and `nStages` stops counting ideal stages and starts counting *real trays*: $y_j = E\,K_j x_j + (1-E)\,y_{j+1}$, folded into an effective K. E is a *declared* number — O'Connell's correlation, a chart,

or plant data; the engine never invents it. `column11_murphree` is `column09` plus that one line: the distillate falls from 98.12 % to 95.05 % benzene, and at $E = 1$ every existing case reproduces bit for bit.

5.6.4 Tray hydraulics: will the column you converged actually run?

> *What you'll learn.* How to ask a converged column whether its trays can pass the traffic they carry — and how the same block, with one keyword removed, stops rating the column and starts sizing it.

A converged column tells you what separates. It says nothing at all about how wide it is, because nothing in the stage equations depends on the diameter. Two columns of very different width give the identical stream table — and one of them floods, which means in practice that it separates nothing. The Theory Guide § "Tray hydraulics" derives why.

Add a `hydraulics {}` block inside `operation` (either column model) and the column runs a rating pass over its own converged profile:

```
hydraulics
{
    trayType              sieve;
    // diameter           1.10;     // GIVE it -> rating.  OMIT it -> design.
    traySpacing           0.50;     // m
    weirHeight            0.050;    // m
    holeDiameter          5.0e-3;   // m
    holeAreaFraction      0.10;     // A_holes / A_active
    downcomerAreaFraction 0.12;     // A_downcomer / A_tower
    weirLengthFraction    0.77;     // l_weir / D
    orificeCoefficient    0.84;     // C_o -- YOU read this off the chart
    floodFraction         0.80;     // design target
    K2                    30.7;     // optional: enables the weep check
}
```

The thermo package must supply the surface tension. Entrainment is a contest between the vapour's drag on a droplet and the surface tension holding it together, so the Souders–Brown flooding velocity cannot be evaluated without σ. Add `transport { surfaceTension { model BrockBird; } }` to the `thermoPhysPropDict`, or declare a constant `sigma <N/m>;` inside the hydraulics block. The column refuses to guess; it will stop and say so.

Two modes, one block. Give `diameter` and the column *rates*: it reports how close each tray runs to its flooding velocity. Omit `diameter` and it *designs*: it finds the width at which no tray exceeds `floodFraction` of flood, and the widest tray sets the column. This is the same rating/design duality as the spray dryer (§5.9.2) — one set of equations, run in whichever direction you need.

Two numbers you must own. The orifice coefficient C_o and the weep constant K_2 are charts in the literature, not formulae. Choupo does not invent charts. C_o is a declared input, printed back at you with its provenance. K_2 is optional: give it and the weep check runs, leave it out and the column tells you plainly that the check was not performed. If a tray lands where the flooding fit is known to be unreliable ($F_{LV} < 0.03$), the column says that too.

What it prints. Run `tutorials/steady/distillation/column09_tray_hydraulics` at `verbosity 3`:

```
--- Sieve-tray hydraulics (DESIGN: the diameter is the result) ---
D = 1.322 m    tray spacing 0.500 m    weir 50.000 mm    holes 5.000 mm ...
C_o = 0.840  (declared -- the Liebson chart value for this A_h/A_p and t/d_h)
sized so no tray passes 80.000 % of its flooding velocity

stage    F_LV    C_SB    u_v    u_flood   %flood    h_t     dP      h_b   h_b,max
 1     0.0389  0.091   1.002   1.575    63.6     112.9   886.1   178.1   275.0
 ...
14     0.0826  0.085   1.081   1.351    80.0     135.3  1017.8   215.8   275.0
worst tray: 80.0 % of flood on stage 14   column dP = 13.26 kPa   ...
```

The last stripping tray sets the diameter, sitting exactly on the 80 % it was asked for. Notice that the flow parameter F_{LV} jumps at the feed — more liquid, less vapour below it — which lowers the capacity parameter C_{SB} and so the flooding velocity. Which tray ends up tightest is not obvious in advance; you find out by looking.

Try it now. Run `column10_flooding`. It is the *same* column, built 1.10 m wide instead of 1.322 m. Compare the stream tables: they are identical to the last digit, the MESH converges just as fast, the distillate is still 98.12 % benzene. Then read the hydraulics: stage 14 runs at 115.6 % of flood, and the downcomer backs up to 271 mm against a 275 mm limit. On paper it is a perfectly good column. It would not work. This is exactly why the check exists.

Common pitfall. Hydraulics is a *rating pass on the converged profile*, not a coupled unknown. The 13 kPa of pressure drop it reports is **not** fed back into the stage temperatures — the column was solved at one pressure. For a low-pressure column where the drop is a large fraction of the operating pressure, that matters, and Choupo says so rather than quietly absorbing it. The KPIs `diameter`, `floodApproach_max`, `floodStage`, `dP_column_kPa` and `downcomerBackup_max_mm` are available to a sweep or an optimiser, so you can trace how far you can push the reflux before a tray floods.

5.6.5 Shortcut column, absorber, stripper, extractor

`shortcutColumn` (alias `FUG`) is the Fenske–Underwood–Gilliland *design* method (constant α, no azeotropes): declare the key components, their recoveries and `refluxFactor` ($R/R_{\min}$); $N_{\min}$, $R_{\min}$, N, and the feed stage are results. Use it to size first, then hand the numbers to `distillationColumn` for a rigorous rating.

`absorber` / `stripper` are counter-current staged gas–liquid contactors solved by the Thomas tridiagonal; the solute(s) need a Henry pair (§4). `extractor` (alias `extract`) is the liquid–liquid analogue: each theoretical stage is one LL equilibrium computed by the *same* Gibbs-minimisation flash kernel the flash uses (never re-implemented), the cascade swept by successive substitution. Hardware in, results out: `stages` is the knob; recovery and the distribution ratio are results. Tutorials: `absorber01_NH3_water`, `extract01_ethanol_water_benzene`.

5.7 Membranes and electrochemical separations

`spiralWoundModule` (alias `membraneSW`). The NF/RO element: solution-diffusion permeation + film-model concentration polarisation, with selectable sub-models for feed-channel mass transfer (`SchockMiquel` or `constant`), pressure drop, and osmotic pressure (`vanHoff` or `Pitzer`):

```
operation
{
    membrane         SW30HR;        // -> data/standards/assets/<name>.dat
    area             35 m2;         // per element
    length           1 m;
    elements         1;             // >1 for a train in one vessel
    P_perm           1.01325 bar;
    dP_feed_total    2 bar;
    massTransfer  { model SchockMiquel;  channelHeight 0.7 mm;
                    diffusivity 1.6e-9 m2/s;  viscosity 1.0e-3 Pa.s; }
    osmotic       { model Pitzer; }
}
```

Inputs `( feed )`, outputs `( retentate permeate )`. KPIs: `J_w_avg`, per-solute observed rejection, `water_recovery`, `dP_feed`. Four membranes ship in the catalogue (a seawater-RO archetype, a loose-NF archetype, a pore-model NF, and an ion-exchange pair); the solutes carry `nonvolatile true;` and a van 't Hoff `dissociation` factor. A `scaling { minerals (calcite gypsum); }` block runs the aqueous speciation solver per module at the bulk *and at the wall*, so you see concentration polarisation as the scaling driver (`membrane07_scaling_si`). Start with `membrane01_RO_NaCl_seawater`, then the concentration-polarisation sweep `membrane03_concentration_polarization`.

`ionExchanger`. A cation-exchange softener: the inlet water analysis is fully equilibrated with a fixed-capacity Na-form resin (Gaines–Thomas mass action), so Ca/Mg drop onto the resin and Na rises equivalent-for-equivalent. `operation { resin SAC_Na; CEC 2.0 eq/L; }`; hardness removal is a *result* (a target-hardness key is refused — that is an outer-driver job). Every run carries a non-suppressible banner: this is the fresh-resin equilibrium limit, not a bed in service (breakthrough is transient). Tutorial: `membrane08_softened_scaling` — softener upstream of the RO train, the scaling driver removed at the source.

electrodialysisStack. N cell pairs of alternating cation/anion-exchange membranes: an applied current drives ions out of the diluate channel into the concentrate. Glass-box electrochemistry, all printed: Faraday transfer per cell pair ($\dot{n}_i = \xi I / z_i F$), Nernst membrane potential from real ion activities, the Cowan–Brown limiting current density (a loud warning — never a clamp — when $i > i_{\mathrm{lim}}$), ohmic drops from the membrane and channel resistances, and the stack power published on an energy wire.

```
{ name edStack;  type electrodialysisStack;
  inputs ( diluateFeed concentrateFeed );  outputs ( diluate concentrate );
  operation
  {
      N_cellpairs 100;   current 8.0;        // A
      xi 0.9;                                 // current efficiency (announced)
      membraneArea 0.2 m2;  channelThickness 0.5 mm;
      channelLength 0.4 m;  linearVelocity 0.05 m/s;
      E_electrodes 1.5;  membrane CMX_AMX;
  }
  energyOutputs ( { name power; kind work; expression "W_electric"; unit W; } ) }
```

Tutorials: `tutorials/electrochem/ed01_nacl_desalination` and `ed02_over_limiting_current` (what the i_{lim} warning looks like when you cross it).

psa. Pressure-swing adsorption as a steady flowsheet unit — the equilibrium-theory (local-equilibrium) shortcut: the swing between P_{high} and P_{low} on a competitive-Langmuir adsorbent sets the captured flow per species, with a declared equilibrium-utilisation derating `eta` and a purge fraction. What the shortcut does *not* model (breakthrough, cycle time, thermal effects) is announced in the run header. A result-named key (`recovery`, `purity`) in `operation` is refused — hitting a target is an outer-driver job. Tutorial: `tutorials/steady/separation/psa01_h2_psa`.

5.8 Solids: separation, conveying, crystallisation

Solids ride streams as a `phases { solid {...} }` decomposition plus an optional particle-size distribution (`psd`); the solids units read and transform both.

Gas–solid separation. `cyclone` is a PSD-aware classifier with five selectable models (`Lapple` default, `LeithLicht`, `IoziaLeith`, `Barth`, `Muschelknautz` — the last adds the solids-loading effect); `bagFilter` is cake-dominated near-total collection whose headline KPI is `dP_filter`; `gasSolidSplitter` is the ideal spec'd split (PSD passes unchanged — it does *not* classify).

pneumaticConveyor. A dilute-phase conveying line: the total ΔP is the sum of announced contributions (gas + solids acceleration, wall friction, static head), particle velocity by Hinkle, and the *Rizk saltation velocity* reported with a loud warning if the gas velocity falls below it (settling / blockage risk) — never a silent clip.

crystalliser. Three models on one saturation foundation:

- `equilibrium` (default) — cooling crystalliser; the mother liquor leaves saturated, yield = solute in − $c_{\mathrm{sat}}(T)$·solvent.
- `MSMPR` — steady continuous population balance by the method of moments: nucleation/growth kinetics from `constant/crystallisation`, supersaturation S a result, and the full crystal-size distribution (mean sizes, CV, magma PSD) reported.
- `FVM` — the discretised population balance on a size grid (size-dependent growth).

All three read the same saturation — a solubility curve (`solubility {}` on the component) or, for a salt, the ion-activity K_{sp} of the electrolyte package (including antisolvent drowning-out; `crystalliser08_nacl_antisolvent_msmpr`). Multiple feeds are first-class (`inputs ( brine antisolvent );`) and `operation { solute <name>; }` names which salt this unit crystallises when the feed carries several (`crystalliser09_KHT_KCl_series`). The declared `outputs` pick the product shape: `( magma )` for one slurry (required by MSMPR/FVM so the PSD travels), or `( crystals motherLiquor )` for a solid–liquid split, where the cake leaves *wet* (`cakeMoisture`, default 0.10 kg liquor per kg dry solid — perfectly dry solids are unphysical) and `solidsRecovery` lets fines escape with the liquor.

5.9 Drying

Three dryers, one honesty rule — the drying air is a *real stream* (it brings the heat and carries the moisture; the outlet temperature is a result, there is no imposed duty):

- `sprayDryer` — solution/suspension → powder (§5.9.1);

- `solidDryer` — polish a *hygroscopic* powder toward the equilibrium moisture set by the air humidity and the solid's GAB sorption isotherm (a `sorption {}` block on the component, typically a case-local overlay — sorption is sample-specific);
- `evaporativeDryer` — dry a *non-sorbing* crystalline solid of its free surface moisture: no isotherm exists or is consulted; constant-rate evaporation until the first of three announced limits binds (water gone, exhaust saturates, air runs out of heat).

5.9.1 Advanced spray drying: the atomiser library

> *What you'll learn.* How the `sprayDryer` atomises a solution into droplets, dries them to a powder of a real particle size, and where each piece of physics — the droplet size, the liquid properties, the drying curve — comes from. "All models are wrong, but some are useful": every correlation is an algebraic fit (no CFD) and announces its source and validity envelope at run time.

A spray dryer takes a liquid feed and hot drying air (both flowsheet *streams*) and returns a powder plus the humid exhaust. Three pieces of physics compose: (i) *atomisation* sets the droplet Sauter diameter d_{32}; (ii) *single-droplet drying* (Ranz–Marshall $\mathrm{Nu} = 2 + 0.6\,\mathrm{Re}^{1/2}\mathrm{Pr}^{1/3}$, a wet-bulb surface) sets the rate; (iii) the droplet shrinks to a powder particle $d_p = d_{32}\,(x_{\mathrm{solid}}\rho_L/\rho_{\mathrm{solid}})^{1/3}$ carrying a residual moisture set by the material's sorption isotherm.

The atomiser is a selectable sub-model.

```
operation
{
    atomiser
    {
        model   pressureNozzle;     // rotary | pressureNozzle | twinFluid
        deltaP  40 bar;             // pressure-nozzle hardware
    }
    chamberDiameter  1.0 m;         // -> residence time is a RESULT
    chamberHeight    5.0 m;
    flow             co;            // co | counter current
}
```

Omit the `atomiser` block and the default `rotary` wheel reads the legacy `wheelSpeed`/`wheelDiameter` keys — older cases are unchanged. The families and their hardware:

`model`	correlation (d_{32})	hardware
`rotary`	Friedman/Marshall wheel	`wheelSpeed` [rpm], `wheelDiameter` [m]
`pressureNozzle`	$9575\,\Delta P^{-1/3}\,\mu$m (class curve)	`deltaP` [Pa]
	or Wang–Lefebvre (two-term)	`correlation wangLefebvre`
`twinFluid`	Nukiyama–Tanasawa airblast	`relativeVelocity`, `airLiquidMassRatio`

Droplet size follows the liquid properties. d_{32} rises with surface tension ($\sigma^{0.1}$–$\sigma^{0.6}$) and viscosity ($\mu_L^{0.2}$), so a concentrated, viscous feed sprays coarser. Choupo pulls σ and μ_L from the property layer when the `thermoPhysPropDict` declares the models (`surfaceTension`, `liquidViscosity`), with an announced fallback otherwise — the σ source is printed in the report.

Drying kinetics — equilibrium vs. curve. The powder does not leave bone dry: it stops at the residual moisture set by (a) the equilibrium X_e from the solid's `sorption` isotherm at the exhaust-air humidity, and (b) the falling-rate approach past the critical moisture X_c (the `dryingCurve` in `constant/dryingKinetics`). Co- vs. counter-current changes the residence time and, in the trajectory models, the droplet thermal history.

Chamber model — named by its authors, not "rigorous". The unit-level `model` slot picks the chamber model: `marshall` (default) — the lumped inlet/outlet mass + energy balance and single-droplet constant-rate time (Marshall 1954); `langrishKockel` — the 1-D axial droplet trajectory with the Characteristic Drying Curve, which reports the profiles of X, d, T_g, T_p *along* the chamber (a `profile.csv` you can plot); or `chen` — the same trajectory with the Reaction Engineering Approach kinetics, a smooth activation-energy rate with no critical-moisture break. Reported KPIs: d_{32}, d_p, Nu, Sh, h, k_c, T_{out}, T_{wetbulb}, the constant-rate droplet time and X_{final} — all feeding sensitivity/sizing/costing.

5.9.2 Simulation vs. design — the model runs both ways

It can feel odd to supply a wheel speed and a chamber size just to *simulate* a dryer. That feeling is correct, and it has a name: what you are doing is *rating* — geometry → performance, "I have this dryer, what does it do?". The opposite question — "I want this powder, what dryer do I need?" — is *design*, targets → geometry. Choupo does not split these into two black boxes; it keeps **one model and runs it in both directions**.

Rating is the `sprayDryer` unit itself. *Design* is a `postDict` sizing pass carrying a `design {}` block, which **inverts the run you just rated** — it does not re-derive the correlations, it *scales from the simulated point*, so every step is visible:

```
sizing
{
    units
    (
        {
            unitName    dryer;
            type        sprayDryer;
            material    SS304;
            designRules
            {
                design
                {
                    targetSize       40;    // um  -- target powder Sauter mean d32
                    residenceFactor  3.0;   // chamber residence >= 3 x drying time
                }
            }
        }
    );
}
```

The pass then reports the geometry the targets demand: the wheel speed from the target drop size (Friedman's $d_{32} \sim N^{-0.6}$) and the chamber size from the required residence, with the Tanasawa regime warning still telling you whether that design lands in the clean ligament regime. In `tutorials/steady/drying/sprayDryer07_design` the sugar dryer rated at 20 000 rpm must spin to ≈40 000 rpm for a 40 µm powder, and its 1×5 m chamber turns out ∼30× oversized — the fast sugar drying needs only 0.24 s of residence. **You simulate with the unit and design with the pass: two directions of one model, both on the page, neither hidden.**

5.10 Flow primitives: mixer, splitter, valve, pipe

`mixer`. Adiabatic merge: $P_{\text{out}} = \min(P_{\text{in}})$, energy balance on the dominant phase, solids summed, Newton-1D in T_{out}. A mixer *mixes* — it never runs an internal flash (separators separate). Declare gas inlets with `phase gas;` in their `0/` files so the energy balance picks the right basis; an optional `operation { T 330 K; }` declares the mix temperature and skips the adiabatic balance (announced in the log).

`splitter`. One inlet, N outlets at `fractions ( 0.6 0.4 );` — a flow split, not a separation: all branches share T, P, composition, vf.

`valve`. Isenthalpic throttle to `operation { P 1 bar; }`; T_{out} and the vapour fraction are results (a real-gas EoS flashes the let-down). Use it upstream of a flash to set the drum pressure; it warns if $P \geq P_{\text{in}}$.

`pipe`. The pressure *drop* is a computed result of geometry + flow — never a spec (a pipe has no knob; a pump raises P, a valve sets a let-down). Darcy–Weisbach friction + minor losses + static head, with the friction-factor sub-model in the `model` slot (`Churchill` default — one explicit correlation across all regimes; `Haaland`; `Colebrook`):

```
{ name P1;  type pipe;  model Churchill;
  in feed;  outputs ( downstream );
  operation {
    geometry {
      D 0.1 m;  L 50 m;  roughness 4.6e-5 m;  dz 2 m;    // rise costs P
      fittings ( { K 0.9; count 2; }      // two elbows
                 { K 10.0;         } );   // one globe valve
    }
  }
```

```
}
```

When the feed flashes two-phase at (T, P), the pipe automatically switches to a gas–liquid model, selected in an `operation { twoPhase { model ...; } }` sub-block: `LockhartMartinelli` (default; Chisholm multiplier + Butterworth holdup), `homogeneous`, `Friedel`, or `BeggsBrill` (flow-pattern map + inclination). KPIs include the five-way ΔP breakdown, velocities, Reynolds, regime, holdup and void fraction. Tutorials: `pipe01_water_line`, `pipe02_airwater_twophase`.

5.11 Heat duties, utilities, heat integration

One rule covers heat: **a duty is carried by a physical stream when one is wired to it, otherwise it is sized to a plant utility by temperature level**. Shaft work is a direct scalar wire (§5.5); no abstract "Q stream" ever travels the flowsheet.

Q then allocate (the default). Most units just report their duty and let a report size the utility. Add to `controlDict`:

```
reports { utilityAllocation { dTmin 10; } }
```

For each duty Q at process temperature T the report picks the lowest-grade catalogue utility that still works (a steam grade with $T_{\text{in}} \geq T + \Delta T_{\min}$ for heating; a coolant with $T_{\text{in}} \leq T - \Delta T_{\min}$ for cooling) and converts $|Q|$ to a mass flow, a load, and a cost. Nine utilities ship in `data/standards/utilities/`: `steamLP/MP/HP`, `coolingWater`, `chilledWater`, `dowthermA`, `hitecSalt`, `refrigerationPG`, and `electricity` — the power tier tallies pump/compressor motors against turbine generators, so a combined cycle shows a net electricity credit.

Heat integration. Reuse a column's condenser heat by wiring the duty on the consumer, like any energy link:

```
{ name preheater;  type heater;  in coldFeed;  outputs ( warmFeed );
  operation { }                                  // Q comes from the link
  energyInputs ( { from column01.condenser;  kind heat;  target Q; } ) }
```

Both ends then read *(carried)* in the allocation — no double-counting, and the report shows the saving. A *forward* link runs in one pass; a *feedback* link (the condenser preheating the column's own feed) is an energy tear — the flowsheet detects it, turns the recycle loop on, and converges the duty by successive substitution. Tutorials: `heatlink01_condenser_to_heater`, `heatlink02_condenser_feed_preheat`.

6 Reports

What you'll learn. One `reports {}` block in `controlDict` turns a converged run into CSV tables and a consolidated spreadsheet — stream tables, balances, utility bills, profiles, costs.

Each key names a report object; after a converged solve each writes into the case's `reports/` directory and announces the file:

Report	What it writes
`streamTable`	one row per stream: role, F, F_{mass}, T, P, vf, compositions
`massBalance`	per-component in/out/net + global closure %, and a per-unit table
`elementBalance`	plant-boundary *atom* conservation, one row per element (kmol-atom/h in, out, residual, closure %) + a status sidecar (§6)
`energyBalance`	per-unit H_{in} / H_{out} / ΔH (the implied duty) + the plant boundary
`utilities`	consumption aggregated by stream `category` (kg/h, MW, cost)
`utilityAllocation`	each duty sized to a catalogue utility by temperature level (§5.11)
`energyStreams`	the W/Q wire ledger
`computed`	the values of computed `$variables` (below)
`profiles`	per-unit internal profiles (column stages, PFR axis, dryer trajectory)
`design` / `economics`	equipment realisation + cost tables (with a `postDict`, §8)
`spreadsheet`	one consolidated, coloured `report.ods` (multi-sheet)

The element balance is a default diagnostic. Species moles may change legally through reactions; the invariant of a reacting flowsheet is the *atoms*. Every converged `choupoSolve` run therefore writes `reports/balances/elementBalance.csv` (one row per element: kmol-atom/h in, out, residual, closure %) and its `elementBalance.meta` status sidecar *without being asked* — a declared `elementBalance {}` block keeps full control, and any report block can opt out with `enabled false;` inside it. The claim is honest by construction: a component whose formula cannot be parsed and that declares no `elementalComposition{}` block makes the status `UNAVAILABLE` *naming the species and the reason*; a declared-composition component with unaccounted mass makes it `PARTIAL`. The GUI's *Element balance* view draws exactly this artefact: total atoms in vs out, sealed only when *every* element closes (a signed total could cancel a +C error against a −O error), with the per-element decomposition one click away. The theory and the parser contract are derived in the Theory Guide (*Balances: mass, elements, energy*).

Computed variables. For post-processing arithmetic, declare `compute` expressions in the `variables` block of `flowsheetDict` — they resolve against unit KPIs, stream fields, and earlier variables, and land in `reports/computed/values.csv`:

```
variables
{
    Q_total    { compute "effect1.Q + effect2.Q + effect3.Q";  unit kW; }
    economy    { compute "water_evap / effect1.F_steam"; }
}
```

7 Sweeps, design specs, optimisation

> *What you'll learn.* How to wrap the simulator in an outer loop without writing a single line of C++: trend one knob (`sweep`), map two (`gridSweep`), meet N specs with N knobs (`designSpec`), or search for the best value (`optimization`).

Drop a `system/outerDict` into a case and the binary switches from one pass to the driver loop automatically. One driver per case; `type` selects it. Ask the right question of the right tool:

Question	Tool
"What if T were 360 K?" (one point)	edit the dict, re-run
"How does V/F trend with T?"	`sweep`
"What does the (T, P) envelope look like?"	`gridSweep`
"What area gives exactly 4500 kg/h?"	`designSpec`
"Which reflux minimises V_{strip}?"	`optimization`
"Fit NRTL to my bubble-T data"	`fitParameters` under `choupoProps` (§11)

Paths. Drivers mutate the case between passes via dot-and-bracket paths (`units[0].operation.refluxRatio`, `variables.A`, `streams.feed.F` — the last steered through an explicit stream-override channel, since stream state lives in `0/`). Values written this way are *raw canonical SI*. Results are read back as `<unit>.<kpi>` or `<stream>.{F,T,P,vf}`.

7.1 `type sweep;` — trend over one parameter

```
type        sweep;

parameter
{
    target    units[0].operation.T;        // any scalar dict path
    range     ( 360.0  385.0 );            // (min max), raw SI of the target
    nPoints   26;
}

responses
(
    flash01.V_over_F       // <unit>.<kpi>
    flash01.Q_kW
);

report { format csv;  file sweep_flashT.csv; }
```

One row per point; a point that fails to converge is reported and written as `nan` — the sweep continues. Canonical case: `tutorials/steady/flash/flash06_sweep_T` (the V/F S-curve across the bubble–dew window). `type gridSweep;` is the 2-D version — a `parameters ( {...} {...} )` list of two targets, producing the long-form CSV a contour plot reads (`flash07_gridsweep_TP`: the flash envelope over (T, P); non-converged cells stay honest gaps, never interpolated).

A sweep may declare CONSTRAINTS — the same block the SQP optimizer uses, same grammar, same refusals:

```
constraints
(
    { kpi column01.x_D_LK;   atLeast 0.985; }
    { kpi column01.V_strip;  atMost  0.065;  }
);
```

The sweep becomes a FEASIBILITY MAP: every row gains, per constraint, the value, the bound, the signed normalised residual and a `satisfied` flag, plus a final `feasible` verdict — and no point is ever filtered. The infeasible rows are the lesson: where the spec fails and where the budget binds, with the feasible window read directly off the CSV (`sensitivity02_feasibility_map`: the reflux window between the purity spec and the boilup budget).

Two objectives that pull against each other build a PARETO FRONT by epsilon-constraint — `type paretoSweep;` — with no multi-objective solver: the second objective becomes a swept bound and each front point is one full, auditable SQP run,

```
type paretoSweep;   method sqp;
variables ( { path units[0].operation.refluxRatio;
              min 1.5; max 6.0; initial 3.0; } );
objective { kind kpi;  path column01.V_strip;  sense minimise; }
tradeoff  { kpi column01.x_D_LK;  bound atLeast;
            range ( 0.96  0.994 );  nPoints 5; }
report    { file pareto.csv; }
```

The CSV records, per point, the bound, the achieved objective and trade-off values, the variables at the optimum and the solver's own converged/feasible verdicts — a failed point stays on the table, marked. Where the bound is slack the point is the unconstrained optimum; where it binds, the SQP's Lagrange multiplier is the marginal price of the second objective (`pareto01_purity_energy`: the purity–energy front of the benzene/toluene column, R^* rising $1.6 \to 3.8$ as the purity bound tightens).

An INTEGER decision — how many stages? — is declared, never relaxed: `integer true;` on a variable makes the optimizer enumerate every value in `min..max`, each one a NEW run (a changed `nStages` rebuilds the column; a late parameter mutation on a sized object would lie), solving the inner continuous problem at each. The FULL table is emitted — failures included, best marked (`optim06_integer_stages`: the classic N-vs-R trade, boilup falling $1.03 \to 0.045$ kmol/s from 9 to 14 stages at the same purity spec, with $N = 8$ refused aloud because the feed cannot sit on the bottom stage).

Try it now. `runCase tutorials/steady/optimisation/sensitivity01_column_reflux` produces `sweep_results.csv`. Plot V_{strip} against R with any tool; you'll see the classic energy asymptote as the reflux ratio grows past the minimum.

7.2 `type designSpec;` — meet N specs with N knobs

The DesignSpec is the visible version of "back-compute the input from the target". Each knob is a case-level `$variable`, declared in `flowsheetDict` and referenced by the unit — the driver iterates the variable, a square Newton meets the targets:

```
// system/flowsheetDict
variables { A 100 m2; }                       // initial guess; DesignSpec iterates it
units ( { name effect1; type evaporator;  ...
          operation { area $A; ... } }
        { name effect2; type evaporator;  ...
          operation { area $A; ... } } );     // the SAME $A: equal areas by construction
```

```
// system/outerDict
type         designSpec;

manipulate
(
    { variable A;   initial 100.0 m2;   min 20.0 m2;   max 500.0 m2; }
    { variable streams.utilitySteam_200kPa.F;
                  initial 400 kmol/h; min 100 kmol/h; max 1500 kmol/h; }
);

targets                         // SAME COUNT as knobs (square Newton)
(
    { path  L3.F_mass;   value 4500.0 kg/h;   tol 1.0 kg/h; }   // value form
    { path  effect3.P;   value 15 kPa;        tol 50 Pa;    }
    // equality form: { lhs effect1.area;  rhs effect2.area;  tol 0.001; }
);

options { maxIter 40;  tolF 1.0;  fdStep 1.0e-4; }
report  { file designspec_history.csv; }
```

All knob entries carry units, converted on read; a `manipulate` variable may also be a stream field (as above — the steam flow the plant must supply). After convergence the simulator is replayed once at the design point so the case ends with full converged output. Canonical cases: `compressor02_designspec_pout` (1×1), `designSpec01_triple_equal_areas` (2×2: one area + one steam flow meet a capacity and a vacuum spec at once).

7.3 `type optimization;` — search for the best value

```
type    optimization;
method  nelderMead;                        // or sqp (adds nonlinear constraints)

variables
(
    { path units[0].operation.refluxRatio;  min 3.0;  max 5.0;  initial 4.0; }
    { path units[0].operation.feedStage;    min 6;    max 12;   initial 8;   }
);

objective
{
    kind     kpi;             // kpi | stream | cost | costTotal
    path     column01.V_strip;
    sense    minimise;        // minimise (default) | maximise
}

options { maxIter 80;  tolX 5e-3;  tolF 1e-4; }
report  { file optimization_history.csv; }
```

`nelderMead` is the derivative-free simplex with hard box bounds — an evaluation that fails to converge gets an announced infinite penalty and the simplex contracts away from it. The `cost` / `costTotal` objective kinds run the `postDict` sizing + costing chain on every evaluation (§8) — see `optim01_column_reflux` and `optim02_process_cost`.

`method sqp;` switches to a hand-rolled line-search Sequential Quadratic Programming (damped BFGS on the Lagrangian, L1 merit, active-set QP) and accepts a `constraints (...)` list with unit-aware `atLeast` / `atMost` / `equals` bounds on any KPI. The trace is glass-box: per-iteration KKT residuals, the active set, the Lagrange multipliers (shadow prices), and a closing five-line KKT certificate. Worked example: `optim04_membrane_recovery_scaling` — brackish-RO recovery maximised subject to wall-scaling constraints, with the calcite constraint active at the optimum and its multiplier printed (the shadow price of scaling head-room). Asking Nelder–Mead for constraints is a loud error — add a penalty term or switch method.

Parameter estimation. Regression of model parameters against experimental data lives in `choupoProps` as the `fitParameters` operation (§11); the old `fitBinaryPair` outer driver was retired, and asking for it points you to the replacement.

8 Equipment sizing and costing

What you'll learn. How a single `postDict` turns a converged flowsheet into a vessel list, wall thicknesses, weights, and bare-module costs.

Drop a `system/postDict` and the simulator result is post-processed after the pass:

```
sizing
{
    units
    (
        {
            unitName    reactor;
            type        stirredTank;
            material    SS316;
            designRules
            {
                L_over_D        2.5;
                pressureDesign  3.0;
                corrosionAllow  0.003;        // m
                jointEfficiency 1.0;
            }
        }
        {
            unitName    heater;
            type        shellTubeHX;
            material    SS304;
            designRules { U 600.0;  LMTD 10.0;  pressureDesign 2.0; }
        }
    );
}

costing
{
    method      Guthrie;
    year        2026;
    cepci       820;
    cepci2001   397;
    usdToEur    0.92;
}
```

`type` selects an `EquipmentSize` model (`stirredTank`, `shellTubeHX`, and the `design {}` inversions such as the spray dryer's, §5.9.2); materials come from `data/standards/assets/` (carbon steel, SS304, SS316, aluminium — each with density, Guthrie material factor, yield stress, ratings). The `Guthrie` costing model uses Turton-style purchased-cost correlations, the ASME wall-thickness pressure factor for vessels, then $C_{BM} = C_p(B_1 + B_2 F_M F_P)$ and $C_{TM} = 1.18\, C_{BM}$, printing a per-unit table:

```
========================  Equipment Costing  ========================
  Method:  Guthrie    Year:  2026.0    CEPCI: 820.0

  unit      equipment       F_M   F_P    C_purchased   C_bare_mod    C_total_mod
  heater    shellTubeHX     2.34  1.09     365 249      2 146 753    2 533 169
  reactor   stirredTank     3.05  1.00       2 055         16 033       18 918
  TOTALS (EUR)                              367 304      2 162 786    2 552 087
```

Cost indices, currency rates and prices are *case data* — dated and cited in the dict, never invented by the engine. Run `tutorials/steady/flowsheets/process02_with_design` for the full chain, and `optim02_process_cost` for an optimiser driving it.

9 Batch simulation: choupoBatch

> *What you'll learn.* How the time-dependent binary integrates closed vessels — batchReactor, batchStill (plain Rayleigh or a staged rectifier), batchCrystalliser — and how a recipe block scripts a campaign of transfers and setpoint changes.

A batch case is the same case shape with time controls in controlDict:

```
application    choupoBatch;
startTime      0;          // s
endTime        600;        // s
deltaT         1.0;        // s   (RK4 step)
writeInterval  30;         // s   (trajectory snapshot interval)
```

Each vessel carries an initial {} charge instead of an inlet stream; the run writes trajectory.csv (one column per state variable, one row per writeInterval).

9.1 batchReactor

```
{ name reactor;  type batchReactor;
  initial
  {
      T 350 K;  P 1.013 bar;  V 0.001;  totalMoles 0.0185;
      molarComposition { ethanol 0.5;  aceticAcid 0.5; }
  }
  operation { mode isothermal;  T_setpoint 350 K; }   // or: mode adiabatic
  reaction  esterification; }                          // or: reactions ( A_to_B B_to_C );
```

mode isothermal holds T; adiabatic integrates the energy balance alongside the species — batch02_adiabatic shows a cold start, a slow induction, then thermal runaway from 345 to 482 K in a hundred seconds. The kinetics vocabulary is shared with the steady reactors: Arrhenius, the gas-kinetics forms (modifiedArrhenius, thirdBody, falloff — real combustion mechanisms run here, see ignition02_h2air_slack), and LHHW — the *same* RateLaw class (§5.3.5), so a batch vessel and a CSTR can never disagree about what an adsorption denominator means.

Stiff chemistry: adaptive time-stepping. The default is the fixed-step RK4 loop. Add timeStepping adaptive; to controlDict and the stiff, error-controlled Rosenbrock23 integrator takes over — the step starts low (deltaT0), grows through quiet stretches, and shrinks through the runaway, with the step history printed at verbosity 3. Trajectory rows still land on the clean writeInterval grid. Compare batch06_adaptive_runaway with its fixed-step twin, and ignition01_h2o2 (real GRI-Mech H/O ignition, $\lambda \sim 10^{15}$) with ignition01_h2o2_rk4, which blows up to NaN at the same deltaT — the stiffness lesson on real chemistry.

9.2 batchStill — Rayleigh and the batch rectifier

The default model rayleigh; is the classical open still: constant boil-off F_vap, the bubble point re-solved each step, the pot trajectory following the Rayleigh integral (still01_benzene_toluene reproduces the closed form to ∼2 %; still02_ethanol_water_azeo shows the pot moving *away* from the azeotrope — a Rayleigh still cannot cross it from below). dischargeTo receiver; routes the condensed vapour into a passive batchAccumulator, so pot + receiver conserve the charge at all times.

model rectifier; adds the classical batch rectification: nStages ideal stages above the pot and a total condenser, with the reflux policy a first-class choice:

```
{ name still;  type batchStill;  model rectifier;
  dischargeTo receiver;
  initial   { T 365 K;  P 1.013 bar;  totalMoles 1e-3;
              molarComposition { benzene 0.5;  toluene 0.5; } }
  operation { P 1.013 bar;  F_vap 1.5e-6;      // boilup V, kmol/s
              nStages 3;
              refluxPolicy constantReflux;  refluxRatio 3.0; } }
```

Under `constantReflux`, `F_vap` is the internal boilup V; the product is $D = V/(R+1)$ and the distillate composition $x_D(t)$ is solved each instant from the quasi-steady stage cascade — it starts rich and drifts down, the constant-R purity trade (`still04_rectifier_benzene_toluene`; with `nStages 0` and $R = 0$ it reproduces plain Rayleigh to the last digit).

`refluxPolicy constantComposition;` holds a light-key purity instead: `lightKey benzene; x_D_target 0.93; refluxMax 8; onRefluxLimit stopDistillation;` — $R(t)$ is solved each step by a bracketed search, climbing as the pot depletes; when the ceiling can no longer reach the target the run *announces* the limit and executes the declared action (distillation stops, the event lands in the trajectory and the KPIs). Distinct refusals exist for a target below the $R = 0$ floor, a target unreachable at `refluxMax`, and an azeotrope barrier — computed from the VLE, not guessed (`still05_rectifier_constant_xD`).

9.3 batchCrystalliser

A supersaturated charge desupersaturates in time: RK4 on the packed moments $(\mu_0, \mu_1, \mu_2, \mu_3, n_{\text{solute}})$, against the same kinetics library the steady MSMPR uses (`constant/crystallisation`), so the crystal-size distribution grows through the batch (`batch05_crystalliser`).

9.4 Recipes — scripting a campaign

A `recipe (...)` list in `flowsheetDict` schedules events by time or by condition:

```
recipe
(
    { time 400;  action transfer;      from reactor;  to still;  fraction 1.0; }
    { time 200;  action setParameter;  unit reactor;  key T_setpoint;  value 350.0; }
    { when { unit reactor;  quantity x_ethanol;  op lt;  value 0.30; }
      action transfer;  from reactor;  to still;  fraction 0.5; }
);
```

Actions: `transfer` (partial via `fraction`; the receiving vessel may start empty), `setParameter`, and condition-triggered `when` events (`quantity` is `T`, `total`, `n_<comp>` or `x_<comp>`; `op` is `gt`/`ge`/`lt`/`le`). Every fired action lands in the result's timeline — the machine-readable campaign sequence — and, underneath it, the MATERIAL LEDGER: one structured record per material edge (interval, vessels, per-component amount, transported enthalpy integrated package-by-package). The run closes a campaign MASS BALANCE over all-phase vessel inventories ($m_0 = m_{final} + m_{external}$; per-ELEMENT for reacting campaigns) and reports it under the `campaign` KPIs; a genuine leak marks the run non-converged. A still without a `dischargeTo` receiver is recorded as an unrouted external outlet, never a silent loss. `recipe01_react_then_distill` runs the classic react-then-still sequence in one case.

The material ledger has an ENERGY twin: per vessel, one record per segment of constant physics (a `setParameter` or a charge closes the segment), with the isothermal reactor's jacket duty priced exactly from committed mole numbers ($Q = \Delta H$ of the closed constant-P segment; a declared `dH_rxn` prices lumped species via reaction extents, announced). A recipe `transfer` conserves enthalpy by construction: the mix temperature solves H-equality on the elements datum (heat-capacity differences matter — hot ethanol into cold water lands $\sim$4 K above the molar average), falling back to the molar average only when a species cannot be priced, announced. Stills carry reboiler and condenser duty records (phase-enthalpy differences of the actual packages at their own temperatures; the rectifier's condenser handles the whole boilup V at the top-stage T). The campaign ENERGY BALANCE only claims a verdict when every piece is ledgered and priceable — otherwise the run prints `energy balance UNAVAILABLE` naming each missing piece (a temperature jump, a fallen-back charge, a species without formation data). KPIs land under `campaign` as `energy_*`; see `batch08_isothermal_duty`, `recipe04_charge_enthalpy` and `recipe01_react_then_distill` (the react-then-distil showcase closes end to end at 10^{-16}). Valid duties are then allocated to catalogue UTILITIES by temperature level (the still's reboiler draws LP steam by its latent heat; a 293 K crystalliser needs chilled water, not cooling water), with campaign totals under `utility_*` — energy, mass and cost per utility; a duty no utility can serve is listed, never dropped.

10 Dynamic simulation and control: choupoCtrl

What you'll learn. How the control binary integrates a continuously fed plant while controllers write manipulated variables — and why the controller sample time and the integrator step are two different things.

choupoCtrl integrates open vessels with continuous feeds — $\mathrm{d}Y/\mathrm{d}t = f(Y, u, t)$ — with a controllers layer writing the manipulated variables u at each sample. In controlDict, deltaT is the controller *sample time*; with timeStepping adaptive; the plant is sub-stepped between samples by the stiff integrator while the manipulated variable is held — two time steps, both correct (ctrl03_adaptive_disturbance).

10.1 dynamicCSTR

```
{ name reactor;  type dynamicCSTR;
  initial { T 320 K;  P 1.013 bar;  V 0.001;  totalMoles 0.0185;
            molarComposition { compA 1.0; } }
  feed    { F 5e-5 kmol/s;  T 320 K;  composition { compA 1.0; } }
  operation { reactions ( a_to_b );  jacketUA 30 W/K; } }
```

Constant-volume continuous CSTR with an optional jacket ($UA\,(T_j - T)$); RK4 on the packed (n_i, T). Its measured (CV) and manipulated (MV) variables are string-keyed so controllers bind by name. Kinetics: Arrhenius or LHHW (the shared rate law — ctrl07_lhhw_inhibition closes a temperature loop on a reaction that poisons itself, the exotherm fading as the product covers the catalyst). Add start steadyState; inside initial and the engine relaxes the holdup ODE to its fixed point at the declared feed and prints the seeded state — a clean start at the operating point.

10.2 Controllers

```
controllers
(
    {
        name  TC1;
        type  PID;
        measurement { unit reactor;  cv T_reactor; }
        actuator    { unit reactor;  mv T_jacket; }
        setpoint    350 K;
        Kp  4.0;  Ki  0.04;  Kd  0.0;
        bias  320;  uLow  280;  uHigh  420;
    }
    {
        name  FeedDist;
        type  Signal;                                  // open-loop forcing
        actuator { unit reactor;  mv T_in; }
        signal   { type sinusoidal;  mean 320 K;  amplitude 15 K;  period 600 s; }
    }
);
```

Three controller types: PID (clamped anti-windup, derivative-on-PV), Schedule (open-loop piecewise-constant steps — the disturbance injector of ctrl02_disturbance_rejection), and Signal — the full forcing-function vocabulary (step, staircase, ramp, pulse, sine, prbs). Inject a sinusoid on an inlet and watch the loop's forced response, attenuated and phase-lagged; sweep the period to build a Bode plot one point at a time (ctrl06_sine_disturbance). For identification experiments, type prbs; generates the classic broadband binary sequence from a declared-seed maximal-length LFSR — deterministic and exactly repeatable, with the period verified at load and the leading bits printed (ctrl08_prbs_ident): signal { type prbs; mean 320 K; amplitude 10 K; bitPeriod 120; registers 4; seed 1; }.

Disturbances can target the unit's INLET FACE by field name — actuator { unit reactor; inletField T; } (or F, moleFraction.<c>, componentMolarFlow.<c>) — the only legal disturbance surface. The inlet's authoritative state is the per-component molar-flow vector: componentMolarFlow.<c> writes one component's flow and re-derives F and z (the disturbance never hides from the other components), while moleFraction.<c> holds the total flow and renormalises the others proportionally — two explicit semantics, each announced at start-up, both validated hard (ctrl09_stream_disturbance pins the derived pair to hand arithmetic). The trajectory carries F_in and z_in_* as

first-class columns. The trajectory CSV carries every CV, MV and setpoint — plot it and read the tuning off the page.

11 The property workbench: choupoProps

What you'll learn. How to interrogate the thermodynamic foundation *before* wiring a flowsheet: single points, 1-D/2-D/ternary scans, data-consistency tests, and parameter fits — all from a `propsDict`.

A props case replaces `flowsheetDict` with a `system/propsDict` carrying `operations (...)`:

type	What it does
`propertyPoint`	one state, properties to stdout
`propertyScan1D`	sweep one variable, long-form CSV ($P^{\mathrm{sat}}(T)$, $\gamma(x)$, ...)
`propertyScan2D`	a (T, P) or composition grid (Z, residuals, ...)
`propertyScanTernary`	walk the ternary simplex at fixed (T, P), classify the phase regime per node (the same VLLE flash kernel as the unit op), collect tie-lines
`fitParameters`	Levenberg–Marquardt regression of any dotted package parameter against data, with identifiability statistics
`estimateComponent`	create a component from molecular groups (Joback + corresponding-states closures), shown against reference data
`vleConsistency`	thermodynamic consistency tests on VLE data (Herington area + Gibbs–Duhem point test)

Two flavours to give the taste:

```
{ name psat;  type propertyScan1D;
  vary { variable T;  from 250 K;  to 400 K;  n 51; }
  properties ( Psat_ethanol  Psat_water );
  output { file psat.csv; } }

{ name fit_nrtl;  type fitParameters;
  parameters
  (
      { path activityModel.pairs[0].a_ij;  initial -0.8;  min -15;  max 15; }
      { path activityModel.pairs[0].b_ij;  initial 246;   min -5000; max 5000; }
  );
  residual
  {
      kind  T_bubble;   P 1.01325 bar;
      data ( 0.05 361.95   0.10 359.95   0.20 358.05  ... );
  } }
```

The fit writes `fit_history.csv` and `parity.csv` and reports which parameters the data can actually identify — the honest part most fitting exercises skip. A fitted pair is written as a dated *proposal* under the case's `constant/`; promoting it into the frozen standard catalogue is a deliberate human act (the engine refuses to write under `data/standards/`).

This is the tip of a much larger workbench — component estimation from groups, model-vs-data overlays, electrolyte activity models (Pitzer, eNRTL), steam-table verification, speciation and scaling indices. The *Properties Guide* (`docs/propsGuide.pdf`) and the *Explorer Guide* cover it in depth; start with `tutorials/props/README.md` and the worked cases under `tutorials/props/`.

12 Tour of the tutorials

What you'll learn. Choupo ships about 200 tutorials under `tutorials/`, in six categories: `steady/`, `batch/`, `ctrl/`, `props/`, `plant/`, `electrochem/`. `listCases` tabulates them all with one-line descriptions. The selection below is the on-ramp — pick the one closest to what you want to do and edit from there.

Tutorial	What you learn
steady/flash, steady/thermoTest	
`flash01_benzene_toluene`	First run: Raoult VLE, Rachford–Rice, Newton trace, the `0/` state files.
`flash02_ethanol_water`	NRTL + Wegstein outer loop.
`flash06_sweep_T` / `flash07_gridsweep_TP`	One knob → a curve; two knobs → the flash envelope.
`flash08_co2_water_package`	The Henry dilute-solution world, declared as a package manifest.
`flash09_n2ch4_stryjek`	The φ–φ world: one cubic, both phases, declared k_{ij}.
`vlle03_audit_artificial`	Three-phase VLLE by Gibbs minimisation, on a constructed global minimum.
steady/reactors, steady/gibbs	
`cstr01_first_order` / `pfr01_first_order`	Newton on the extent; RK4 by hand vs the analytic $X = 1 - e^{-k\tau}$.
`cstr03` / `pfr03_series_selectivity`	Series selectivity: back-mixing eats the intermediate (53 % vs 79 %).
`cstr04_adiabatic` / `cstr05_multiplicity`	Non-isothermal CSTR; three steady states, all printed.
`cstr07_lhhw_methylAcetate`	A published LHHW law on UNIQUAC activities (§5.3.5).
`conversion01_hda`	Specified per-pass conversion + recycle — the honest gas-phase reactor.
`equil01_reforming`	Reforming + shift driven to simultaneous $K_p(T)$ equilibrium.
`gibbs01_water_gas_shift` / `gibbs05_adiabatic_flame`	Gibbs minimisation; the flame temperature as a DesignSpec.
steady/distillation, absorption	
`column01_benzene_toluene`	Wang–Henke bubble-point column.
`column03_azeotrope_mesh`	The simultaneous MESH through the ethanol/water azeotrope.
`column05_reactive_methylacetate`	Reactive distillation, paper-validated (§5.6.2).
`column09` / `column10` / `column11`	Tray hydraulics design; the flooded twin; Murphree trays.
`shortcut01_benzene_toluene`	Fenske–Underwood–Gilliland sizing.
`absorber01_NH3_water` / `extract01_ethanol_water_benzene`	Henry absorber; staged LL extraction.

Tutorial	What you learn
steady/heat, power, rotating, hydraulics	
`hx01_heater` / `heatExchanger02_geometry_U`	The duty device; U computed from the bundle geometry.
`hxWorkflow1` / `hxWorkflow2`	Design from a duty target, then rate the designed bundle.
`condenser01_film_nusselt` / `reboiler_water_copper`	Duty emerging from film condensation / nucleate boiling (CHF guard).
`rankine02_water` / `combined02_brayton_rankine_shaft`	IF97 Rankine cycle; the shaft-split combined cycle.
`compressor01_air`	$W_{\text{shaft}} + \eta$; P_{out} a result.
`pipe02_airwater_twophase`	Lockhart–Martinelli two-phase pressure drop.
steady/membranes, separation, solids, drying, crystallisation	
`membrane01_RO_NaCl_seawater`	The canonical seawater-RO element.
`membrane07_scaling_si`	Speciation × membrane: wall scaling three modules ahead of the bulk.
`psa01_h2_psa`	Equilibrium-theory pressure-swing adsorption.
`cyclone01_dust_removal`	PSD-aware cyclone.
`crystalliser01_sugar` / `crystalliser02_msmpr`	Cooling equilibrium yield; the MSMPR population balance.
`sprayDryer01_sugar`	Atomisation + droplet drying to a real particle size.
steady/flowsheets, optimisation, evaporation	
`process01_reactor_flash` / `process03_recycle`	First flowsheet; the tear + Newton-on-tears recycle.
`sensitivity01_column_reflux`	The reflux sweep to CSV.
`designSpec01_triple_equal_areas`	Two knobs, two specs on a triple-effect evaporator train.
`optim01_column_reflux` / `optim04_membrane_recovery_scaling`	Nelder–Mead; constrained SQP with shadow prices.

Tutorial	What you learn
batch/, ctrl/, electrochem/, plant/	
`batch01_first_order` / `batch02_adiabatic`	RK4 vs the analytic decay; thermal runaway.
`still01` / `still04` / `still05`	Rayleigh; constant-R rectifier; constant-x_D policy at the reflux ceiling.
`recipe01_react_then_distill`	A scripted transfer between vessels.
`ctrl01_cstr_temp_control` / `ctrl02_disturbance_rejection`	PID tracking; step-disturbance rejection.
`ed01_nacl_desalination`	The electrodialysis stack, below and at the limiting current.
`plant/lithiumBrinePlant`	A fractal multi-sector plant across four thermodynamic worlds.

Every tutorial's header comment states what it demonstrates and, where applicable, the published number it reproduces. The *Tutorials Guide* (`docs/tutorialsGuide.pdf`) is the long-form companion, one chapter per family.

13 Output anatomy

What you'll learn. What every section of a run's log means, and why the inner iteration trace is the part you should read.

A typical steady run with `verbosity 3` prints, in order:

1. **Banner** — version + licence.
2. **Case header** — case dir, database root, the property package that was found, which optional dicts are present, and the `[overlay]`/`[estimate]` announcements for any case-local component data.
3. **State seeding** — `[state] seeded N stream(s) from 0/` and the completeness check.
4. **Topology** — the source streams and the per-unit wiring.
5. **Per-unit run** — for each unit operation:
   - the thermo provenance line (`thermo: inherited (global package)` or the local override);
   - inputs echo ($F, T, P, \mathbf{z}$);
   - the inner iteration trace (Newton steps with $g(x)$, $\mathrm{d}g/\mathrm{d}x$, Δ);
   - for γ-models, the outer composition loop;
   - the result block (regime, V/F, $\mathbf{x}$, $\mathbf{y}$, $\mathbf{K}$, duty).
6. **Recycle loop** (if tears exist) — per-iteration tear residuals, Newton or Wegstein steps, announced convergence aids.
7. **Reports** — one `[report]` line per file written.
8. **State writing** — `[state] wrote converged/`.
9. **Structured result** — a JSON block between `«<Choupo:result-begin»>` markers carrying every stream, KPI, convergence history and advisory (this is what the GUI and scripts read).

An excerpt of the part that matters, from the first tutorial:

```
[outer 0] Rachford-Rice Newton (inner):
  it           V            g(V)          dg/dV          dV
  ----  -----------  -------------  -------------  -------------
  0     0.50000000   -3.75806e-02   -1.87920e-01   -1.99981e-01
  1     0.30001863   7.81978e-04    -1.97366e-01   3.96206e-03
  2     0.30398070   5.66359e-07    -1.97081e-01   2.87374e-06
  3     0.30398357   2.94959e-13    -1.97081e-01   0.00000e+00
```

Key idea. The inner iteration trace is where the simulator stops being a black box. The residual column tells you whether Newton is converging quadratically (the exponent doubling per line, as above) or linearly (the problem is ill-conditioned) — diagnosis you can do by eye.

Batch and control runs additionally write `trajectory.csv` on the `writeInterval` grid, and print an epilogue of headline events (conversions reached, recipe actions fired, reflux limits hit, ignition delay where relevant).

14 The web GUI: a runner and visualiser

> *What you'll learn.* How to open a case in the browser, run it, and read the result — and why the GUI never edits your dicts.

Choupo ships a browser front-end under `gui/`. Its design stance is deliberate and worth stating up front: **the GUI is a runner and visualiser, not an editor.** Cases are authored as plain-text dicts on disk, edited with any text editor (increasingly with LLM help). The GUI loads those dicts, draws the flowsheet, runs the solver *in the browser* (compiled to WebAssembly), and renders the log + plots. The files on disk remain the single source of truth. The architectural inspiration is ParaView — a text-defined case plus a viewer — not a block-diagram editor.

14.1 Starting the GUI

```
make wasm-gui          # build the WASM solver (choupoSolve + choupoProps)
bin/runGui             # boots the dev server and opens a browser window
bin/runGui --kill      # stops everything
```

14.2 The layout and interactions

The shell is a thin top bar over a full-screen flowsheet canvas — no left tree, no docked panels. Auxiliary views (log, plots, stream table, thermo foundation) open on demand as workspaces.

- **Single-click** a node or edge — raises a floating, read-only selection card (labels, units, help tooltips).
- **Double-click** — pops the unit out, or drills into a composite sector (the drilled sub-case is seeded from the parent's persisted `converged/` state).
- **Drag** nodes and edge bends to lay the flowsheet out; the layout persists per case in the browser's `localStorage`.
- **Stream-class toggles** — show/hide whole classes of streams (e.g. energy wires) to declutter.
- **Run** — streams the same log you would see from `runCase`; **Stop** cancels mid-solve.

Stream colour encodes phase and comes straight from the solver (vf plus solid mass fraction); it never changes on hover or selection — selection is signalled by a halo, not a recolour.

> **Common pitfall.** **MOCK vs WASM.** If the WASM solver has not been built (`make wasm-gui`), the GUI falls back to a *mock* adapter that returns canned numbers so the UI still renders. The Run badge reads `MOCK` in that case, not `WASM` — check it before trusting any result. After changing any C++ source, rebuild the WASM (it is a separate artefact from the native binary) or the browser keeps running stale code.

15 Building your own case

> *What you'll learn.* The fastest path from a blank idea to a converged result: scaffold or clone, wire the topology, author the inlets, initialise, run.

Two starting points:

```
# (a) scaffold a fresh case in your workspace:
newCase myFlash "isothermal flash of my mixture"

# (b) clone the tutorial closest to your idea:
cp -r tutorials/steady/flash/flash01_benzene_toluene tutorials/steady/flash/myFlash
```

Then the loop is always the same:

1. edit `system/flowsheetDict` (topology) and `constant/thermoPhysPropDict` (components + models);
2. author the *inlet* files in `0/` (and a seed for any tear);
3. `bin/choupo-init0 <case>` to materialise the remaining stream files;
4. `runCase <case>`, read the log, iterate.

Adding a missing component. Two locations — pick the right tier for what you are adding:

- `data/standards/components/<name>.dat` — a contribution to the *shared catalogue* of well-curated chemicals (primary-cited, per-value provenance; the engine refuses to write here — promotion is a human act).
- `<case>/constant/components/<name>.dat` — *sample-specific or one-off data*: a particular powder's sorption isotherm, a laboratory measurement, a NASA-7 C_p override. Case-local files are overlaid block-by-block on the standard entry of the same name, so a slim file with just the bespoke blocks is enough — and every overlay is announced in the log (`[overlay] ...`).

When a new file carries a `standardThermochemistry` block, declare the `phase` of the tabulated Δh_f° explicitly (`gas` / `liquid` / `solid`) — it tells the solver which rung of the enthalpy ladder the datum pins (Theory Guide § "Which rung does the data file pin?"). A component with no catalogue entry at all can be *estimated* from its molecular groups on the props bench (§11) and promoted as a dated, glass-box proposal.

Using a reaction. Declare it in `constant/reactions` and reference it by name (`reaction <name>;` or `reactions (...)`) — §5.3.

Custom unit operations. A case may carry its own C++ unit under `<case>/code/` (`MyUnit.H/.cpp` + an explicit registration file); `bin/buildCode` compiles it into a per-case binary. See `tutorials/steady/userops/userOp01_yield_reactor` and the Developer Guide.

Archiving. Seal the case (`bin/choupo-import`, §4.10) so it runs without the installation catalogue — the right form for a thesis appendix or a paper's supplementary material.

16 Troubleshooting

What you'll learn. The failure modes that account for nearly every support question, and exactly what to do about each.

"0/ is incomplete" / missing or orphan stream file

The completeness contract (§3.4): every stream in the graph needs a file in `0/`, and every file must correspond to a graph stream. The error names the offending stream. Author the inlets, then `bin/choupo-init0 <case>` writes the rest; a leftover file from a renamed stream must be deleted (the validator refuses to guess which stream you meant).

"Component does not exist"

```
ERROR: Database: component file not found: .../components/foo.dat
```

The component name is misspelt, or the file is missing. Names are case-sensitive and resolved exactly — check `data/standards/components/` and your `constant/components/` overlay.

"moleFractions do not close"

Stream-state fractions must sum to 1 within 10^{-6} (§3.4); the error names the stream and the actual sum. This is deliberate — a state file is data, and data with a hole in it should stop the run, not be silently patched.

Dimension mismatch on a parameter

```
parameter 'P' expected dimensions [1 -1 -2 0 0] (Pa) but the dict
declared [1 0 0 0 0] (kg).
```

A wrong unit suffix (§3.7). Fix the suffix; do not strip it (raw numbers pass unchecked).

Build fails after pulling new code

```
make distclean && make all
```

Necessary when headers change incompatibly. If the GUI misbehaves after a source change, rebuild the WASM too (`make wasm-gui`) — it is a separate artefact.

Simulator runs but a stream shows NaN

Most common: an upstream unit was fed an empty stream ($F = 0$ emptying the composition vector), or Antoine was evaluated far outside its range (e.g. T so low the extrapolated P^{sat} turns non-physical). Re-read the input echo of the first unit whose output is NaN; `bin/runTests` guards every tutorial against silent NaN/inf for exactly this reason.

Newton fails to converge

- **Outer composition loop** (γ-model refresh): try `outerAccelerator directSubstitution` in `solverDict` for strongly non-ideal systems, or raise `maxOuterIter`.
- **Inner Newton** (Rachford–Rice, bubble-T): raise `maxIter`; near a phase boundary, tighten `compositionTol`.
- **Recycle**: give the tear a better `0/` seed (run `choupo-init0 -force` after improving the inlets), or switch `recycleSolver` between `Newton` and `Wegstein` — they have different failure modes. Every aid the solver applies (auto-seed, bound, fallback) is announced in the log — read what it did before overriding it.

Wang–Henke column oscillates or crosses an azeotrope

The bubble-point method is not stable on strongly non-ideal systems. Select `model simultaneous;` on the column — the whole-block Newton converges quadratically and refuses to invent a "past-the-azeotrope" answer (§5.6).

A sweep fails on some points

Almost always the solver's iteration caps are too small for the edge points (high reflux, high pressure). Raise the limits in `solverDict`; failed points are written as `nan` and the sweep continues, so the CSV tells you exactly where the envelope ends.

Key idea. If a case behaves unexpectedly, *read the log*. Every Newton step, every Wegstein update, every γ value, every announced default is printed. Black-box debugging is exactly the failure mode Choupo refuses to participate in.

17 Where to look next

Choupo's documentation is a family of guides, each with its own job:

- `docs/theoryGuide.pdf` — every equation the engine solves, derived: flashes, MESH, kinetics, enthalpy ladders, population balances, the numerical methods.
- `docs/tutorialsGuide.pdf` — the long-form walk through the tutorial corpus, family by family.
- `docs/propsGuide.pdf` — the property workbench in depth: estimation, fitting, consistency, electrolytes.
- `docs/explorerGuide.pdf` — the property-explorer view of the GUI.
- `docs/designGuide.pdf` — sizing, costing, and design heuristics.
- `docs/developerGuide.pdf` — adding unit operations, thermo models, components, outer drivers, post-processors.
- `tutorials/<category>/<family>/<name>/` — every tutorial is a fully-worked, runnable example with an explanatory header.
- `src/` — the ultimate reference. Every model documented here is a readable C++ class; start from `src/applications/choupoSolve/main.cpp` and follow the calls.

Colophon. This manual was typeset in Latin Modern with the Choupo shared LaTeX preamble. Sources live in `docs/`; rebuild with `cd docs && make user`.