Appendix C – Python Codebase Author: Saif-Aldain Aqel

This document contains the complete Python source code and configuration files used to generate all results and figures presented in this thesis. The code is provided verbatim for reproducibility and archival purposes.

== FILE: **init**.py ============================

== FILE: main.py ============================
import logging from common.logging_utils import setup_logging from common.units import Q_ from common.boiler_loop import run_boiler_case

log = logging.getLogger(**name**)

def run_default_case() -> None: run_boiler_case(run_id="default_case")

def run_excess_air_sensitivity() -> None: ea_values = [1.00, 1.05, 1.10, 1.15, 1.20, 1.30]

```
for ea in ea_values:
    logging.getLogger(__name__).info(f"Running case with excess_air_ratio={ea}")

    run_boiler_case(
        operation_overrides={"excess_air_ratio": Q_(ea, "")},
        tol_m=Q_(1e-3, "kg/s"),
        max_iter=20,
        write_csv=True,
        run_id=f"excess_air_{ea}",
    )
```

def run_water_pressure_sensitivity() -> None: Pbar_values = [4.0, 10.0, 16.0]

```
for P_bar in Pbar_values:
    logging.getLogger(__name__).info(f"Running case with drum pressure={P_bar} bar")

    run_boiler_case(
        operation_overrides={"drum_pressure": Q_(P_bar, "bar")},
        tol_m=Q_(1e-3, "kg/s"),
        max_iter=20,
        write_csv=True,
        run_id=f"drum_pressure_{P_bar}bar",
    )
```

def run_fuel_flow_sensitivity() -> None: mdot_values = [0.025, 0.050, 0.075, 0.10, 0.125] # kg/s

```
for mdot in mdot_values:
    logging.getLogger(__name__).info(f"Running case with fuel mass_flow={mdot} kg/s")

    run_boiler_case(
        fuel_overrides={"mass_flow": Q_(mdot, "kg/s")},
        tol_m=Q_(1e-3, "kg/s"),
        max_iter=20,
        write_csv=True,
        run_id=f"fuel_flow_{mdot}kgs",
    )
```

def run_fouling_sensitivity() -> None: factors = [1, 5, 10] for f in factors: logging.getLogger(**name**).info(f"Running case with fouling_factor={f}")

```
    run_boiler_case(
        fouling_factor=f,
        tol_m=Q_(1e-3, "kg/s"),
```

```
        max_iter=20,
        write_csv=True,
        run_id=f"fouling_{f}",
    )
```

def main() -> None: setup_logging("INFO")

```
log.info("start")

run_default_case()
# run_excess_air_sensitivity()
# run_water_pressure_sensitivity()
# run_fuel_flow_sensitivity()
# run_fouling_sensitivity()
log.info("end")
```

if **name** == "**main**": main()

== FILE: analysis/analyze_boiler_runs.py == from pathlib import Path import re import pandas as pd

RESULTS_DIR = Path("results/runs") SUMMARY_DIR = Path("results/summary")

FILE_RE = re.compile(r"(?P *]+)/default_case)*(?Pboiler_summary|stages_summary|steps).csv$")

def parse_param_value(raw): if raw is None: return None

```
for suffix in ("kgs", "bar"):
    if raw.endswith(suffix):
        raw = raw[: -len(suffix)]

try:
    return float(raw)
except ValueError:
    return raw
```

def discover_runs(results_dir: Path):

```
runs = {}

for path in results_dir.glob("*.csv"):
    m = FILE_RE.match(path.name)
    if not m:
        continue

    case = m.group("case")
    param = m.group("param")
    raw_value = m.group("value")
    kind = m.group("kind")

    info = runs.setdefault(
        case,
        {
            "case": case,
            "param": param if param is not None else "control",
            "value": parse_param_value(raw_value),
            "files": {},
        },
```

```
    )
    info["files"][kind] = path

return runs
```

def load_boiler_as_series(path: Path, run_name: str, param_group: str, param_value): df = pd.read_csv(path) s = df.set_index("parameter")["value"] s = s.copy() s["run"] = run_name s["param_group"] = param_group s["param_value"] = param_value return s

def load_stage_as_tidy(path: Path, run_name: str, param_group: str, param_value): df_raw = pd.read_csv(path, index_col=0) df = df_raw.T.reset_index().rename(columns={"index": "stage"}) df.insert(0, "run", run_name) df.insert(1, "param_group", param_group) df.insert(2, "param_value", param_value) return df

def load_steps(path: Path): return pd.read_csv(path)

def main(): RESULTS_DIR.mkdir(exist_ok=True) SUMMARY_DIR.mkdir(parents=True, exist_ok=True)

```
runs = discover_runs(RESULTS_DIR)

if not runs:
    print("[INFO] No runs discovered in 'results/' matching expected patterns.")
    return

boiler_rows = []
stage_rows = []

for case, info in runs.items():
    files = info["files"]
    run_name = info["case"]
    param_group = info["param"]
    param_value = info["value"]

    if "boiler_summary" in files:
        s = load_boiler_as_series(files["boiler_summary"], run_name, param_group, param_value)
        boiler_rows.append(s)
    else:
        print(f"[WARN] run {run_name}: boiler_summary file missing.")

    if "stages_summary" in files:
       df_stages = load_stage_as_tidy(files["stages_summary"], run_name, param_group, param_value)
        stage_rows.append(df_stages)
    else:
        print(f"[WARN] run {run_name}: stages_summary file missing.")

    if "steps" not in files:
        print(f"[WARN] run {run_name}: steps file missing.")

if boiler_rows:
    boiler_df = pd.DataFrame(boiler_rows)

    cols = list(boiler_df.columns)
    for key in ["run", "param_group", "param_value"]:
        if key in cols:
            cols.remove(key)
    ordered_cols = ["run", "param_group", "param_value"] + cols
```

```
    boiler_df = boiler_df[ordered_cols]

    boiler_df = boiler_df.sort_values(
        by=["param_group", "param_value"],
        ascending=[True, True],
        kind="mergesort",
        na_position="last",
    )

    boiler_df.to_csv(SUMMARY_DIR / "boiler_kpis_all_runs.csv", index=False)
   print(f"[INFO] Wrote boiler KPIs table (with deviations): {SUMMARY_DIR / 'boiler_kpis_all_runs.csv'}")
else:
    print("[INFO] No boiler KPI data collected.")

if stage_rows:
    non_empty_stage_rows = [df for df in stage_rows if not df.empty]

    if non_empty_stage_rows:
        stages_df = pd.concat(non_empty_stage_rows, ignore_index=True)

        stages_df = stages_df.sort_values(
            by=["param_group", "param_value"],
            ascending=[True, True],
            kind="mergesort",
            na_position="last",
        )

        stages_df.to_csv(SUMMARY_DIR / "stages_summary_all_runs.csv", index=False)
       print(f"[INFO] Wrote stages summary table: {SUMMARY_DIR / 'stages_summary_all_runs.csv'}")
    else:
        print("[INFO] Stage summary dataframes are all empty; nothing to write.")
else:
    print("[INFO] No stage summary data collected.")
```

if **name** == “**main**”: main()

== FILE: analysis/map.py ======================== from pathlib import Path import numpy as np import pandas as pd import matplotlib.pyplot as plt

HX_CSV = Path(r”results/summary/stages_summary_all_runs.csv”) OUTDIR = Path(r”results/plots/map”)

def ensure_outdir(path: Path) -> Path: path.mkdir(parents=True, exist_ok=True) return path

def rename_hx_columns(df: pd.DataFrame) -> pd.DataFrame: mapping = { “gas in pressure[pa]”: “p_gas_in”, “gas in temp[°C]”: “T_gas_in”, “gas in enthalpy[kJ/kg]”: “h_gas_in”, “gas out pressure[pa]”: “p_gas_out”, “gas out temp[°C]”: “T_gas_out”, “gas out enthalpy[kJ/kg]”: “h_gas_out”, “water pressure[pa]”: “p_water”, “water in temp[°C]”: “T_water_in”, “water in enthalpy[kJ/kg]”: “h_water_in”, “water out temp[°C]”: “T_water_out”, “water out enthalpy[kJ/kg]”: “h_water_out”, “gas avg velocity[m/s]”: “v_gas”, “water avg velocity[m/s]”: “v_water”, “pressure drop fric[pa]”: “dp_fric”, “pressure drop minor[pa]”: “dp_minor”, “pressure drop total[pa]”: “dp_total”, “Q conv[MW]”: “Q_conv”, “Q rad[MW]”: “Q_rad”, “Q total[MW]”: “Q_total”, “UA[MW/K]”: “UA”, “steam capacity[t/h]”: “steam_capacity”, } df = df.rename(columns={k: v for k, v in mapping.items() if k in df.columns})

```
if "stage" in df.columns:
    df["stage_index"] = (
        df["stage"]
        .astype(str)
```

```
        .str.extract(r"(\d+)", expand=False)
        .astype(int)
    )
return df
```

def __plot_single_heatmap(df: pd.DataFrame, value_col: str, outdir: Path, filename: str, title: str, cbar_label: str,) -> None: if "run" not in df.columns or "stage_index" not in df.columns: return if value_col not in df.columns: return

```
table = df.pivot_table(
    index="run",
    columns="stage_index",
    values=value_col,
    aggfunc="mean",
)

if table.empty:
    return

fig, ax = plt.subplots()
im = ax.imshow(table.values, aspect="auto")

ax.set_yticks(np.arange(len(table.index)))
ax.set_yticklabels(table.index)
ax.set_xticks(np.arange(len(table.columns)))
ax.set_xticklabels(table.columns)

ax.set_xlabel("HX stage index [-]")
ax.set_ylabel("Run")
ax.set_title(title)

fig.colorbar(im, ax=ax, label=cbar_label)
fig.tight_layout()
fig.savefig(outdir / filename, dpi=200)
plt.close(fig)
```

def plot_hx_heatmaps(hx: pd.DataFrame, outdir: Path) -> None: df = hx.copy() outdir = ensure_outdir(outdir) if df.empty: return

```
heatmaps = {
    "Q_total": (
        "heatmap_Q_total.png",
        "Heat duty per stage and run [MW]",
        "Q_total [MW]",
    ),
    "T_gas_out": (
        "heatmap_T_gas_out.png",
        "Gas outlet temperature per stage and run [°C]",
        "T_gas_out [°C]",
    ),
    "T_gas_in": (
        "heatmap_T_gas_in.png",
        "Gas inlet temperature per stage and run [°C]",
        "T_gas_in [°C]",
    ),
    "T_water_in": (
```

```
        "heatmap_T_water_in.png",
        "Water inlet temperature per stage and run [°C]",
        "T_water_in [°C]",
    ),
    "T_water_out": (
        "heatmap_T_water_out.png",
        "Water outlet temperature per stage and run [°C]",
        "T_water_out [°C]",
    ),
    "v_gas": (
        "heatmap_v_gas.png",
        "Gas velocity per stage and run [m/s]",
        "v_gas [m/s]",
    ),
    "v_water": (
        "heatmap_v_water.png",
        "Water velocity per stage and run [m/s]",
        "v_water [m/s]",
    ),
    "dp_total": (
        "heatmap_dp_total.png",
        "Total pressure drop per stage and run [Pa]",
        "Δp_total [Pa]",
    ),
    "Q_conv": (
        "heatmap_Q_conv.png",
        "Convective heat duty per stage and run [MW]",
        "Q_conv [MW]",
    ),
    "Q_rad": (
        "heatmap_Q_rad.png",
        "Radiative heat duty per stage and run [MW]",
        "Q_rad [MW]",
    ),
    "UA": (
        "heatmap_UA.png",
        "UA per stage and run [MW/K]",
        "UA [MW/K]",
    ),
    "steam_capacity": (
        "heatmap_steam_capacity.png",
        "Steam capacity per stage and run [t/h]",
        "Steam capacity [t/h]",
    ),
}

for col, (fname, title, cbar) in heatmaps.items():
    _plot_single_heatmap(df, col, outdir, fname, title, cbar)
```

def main(): ensure_outdir(OUTDIR) hx = pd.read_csv(HX_CSV) hx = rename_hx_columns(hx) plot_hx_heatmaps(hx, OUTDIR)

if **name** == "**main**": main()

====================================== FILE: analysis/per_run.py ===================

from pathlib import Path import sys from typing import Dict import pandas as pd import matplotlib.pyplot as plt from matplotlib.lines import Line2D

style_colors: Dict[str, str] = { “Q_in”: “tab:blue”, “Q_useful”: “tab:orange”, “eta_direct”: “tab:green”, “eta_indirect”: “tab:red”, “feedwater_flow”: “tab:blue”, “steam_capacity”: “tab:orange”, “stack_temperature”: “tab:red”, “pressure_drop”: “tab:purple”, }

style_linestyles: Dict[str, str] = { “Q_in”: “-”, “Q_useful”: “–”, “eta_direct”: “-”, “eta_indirect”: “–”, “feedwater_flow”: “-”, “steam_capacity”: “–”, “stack_temperature”: “-”, “pressure_drop”: “–”, }

style_markers: Dict[str, str] = { “excess_air”: “o”, “fuel_flow”: “s”, “drum_pressure”: “D”, “control”: “x”, “fouling”: “^”, }

param_xlabels: Dict[str, str] = { “excess_air”: “Excess air [-]”, “fuel_flow”: “Fuel flow [kg/s]”, “drum_pressure”: “Drum pressure [bar]”, “control”: “Control case [-]”, “fouling”: “Fouling factor [-]”, }

def _get_xlabel(param_group: str) -> str: return param_xlabels.get(param_group, “Parameter value [-]”)

param_group_style = { “excess_air”: dict(color=“tab:blue”, marker=“o”, linestyle=“-”), “fuel_flow”: dict(color=“tab:orange”, marker=“s”, linestyle=“–”), “drum_pressure”: dict(color=“tab:green”, marker=“D”, linestyle=“-”), “control”: dict(color=“tab:red”, marker=“x”, linestyle=“-”), “fouling”: dict(color=“tab:purple”, marker=“^”, linestyle=“-.”), }

def _get_pg_style(param_group: str) -> dict: return param_group_style.get(param_group, dict(color=“black”, marker=“o”, linestyle=“-”),)

plt.rcParams[“font.size”] = 11 plt.rcParams[“axes.titlesize”] = 12 plt.rcParams[“axes.labelsize”] = 11 plt.rcParams[“legend.fontsize”] = 9 plt.rcParams[“figure.dpi”] = 100 plt.rcParams[“font.family”] = “DejaVu Sans” plt.rcParams[“axes.grid”] = True plt.rcParams[“grid.linestyle”] = “:” plt.rcParams[“grid.linewidth”] = 0.5 plt.rcParams[“grid.alpha”] = 0.7

def load_data(csv_path: str) -> pd.DataFrame: csv_file = Path(csv_path) df = pd.read_csv(csv_file) df[“param_value”] = pd.to_numeric(df[“param_value”], errors=“coerce”) return df

def load_steps_data(csv_path: str) -> pd.DataFrame: csv_file = Path(csv_path) df = pd.read_csv(csv_file)

```
num_cols = [
    "x[m]",
    "gas_T[°C]", "water_T[°C]",
    "gas_P[kPa]", "water_P[kPa]",
    "gas_V[m/s]", "water_V[m/s]",
    "h_gas[W/m^2/K]", "h_water[W/m^2/K]",
]
for c in num_cols:
    if c in df.columns:
        df[c] = pd.to_numeric(df[c], errors="coerce")

return df
```

def _sweep_percent(series: pd.Series) -> pd.Series: s = pd.to_numeric(series, errors=“coerce”) smin = s.min() smax = s.max() if pd.isna(smin) or pd.isna(smax) or smax == smin: return pd.Series([0.0] * len(s), index=s.index) return (s - smin) / (smax - smin) * 100.0

def _get_marker(param_group: str): return style_markers.get(param_group, None)

def plot_Qin_Quseful(ax, df_group: pd.DataFrame, param_group: str) -> None: df_plot = df_group.dropna(subset=[“param_value”]).sort_values(“param_value”)

```
marker = _get_marker(param_group)

x = df_plot["param_value"]
```

```
y_qin = df_plot["Q_in total[MW]"]
y_quse = df_plot["Q_useful[MW]"]

line_qin, = ax.plot(
    x,
    y_qin,
    label="Total heat input $Q_{in}$",
    color=style_colors.get("Q_in", None),
    linestyle=style_linestyles.get("Q_in", "-"),
    marker=marker,
)
line_quse, = ax.plot(
    x,
    y_quse,
    label="Useful heat $Q_{useful}$",
    color=style_colors.get("Q_useful", None),
    linestyle=style_linestyles.get("Q_useful", "--"),
    marker=marker,
)

ax.set_xlabel(_get_xlabel(param_group))
ax.set_ylabel("Heat rate [MW]")
ax.set_title("Heat rates")
ax.grid(True, which="both")
ax.legend(loc="best", framealpha=0.8)
```

def plot_eta(ax, df_group: pd.DataFrame, param_group: str) -> None: df_plot = df_group.dropna(subset=["param_value"

```
marker = _get_marker(param_group)

x = df_plot["param_value"]
y_eta_dir = df_plot["eta direct[-]"]
y_eta_ind = df_plot["eta indirect[-]"]

line_eta_dir, = ax.plot(
    x,
    y_eta_dir,
    label="Direct efficiency",
    color=style_colors.get("eta_direct", None),
    linestyle=style_linestyles.get("eta_direct", "-"),
    marker=marker,
)
line_eta_ind, = ax.plot(
    x,
    y_eta_ind,
    label="Indirect efficiency",
    color=style_colors.get("eta_indirect", None),
    linestyle=style_linestyles.get("eta_indirect", "--"),
    marker=marker,
)

ax.set_xlabel(_get_xlabel(param_group))
ax.set_ylabel("Efficiency [-]")
ax.set_title("Boiler efficiency")
ax.grid(True, which="both")
```

```
ax.legend(loc="best", framealpha=0.8)
```

def plot_water_steam(ax, df_group: pd.DataFrame, param_group: str) -> None: df_plot = df_group.dropna(subset=[“param_value”]).sort_values(“param_value”) marker = _get_marker(param_group)

```
x = df_plot["param_value"]
y_water = df_plot["feedwater flow[kg/s]"]
y_steam = df_plot["steam capacity[t/h]"]

line_water, = ax.plot(
    x,
    y_water,
    label="feedwater_flow",
    color=style_colors.get("feedwater_flow", None),
    linestyle=style_linestyles.get("feedwater_flow", "-"),
    marker=marker,
)
ax.set_xlabel(_get_xlabel(param_group))
ax.set_ylabel("feedwater flow [kg/s]")
ax.set_title("Water and steam")

ax2 = ax.twinx()
line_steam, = ax2.plot(
    x,
    y_steam,
    label="Steam capacity",
    color=style_colors.get("steam_capacity", None),
    linestyle=style_linestyles.get("steam_capacity", "--"),
    marker=marker,
)
ax2.set_ylabel("Steam capacity [t/h]")

ax.grid(True, which="both")

lines = [line_water, line_steam]
labels = [l.get_label() for l in lines]
ax.legend(lines, labels, loc="best", framealpha=0.8)
```

def plot_stack_pressure(ax, df_group: pd.DataFrame, param_group: str) -> None: df_plot = df_group.dropna(subset=[“param_value”]).sort_values(“param_value”) marker = _get_marker(param_group)

```
x = df_plot["param_value"]
y_stack = df_plot["stack temperature[°C]"]
y_dp = df_plot["pressure drop total[kPa]"].abs()

line_stack, = ax.plot(
    x,
    y_stack,
    label="Stack temperature",
    color=style_colors.get("stack_temperature", None),
    linestyle=style_linestyles.get("stack_temperature", "-"),
    marker=marker,
)
ax.set_xlabel(_get_xlabel(param_group))
ax.set_ylabel("Stack temperature [°C]")
ax.set_title("Stack and pressure drop")
```

```
ax2 = ax.twinx()
line_dp, = ax2.plot(
    x,
    y_dp,
    label="Total pressure drop",
    color=style_colors.get("pressure_drop", None),
    linestyle=style_linestyles.get("pressure_drop", "--"),
    marker=marker,
)
ax2.set_ylabel("Total pressure drop [Pa]")

ax.grid(True, which="both")

lines = [line_stack, line_dp]
labels = [l.get_label() for l in lines]
ax.legend(lines, labels, loc="best", framealpha=0.8)
```

def generate_overall_kpi_figure(csv_path: str, output_dir: str = "figures") -> None: df = load_data(csv_path) out_dir = Path(output_dir) out_dir.mkdir(parents=True, exist_ok=True)

```
df = df.dropna(subset=["param_value"]).copy()
df_all = df.copy()
kpi_defs = [
    {"column": "Tad[°C]",                 "ylabel": "Tad [°C]"},
    {"column": "stack temperature[°C]",   "ylabel": "Stack temperature [°C]"},
    {"column": "air flow[kg/s]",          "ylabel": "Air flow [kg/s]"},
    {"column": "feedwater flow[kg/s]",        "ylabel": "feedwater flow [kg/s]"},
    {"column": "Q_in total[MW]",          "ylabel": "Heat input $Q_{in}$ [MW]"},
    {"column": "steam capacity[t/h]",     "ylabel": "Steam capacity [t/h]"},
    {"column": "pressure drop total[kPa]", "ylabel": "Total pressure drop [kPa]", "abs": True},
    {"column": "eta direct[-]",           "ylabel": "Direct efficiency [-]"},
]

fig, axes = plt.subplots(4, 2, figsize=(9, 10))
axes_flat = axes.flatten()

for ax in axes_flat:
    ax.set_xlim(0, 100)
    ax.set_xticks([0, 100])
    ax.set_xticklabels(["Min", "Max"])

for ax, kpi in zip(axes_flat, kpi_defs):
    ax.set_xlabel("Parameter range")
    ax.set_ylabel(kpi["ylabel"])
    ax.grid(True, which="both")

legend_handles = {}

for param_group, df_group in df.groupby("param_group"):
    df_group_sorted = df_group.sort_values("param_value")
    x = _sweep_percent(df_group_sorted["param_value"])

    pg_style = _get_pg_style(param_group)
    marker = pg_style["marker"]
```

```
    color = pg_style["color"]
    line_style = pg_style["linestyle"]

    for ax, kpi in zip(axes_flat, kpi_defs):
        col = kpi["column"]
        if col not in df_group_sorted.columns:
            continue

        y_raw = df_group_sorted[col]
        y = y_raw.abs() if kpi.get("abs", False) else y_raw

        line, = ax.plot(
            x, y,
            marker=marker,
            linestyle=line_style,
            color=color,
            label=param_group,
        )

        if param_group not in legend_handles:
            legend_handles[param_group] = line

for ax, kpi in zip(axes_flat, kpi_defs):
    col = kpi["column"]
    if col not in df_all.columns:
        continue

    df_overlay = df_all.dropna(subset=["param_value", col]).copy()
    if kpi.get("abs", False):
        df_overlay[col] = df_overlay[col].abs()

if legend_handles:
    fig.legend(
        handles=list(legend_handles.values()),
        labels=list(legend_handles.keys()),
        loc="lower center",
        ncol=min(len(legend_handles), 4),
        framealpha=0.8,
        bbox_to_anchor=(0.5, 0.02),
    )

fig.tight_layout(rect=(0.0, 0.05, 1.0, 1.0))

out_path = out_dir / "kpi_overview_all_param_groups.png"
fig.savefig(out_path, dpi=300)
plt.close(fig)
```

def generate_eff_stack_scatter(csv_path: str = "results/summary/boiler_kpis_all_runs.csv", output_dir: str = "figures",) -> None: df = load_data(csv_path) out_dir = Path(output_dir) out_dir.mkdir(parents=True, exist_ok=True)

```
needed = ["param_group", "stack temperature[°C]", "eta direct[-]", "eta indirect[-]"]
missing = [c for c in needed if c not in df.columns]
if missing:
    raise KeyError(f"Missing columns for scatter: {missing}")
```

```
dfp = df.dropna(subset=["stack temperature[°C]", "eta direct[-]", "eta indirect[-]"]).copy()

fig, ax_dir = plt.subplots(1, 1, figsize=(7.5, 4))

df_non_control = dfp[dfp["param_group"].astype(str).str.lower() != "control"].copy()
df_control     = dfp[dfp["param_group"].astype(str).str.lower() == "control"].copy()

for pg, dpg in df_non_control.groupby("param_group"):
    st = _get_pg_style(str(pg))
    ax_dir.scatter(
        dpg["stack temperature[°C]"],
        dpg["eta direct[-]"],
        label=str(pg),
        color=st["color"],
        marker=st["marker"],
        s=28,
        alpha=0.85,
        zorder=2,
    )

if not df_control.empty:
    stc = _get_pg_style("control")
    ax_dir.scatter(
        df_control["stack temperature[°C]"],
        df_control["eta direct[-]"],
        label="control",
        color=stc["color"],
        marker=stc["marker"],
        s=60,
        alpha=1.0,
        linewidths=0.6,
        zorder=10,
    )

ax_dir.set_title("Direct efficiency vs stack temperature")
ax_dir.set_xlabel("Stack temperature [°C]")
ax_dir.set_ylabel("Direct efficiency [-] (LHV)")
ax_dir.grid(True, which="both")

handles, labels = ax_dir.get_legend_handles_labels()
if handles:
    ax_dir.legend(
        handles=handles,
        labels=labels,
        loc="best",
        framealpha=0.8,
    )

fig.tight_layout()

out_path = out_dir / "scatter_efficiency_vs_stack_temperature_all_runs.png"
fig.savefig(out_path, dpi=300, bbox_inches="tight")
plt.close(fig)
```

def generate_stage_combined_control_figure(csv_path: str = "results/summary/stages_summary_all_runs.csv", output_dir: str = "figures",) -> None: df = load_data(csv_path) out_dir = Path(output_dir) out_dir.mkdir(parents=True, exist_ok=True)

```
df = df[df["param_group"].astype(str).str.lower() == "control"].copy()
if df.empty:
    raise ValueError("No rows found for param_group == 'control' in stages summary CSV.")

if df["stage"].dtype == object:
    df["stage_index"] = (
        df["stage"].astype(str).str.extract(r"(\d+)", expand=False).astype(int)
    )
else:
    df["stage_index"] = df["stage"].astype(int)

df = df.dropna(subset=["stage_index"]).copy()

plot_cols = [
    "gas out temp[°C]",
    "Q total[MW]",
    "Q rad[MW]",
    "Q conv[MW]",
    "UA[MW/K]",
    "gas out pressure[kpa]",
    "pressure drop total[kpa]",
    "gas avg velocity[m/s]",
]
for c in plot_cols:
    if c in df.columns:
        df[c] = pd.to_numeric(df[c], errors="coerce")

fig, axes = plt.subplots(4, 2, figsize=(10, 10))
ax_Tg   = axes[0, 0]
ax_Q    = axes[0, 1]
ax_Qrad = axes[1, 0]
ax_Qconv= axes[1, 1]
ax_UA   = axes[2, 0]
ax_p    = axes[2, 1]
ax_dp   = axes[3, 0]
ax_vel  = axes[3, 1]

stage_ticks = sorted(df["stage_index"].unique())
for ax in (ax_Tg, ax_Q, ax_Qrad, ax_Qconv, ax_UA, ax_p, ax_dp, ax_vel):
    ax.set_xlabel("Stage [-]")
    ax.set_xticks(stage_ticks)
    ax.grid(True, which="both")

ax_Tg.set_ylabel("Gas outlet temperature [°C]")
ax_Q.set_ylabel("Stage duty $Q_{\\mathrm{stage}}$ [MW]")
ax_Qrad.set_ylabel("$Q_{\\mathrm{rad}}$ [MW]")
ax_Qconv.set_ylabel("$Q_{\\mathrm{conv}}$ [MW]")
ax_UA.set_ylabel("Stage conductance $UA$ [MW/K]")
ax_p.set_ylabel("Gas outlet pressure [kPa]")
ax_dp.set_ylabel("Total pressure drop [kPa]")
```

```
ax_vel.set_ylabel("Gas average velocity [m/s]")

st = _get_pg_style("control")
color = st["color"]
marker = st["marker"]
line_style = st["linestyle"]
lw = 0.9

for run_name, df_run in df.groupby("run"):
    df_run = df_run.sort_values("stage_index")

    x = df_run["stage_index"]
    y_Tg   = df_run["gas out temp[°C]"]
    y_Q    = df_run["Q total[MW]"]
    y_Qrad = df_run["Q rad[MW]"]
    y_Qconv= df_run["Q conv[MW]"]
    y_UA   = df_run["UA[MW/K]"]
    y_p    = df_run["gas out pressure[kpa]"]
    y_dp   = df_run["pressure drop total[kpa]"].abs()
    y_vel  = df_run["gas avg velocity[m/s]"]

   ax_Tg.plot(x, y_Tg,  marker=marker, linestyle=line_style, color=color, linewidth=lw, label="Control case")
    ax_Q.plot(x, y_Q,     marker=marker, linestyle=line_style, color=color, linewidth=lw)
    ax_Qrad.plot(x, y_Qrad, marker=marker, linestyle=line_style, color=color, linewidth=lw)
    ax_Qconv.plot(x, y_Qconv, marker=marker, linestyle=line_style, color=color, linewidth=lw)
    ax_UA.plot(x, y_UA,   marker=marker, linestyle=line_style, color=color, linewidth=lw)
    ax_p.plot(x, y_p,     marker=marker, linestyle=line_style, color=color, linewidth=lw)
    ax_dp.plot(x, y_dp,   marker=marker, linestyle=line_style, color=color, linewidth=lw)
    ax_vel.plot(x, y_vel, marker=marker, linestyle=line_style, color=color, linewidth=lw)

handles, labels = ax_Tg.get_legend_handles_labels()
if handles:
    fig.legend(
        handles=handles,
        labels=labels,
        loc="lower center",
        ncol=min(len(labels), 4),
        framealpha=0.8,
        bbox_to_anchor=(0.5, 0.02),
    )

fig.tight_layout(rect=(0.0, 0.06, 1.0, 0.98))

out_path = out_dir / "stages_control_combined_8plots.png"
fig.savefig(out_path, dpi=300)
plt.close(fig)
```

def generate_all_figures(csv_path: str, output_dir: str = “figures”) -> None:

```
df = load_data(csv_path)
out_dir = Path(output_dir)
out_dir.mkdir(parents=True, exist_ok=True)

grouped = df.groupby("param_group")
```

```
for param_group, df_group in grouped:
    df_group_numeric = df_group.dropna(subset=["param_value"])
    if df_group_numeric["param_value"].nunique(dropna=True) < 2:
        continue

    fig, axes = plt.subplots(2, 2, figsize=(8, 6))
    ax_tl = axes[0, 0]
    ax_tr = axes[0, 1]
    ax_bl = axes[1, 0]
    ax_br = axes[1, 1]

    plot_Qin_Quseful(ax_tl, df_group_numeric, param_group)
    plot_eta(ax_tr, df_group_numeric, param_group)
    plot_water_steam(ax_bl, df_group_numeric, param_group)
    plot_stack_pressure(ax_br, df_group_numeric, param_group)

    fig.tight_layout()

    safe_group = str(param_group).replace(" ", "_")
    png_path = out_dir / f"performance_{safe_group}.png"

    fig.savefig(png_path, dpi=300)
    plt.close(fig)
```

if **name** == "**main**": default_csv = "results/summary/boiler_kpis_all_runs.csv" default_output = "results/plots/per_run"

```
if len(sys.argv) >= 2:
    csv_arg = sys.argv[1]
else:
    csv_arg = default_csv

if len(sys.argv) >= 3:
    out_arg = sys.argv[2]
else:
    out_arg = default_output

generate_all_figures(csv_arg, output_dir=out_arg)

generate_overall_kpi_figure(csv_arg, output_dir=out_arg)

generate_eff_stack_scatter(csv_arg, output_dir=out_arg)

stage_csv = "results/summary/stages_summary_all_runs.csv"
generate_stage_combined_control_figure(stage_csv, output_dir=out_arg)
```

== FILE: analysis/tables.py ========================
import os import pandas as pd

INPUT_CSV = os.path.join("results", "summary", "boiler_kpis_all_runs.csv") OUTPUT_MD = os.path.join("results", "summary", "boiler_kpis_tables.md") STAGES_CSV = os.path.join("results", "runs", "default_case_stages_summary.csv")

def pretty_kpi_label(name: str) -> str: stripped = name.strip()

```
if "[" in stripped and stripped.endswith("]"):
    base, unit = stripped.split("[", 1)
```

```
    base = base.strip()
    unit = "[" + unit
else:
    base, unit = stripped, ""

b = base.lower()

if b.startswith("eta direct"):
    return r"$\eta_{\mathrm{direct}}$ [-]"
if b.startswith("eta indirect"):
    return r"$\eta_{\mathrm{indirect}}$ [-]"
if b.startswith("tad"):
    return f"adiabatic temperature {unit}".strip()
if b.startswith("ua"):
    return f"conductance {unit}".strip()
if b.startswith("q_in total"):
    return f"input heat {unit}".strip()
if b.startswith("q_useful"):
    return f"useful heat {unit}".strip()
if b.startswith("p-lhv"):
    return f"firing rate {unit}".strip()

return stripped.title()
```

def is_number(x) -> bool: try: float(x) return True except Exception: return False

def format_param_value(v) -> str: if is_number(v): return f"{float(v):.2f}" return str(v)

def format_stage_row_label(name: str) -> str: # reuse KPI prettifier for consistent naming return pretty_kpi_label(name)

def format_stage_cell(v) -> str: # keep blanks as blank (your CSV has empty cells) if pd.isna(v) or v == " ": return" " if is_number(v): return f"{float(v):.2f}" return str(v)

def append_stages_summary_table(lines: list[str], stages_csv_path: str) -> None: if not os.path.exists(stages_csv_path): return

```
stages = pd.read_csv(stages_csv_path)

# First column is "name" (row labels)
if "name" not in stages.columns:
    return

# Set index to the row label column, keep stage columns as columns
stages = stages.set_index("name")

# Format values cell-by-cell (handles numeric + blanks)
stages = stages.applymap(format_stage_cell)

# Pretty row labels
stages.index = [format_stage_row_label(i) for i in stages.index]
stages.index.name = "stage KPI"

# Add to markdown output
lines.append("## default case stages summary\n")
lines.append(stages.to_markdown())
lines.append("")
```

def main(): df = pd.read_csv(INPUT_CSV)

```
if "LHV[kJ/kg]" in df.columns:
    df["LHV[kJ/kg]"] = df["LHV[kJ/kg]"] / 1000.0
    df = df.rename(columns={"LHV[kJ/kg]": "LHV [MJ/kg]"})

num_cols = df.select_dtypes(include="number").columns
df[num_cols] = df[num_cols].round(2)

id_vars = ["run", "param_group", "param_value"]
value_vars = [c for c in df.columns if c not in id_vars]

group_meta = {
    "control": ("Control", "control"),
    "excess_air": ("Excess air", "excess air [-]"),
    "fuel_flow": ("Fuel flow", "fuel flow [kg/s]"),
    "drum_pressure": ("Drum pressure", "drum pressure [bar]"),
    "fouling": ("Fouling", "fouling [-]"),
}

lines = []

melted = df.melt(
    id_vars=id_vars,
    value_vars=value_vars,
    var_name="kpi",
    value_name="value",
)

for group_name, (title, index_label) in group_meta.items():

    if group_name == "control":
        sub = df[df["param_group"] == "control"]
        if sub.empty:
            continue

        row = sub.iloc[0]
        data = {"control": [row[k] for k in value_vars]}
        table = pd.DataFrame(data, index=value_vars)

        table.index = [pretty_kpi_label(i) for i in table.index]
        table.index.name = index_label

        lines.append(f"## {title.lower()}\n")
        lines.append(table.to_markdown())
        lines.append("")
        continue

    sub = melted[melted["param_group"] == group_name]
    if sub.empty:
        continue

    group_values = df.loc[df["param_group"] == group_name, "param_value"].tolist()
    seen = set()
```

```
    ordered = []
    for v in group_values:
        if v not in seen:
            seen.add(v)
            ordered.append(v)

    if group_name == "drum_pressure":
        desired = [4.0, 10.0, 16.0]
        present = {float(v) for v in ordered if is_number(v)}
        ordered = [v for v in desired if v in present]

    table = sub.pivot_table(
        index="kpi", columns="param_value", values="value", aggfunc="first"
    )

    table = table.reindex(index=value_vars)
    table = table[ordered]
    table = table.dropna(how="all")

    table.index = [pretty_kpi_label(i) for i in table.index]
    table.columns = [format_param_value(c) for c in table.columns]
    table.index.name = index_label

    lines.append(f"## {title.lower()}\n")
    lines.append(table.to_markdown())
    lines.append("")

append_stages_summary_table(lines, STAGES_CSV)

os.makedirs(os.path.dirname(OUTPUT_MD), exist_ok=True)
with open(OUTPUT_MD, "w", encoding="utf-8") as f:
    f.write("\n".join(lines))
```

if **name** == "**main**": main()

======================================= FILE: combustion/**init**.py ====================

======================================= FILE: combustion/adiabatic_flame_temperature.py ======================================= from common.units import ureg, Q_ from common.models import GasStream from scipy.optimize import root_scalar from common.props import GasProps import cantera as ct from combustion.mass_mole import to_mole, molar_flow from combustion.flue import from_fuel_and_air

def adiabatic_flame_T(air: GasStream, fuel: GasStream) -> GasStream: P_Pa = air.P.to("Pa").magnitude T_air = air.T.to("K").magnitude T_fuel = fuel.T.to("K").magnitude

```
m_air  = air.mass_flow.to("kg/s").magnitude
m_fuel = fuel.mass_flow.to("kg/s").magnitude
m_tot  = m_air + m_fuel
if m_tot <= 0.0:
    raise ValueError("adiabatic_flame_T: total mass flow must be > 0")

X_air  = to_mole({k: v.to("").magnitude for k, v in (air.comp  or {}).items() if v.to("").magnitude > 0})
X_fuel = to_mole({k: v.to("").magnitude for k, v in (fuel.comp or {}).items() if v.to("").magnitude > 0})

gas_air  = ct.Solution("config/flue_cantera.yaml", "gas_mix")
```

```
gas_fuel = ct.Solution("config/flue_cantera.yaml", "gas_mix")
gas_mix  = ct.Solution("config/flue_cantera.yaml", "gas_mix")

gas_air.TPX  = T_air,  P_Pa, X_air
gas_fuel.TPX = T_fuel, P_Pa, X_fuel

Hdot_react = m_air * gas_air.enthalpy_mass + m_fuel * gas_fuel.enthalpy_mass
h_target   = Hdot_react / m_tot

n_air  = molar_flow(air.comp,  air.mass_flow)
n_fuel = molar_flow(fuel.comp, fuel.mass_flow)

def _mol_rate(X, n_tot): return {k: n_tot * float(x) for k, x in X.items()}
n_dot_sp = {}
for d in (_mol_rate(X_air, n_air), _mol_rate(X_fuel, n_fuel)):
    for k, v in d.items():
        n_dot_sp[k] = n_dot_sp.get(k, 0.0) + v
n_sum = sum(n_dot_sp.values())
if n_sum <= 0.0:
    raise ValueError("adiabatic_flame_T: empty reactant composition")
X_react = {k: v / n_sum for k, v in n_dot_sp.items()}

gas_mix.TPX = 300.0, P_Pa, X_react
gas_mix.HP  = h_target, P_Pa
gas_mix.equilibrate("HP")

Y_eq = gas_mix.Y
comp_eq = {sp: Q_(float(Y_eq[i]), "") for i, sp in enumerate(gas_mix.species_names) if Y_eq[i] > 1e-
15}

return GasStream(
    mass_flow=Q_(m_tot, "kg/s"),
    T=Q_(gas_mix.T, "K"),
    P=air.P,
    comp=comp_eq,
)
```

def adiabatic_flame_T_no_dissociation(air: GasStream, fuel: GasStream) -> GasStream: m_air = air.mass_flow.to("kg/s").magnitude m_fuel = fuel.mass_flow.to("kg/s").magnitude m_tot = m_air + m_fuel if m_tot <= 0.0: raise ValueError("adiabatic_flame_T_no_dissociation: total mass flow must be > 0")

```
gasprops = GasProps()

h_air  = gasprops.h(air.T,  air.P,  air.comp).to("J/kg").magnitude
h_fuel = gasprops.h(fuel.T, fuel.P, fuel.comp).to("J/kg").magnitude

Hdot_react = m_air * h_air + m_fuel * h_fuel
h_target   = Hdot_react / m_tot

mass_comp_burnt, m_dot_flue = from_fuel_and_air(fuel, air)
comp_prod = {sp: Q_(float(y), "") for sp, y in mass_comp_burnt.items() if float(y) > 1e-15}

def f(T_K: float) -> float:
    hP = gasprops.h(Q_(T_K, "K"), air.P, comp_prod).to("J/kg").magnitude
```

```
    return hP - h_target

T_lo, T_hi = 250.0, 3500.0
f_lo, f_hi = f(T_lo), f(T_hi)
if f_lo * f_hi > 0.0:
    T_hi = 4500.0
    f_hi = f(T_hi)
    if f_lo * f_hi > 0.0:
        raise RuntimeError(
            "adiabatic_flame_T_no_dissociation: could not bracket root for Tad "
            f"(f({T_lo})={f_lo:.3e}, f({T_hi})={f_hi:.3e})"
        )

sol = root_scalar(f, bracket=(T_lo, T_hi), method="brentq", xtol=1e-6, rtol=1e-8)
if not sol.converged:
    raise RuntimeError("adiabatic_flame_T_no_dissociation: root solve did not converge")

Tad = float(sol.root)

return GasStream(
    mass_flow=m_dot_flue.to("kg/s"),
    T=Q_(Tad, "K"),
    P=air.P,
    comp=comp_prod,
)
```

==================================== FILE: combustion/combustor.py ================
from combustion.adiabatic_flame_temperature import adiabatic_flame_T from combustion.heat import total_input_heat, compute_LHV_HHV from combustion.flue import air_flow_rates from common.results import CombustionResult from common.models import GasStream from common.units import Q_ from combustion.adiabatic_flame_temperature import adiabatic_flame_T_no_dissociation

class Combustor: def **init**(self, air: GasStream, fuel: GasStream, excess_air_ratio: Q_): self.air = air self.fuel = fuel self.excess_air_ratio = excess_air_ratio

```
def run(self) -> CombustionResult:
    air = self.air
    fuel = self.fuel

    air.mass_flow = air_flow_rates(air, fuel, self.excess_air_ratio)

    power_LHV, Q_in = total_input_heat(fuel, air)
    HHV_mass, LHV_mass, P_HHV, P_LHV = compute_LHV_HHV(fuel, air)

    flue_ad = adiabatic_flame_T(air, fuel)
    T_ad = flue_ad.T

    flue_boiler = adiabatic_flame_T_no_dissociation(air, fuel)


    return CombustionResult(
        LHV           = power_LHV,
        Q_in          = Q_in,
        T_ad          = T_ad,
        flue          = flue_boiler,
```

```
        flue_ad        = flue_ad,
        fuel_LHV_mass  = LHV_mass,
        fuel_P_LHV     = P_LHV,
        fuel_mass_flow = fuel.mass_flow,
        excess_air_ratio=self.excess_air_ratio,
        air_mass_flow  = air.mass_flow,
    )
```

======================================= FILE: combustion/flue.py ====================

from common.models import GasStream from common.units import Q_ from common.constants import O2_per_mol from combustion.mass_mole import to_mole, molar_flow, mass_flow, to_mass, mix_molar_mass

def stoich_O2_required_per_mol_fuel(fuel: GasStream) -> Q_: fuel_x = to_mole(fuel.comp) total = sum(fuel_x[k] * O2_per_mol.get(k, 0.0) for k in fuel_x) return Q_(total, "dimensionless")

def air_flow_rates(air: GasStream, fuel: GasStream, excess: Q_) -> Q_: air_x = to_mole(air.comp) fuel_n_dot = molar_flow(fuel.comp, fuel.mass_flow) O2_x = air_x["O2"] O2_req = stoich_O2_required_per_mol_fuel(fu O2_stoich = fuel_n_dot * O2_req O2_actual = O2_stoich * excess air_n = O2_actual / O2_x M_air = mix_molar_mass(air_x) air_m = air_n * M_air return air_m

def from_fuel_and_air(fuel: GasStream, air: GasStream) -> tuple[dict[str, float], float]: O2_req = stoich_O2_required_per_mol_fuel(fuel) fuel_x = to_mole(fuel.comp) fuel_n = molar_flow(fuel.comp, fuel.mass_flow) air_x = to_mole(air.comp) air_n = molar_flow(air.comp, air.mass_flow)

```
gf=lambda k:fuel_x.get(k,0.0); ga=lambda k:air_x.get(k,0.0)
n_CO2 = air_n*ga("CO2")+fuel_n*gf("CO2")+fuel_n*(gf("CH4")+2*gf("C2H6")+3*gf("C3H8")+4*gf("C4H10"))
n_H2O = fuel_n*gf("H2O")+fuel_n*(2*gf("CH4")+3*gf("C2H6")+4*gf("C3H8")+5*gf("C4H10"))+fuel_n*gf("H2S")+air_n*ga("
n_SO2 = fuel_n*gf("H2S")
n_O2  = air_n*ga("O2") - fuel_n*O2_req
n_N2  = air_n*ga("N2") + fuel_n*gf("N2")
n_Ar  = air_n*ga("Ar")
flows={"CO2":n_CO2,"H2O":n_H2O,"SO2":n_SO2,"O2":n_O2,"N2":n_N2,"Ar":n_Ar}

n_tot=sum(flows.values())
mol_comp={k:(v/n_tot if n_tot!=0 else 0.0) for k,v in flows.items()}
mass_comp = to_mass(mol_comp)
m_dot = mass_flow(mol_comp, n_tot)

return mass_comp, m_dot
```

======================================= FILE: combustion/heat.py ====================

from common.units import Q_ import re import cantera as ct from common.props import WaterProps, GasProps from common.models import GasStream from combustion.mass_mole import to_mole from common.constants import molar_masses, T_ref, P_ref, O2_per_mol

_gasprops = GasProps()

def parse_CH(s: str): m = re.fullmatch(r'C(*)H(+)', s) if not m: return None, None C = int(m.group(1)) if m.group(1) else 1 H = int(m.group(2)) return C, H

def compute_LHV_HHV(fuel: GasStream, air: GasStream) -> tuple[Q_, Q_, Q_, Q_]: gas = ct.Solution("config/flue_cantera.yaml", "gas_mix")

```
fuel_x = to_mole({k: float(v.to("").magnitude) for k, v in (fuel.comp or {}).items()
                  if float(v.to("").magnitude) > 0.0})
if not fuel_x:
    raise ValueError("compute_LHV_HHV: empty fuel composition")
```

```
air_x = to_mole({k: float(v.to("").magnitude) for k, v in (air.comp or {}).items()
                 if float(v.to("").magnitude) > 0.0})
if not air_x or air_x.get("O2", 0.0) <= 0.0:
    raise ValueError("compute_LHV_HHV: air composition missing O2")

def xf(sp: str) -> float:
    return float(fuel_x.get(sp, 0.0))

def xa(sp: str) -> float:
    return float(air_x.get(sp, 0.0))

O2_req = 0.0
for sp, x in fuel_x.items():
    req = O2_per_mol.get(sp, Q_(0.0, "")).to("").magnitude
    O2_req += float(x) * float(req)

n_air = O2_req / max(xa("O2"), 1e-30)

n_react = {}
for sp, x in fuel_x.items():
    if x > 0.0:
        n_react[sp] = n_react.get(sp, 0.0) + x

for sp, x in air_x.items():
    if x > 0.0:
        n_react[sp] = n_react.get(sp, 0.0) + n_air * x

nR_tot = sum(n_react.values())
X_react = {sp: n / nR_tot for sp, n in n_react.items() if n > 0.0}

n_CO2 = n_air * xa("CO2") + xf("CO2") + (xf("CH4") + 2*xf("C2H6") + 3*xf("C3H8") + 4*xf("C4H10"))
n_H2O = n_air * xa("H2O") + xf("H2O") + (2*xf("CH4") + 3*xf("C2H6") + 4*xf("C3H8") + 5*xf("C4H10")) + xf("H2S")
n_SO2 = xf("H2S")
n_O2  = n_air * xa("O2") - O2_req
n_N2  = n_air * xa("N2") + xf("N2")
n_Ar  = n_air * xa("Ar")

n_prod = {
    "CO2": n_CO2,
    "H2O": n_H2O,
    "SO2": n_SO2,
    "O2":  n_O2,
    "N2":  n_N2,
    "Ar":  n_Ar,
}
for k in list(n_prod.keys()):
    if n_prod[k] < 0.0 and abs(n_prod[k]) < 1e-12:
        n_prod[k] = 0.0

nP_tot = sum(v for v in n_prod.values() if v > 0.0)
if nP_tot <= 0.0:
    raise ValueError("compute_LHV_HHV: empty products")

X_prod = {sp: n / nP_tot for sp, n in n_prod.items() if n > 0.0}
```

```
T0 = T_ref.to("K").magnitude
P0 = P_ref.to("Pa").magnitude

gas.TPX = T0, P0, X_react
hR_molar = gas.enthalpy_mole

gas.TPX = T0, P0, X_prod
hP_molar = gas.enthalpy_mole

HR = hR_molar * (nR_tot / 1000.0)
HP = hP_molar * (nP_tot / 1000.0)

LHV_mol_J = HR - HP
if LHV_mol_J <= 0.0:
    raise ValueError(f"compute_LHV_HHV: non-positive LHV ({LHV_mol_J} J/mol basis)")

M_mix = Q_(0.0, "kg/mol")
for sp, x in fuel_x.items():
    M_mix = M_mix + Q_(float(x), "") * molar_masses[sp].to("kg/mol")

if M_mix.to("kg/mol").magnitude <= 0.0:
    raise ValueError("compute_LHV_HHV: invalid fuel mixture molar mass")

LHV_mol = Q_(float(LHV_mol_J), "J/mol")

LHV_kg = (LHV_mol / M_mix).to("kJ/kg")

latent = (WaterProps.h_g(P_ref) - WaterProps.h_f(P_ref)).to("J/kg")
h_fg_mol = (latent * molar_masses["H2O"].to("kg/mol")).to("J/mol")

HHV_mol = (LHV_mol + Q_(float(n_H2O), "") * h_fg_mol).to("J/mol")
HHV_kg = (HHV_mol / M_mix).to("kJ/kg")

P_LHV = (LHV_kg.to("J/kg") * fuel.mass_flow).to("kW")
P_HHV = (HHV_kg.to("J/kg") * fuel.mass_flow).to("kW")

return HHV_kg, LHV_kg, P_HHV, P_LHV
```

def sensible_heat(stream: GasStream) -> Q_: s = stream h_sens = _gasprops.h_sensible(s.T, s.P, s.comp, Tref=T_ref).to("J/kg") return (s.mass_flow * h_sens).to("kW")

def total_input_heat(fuel, air) -> Q_: , , _, power_LHV = compute_LHV_HHV(fuel, air) fuel_sens = sensible_heat(fuel) air_sens = sensible_heat(air) Q_in = (power_LHV.to("kW") + fuel_sens.to("kW") + air_sens.to("kW")).to("kW") return power_LHV, Q_in

== FILE: combustion/mass_mole.py == from typing import Dict from common.constants import molar_masses

def to_mole(mass_comp: Dict[str, float]) -> Dict[str, float]: n = {sp: mass_comp[sp] / molar_masses[sp] for sp in mass_comp} tot = sum(n.values()) return {sp: v / tot for sp, v in n.items()}

def to_mass(mol_comp: Dict[str, float]) -> Dict[str, float]: m = {sp: mol_comp[sp] * molar_masses[sp] for sp in mol_comp} tot = sum(m.values()) return {sp: v / tot for sp, v in m.items()}

def mix_molar_mass(mol_comp: Dict[str, float]) -> float: return sum(mol_comp[sp] * molar_masses[sp] for sp in mol_comp)

def molar_flow(mass_comp: Dict[str, float], m_dot: float) -> float: return sum((mass_comp[sp] * m_dot) / (molar_masses[sp]) for sp in mass_comp)

def mass_flow(mol_comp: Dict[str, float], n_dot: float) -> float: return sum(mol_comp[sp] * n_dot * molar_masses[sp] for sp in mol_comp)

== FILE: common/**init**.py ========================

== FILE: common/boiler_loop.py ==================
from **future** import annotations import logging from typing import Dict, Any, Tuple from common.new_loader import load_all from combustion.combustor import Combustor from heat.runner import run_hx from common.units import Q_ from common.props import WaterProps from common.models import WaterStream from common.results import CombustionResult, write_results_csvs

log = logging.getLogger(**name**)

def _drum_steam_rate(*, P_drum: Q_, Q_evap: Q_, m_fw: Q_, h_fw_out: Q_, steam_quality_out: float = 1.0,) -> Q_: hf = WaterProps.h_f(P_drum).to("J/kg") hg = WaterProps.h_g(P_drum).to("J/kg") h_s = (hf + Q_(steam_quality_out, " ") * (hg - hf)).to("J/kg")

```
denom = (h_s - hf).to("J/kg")
if denom.magnitude <= 0:
    raise ValueError("Invalid drum latent enthalpy (h_s - h_f) <= 0.")

m_s = (Q_evap.to("W") + m_fw * (h_fw_out - hf)) / denom
return m_s.to("kg/s")
```

def run_boiler_case(stages_path: str = "config/stages.yaml", air_path: str = "config/air.yaml", fuel_path: str = "config/fuel.yaml", water_path: str = "config/water.yaml", drum_path: str = "config/drum.yaml", operation_path: str = "config/operation.yaml", *, tol_m: Q_ = Q_(1e-3, "kg/s"), max_iter: int = 20, write_csv: bool = True, operation_overrides: Dict[str, Q_] | None = None, fuel_overrides: Dict[str, Q_] | None = None, fouling_factor: float = 1.0, run_id: str | None = None,) -> Dict[str, Any]: log.info(f"Load config") stages, air, fuel, water, drum, operation = load_all(stages_path=stages_path, air_path=air_path, fuel_path=fuel_path, water_path=water_path, drum_path=drum_path, operation_path=operation_path,)

```
f = float(fouling_factor)
if f <= 0.0:
    raise ValueError(f"fouling_factor must be > 0. Got {fouling_factor!r}")

if abs(f - 1.0) > 1e-12:
    for st in stages:
        spec = st.spec
        if "foul_t_in" in spec:
            spec["foul_t_in"] = (spec["foul_t_in"] * Q_(f, "")).to(spec["foul_t_in"].units)
        if "foul_t_out" in spec:
            spec["foul_t_out"] = (spec["foul_t_out"] * Q_(f, "")).to(spec["foul_t_out"].units)

if operation_overrides:
    operation.update(operation_overrides)

if fuel_overrides:
    for attr, val in fuel_overrides.items():
        if hasattr(fuel, attr):
            setattr(fuel, attr, val)
        else:
```

```
            log.warning(f"GasStream (fuel) has no attribute '{attr}', ignoring override.")

P_drum: Q_ | None = operation.get("drum_pressure", None)
blowdown_fraction = operation.get("blowdown_fraction", Q_(0.0, ""))
steam_quality_out = float(operation.get("steam_quality_out", Q_(1.0, "")).to("").magnitude)

water_template: WaterStream = water

if P_drum is not None:
    water_template.P = P_drum

log.info(f"Running Combustor")
svc = Combustor(air, fuel, operation["excess_air_ratio"])
combustion_results = svc.run()
log.info(f"Combustion Done")

if P_drum is None:
    raise ValueError("Option B requires operation.drum_pressure")

hf = WaterProps.h_f(P_drum).to("J/kg")
hg = WaterProps.h_g(P_drum).to("J/kg")
h_feed = water_template.h.to("J/kg")
latent = (hg - hf).to("J/kg")
m_fw = ((Q_(0.94, "") * combustion_results.Q_in.to("W")) / (latent + (hf - h_feed))).to("kg/s")

prev_m = None
final_result = None
final_m_fw = None
final_m_s = None

tol_m_fw = tol_m.to("kg/s").magnitude

for it in range(max_iter):
    m_bd = (blowdown_fraction.to("").magnitude * m_fw).to("kg/s")

    water_in = WaterStream(mass_flow=m_fw, h=water_template.h, P=water_template.P)

    final_result = run_hx(
        stages_raw=stages,
        water=water_in,
        gas=combustion_results.flue,
        drum=drum,
        drum_pressure=P_drum,
        target_dx="0.1 m",
        combustion=combustion_results,
        write_csv=False,
    )

    h_fw_out = final_result["water_out"].h.to("J/kg")

    evap_names = {f"HX_{i}" for i in range(1, 6)}
    Q_evap_W = 0.0
    for r in final_result["summary_rows"]:
        if r.get("stage_name") in evap_names and isinstance(r.get("Q_stage[MW]"), (int, float)):
```

```
            Q_evap_W += Q_(r["Q_stage[MW]"], "MW").to("W").magnitude
    Q_evap = Q_(Q_evap_W, "W")

    m_s = _drum_steam_rate(
        P_drum=P_drum,
        Q_evap=Q_evap,
        m_fw=m_fw,
        h_fw_out=h_fw_out,
        steam_quality_out=steam_quality_out,
    )

    resid = (m_s + m_bd - m_fw).to("kg/s").magnitude

    final_m_fw = m_fw
    final_m_s = m_s

    if prev_m is not None:
        dm = (m_fw - prev_m).to("kg/s").magnitude
        if abs(dm) < tol_m_fw and abs(resid) < tol_m_fw:
            break

    m_fw_new = (m_s + m_bd).to("kg/s")
    m_fw = (Q_(0.5, "") * m_fw_new + Q_(0.5, "") * m_fw).to("kg/s")

    prev_m = final_m_fw

else:
    log.warning("Did not reach drum mass-balance convergence within max_iter.")

feed_P: Q_ | None = None

if P_drum is not None and final_m_fw is not None:
    P_drum_Pa = P_drum.to("Pa")

    P_in = (P_drum_Pa * Q_(1.01, "")).to("Pa")

    max_p_iter = 30
    tol_P = Q_(1.0, "Pa")

    feed_P = None
    last_result: Dict[str, Any] | None = None

    for _ in range(max_p_iter):
        water_trial = WaterStream(
            mass_flow=final_m_fw,
            h=water_template.h,
            P=P_in,
        )

        last_result = run_hx(
            stages_raw=stages,
            water=water_trial,
            gas=combustion_results.flue,
            drum=drum,
```

```
            drum_pressure=P_drum,
            target_dx="0.1 m",
            combustion=combustion_results,
            write_csv=False,
        )

        P_out = last_result["water_out"].P.to("Pa")

        err = (P_drum_Pa - P_out).to("Pa")

        if abs(err).to("Pa").magnitude < tol_P.to("Pa").magnitude:
            feed_P = P_in
            break

        P_in = (P_in + err).to("Pa")

        if P_in < P_drum_Pa:
            P_in = P_drum_Pa

    if feed_P is None:
        feed_P = P_in

    log.info(f"Solved feedwater inlet pressure: {feed_P:~P} for drum pressure {P_drum:~P}")

if feed_P is None:
    feed_P = P_drum

water_final_in = WaterStream(
    mass_flow=final_m_fw,
    h=water_template.h,
    P=feed_P,
)

log.info(f"Final feedwater inlet mass flow: {water_final_in.mass_flow:~P}")

final_result = run_hx(
    stages_raw=stages,
    water=water_final_in,
    gas=combustion_results.flue,
    drum=drum,
    drum_pressure=P_drum,
    target_dx="0.1 m",
    combustion=combustion_results,
    write_csv=write_csv,
)

csv_paths: Tuple[str, str, str] | None = None
if write_csv:
    effective_run_id = run_id if run_id is not None else final_result["run_id"]

    steps_csv, stages_summary_csv, boiler_summary_csv = write_results_csvs(
        global_profile=final_result["global_profile"],
        combustion=final_result["combustion"],
        outdir=final_result["outdir"],
```

```
        run_id=effective_run_id,
        drum_pressure=P_drum,
        feed_pressure=feed_P,
    )
    csv_paths = (steps_csv, stages_summary_csv, boiler_summary_csv)

m_bd_final = None
if final_m_fw is not None:
    m_bd_final = (blowdown_fraction.to("").magnitude * final_m_fw).to("kg/s")
else:
    m_bd_final = Q_(0.0, "kg/s")

return {
    "result": final_result,
    "m_fw": final_m_fw,
    "m_s": final_m_s,
    "m_bd": m_bd_final,
    "drum_pressure": P_drum,
    "combustion": combustion_results,
    "csv_paths": csv_paths,
}
```

====================================== FILE: common/constants.py ======================================
from common.units import Q_

T_ref = Q_(298.15, "kelvin") P_ref = Q_(101325, "pascal")

molar_masses = { "CH4": Q_(0.01604, "kilogram / mole"), "C2H6": Q_(0.03007, "kilogram / mole"), "C3H8": Q_(0.04410, "kilogram / mole"), "C4H10": Q_(0.05812, "kilogram / mole"), "H2S": Q_(0.03408, "kilogram / mole"), "N2": Q_(0.02802, "kilogram / mole"), "CO2": Q_(0.04401, "kilogram / mole"), "H2O": Q_(0.018015, "kilogram / mole"), "O2": Q_(0.031998, "kilogram / mole"), "Ar": Q_(0.039948, "kilogram / mole"), "SO2": Q_(0.06407, "kilogram / mole"), }

O2_per_mol = { "CH4": Q_(2.0, "dimensionless"), "C2H6": Q_(3.5, "dimensionless"), "C3H8": Q_(5.0, "dimensionless"), "C4H10": Q_(6.5, "dimensionless"), "H2S": Q_(1.0, "dimensionless"), "N2": Q_(0.0, "dimensionless"), "CO2": Q_(0.0, "dimensionless"), "H2O": Q_(0.0, "dimensionless"), }

====================================== FILE: common/logging_utils.py ======================================
import logging, functools, time

TRACE_LEVEL_NUM = 5 logging.addLevelName(TRACE_LEVEL_NUM, "TRACE")

class _TraceLogger(logging.Logger): def trace(self, msg, *a, **k): if self.isEnabledFor(TRACE_LEVEL_NUM): self._log(TRACE_LEVEL_NUM, msg, a, **k) logging.setLoggerClass(_TraceLogger)

_FMT = "%(asctime)s | %(levelname)s | %(name)s | stage=%(stage)s step=%(step)s | %(message)s" _DATE = "%Y-%m-%d %H:%M:%S"

class _Stage(logging.Filter): def filter(self, r): if not hasattr(r,"stage"): r.stage="-" if not hasattr(r,"step"): r.step="-" return True

def setup_logging(level: int | str = logging.INFO): if isinstance(level,str): lvl = logging.getLevelName(level.upper()) level = lvl if isinstance(lvl,int) else logging.INFO root = logging.getLogger() root.handlers.clear() root.setLevel(level) h = logging.StreamHandler() h.setFormatter(logging.Formatter(_FMT,_DATE)) h.addFilter(_Stage()) root.addHandler(h)

def _fmt(v): try: return f"{v:.6g~P}" except: return repr(v)

def trace_calls(name: str|None=None, values: bool=False): def _wrap(fn): qual = name or f"{fn.**module**}.{fn.**qualname**}" log = logging.getLogger(qual) @functools.wraps(fn) def _inner(*a,

k): stage = k.get("stage","-") log.trace("enter", extra={"stage": stage, "step": fn.___name___}) if values: arg_s = ",".join([*map(_fmt, a), *[f"{kk}={_fmt(v)}" for kk,v in k.items()]]) log.trace(f"args: {arg_s}", extra={"stage": stage, "step": fn.___name___}) t0=time.perf_counter() try: out=fn(*a, **k) dt=(time.perf_counter()-t0)*1000 if values: log.trace(f"ret: {_fmt(out)}", extra={"stage": stage, "step": fn.name**}) log.trace(f"exit ok in {dt:.2f} ms", extra={"stage": stage, "step": fn.**name**}) return out except Exception as e: log.exception(f"exit err: {e}", extra={"stage": stage, "step": fn.**name**}) raise return _inner return _wrap

== FILE: common/models.py ======================
from dataclasses import dataclass from typing import Dict, Any from common.units import Q_

@dataclass class GasStream: mass_flow: Q_ T: Q_ P: Q_ comp: Dict[str, Q_]

@dataclass class WaterStream: mass_flow: Q_ h: Q_ P: Q_

@dataclass class HXStage: name: str kind: str spec: Dict[str, Any]

@dataclass class Drum: Di: Q_ L: Q_
====================================== FILE: common/new_loader.py ==================
from **future** import annotations from typing import Tuple, List, Dict, Any import yaml from common.units import Q_ from common.models import HXStage, GasStream, WaterStream, Drum

def *q(node: Any) -> Q*: if isinstance(node, dict) and "value" in node and "unit" in node: return Q_(node["value"], str(node["unit"])) raise ValueError(f"Invalid quantity format: {node!r}")

def _get(d: Dict[str, Any] | None, key: str, default=None): return d.get(key, default) if isinstance(d, dict) else default

def _wall_to_spec(wall: Dict[str, Any] | None, spec: Dict[str, Q_]): if not wall: return if _get(wall, "thickness"): spec["wall_t"] = _q(_get(wall, "thickness")) if _get(wall, "conductivity"): spec["wall_k"] = _q(_get(wall, "conductivity"))

```
surf_in = _get(_get(wall, "surfaces"), "inner") or {}
if _get(surf_in, "roughness"):            spec["roughness_in"] = _q(_get(surf_in, "roughness"))
if _get(surf_in, "emissivity"):           spec["eps_in"] = _q(_get(surf_in, "emissivity"))
if _get(surf_in, "fouling_thickness"):   spec["foul_t_in"] = _q(_get(surf_in, "fouling_thickness"))
if _get(surf_in, "fouling_conductivity"): spec["foul_k_in"] = _q(_get(surf_in, "fouling_conductivity"))

surf_out = _get(_get(wall, "surfaces"), "outer") or {}
if _get(surf_out, "roughness"):            spec["roughness_out"] = _q(_get(surf_out, "roughness"))
if _get(surf_out, "emissivity"):           spec["eps_out"] = _q(_get(surf_out, "emissivity"))
if _get(surf_out, "fouling_thickness"):   spec["foul_t_out"] = _q(_get(surf_out, "fouling_thickness"))
if _get(surf_out, "fouling_conductivity"): spec["foul_k_out"] = _q(_get(surf_out, "fouling_conductivity"))
```

def _map_K(node: Dict[str, Any], spec: Dict[str, Q_]): K_node = _get(node, "K") or {} if not isinstance(K_node, dict): return

```
mapping = {
    "hot_inlet": "K_hot_inlet",
    "hot_outlet": "K_hot_outlet",
    "hot_bend": "K_hot_bend",
    "cold_inlet": "K_cold_inlet",
    "cold_outlet": "K_cold_outlet",
    "cold_bend": "K_cold_bend",
}

for yaml_key, spec_key in mapping.items():
    v = _get(K_node, yaml_key)
    if v is not None:
```

```
        spec[spec_key] = Q_(float(v), "dimensionless")
```

def load_air(path: str) -> Dict[str, Any]: doc = yaml.safe_load(open(path, “r”, encoding=“utf-8”)) comp = {k: *q(v) for k, v in (doc.get(“composition”) or {}).items()} return GasStream(mass_flow=Q*(0, “kg/s”), T=_q(doc[“T”]), P=_q(doc[“P”]), comp=comp)

def load_fuel(path: str) -> GasStream: doc = yaml.safe_load(open(path, “r”, encoding=“utf-8”)) comp = {k: _q(v) for k, v in (doc.get(“composition”) or {}).items()} return GasStream(mass_flow=_q(doc[“mass_flow”]), T=_q(doc[“T”]), P=_q(doc[“P”]), comp=comp)

def load_drum(path: str) -> Drum: doc = yaml.safe_load(open(path, “r”, encoding=“utf-8”)) return Drum(Di=_q(doc[“inner_diameter”]).to(“m”), L=_q(doc[“length”]).to(“m”))

def load_stages(path: str) -> List[HXStage]: sdoc = yaml.safe_load(open(path, “r”, encoding=“utf-8”)) stages: List[HXStage] = []

```
for name, node in sdoc.items():

    spec: Dict[str, Q_] = {
        "inner_diameter": _q(node["inner_diameter"]),
    }
    if "inner_length" in node:          spec["inner_length"]     = _q(node["inner_length"])
    if "pool_boiling" in node:         spec["pool_boiling"]     = bool(node["pool_boiling"])
    if "curvature_radius" in node:     spec["curvature_radius"] = _q(node["curvature_radius"])
    if "tubes_number" in node:         spec["tubes_number"]     = _q(node["tubes_number"])
    if "ST" in node:                   spec["ST"]               = _q(node["ST"])
    if "SL" in node:                   spec["SL"]               = _q(node["SL"])
    if "arrangement" in node:          spec["arrangement"]      = str(node["arrangement"])
    if "N_rows" in node:               spec["N_rows"]           = _q(node["N_rows"])
    if "baffle_spacing" in node:       spec["baffle_spacing"]   = _q(node["baffle_spacing"])
   if "shell_inner_diameter" in node: spec["shell_inner_diameter"] = _q(node["shell_inner_diameter"])
    if "baffle_cut" in node:           spec["baffle_cut"]       = _q(node["baffle_cut"])
    if "bundle_clearance" in node:     spec["bundle_clearance"] = _q(node["bundle_clearance"])
    if "n_tubes" in node:              spec["n_tubes"]      = _q(node["n_tubes"])
    if "n_circuits" in node:           spec["n_circuits"]   = _q(node["n_circuits"])
    if "tube_length" in node:          spec["tube_length"]  = _q(node["tube_length"])


    _wall_to_spec(_get(node, "wall"), spec)
    _map_K(node, spec)

    stages.append(HXStage(name=name, kind=str(node["kind"]), spec=spec))

return stages
```

def load_operation(path: str) -> Dict[str, Q_]: doc = yaml.safe_load(open(path, “r”, encoding=“utf-8”))

```
out: Dict[str, Q_] = {
    "excess_air_ratio": _q(doc["excess_air_ratio"]),
}

if "drum_pressure" in doc:
    out["drum_pressure"] = _q(doc["drum_pressure"])
if "circulation_ratio" in doc:
    out["circulation_ratio"] = _q(doc["circulation_ratio"])
if "blowdown_fraction" in doc:
    out["blowdown_fraction"] = _q(doc["blowdown_fraction"])
```

```
if "steam_quality_out" in doc:
    out["steam_quality_out"] = _q(doc["steam_quality_out"])

return out
```

def load_water_stream(path: str) -> WaterStream: doc = yaml.safe_load(open(path, “r”, encoding=“utf-8”)) h = _q(doc[“enthalpy”])

```
P_node = doc.get("pressure", None)
if P_node is not None:
    P = _q(P_node)
else:
    P = Q_(1.0, "megapascal")

return WaterStream(
    mass_flow=Q_(0, "kg/s"),
    h=h,
    P=P,
)
```

def load_all(stages_path: str, water_path: str, drum_path: str, air_path: str, fuel_path: str, operation_path: str,) -> Tuple[List[HXStage], GasStream, GasStream, WaterStream, Drum, Dict[str, Q_]]: stages = load_stages(stages_path) if stages_path else None water = load_water_stream(water_path) if water_path else None drum = load_drum(drum_path) if drum_path else None air = load_air(air_path) if air_path else None fuel = load_fuel(fuel_path) if fuel_path else None operation = load_operation(operation_path) if operation_path else None return stages, air, fuel, water, drum, operation

== FILE: common/props.py ==
from **future** import annotations from typing import Dict, Optional from common.units import Q_ import cantera as ct from iapws import IAPWS97

class GasProps: def **init**(self, mech_path: str = “config/flue_cantera.yaml”, phase: str = “gas_mix”): self._sol = ct.Solution(mech_path, phase)

```
def _set(self, T: Q_, P: Q_, Y: Dict[str, Q_], film_T: Optional[Q_] = None):
    T_K = (film_T or T).to("K").magnitude
    P_Pa = P.to("Pa").magnitude
    Y_map = {k: v.to("dimensionless").magnitude for k, v in Y.items()}
    self._sol.TPY = T_K, P_Pa, Y_map
    return self._sol

def cp(self, T: Q_, P: Q_, X: Dict[str, Q_], film_T: Optional[Q_] = None) -> Q_:
    return Q_(self._set(T,P,X,film_T).cp_mass, "J/kg/K")

def k(self, T: Q_, P: Q_, X: Dict[str, Q_], film_T: Optional[Q_] = None) -> Q_:
    return Q_(self._set(T,P,X,film_T).thermal_conductivity, "W/m/K")

def mu(self, T: Q_, P: Q_, X: Dict[str, Q_], film_T: Optional[Q_] = None) -> Q_:
    return Q_(self._set(T,P,X,film_T).viscosity, "Pa*s")

def rho(self, T: Q_, P: Q_, X: Dict[str, Q_], film_T: Optional[Q_] = None) -> Q_:
    return Q_(self._set(T,P,X,film_T).density, "kg/m^3")

def h(self, T: Q_, P: Q_, X: Dict[str, Q_], film_T: Optional[Q_] = None) -> Q_:
    return Q_(self._set(T,P,X,film_T).enthalpy_mass, "J/kg")
```

```
def h_sensible(self, T: Q_, P: Q_, X: dict, Tref: Q_ = Q_(298.15, "K"), film_T: Q_ | None = None) -
> Q_:
    hT   = self.h(T,   P, X, film_T)
    href = self.h(Tref, P, X, film_T)
    return (hT - href).to("J/kg")
```

class WaterProps: @staticmethod def *Ph(P: Q*, h: Q__) -> IAPWS97: return IAPWS97(P=P.to("megapascal").magnitude, h=h.to("kJ/kg").magnitude)

```
@staticmethod
def _PT(P: Q_, T: Q_) -> IAPWS97:
    return IAPWS97(P=P.to("megapascal").magnitude, T=T.to("K").magnitude)

@staticmethod
def T_from_Ph(P: Q_, h: Q_) -> Q_: return Q_(WaterProps._Ph(P,h).T, "K")
@staticmethod
def rho_from_Ph(P: Q_, h: Q_) -> Q_: return Q_(WaterProps._Ph(P,h).rho, "kg/m^3")
@staticmethod
def mu_from_Ph(P: Q_, h: Q_) -> Q_:  return Q_(WaterProps._Ph(P,h).mu, "Pa*s")
@staticmethod
def k_from_Ph(P: Q_, h: Q_) -> Q_:   return Q_(WaterProps._Ph(P,h).k, "W/m/K")
@staticmethod
def cp_from_Ph(P: Q_, h: Q_) -> Q_:  return Q_(WaterProps._Ph(P,h).cp, "kJ/kg/K").to("J/kg/K")

@staticmethod
def quality_from_Ph(P: Q_, h: Q_) -> Q_ | None:
    Pcrit = Q_(22.064, "MPa")
    if P >= Pcrit:
        return None
    hf = WaterProps.h_f(P)
    hg = WaterProps.h_g(P)
    dh = hg - hf
    if abs(dh.to("J/kg").magnitude) < 1e-9:
        return None

    x = ((h - hf) / dh).to("")
    xm = x.magnitude
    if xm < -1e-6 or xm > 1 + 1e-6:
        return None

    xm = min(1.0, max(0.0, xm))
    return Q_(xm, "")

@staticmethod
def Tsat(P: Q_) -> Q_: return Q_(IAPWS97(P=P.to("megapascal").magnitude, x=0.0).T, "K")
@staticmethod
def h_f(P: Q_) -> Q_:  return Q_(IAPWS97(P=P.to("megapascal").magnitude, x=0.0).h, "kJ/kg").to("J/kg")
@staticmethod
def h_g(P: Q_) -> Q_:  return Q_(IAPWS97(P=P.to("megapascal").magnitude, x=1.0).h, "kJ/kg").to("J/kg")

@staticmethod
def cp_from_PT(P: Q_, T: Q_) -> Q_:
    return Q_(WaterProps._PT(P,T).cp, "kJ/kg/K").to("J/kg/K")

@staticmethod
```

```
def mu_from_PT(P: Q_, T: Q_) -> Q_:  return Q_(WaterProps._PT(P,T).mu, "Pa*s")

@staticmethod
def k_from_PT(P: Q_, T: Q_) -> Q_:   return Q_(WaterProps._PT(P,T).k, "W/m/K")

@staticmethod
def rho_from_PT(P: Q_, T: Q_) -> Q_: return Q_(WaterProps._PT(P,T).rho, "kg/m^3")

@staticmethod
def rho_from_Px(P: Q_, x: Q_) -> Q_:
    P_MPa = P.to("megapascal").magnitude
    if x.magnitude <= 0:
        return Q_(IAPWS97(P=P_MPa, x=0.0).rho, "kg/m^3")
    if x.magnitude >= 1:
        return Q_(IAPWS97(P=P_MPa, x=1.0).rho, "kg/m^3")

    rho_f = IAPWS97(P=P_MPa, x=0.0).rho
    rho_g = IAPWS97(P=P_MPa, x=1.0).rho

    v_mix = (1 - x.magnitude) / rho_f + x.magnitude / rho_g
    return Q_(1 / v_mix, "kg/m^3")
```

======================================= FILE: common/results.py =======================

from **future** import annotations from dataclasses import dataclass, field from typing import List, Sequence, Tuple from common.units import Q_ from common.models import GasStream from pathlib import Path import pandas as pd

@dataclass(frozen=True) class CombustionResult: LHV: Q_ Q_in: Q_ T_ad: Q_ flue: GasStream flue_ad: GasStream | None = None fuel_LHV_mass: Q_ | None = None fuel_P_LHV: Q_ | None = None fuel_mass_flow: Q_ | None = None air_mass_flow: Q_ | None = None excess_air_ratio: Q_ | None = None

@dataclass(frozen=True) class StepResult: i: int x: Q_ dx: Q_ gas: object water: object Tgw: Q_ Tww: Q_ UA_prime: Q_ qprime: Q_ boiling: bool h_g: Q_ h_c: Q_ stage_name: str = “” stage_index: int = -1 dP_fric: Q_ = field(default_factory=lambda: Q_(0.0, “kPa”)) dP_minor: Q_ = field(default_factory=lambda: Q_(0.0, “kPa”)) dP_total: Q_ = field(default_factory=lambda: Q_(0.0, “kPa”)) w_dP_fric: Q_ = field(default_factory=lambda: Q_(0.0, “kPa”)) w_dP_minor: Q_ = field(default_factory=lambda: Q_(0.0, “kPa”)) w_dP_tot: Q_ = field(default_factory=lambda: Q_(0.0, “kPa”)) qprime_conv: Q_ = field(default_factory=lambda: Q_(0.0, “W/m”)) qprime_rad: Q_ = field(default_factory=lambda: Q_(0.0, “W/m”))

@dataclass(frozen=True) class StageResult: stage_name: str stage_kind: str steps: Sequence[StepResult] Q_stage: Q_ UA_stage: Q_ dP_stage_fric: Q_ = field(default_factory=lambda: Q_(0.0, “kPa”)) dP_stage_minor: Q_ = field(default_factory=lambda: Q_(0.0, “kPa”)) dP_stage_total: Q_ = field(default_factory=lambda: Q_(0.0, “kPa”)) dP_water_stage_fric: Q_ = field(default_factory=lambda: Q_(0.0, “kPa”)) dP_water_stage_minor: Q_ = field(default_factory=lambda: Q_(0.0, “kPa”)) dP_water_stage_total: Q_ = field(default_factory=lambda: Q_(0.0, “kPa”)) hot_flow_A: Q_ = field(default_factory=lambda: Q_(0.0, “m^2”)) cold_flow_A: Q_ = field(default_factory=lambda: Q_(0.0, “m^2”)) hot_Dh: Q_ = field(default_factory=lambda: Q_(0.0, “m”)) cold_Dh: Q_ = field(default_factory=lambda: Q_(0.0, “m”))

@dataclass(frozen=True) class GlobalProfile: x: List[Q_] dx: List[Q_] gas: List[object] water: List[object] qprime: List[Q_] UA_prime: List[Q_] h_g: List[Q_] h_c: List[Q_] stage_index: List[int] stage_name: List[str] dP_fric: List[Q_] dP_minor: List[Q_] dP_total: List[Q_] w_dP_fric: List[Q_] w_dP_minor: List[Q_] w_dP_tot: List[Q_] stage_results: List[StageResult]

def build_global_profile(stage_results: Sequence[StageResult]) -> GlobalProfile: xs: List[Q_] = [] dxs:

List[Q_] = [] gas: List[object] = [] water: List[object] = [] qprime: List[Q_] = [] UA_prime: List[Q_] = [] h_g: List[Q_] = [] h_c: List[Q_] = [] sidx: List[int] = [] sname: List[str] = [] dP_fric: List[Q_] = [] dP_minor: List[Q_] = [] dP_total: List[Q_] = [] w_dP_fric: List[Q_] = [] w_dP_minor: List[Q_] = [] w_dP_tot: List[Q_] = []

```
for k, sr in enumerate(stage_results):
    for st in sr.steps:
        xs.append(st.x)
        dxs.append(st.dx)
        gas.append(st.gas)
        water.append(st.water)
        qprime.append(st.qprime)
        UA_prime.append(st.UA_prime)
        h_g.append(st.h_g)
        h_c.append(st.h_c)
        sidx.append(k if st.stage_index < 0 else st.stage_index)
        sname.append(sr.stage_name if not st.stage_name else st.stage_name)
        dP_fric.append(st.dP_fric)
        dP_minor.append(st.dP_minor)
        dP_total.append(st.dP_total)
        w_dP_fric.append(st.w_dP_fric)
        w_dP_minor.append(st.w_dP_minor)
        w_dP_tot.append(st.w_dP_tot)

return GlobalProfile(
    x=xs, dx=dxs, gas=gas, water=water,
    qprime=qprime, UA_prime=UA_prime, h_g=h_g, h_c=h_c,
    stage_index=sidx, stage_name=sname,
    dP_fric=dP_fric, dP_minor=dP_minor, dP_total=dP_total,
    w_dP_fric=w_dP_fric, w_dP_minor=w_dP_minor, w_dP_tot=w_dP_tot,
    stage_results=list(stage_results),
)
```

def write_results_csvs(global_profile: GlobalProfile, combustion: CombustionResult | None, outdir: str | Path, run_id: str, drum_pressure: Q_ | None = None, feed_pressure: Q_ | None = None,

) -> Tuple[str, str, str]:

```
from heat.postproc import profile_to_dataframe, summary_from_profile

outdir = Path(outdir)
outdir.mkdir(parents=True, exist_ok=True)

steps_path = outdir / f"{run_id}_steps.csv"
stages_summary_path = outdir / f"{run_id}_stages_summary.csv"
boiler_summary_path = outdir / f"{run_id}_boiler_summary.csv"

df_steps = profile_to_dataframe(
    global_profile,
    remap_water=False,
)
df_steps.set_index("stage_name").to_csv(steps_path)

rows, _, _ = summary_from_profile(
    global_profile,
    combustion=combustion,
```

```
    drum_pressure=drum_pressure
)


df_summary = pd.DataFrame(
    rows,
    columns=[
        "stage_index", "stage_name", "stage_kind",
        "Q_stage[MW]", "UA_stage[MW/K]",

        "gas_in_P[kPa]", "gas_in_T[°C]", "gas_in_h[kJ/kg]",
        "gas_out_P[kPa]", "gas_out_T[°C]", "gas_out_h[kJ/kg]",

        "water_in_P[kPa]", "water_in_T[°C]", "water_in_h[kJ/kg]",
        "water_out_P[kPa]", "water_out_T[°C]", "water_out_h[kJ/kg]",

        "gas_V_avg[m/s]", "water_V_avg[m/s]",

        "ΔP_stage_fric[kPa]", "ΔP_stage_minor[kPa]", "ΔP_stage_total[kPa]",
        "ΔP_water_stage_fric[kPa]", "ΔP_water_stage_minor[kPa]", "ΔP_water_stage_total[kPa]",
        "stack_temperature[°C]",
        "Q_conv_stage[MW]", "Q_rad_stage[MW]",

        "steam_capacity[kg/s]", "steam_capacity[t/h]",

        "η_direct[-]", "η_indirect[-]", "Stack_loss_fraction[-]",
        "Q_total_useful[MW]", "Q_in_total[MW]", "Q_flue_out[MW]","Q_balance_error[MW]",

        "P_LHV[MW]", "LHV_mass[kJ/kg]",
    ],
).set_index("stage_index")


df_stages = df_summary[df_summary["stage_name"] != "TOTAL_BOILER"].copy()
df_boiler = df_summary[df_summary["stage_name"] == "TOTAL_BOILER"].copy()

df_stages = df_stages.set_index("stage_name")

table = pd.DataFrame(
    {
        "name": df_stages.index,
        "kind": df_stages["stage_kind"],
        "gas in pressure[kpa]": df_stages["gas_in_P[kPa]"],
        "gas in temp[°C]": df_stages["gas_in_T[°C]"],
        "gas in enthalpy[kJ/kg]": df_stages["gas_in_h[kJ/kg]"],
        "gas out pressure[kpa]": df_stages["gas_out_P[kPa]"],
        "gas out temp[°C]": df_stages["gas_out_T[°C]"],
        "gas out enthalpy[kJ/kg]": df_stages["gas_out_h[kJ/kg]"],
        "water in temp[°C]": df_stages["water_in_T[°C]"],
        "water in enthalpy[kJ/kg]": df_stages["water_in_h[kJ/kg]"],
        "water in pressure[kpa]": df_stages["water_in_P[kPa]"],
        "water out temp[°C]": df_stages["water_out_T[°C]"],
        "water out enthalpy[kJ/kg]": df_stages["water_out_h[kJ/kg]"],
        "water out pressure[kpa]": df_stages["water_out_P[kPa]"],
```

```
        "gas avg velocity[m/s]": df_stages["gas_V_avg[m/s]"],
        "water avg velocity[m/s]": df_stages["water_V_avg[m/s]"],
        "pressure drop fric[kpa]": df_stages["ΔP_stage_fric[kPa]"],
        "pressure drop minor[kpa]": df_stages["ΔP_stage_minor[kPa]"],
        "pressure drop total[kpa]": df_stages["ΔP_stage_total[kPa]"],
        "water pressure drop fric[kpa]": df_stages["ΔP_water_stage_fric[kPa]"],
        "water pressure drop minor[kpa]": df_stages["ΔP_water_stage_minor[kPa]"],
        "water pressure drop total[kpa]": df_stages["ΔP_water_stage_total[kPa]"],
        "Q conv[MW]": df_stages["Q_conv_stage[MW]"],
        "Q rad[MW]": df_stages["Q_rad_stage[MW]"],
        "Q total[MW]": df_stages["Q_stage[MW]"],
        "UA[MW/K]": df_stages["UA_stage[MW/K]"],
        "steam capacity[t/h]": df_stages["steam_capacity[t/h]"],
    }
).set_index("name")

table = table.drop(columns=["name"], errors="ignore")
table.T.to_csv(stages_summary_path, index_label="name")

if not df_boiler.empty:
    boiler_row = df_boiler.iloc[0].copy()

    try:
        m_fw = global_profile.stage_results[5].steps[0].water.mass_flow.to("kg/s").magnitude
    except Exception:
        m_fw = float("nan")

    boiler_row["feedwater_mass_flow[kg/s]"] = m_fw

    if feed_pressure is not None:
        try:
            boiler_row["feedwater_P[kPa]"] = feed_pressure.to("kPa").magnitude
        except Exception:
            boiler_row["feedwater_P[kPa]"] = ""
    else:
        boiler_row["feedwater_P[kPa]"] = ""

    if drum_pressure is not None:
        try:
            boiler_row["drum_P[kPa]"] = drum_pressure.to("kPa").magnitude
        except Exception:
            boiler_row["drum_P[kPa]"] = ""
    else:
        boiler_row["drum_P[kPa]"] = ""

    if combustion is not None:
        try:
            boiler_row["T_ad[°C]"] = combustion.T_ad.to("degC").magnitude
        except Exception:
            boiler_row["T_ad[°C]"] = ""

        if combustion.fuel_LHV_mass is not None:
            boiler_row["fuel_LHV_mass[kJ/kg]"] = combustion.fuel_LHV_mass.to("kJ/kg").magnitude
        else:
```

```
        boiler_row["fuel_LHV_mass[kJ/kg]"] = ""

    if combustion.fuel_P_LHV is not None:
        boiler_row["fuel_P_LHV[MW]"] = combustion.fuel_P_LHV.to("MW").magnitude
    else:
        boiler_row["fuel_P_LHV[MW]"] = ""

    if combustion.fuel_mass_flow is not None:
        boiler_row["fuel_mass_flow[kg/s]"] = combustion.fuel_mass_flow.to("kg/s").magnitude
    else:
        boiler_row["fuel_mass_flow[kg/s]"] = ""

    if combustion.excess_air_ratio is not None:
        boiler_row["excess_air_ratio[-]"] = combustion.excess_air_ratio.to("").magnitude
    else:
        boiler_row["excess_air_ratio[-]"] = ""

    if combustion.air_mass_flow is not None:
        boiler_row["air_mass_flow[kg/s]"] = combustion.air_mass_flow.to("kg/s").magnitude
    else:
        boiler_row["air_mass_flow[kg/s]"] = ""

else:
    boiler_row["T_ad[°C]"] = ""
    boiler_row["fuel_LHV_mass[kJ/kg]"] = ""
    boiler_row["fuel_P_LHV[MW]"] = ""
    boiler_row["fuel_mass_flow[kg/s]"] = ""
    boiler_row["air_mass_flow[kg/s]"] = ""
    boiler_row["excess_air_ratio[-]"] = ""

boiler_df = pd.DataFrame([boiler_row])

summary_mapping = [
    ("fuel mass flow[kg/s]",            "fuel_mass_flow[kg/s]"),
    ("air flow[kg/s]",                  "air_mass_flow[kg/s]"),
    ("excess air ratio[-]",             "excess_air_ratio[-]"),
    ("feedwater flow[kg/s]",            "feedwater_mass_flow[kg/s]"),
    ("steam capacity[t/h]",             "steam_capacity[t/h]"),
    ("eta direct[-]",                   "η_direct[-]"),
    ("eta indirect[-]",                 "η_indirect[-]"),
    ("Stack loss fraction[-]",         "Stack_loss_fraction[-]"),
    ("Q_flue_out[MW]",                  "Q_flue_out[MW]"),
    ("UA[MW/K]",                        "UA_stage[MW/K]"),
    ("Q_in total[MW]",                  "Q_in_total[MW]"),
    ("Q_useful[MW]",                     "Q_total_useful[MW]"),
    ("Q_balance_error[MW]",                  "Q_balance_error[MW]"),
    ("pressure drop fric total[kPa]",      "ΔP_stage_fric[kPa]"),
    ("pressure drop minor total[kPa]",   "ΔP_stage_minor[kPa]"),
    ("pressure drop total[kPa]",         "ΔP_stage_total[kPa]"),
    ("water pressure drop fric total[kPa]",  "ΔP_water_stage_fric[kPa]"),
    ("water pressure drop minor total[kPa]", "ΔP_water_stage_minor[kPa]"),
    ("water pressure drop total[kPa]",       "ΔP_water_stage_total[kPa]"),
    ("LHV[kJ/kg]",                      "LHV_mass[kJ/kg]"),
    ("P-LHV[MW]",                       "P_LHV[MW]"),
```

```
        ("Tad[°C]",                        "T_ad[°C]"),
        ("stack temperature[°C]",          "stack_temperature[°C]"),
        ("feedwater pressure[kPa]",             "feedwater_P[kPa]"),
        ("drum pressure[kPa]",                  "drum_P[kPa]"),
    ]

    out_row = {}
    row0 = boiler_df.iloc[0]
    for new_name, src_col in summary_mapping:
        out_row[new_name] = row0.get(src_col, "")

    boiler_out_df = pd.DataFrame([out_row]).T
    boiler_out_df = boiler_out_df.reset_index()
    boiler_out_df.columns = ["parameter", "value"]

    boiler_out_df.to_csv(boiler_summary_path, index=False)

else:
    empty_cols = list(df_summary.columns) + [
        "water_mass_flow[kg/s]",
        "T_ad[°C]",
        "fuel_LHV_mass[kJ/kg]",
        "fuel_P_LHV[MW]",
        "fuel_mass_flow[kg/s]",
        "air_mass_flow[kg/s]",
        "excess_air_ratio[-]",
    ]
    pd.DataFrame(columns=empty_cols).to_csv(boiler_summary_path, index=False)

return str(steps_path), str(stages_summary_path), str(boiler_summary_path)
```

== FILE: common/units.py ==
from **future** import annotations import pint

ureg = pint.UnitRegistry() Q_ = ureg.Quantity

== FILE: heat/**init**.py ==

== FILE: heat/gas_htc.py ==
from **future** import annotations from typing import Dict, Any, Tuple import numpy as np from common.units import Q_ from common.models import GasStream from common.props import GasProps from combustion.mass_mole import to_mole

_gas = GasProps(mech_path="config/flue_cantera.yaml", phase="gas_mix")

def cp_gas(g: GasStream) -> Q_: return _gas.cp(g.T, g.P, g.comp or {})

def _gas_partials(g): P = g.P.to("Pa").magnitude Y = {k: v.to(" ").magnitude for k,v in (g.comp or {}).items()} X = to_mole(Y) xH2O = X.get("H2O", 0.0) xCO2 = X.get("CO2", 0.0) return xH2O * P, xCO2 * P

_A = np.array([0.434, 0.313, 0.180, 0.073]) _K = np.array([0.0, 2.3, 11.6, 30.4])

def *mean_beam_length(spec: dict)* -> *Q*: if "rad_Lb" in spec: return spec["rad_Lb"].to("m") Dh = spec["hot_Dh"].to("m") return (0.9 * Dh).to("m")

def emissivity(T_K: float, pH2O_Pa: float, pCO2_Pa: float, L_m: float, *, Texp: float = 0.65) -> float: T = float(np.clip(T_K, 300.0, 3000.0)) scale_T = (T / 1000.0) ** Texp

```
p_ratio = (pH2O_Pa + pCO2_Pa) / 101325.0
tau = (_K * scale_T) * p_ratio * L_m

eps = 1.0 - float(np.sum(_A * np.exp(-tau)))
return float(np.clip(eps, 0.0, 1.0))
```

def h_rad(Tfilm_K: float, eps_g: float, F: float = 1.0) -> float: sigma = 5.670374419e-8 return 4.0 * sigma * F * eps_g * Tfilm_K**3

def h_gas_rad_smith(T_K: float, pH2O_Pa: float, pCO2_Pa: float, L_m: float, Twall_K: float, F: float = 1.0, , *Texp: float = 0.65) -> float: eps = emissivity(T_K, pH2O_Pa, pCO2_Pa, L_m, Texp=Texp) Tfilm = 0.5* (T_K + Twall_K) return h_rad(Tfilm, eps, F)

def *h_rad(g: GasStream, spec: Dict[str, Any], Tgw: Q*) -> Q_: Tg = g.T pH2O, pCO2 = _gas_partials(g) Lb = *mean_beam_length(spec).to("m").magnitude F = float(spec.get("rad_F", 1.0)) Texp = float(spec.get("rad_Texp", 0.65)) h_val = h_gas_rad_smith(Tg.to("K").magnitude, pH2O, pCO2, Lb, Tgw.to("K").magnitude, F, Texp=Texp) return Q*(max(h_val, 0.0), "W/m^2/K")

def *reynolds(rho: Q*, V: Q_, D: Q_, mu: Q_) -> float: Re = (rho * V * D / mu).to("").magnitude return max(Re, 1e-12)

def *prandtl(cp: Q*, mu: Q_, k: Q_) -> float: Pr = (cp * mu / k).to("").magnitude return max(Pr, 1e-12)

def *vel_internal(g: GasStream, A: Q*) -> Q_: rho = _gas.rho(g.T, g.P, g.comp) return (g.mass_flow / (rho * A)).to("m/s")

def *vel_external(g: GasStream, A_bulk: Q*, umax_factor: Q_ | float | None) -> Q_: rho = _gas.rho(g.T, g.P, g.comp) V_bulk = (g.mass_flow / (rho * A_bulk)).to("m/s") if umax_factor is None: return V_bulk f = umax_factor if isinstance(umax_factor, (int, float)) else umax_factor.to(" ").magnitude return (max(f, 1.0) * V_bulk).to("m/s")

def *nu_internal(Re: float, Pr: float, D: Q*, L: Q_) -> float: if Re < 2300.0: Gz = Re * Pr * (D.to("m").magnitude / max(L.to("m").magnitude, 1e-12)) return 3.66 + 0.0668 * Gz / (1.0 + 0.04 * (Gz ** (2.0/3.0))) f = (0.79 * np.log(Re) - 1.64) ** -2 num = (f/8.0) * (Re - 1000.0) * Pr den = 1.0 + 12.7 * np.sqrt(f/8.0) * ((Pr ** (2.0/3.0)) - 1.0) return max(num / max(den, 1e-12), 1e-12)

def _nu_churchill_bernstein(Re: float, Pr: float) -> float: a = 0.3 b = (0.62 * Re**0.5 * Pr**(1/3.0)) / ((1.0 + (0.4/Pr)**(2.0/3.0))** 0.25) c = (1.0 + (Re / 282000.0) ** (5.0/8.0)) ** (4.0/5.0) return max(a + b * c, 1e-12)

def _nu_zukauskas(Re: float, Pr: float, arrangement: str) -> float | None: bands = [(1e3, 2e3, {"inline": (0.90, 0.40), "staggered": (1.04, 0.40)}), (2e3, 4e3, {"inline": (0.52, 0.50), "staggered": (0.71, 0.50)}), (4e3, 1e5, {"inline": (0.27, 0.63), "staggered": (0.35, 0.60)}), (1e5, 2e6, {"inline": (0.021, 0.84), "staggered": (0.022, 0.84)}),] C = m = None arr = "staggered" if arrangement == "staggered" else "inline" for Re_min, Re_max, table in bands: if Re_min <= Re < Re_max: C, m = table[arr] break if C is None: return None n = 0.36 if Pr <= 10.0 else 0.25 return max(C * (Re**m) * (Pr**n), 1e-12)

def *h_from_Nu(Nu: float, k: Q*, D: Q_) -> Q_: return (Q_(Nu, " ") * k / D).to("W/m^2/K")

def *h_conv_internal(g: GasStream, spec: dict) -> Q*: D = spec["inner_diameter"].to("m") L = spec["inner_length"].to("m") A = spec["hot_flow_A"].to("m^2")

```
V   = _vel_internal(g, A)
rho = _gas.rho(g.T, g.P, g.comp)
mu  = _gas.mu(g.T, g.P, g.comp)
k   = _gas.k(g.T, g.P, g.comp)
cp  = _gas.cp(g.T, g.P, g.comp)

Re = _reynolds(rho, V, D, mu)
Pr = _prandtl(cp, mu, k)
```

```
Nu = _nu_internal(Re, Pr, D, L)
return _h_from_Nu(Nu, k, D)
```

def *h_conv_economiser_external(g: GasStream, spec: dict) -> Q*: D = spec[“outer_diameter”].to(“m”) A_bulk = spec[“hot_flow_A”].to(“m^2”) umax = spec.get(“umax_factor”, None)

```
V   = _vel_external(g, A_bulk, umax)
rho = _gas.rho(g.T, g.P, g.comp)
mu  = _gas.mu(g.T, g.P, g.comp)
k   = _gas.k(g.T, g.P, g.comp)
cp  = _gas.cp(g.T, g.P, g.comp)

Re = _reynolds(rho, V, D, mu)
Pr = _prandtl(cp, mu, k)

arrangement = spec.get("arrangement", "inline")
Nu_z = _nu_zukauskas(Re, Pr, arrangement)
if Nu_z is None:
    Nu_z = _nu_churchill_bernstein(Re, Pr)
return _h_from_Nu(Nu_z, k, D)
```

def gas_htc_parts(g: GasStream, spec: dict, Tgw: Q__, *, stage_kind: str | None = None) -> Tuple[Q__, Q__]: kind = (stage_kind or spec.get(“stage_kind”) or “single_tube”).lower()

```
if kind in ("single_tube", "reversal_chamber", "tube_bank"):
    h_conv = _h_conv_internal(g, spec)
elif kind == "economiser":
    h_conv = _h_conv_economiser_external(g, spec)
else:
    h_conv = _h_conv_internal(g, spec)

h_rad = _h_rad(g, spec, Tgw)
return h_conv.to("W/m^2/K"), h_rad.to("W/m^2/K")
```

def gas_htc(g: GasStream, spec: dict, Tgw: Q__, *, stage_kind: str | None = None) -> Q__: h_conv, h_rad = gas_htc_parts(g, spec, Tgw, stage_kind=stage_kind) return (h_conv + h_rad).to(“W/m^2/K”)

====================================== FILE: heat/geometry.py ======================
from math import pi from dataclasses import replace from typing import List from common.models import HXStage, Drum from common.units import Q__

class GeometryBuilder: def **init**(self, drum: Drum): self.drum = drum

```
def enrich(self, stages: List[HXStage]) -> List[HXStage]:
    out: List[HXStage] = []
    for stg in stages:
        if stg.kind == "single_tube":
            spec = dict(stg.spec)
            Di_t = spec["inner_diameter"].to("m")
            t = spec["wall_t"].to("m")
            Do_t = (Di_t + 2*t).to("m")
            Di_drum = self.drum.Di.to("m")

            A_drum = (pi * (Di_drum/2)**2).to("m^2")
            A_tube_out = (pi * (Do_t/2)**2).to("m^2")

            spec["outer_diameter"] = Do_t
```

```
        spec["roughness_cold_surface"] = spec["roughness_out"]

        hot_wet_P = (pi * Di_t).to("m")
        hot_flow_A = (pi * (Di_t/2)**2).to("m^2")
        hot_Dh = (4 * hot_flow_A / hot_wet_P).to("m")

        cold_wet_P = (pi * Do_t).to("m")
        cold_flow_A = (A_drum - A_tube_out).to("m^2")
        cold_Dh = (4 * cold_flow_A / cold_wet_P).to("m")

        spec.update({
            "hot_wet_P": hot_wet_P, "hot_flow_A": hot_flow_A, "hot_Dh": hot_Dh,
            "cold_wet_P": cold_wet_P, "cold_flow_A": cold_flow_A, "cold_Dh": cold_Dh
        })
        out.append(replace(stg, spec=spec))

    elif stg.kind == "tube_bank":
        spec = dict(stg.spec)
        Di_t = spec["inner_diameter"].to("m")
        t = spec["wall_t"].to("m")
        Nt = spec["tubes_number"].to("")
        Do_t = (Di_t + 2*t).to("m")

        Ds = self.drum.Di.to("m")
        B  = spec["baffle_spacing"].to("m")
        ST = spec["ST"].to("m")
        SL = spec["SL"].to("m")

        FAR_T = (1 - (Do_t / ST)).to("")
        A_gross = (Ds * B).to("m^2")
        A_cross = (A_gross * FAR_T).to("m^2")
        arr = (spec.get("arrangement","inline") or "inline").lower()
        if arr == "staggered":
            FAR_L = (1 - (0.5 * Do_t / SL)).to("")
            umax = (ST / (ST - Do_t)) * ((SL / (SL - 0.5*Do_t)) ** 0.5)
        else:
            umax = (ST / (ST - Do_t))
        spec["umax_factor"] = umax.to("dimensionless")

        spec["roughness_cold_surface"] = spec["roughness_out"]
        spec["outer_diameter"] = Do_t


        A_drum = (pi * (Ds/2)**2).to("m^2")
        A_tube_out = (pi * (Do_t/2)**2).to("m^2")

        cold_wet_P = (Nt * pi * Do_t).to("m")
        cold_flow_A = A_cross
        cold_Dh = (4 * cold_flow_A / cold_wet_P).to("m")

        hot_wet_P = (Nt * pi * Di_t).to("m")
        hot_flow_A = (Nt * (pi * (Di_t/2)**2)).to("m^2")
        hot_Dh = (4 * hot_flow_A / hot_wet_P).to("m")
```

```
        spec.update({
            "hot_wet_P": hot_wet_P, "hot_flow_A": hot_flow_A, "hot_Dh": hot_Dh,
            "cold_wet_P": cold_wet_P, "cold_flow_A": cold_flow_A, "cold_Dh": cold_Dh
        })
        out.append(replace(stg, spec=spec))


    elif stg.kind == "reversal_chamber":
        spec = dict(stg.spec)

        Di_t = spec["inner_diameter"].to("m")
        t = spec["wall_t"].to("m")
        Do_t = (Di_t + 2*t).to("m")

        Di_drum = self.drum.Di.to("m")
        A_drum = (pi * (Di_drum/2)**2).to("m^2")
        A_tube_out = (pi * (Do_t/2)**2).to("m^2")

        spec["outer_diameter"] = Do_t
        spec["roughness_cold_surface"] = spec["roughness_out"]

        hot_wet_P = (pi * Di_t).to("m")
        hot_flow_A = (pi * (Di_t/2)**2).to("m^2")
        hot_Dh = (4 * hot_flow_A / hot_wet_P).to("m")

        cold_wet_P = (pi * Do_t).to("m")
        cold_flow_A = (A_drum - A_tube_out).to("m^2")
        cold_Dh = (4 * cold_flow_A / cold_wet_P).to("m")

        spec.update({
            "hot_wet_P": hot_wet_P, "hot_flow_A": hot_flow_A, "hot_Dh": hot_Dh,
            "cold_wet_P": cold_wet_P, "cold_flow_A": cold_flow_A, "cold_Dh": cold_Dh
        })
        out.append(replace(stg, spec=spec))

    elif stg.kind == "economiser":
        spec = dict(stg.spec)

        Di_t = spec["inner_diameter"].to("m")
        t    = spec["wall_t"].to("m")
        Do_t = (Di_t + 2*t).to("m")

        n_tubes_q  = spec.get("n_tubes", None)
        tube_len_q = spec.get("tube_length", None)

        if n_tubes_q is None or tube_len_q is None:
            raise KeyError(f"{stg.name}: economiser requires n_tubes, tube_length")

        N_tubes  = int(round(n_tubes_q.to("").magnitude))
        tube_len = tube_len_q.to("m")

        N_tubes = max(N_tubes, 1)

        Ds = spec["shell_inner_diameter"].to("m")
```

```
            B  = spec["baffle_spacing"].to("m")
            ST = spec["ST"].to("m")
            SL = spec["SL"].to("m")

            A_gross = (Ds * B).to("m^2")
            FAR_T   = (1 - (Do_t / ST)).to("")
            A_cross = (A_gross * FAR_T).to("m^2")

            arr = (spec.get("arrangement", "inline") or "inline").lower()
            if arr == "staggered":
                umax = (ST / (ST - Do_t)) * ((SL / (SL - 0.5*Do_t)) ** 0.5)
            else:
                umax = (ST / (ST - Do_t))
            spec["umax_factor"] = umax.to("dimensionless")

            spec["outer_diameter"] = Do_t
            spec["roughness_cold_surface"] = spec["roughness_in"]

            hot_wet_P  = (N_tubes * pi * Do_t).to("m")
            hot_flow_A = A_cross
            hot_Dh     = (4 * hot_flow_A / hot_wet_P).to("m")

            cold_wet_P  = (N_tubes * pi * Di_t).to("m")
            cold_flow_A = (N_tubes * (pi * (Di_t/2)**2)).to("m^2")
            cold_Dh     = (4 * cold_flow_A / cold_wet_P).to("m")

            N_rows_q = spec.get("N_rows", None)
            N_rows = int(round(N_rows_q.to("").magnitude)) if N_rows_q is not None else 1
            N_rows = max(N_rows, 1)
            hot_flow_L = spec.get("hot_flow_length", (Q_(N_rows, "") * SL)).to("m")

            spec.update({
                "hot_wet_P": hot_wet_P, "hot_flow_A": hot_flow_A, "hot_Dh": hot_Dh,
                "cold_wet_P": cold_wet_P, "cold_flow_A": cold_flow_A, "cold_Dh": cold_Dh,
                "tube_length": tube_len,
                "hot_flow_length": hot_flow_L,
                "water_dx_factor": (tube_len / hot_flow_L).to("dimensionless"),
            })
            out.append(replace(stg, spec=spec))
        else:
            raise ValueError("unknown stage kind")
    return out
```

==================================== FILE: heat/physics.py ====================
from math import pi, log from common.units import Q_

def _nt(spec) -> float: Nt = spec.get("tubes_number", None) return (Nt.to("").magnitude if Nt is not None else 1.0)

def fouling_resistances(spec: dict) -> tuple[Q_, Q_]: di = spec["inner_diameter"].to("m") do = spec["outer_diameter"].to("m") tfi = spec["foul_t_in"].to("m") kfi = spec["foul_k_in"].to("W/m/K") tfo = spec["foul_t_out"].to("m") kfo = spec["foul_k_out"].to("W/m/K") Di_new = di - 2*tfi Rfi = log(di/Di_new)/(2*pi*kfi) do_new = do + 2*tfo Rfo = log(do_new/do)/(2*pi*kfo)

```
Nt = _nt(spec)
return (Rfi / Nt).to("K*m/W"), (Rfo / Nt).to("K*m/W")
```

def wall_resistance(spec: dict) -> Q_: di = spec["inner_diameter"].to("m") do = spec["outer_diameter"].to("m") k = spec["wall_k"].to("W/m/K") R = log(do/di) / (2pik) Nt = _nt(spec) return (R / Nt).to("K*m/W") ==================================== FILE: heat/postproc.py ====================================
from **future** import annotations import pandas as pd from common.results import GlobalProfile, CombustionResult from common.props import WaterProps, GasProps from common.units import Q_ from heat.gas_htc import emissivity from combustion.mass_mole import to_mole import cantera as ct from common.units import Q_ from common.constants import T_ref, P_ref

_gas = GasProps()

def _mag_or_nan(q, unit): return q.to(unit).magnitude if q is not None else float("nan")

def flue_sensible_to_ref(g) -> Q_: h_sens = _gas.h_sensible(g.T, g.P, g.comp).to("J/kg") return (g.mass_flow * h_sens).to("MW")

def profile_to_dataframe(gp: "GlobalProfile", *, remap_water: bool = True) -> "pd.DataFrame": stage_ranges: dict[int, tuple[int, int]] = {} for i in range(len(gp.x)): k = gp.stage_index[i] if k not in stage_ranges: stage_ranges[k] = [i, i] else: stage_ranges[k][1] = i stage_ranges = {k: (v[0], v[1]) for k, v in stage_ranges.items()}

```
stage_offsets: dict[int, Q_] = {}
offset = Q_(0.0, "m")
for k, sr in enumerate(gp.stage_results):
    stage_offsets[k] = offset
    if sr.steps:
        last = sr.steps[-1]
        offset = (offset + last.x + last.dx).to("m")

rows = []
for i in range(len(gp.x)):
    g = gp.gas[i]

    k_stage = gp.stage_index[i]
    disable_water_hydraulics = (k_stage <= 4)
    sr_stage = gp.stage_results[k_stage]
    A_hot = sr_stage.hot_flow_A
    A_cold = sr_stage.cold_flow_A
    Dh_hot = sr_stage.hot_Dh
    Dh_cold = sr_stage.cold_Dh

    if remap_water:
        i0, iN = stage_ranges[gp.stage_index[i]]
        i_local = i - i0
        j = iN - i_local
    else:
        j = i

    w = gp.water[j]

    xq = WaterProps.quality_from_Ph(w.P, w.h)
    Two_phase = xq is not None

    if Two_phase:
        Tw = WaterProps.Tsat(w.P)
        w_cp = w_mu = w_k = None
        w_rho = WaterProps.rho_from_Px(w.P, xq) if xq is not None else None
```

```
else:
    Tw   = WaterProps.T_from_Ph(w.P, w.h)
    w_cp = WaterProps.cp_from_Ph(w.P, w.h)
    w_mu = WaterProps.mu_from_Ph(w.P, w.h)
    w_k  = WaterProps.k_from_Ph(w.P, w.h)
    w_rho = WaterProps.rho_from_Ph(w.P, w.h)

g_h   = _gas.h_sensible(g.T, g.P, g.comp)
g_mu  = _gas.mu(g.T, g.P, g.comp)
g_rho = _gas.rho(g.T, g.P, g.comp)

gas_V = (g.mass_flow / (g_rho * A_hot)).to("m/s")
Re_gas = (g_rho * gas_V * Dh_hot / g_mu).to("").magnitude

if w_rho is not None and A_cold is not None:
    water_V = (w.mass_flow / (w_rho * A_cold)).to("m/s")
else:
    water_V = None

if w_rho is not None and w_mu is not None and water_V is not None:
    Re_water = (w_rho * water_V * Dh_cold / w_mu).to("").magnitude
else:
    Re_water = float("nan")

if disable_water_hydraulics:
    water_V = None
    Re_water = float("nan")
    w_cp = w_mu = w_k = w_rho = None
    w_dP_fric = w_dP_minor = w_dP_tot = float("nan")
else:
    if Two_phase:
        Tw = WaterProps.Tsat(w.P)
        w_cp = w_mu = w_k = None
        w_rho = WaterProps.rho_from_Px(w.P, xq) if xq is not None else None
    else:
        Tw   = WaterProps.T_from_Ph(w.P, w.h)
        w_cp = WaterProps.cp_from_Ph(w.P, w.h)
        w_mu = WaterProps.mu_from_Ph(w.P, w.h)
        w_k  = WaterProps.k_from_Ph(w.P, w.h)
        w_rho = WaterProps.rho_from_Ph(w.P, w.h)

    if w_rho is not None and A_cold is not None:
        water_V = (w.mass_flow / (w_rho * A_cold)).to("m/s")
    else:
        water_V = None

    if w_rho is not None and w_mu is not None and water_V is not None:
        Re_water = (w_rho * water_V * Dh_cold / w_mu).to("").magnitude
    else:
        Re_water = float("nan")

    w_dP_fric = gp.w_dP_fric[i].to("Pa").magnitude
    w_dP_minor = gp.w_dP_minor[i].to("Pa").magnitude
    w_dP_tot = gp.w_dP_tot[i].to("Pa").magnitude
```

```
Y = {sp: float(q.to("").magnitude) for sp, q in (g.comp or {}).items()}
X = to_mole(Y)

xH2O = X.get("H2O", 0.0)
xCO2 = X.get("CO2", 0.0)

P_Pa = g.P.to("Pa").magnitude
pH2O = xH2O * P_Pa
pCO2 = xCO2 * P_Pa

Lb_m = (0.9 * Dh_hot).to("m").magnitude
gas_eps = emissivity(
    g.T.to("K").magnitude,
    pH2O,
    pCO2,
    Lb_m,
)

x_local = gp.x[i].to("m")
x_global = (stage_offsets[k_stage] + x_local).to("m")

row = {
    "stage_name": gp.stage_name[i],
    "i": i,
    "x[m]": x_global.magnitude,
    "dx[m]": gp.dx[i].to("m").magnitude,
    "qprime[MW/m]": gp.qprime[i].to("MW/m").magnitude,
    "UA_prime[MW/K/m]": gp.UA_prime[i].to("MW/K/m").magnitude,
    "gas_P[kPa]": g.P.to("kPa").magnitude,
    "gas_T[°C]": g.T.to("degC").magnitude,
    "gas_h[kJ/kg]": g_h.to("kJ/kg").magnitude,
    "water_P[kPa]": w.P.to("kPa").magnitude,
    "water_T[°C]": Tw.to("degC").magnitude,
    "water_h[kJ/kg]": w.h.to("kJ/kg").magnitude,
    "gas_eps[-]": gas_eps,
    "water_x[-]": _mag_or_nan(xq, ""),
    "boiling": "true" if xq is not None else "false",
    "gas_V[m/s]": gas_V.to("m/s").magnitude,
    "Re_gas[-]": Re_gas,
    "h_gas[W/m^2/K]": gp.h_g[i].to("W/m^2/K").magnitude,
   "water_V[m/s]": (_mag_or_nan(water_V, "m/s") if isinstance(water_V, Q_) else float("nan")),
    "Re_water[-]": Re_water,
    "h_water[W/m^2/K]": gp.h_c[i].to("W/m^2/K").magnitude,
    "dP_fric[kPa]": gp.dP_fric[i].to("kPa").magnitude,
    "dP_minor[kPa]": gp.dP_minor[i].to("kPa").magnitude,
    "dP_total[kPa]": gp.dP_total[i].to("kPa").magnitude,
  "water_dP_fric[kPa]":  (w_dP_fric if disable_water_hydraulics else gp.w_dP_fric[i].to("kPa").magnitude),
  "water_dP_minor[kPa]": (w_dP_minor if disable_water_hydraulics else gp.w_dP_minor[i].to("kPa").magnitude),
  "water_dP_total[kPa]": (w_dP_tot if disable_water_hydraulics else gp.w_dP_tot[i].to("kPa").magnitude),
    "water_cp[kJ/kg/K]": _mag_or_nan(w_cp, "kJ/kg/K"),
    "water_mu[Pa*s]": _mag_or_nan(w_mu, "Pa*s"),
    "water_k[W/m/K]": _mag_or_nan(w_k, "W/m/K"),
    "water_rho[kg/m^3]": _mag_or_nan(w_rho, "kg/m^3"),
```

```
    }

    rows.append(row)

return pd.DataFrame(rows)
```

def summary_from_profile(gp: “GlobalProfile”, combustion: CombustionResult | None = None, drum_pressure: Q_ | None = None) -> tuple[list[dict], float, float]: rows = [] Q_total = 0.0 UA_total = 0.0 Q_total_conv = 0.0 Q_total_rad = 0.0 dP_total_fric = 0.0 dP_total_minor = 0.0 dP_total_total = 0.0 w_dP_tot_fric = 0.0 w_dP_tot_minor = 0.0 w_dP_tot_total = 0.0 stack_T_C = None feedwater_mdot_kg_s = None circulation_mdot_kg_s = None flue_mdot_kg_s = None boiler_water_in_P_kPa = None boiler_water_in_T_C = None boiler_water_out_T_C = None boiler_water_Tsat_C = None econ_out_h_Jkg = None feedwater_mdot_q = None

```
import itertools
for k, grp in itertools.groupby(range(len(gp.x)), key=lambda i: gp.stage_index[i]):
    disable_water_hydraulics = (k <= 4)
    idxs = list(grp)
    name = gp.stage_name[idxs[0]]
    sr_stage = gp.stage_results[k]

    A_hot = sr_stage.hot_flow_A
    A_cold = sr_stage.cold_flow_A

    gas_V_sum = 0.0
    water_V_sum = 0.0
    n_steps = len(idxs)

    for i in idxs:
        g = gp.gas[i]
        w = gp.water[i]

        g_rho = _gas.rho(g.T, g.P, g.comp)
        gas_V = (g.mass_flow / (g_rho * A_hot)).to("m/s").magnitude
        gas_V_sum += gas_V

        if A_cold is not None:
            w_rho = WaterProps.rho_from_Ph(w.P, w.h)
            water_V = (w.mass_flow / (w_rho * A_cold)).to("m/s").magnitude
            water_V_sum += water_V

    gas_V_avg = gas_V_sum / max(n_steps, 1)
    if disable_water_hydraulics or A_cold is None:
        water_V_avg = float("nan")
    else:
        water_V_avg = water_V_sum / max(n_steps, 1)

    Q_stage = sum((gp.qprime[i] * gp.dx[i]).to("MW").magnitude for i in idxs)
    UA_stage = sum((gp.UA_prime[i] * gp.dx[i]).to("MW/K").magnitude for i in idxs)

    Q_stage_conv = sum((st.qprime_conv * st.dx).to("MW").magnitude for st in sr_stage.steps)
    Q_stage_rad  = sum((st.qprime_rad  * st.dx).to("MW").magnitude for st in sr_stage.steps)

    dP_fric = sum(gp.dP_fric[i].to("kPa").magnitude for i in idxs)
    dP_minor = sum(gp.dP_minor[i].to("kPa").magnitude for i in idxs)
```

```
dP_total = sum(gp.dP_total[i].to("kPa").magnitude for i in idxs)

w_dP_fric = sum(gp.w_dP_fric[i].to("kPa").magnitude for i in idxs)
w_dP_minor = sum(gp.w_dP_minor[i].to("kPa").magnitude for i in idxs)
w_dP_tot = sum(gp.w_dP_tot[i].to("kPa").magnitude for i in idxs)

g_in  = gp.gas[idxs[0]]
g_out = gp.gas[idxs[-1]]

gas_in_T  = g_in.T.to("degC").magnitude
gas_out_T = g_out.T.to("degC").magnitude

gas_in_P  = g_in.P.to("kPa").magnitude
gas_out_P = g_out.P.to("kPa").magnitude

gas_in_h  = _gas.h_sensible(g_in.T,  g_in.P,  g_in.comp).to("kJ/kg").magnitude
gas_out_h = _gas.h_sensible(g_out.T, g_out.P, g_out.comp).to("kJ/kg").magnitude

w_in  = gp.water[idxs[0]]
w_out = gp.water[idxs[-1]]

water_in_h  = w_in.h.to("kJ/kg").magnitude
water_out_h = w_out.h.to("kJ/kg").magnitude

water_in_P  = w_in.P.to("kPa").magnitude
water_out_P = w_out.P.to("kPa").magnitude

water_in_T  = WaterProps.T_from_Ph(w_in.P,  w_in.h).to("degC").magnitude
water_out_T = WaterProps.T_from_Ph(w_out.P, w_out.h).to("degC").magnitude

if flue_mdot_kg_s is None:
    flue_mdot_kg_s = g_in.mass_flow.to("kg/s").magnitude

if k == 0:
    boiler_water_out_T_C = water_out_T

if k == len(gp.stage_results) - 1:
    boiler_water_in_T_C = water_in_T
    boiler_water_in_P_kPa = water_in_P

    try:
        feedwater_mdot_q = gp.stage_results[k].steps[0].water.mass_flow.to("kg/s")
        feedwater_mdot_kg_s = feedwater_mdot_q.magnitude
    except Exception:
        feedwater_mdot_q = None
        feedwater_mdot_kg_s = None

    try:
        econ_out_h_Jkg = w_out.h.to("J/kg")
    except Exception:
        econ_out_h_Jkg = None
```

```
row = {
    "stage_index": k,
    "stage_name": name,
    "stage_kind": gp.stage_results[k].stage_kind,
    "Q_stage[MW]": Q_stage,
    "UA_stage[MW/K]": UA_stage,
    "gas_V_avg[m/s]": gas_V_avg,
    "water_V_avg[m/s]": water_V_avg,
    "gas_in_P[kPa]": gas_in_P,
    "gas_in_T[°C]": gas_in_T,
    "gas_in_h[kJ/kg]": gas_in_h,
    "gas_out_P[kPa]": gas_out_P,
    "gas_out_T[°C]": gas_out_T,
    "gas_out_h[kJ/kg]": gas_out_h,
    "water_in_P[kPa]": water_in_P,
    "water_in_T[°C]": water_in_T,
    "water_in_h[kJ/kg]": water_in_h,
    "water_out_P[kPa]": water_out_P,
    "water_out_T[°C]": water_out_T,
    "water_out_h[kJ/kg]": water_out_h,
    "ΔP_stage_fric[kPa]": dP_fric,
    "ΔP_stage_minor[kPa]": dP_minor,
    "ΔP_stage_total[kPa]": dP_total,
    "ΔP_water_stage_fric[kPa]": (float("nan") if disable_water_hydraulics else w_dP_fric),
    "ΔP_water_stage_minor[kPa]": (float("nan") if disable_water_hydraulics else w_dP_minor),
    "ΔP_water_stage_total[kPa]": (float("nan") if disable_water_hydraulics else w_dP_tot),
    "Q_conv_stage[MW]": Q_stage_conv,
    "Q_rad_stage[MW]": Q_stage_rad,
    "steam_capacity[kg/s]": "",
    "steam_capacity[t/h]": "",
    "η_direct[-]": "",
    "η_indirect[-]": "",
    "Q_total_useful[MW]": "",
    "Q_in_total[MW]": "",
    "P_LHV[MW]": "",
    "LHV_mass[kJ/kg]": "",
    "flue_mdot[kg/s]": "",
    "boiler_water_in_T[°C]": "",
    "boiler_water_out_T[°C]": "",
    "boiler_water_P[kPa]": "",
    "boiler_water_Tsat[°C]": "",
}

rows.append(row)

Q_total += Q_stage
UA_total += UA_stage
Q_total_conv += Q_stage_conv
Q_total_rad  += Q_stage_rad
dP_total_fric  += dP_fric
dP_total_minor += dP_minor
dP_total_total += dP_total
w_dP_tot_fric  += w_dP_fric
w_dP_tot_minor += w_dP_minor
```

```
    w_dP_tot_total += w_dP_tot
    stack_T_C = gas_out_T

steam_capacity_total_kg_s = None
steam_capacity_total_tph = None

P_for_evap: Q_ | None = drum_pressure
if P_for_evap is None and boiler_water_in_P_kPa is not None:
    P_for_evap = Q_(boiler_water_in_P_kPa, "kPa")

if P_for_evap is not None:
    P_q = P_for_evap.to("Pa")
    boiler_water_Tsat_C = WaterProps.Tsat(P_q).to("degC").magnitude

    hf = WaterProps.h_f(P_q).to("J/kg")
    hg = WaterProps.h_g(P_q).to("J/kg")

    evap_stage_names = {f"HX_{i}" for i in range(1, 6)}
    Q_evap_W = 0.0
    for r in rows:
      if r.get("stage_name") in evap_stage_names and isinstance(r.get("Q_stage[MW]"), (int, float)):
            Q_evap_W += Q_(r["Q_stage[MW]"], "MW").to("W").magnitude
    Q_evap = Q_(Q_evap_W, "W")

    x_out = 1.0
    h_s = (hf + Q_(x_out, "") * (hg - hf)).to("J/kg")
    denom = (h_s - hf).to("J/kg")

    if denom.magnitude <= 0:
        steam_capacity_total_kg_s = None
        steam_capacity_total_tph = None
    elif (feedwater_mdot_q is None) or (econ_out_h_Jkg is None):
        steam_capacity_total_kg_s = None
        steam_capacity_total_tph = None
    else:
        Q_sens = (feedwater_mdot_q * (hf - econ_out_h_Jkg)).to("W")
        if Q_sens.magnitude < 0:
            Q_sens = Q_(0.0, "W")

        Q_evap_net = (Q_evap - Q_sens).to("W")
        if Q_evap_net.magnitude < 0:
            Q_evap_net = Q_(0.0, "W")

        m_s_q = (Q_evap_net / denom).to("kg/s")
        steam_capacity_total_kg_s = m_s_q.magnitude
        steam_capacity_total_tph = m_s_q.to("tonne/hour").magnitude

        Q_evap_pos = max(Q_evap.to("W").magnitude, 1e-12)

        for r in rows:
        if r.get("stage_name") in evap_stage_names and isinstance(r.get("Q_stage[MW]"), (int, float)):
                Q_stage_W = Q_(r["Q_stage[MW]"], "MW").to("W")

                frac = Q_stage_W.to("W").magnitude / Q_evap_pos
```

```
                frac = max(0.0, min(1.0, frac))

                Q_stage_eff = (Q_stage_W - Q_sens * Q_(frac, "")).to("W")
                if Q_stage_eff.magnitude < 0:
                    Q_stage_eff = Q_(0.0, "W")

                m_s_stage = (Q_stage_eff / denom).to("kg/s")
                r["steam_capacity[kg/s]"] = m_s_stage.magnitude
                r["steam_capacity[t/h]"]  = m_s_stage.to("tonne/hour").magnitude

    if steam_capacity_total_kg_s is not None:
       steam_capacity_total_tph = Q_(steam_capacity_total_kg_s, "kg/s").to("tonne/hour").magnitude
    else:
        steam_capacity_total_tph = None

Q_useful_hx = Q_total
Q_in_total = None
P_LHV_W = None
LHV_mass_kJkg = None
eta_direct = None
eta_indirect = None
Stack_loss_fraction = None
Q_flue_out_MW = None

Q_useful = Q_useful_hx

if combustion is not None:
    # "Declared" input from combustion model (may not match postproc reference basis)
    Q_in_declared = combustion.Q_in.to("MW").magnitude

    if combustion.fuel_P_LHV is not None:
        P_LHV_W = combustion.fuel_P_LHV.to("MW").magnitude
    else:
        P_LHV_W = combustion.LHV.to("MW").magnitude

    if combustion.fuel_LHV_mass is not None:
        LHV_mass_kJkg = combustion.fuel_LHV_mass.to("kJ/kg").magnitude

    # Stack sensible loss to reference
    try:
        g_stack = gp.gas[-1]
        Q_flue_out_MW = flue_sensible_to_ref(g_stack).to("MW").magnitude
    except Exception:
        Q_flue_out_MW = None

    # ---- ENFORCE: Q_in_used = Q_useful + Q_flue_out ----
    if Q_flue_out_MW is not None:
        Q_in_used = Q_useful + Q_flue_out_MW
    else:
        # If we can't compute flue loss, fall back to declared input
        Q_in_used = Q_in_declared

    # Report balance error against the declared combustion input (diagnostic)
    if (Q_in_declared is not None) and (Q_flue_out_MW is not None):
```

```
        Q_balance_err_MW = Q_in_declared - (Q_useful + Q_flue_out_MW)
    else:
        Q_balance_err_MW = None

    # Efficiencies computed from enforced balance => ALWAYS equal
    if Q_in_used and Q_in_used > 0.0 and Q_flue_out_MW is not None:
        eta_direct = Q_useful / Q_in_used
        Stack_loss_fraction = Q_flue_out_MW / Q_in_used
        eta_indirect = 1.0 - Stack_loss_fraction
    elif Q_in_used and Q_in_used > 0.0:
        eta_direct = Q_useful / Q_in_used
        eta_indirect = eta_direct
        Stack_loss_fraction = None

    # overwrite the variable used downstream for reporting
    Q_in_total = Q_in_used


total_row = {
    "stage_index": "",
    "stage_name": "TOTAL_BOILER",
    "stage_kind": "",
    "Q_stage[MW]": Q_useful_hx,
    "UA_stage[MW/K]": UA_total,
    "gas_V_avg[m/s]": "",
    "water_V_avg[m/s]": "",
    "gas_in_P[kPa]": "",
    "gas_in_T[°C]": "",
    "gas_in_h[kJ/kg]": "",
    "gas_out_P[kPa]": "",
    "gas_out_T[°C]": "",
    "gas_out_h[kJ/kg]": "",
    "water_in_P[kPa]": "",
    "water_in_T[°C]": "",
    "water_in_h[kJ/kg]": "",
    "water_out_P[kPa]": "",
    "water_out_T[°C]": "",
    "water_out_h[kJ/kg]": "",
    "ΔP_stage_fric[kPa]": dP_total_fric,
    "ΔP_stage_minor[kPa]": dP_total_minor,
    "ΔP_stage_total[kPa]": dP_total_total,
    "ΔP_water_stage_fric[kPa]": w_dP_tot_fric,
    "ΔP_water_stage_minor[kPa]": w_dP_tot_minor,
    "ΔP_water_stage_total[kPa]": w_dP_tot_total,
    "stack_temperature[°C]": stack_T_C,
    "feedwater_mdot[kg/s]": feedwater_mdot_kg_s if feedwater_mdot_kg_s is not None else "",
    "circulation_mdot[kg/s]": circulation_mdot_kg_s if circulation_mdot_kg_s is not None else "",
    "Q_conv_stage[MW]": Q_total_conv,
    "Q_rad_stage[MW]": Q_total_rad,
    "steam_capacity[kg/s]": steam_capacity_total_kg_s,
    "steam_capacity[t/h]": steam_capacity_total_tph,
    "η_direct[-]": eta_direct if eta_direct is not None else "",
    "η_indirect[-]": eta_indirect if eta_indirect is not None else "",
    "Stack_loss_fraction[-]": Stack_loss_fraction if Stack_loss_fraction is not None else "",
```

```
    "Q_total_useful[MW]": Q_useful,
    "Q_flue_out[MW]": Q_flue_out_MW if Q_flue_out_MW is not None else "",
    "Q_balance_error[MW]": (Q_balance_err_MW if Q_balance_err_MW is not None else ""),
    "Q_in_total[MW]": Q_in_total if Q_in_total is not None else "",
    "P_LHV[MW]": P_LHV_W if P_LHV_W is not None else "",
    "LHV_mass[kJ/kg]": LHV_mass_kJkg if LHV_mass_kJkg is not None else "",
    "flue_mdot[kg/s]": flue_mdot_kg_s if flue_mdot_kg_s is not None else "",
    "boiler_water_in_T[°C]": boiler_water_in_T_C if boiler_water_in_T_C is not None else "",
    "boiler_water_out_T[°C]": boiler_water_out_T_C if boiler_water_out_T_C is not None else "",
    "boiler_water_P[kPa]": boiler_water_in_P_kPa if boiler_water_in_P_kPa is not None else "",
    "boiler_water_Tsat[°C]": boiler_water_Tsat_C if boiler_water_Tsat_C is not None else "",
}


rows.append(total_row)

return rows, Q_total, UA_total
```

== FILE: heat/runner.py ========================
from **future** import annotations from pathlib import Path from typing import List, Optional, Dict, Any from common.units import Q_ from heat.geometry import GeometryBuilder from heat.solver import solve_exchanger from common.models import HXStage, WaterStream, GasStream, Drum from common.results import build_global_profile,CombustionResult from heat.postproc import profile_to_dataframe, summary_from_profile from common.props import WaterProps

def *q_or_none(s: Optional[str]) -> Optional[Q_]: if s is None: return None s = s.strip() return Q*(s) if s else None

def run_hx(*, stages_raw: List[HXStage], water: WaterStream, gas: GasStream, drum: Drum, drum_pressure: Q_ | None = None, target_dx: str | None = None, min_steps: int = 20, max_steps: int = 400, max_passes: int = 20, tol_Q: str = "1e-3 W", tol_end: str = "1e-3 J/kg", write_csv: bool, outdir: str | Path = "results/runs", run_id: str | None = None, log_level: str = "INFO", combustion: CombustionResult | None = None,) -> Dict[str, Any]:

```
outdir = Path(outdir)
outdir.mkdir(parents=True, exist_ok=True)

if not run_id:
    from datetime import datetime
    run_id = datetime.now().strftime("%Y%m%d-%H%M%S")

target_dx_q = _q_or_none(target_dx)
tol_Q_q = Q_(tol_Q)
tol_end_q = Q_(tol_end)

if gas is None or water is None or drum is None:
    raise ValueError("missing required inputs: 'gas', 'water', and 'drum'")

stages: List[HXStage] = GeometryBuilder(drum).enrich(stages_raw)

drum_pool = None
if drum_pressure is not None:
    P_d = drum_pressure.to("Pa")
    h_f = WaterProps.h_f(P_d).to("J/kg")

    drum_pool = WaterStream(mass_flow=water.mass_flow, h=h_f, P=P_d)
```

```
stage_results, gas_out, water_out = solve_exchanger(
    stages,
    gas,
    water,
    drum_pool=drum_pool,
    drum_pool_stage_count=5,
    target_dx=target_dx_q,
    min_steps_per_stage=min_steps,
    max_steps_per_stage=max_steps,
    max_passes=max_passes,
    tol_Q=tol_Q_q,
    tol_end=tol_end_q,
    log_level=log_level,
)

global_profile = build_global_profile(stage_results)

if write_csv:
    df_steps = profile_to_dataframe(global_profile)
else:
    df_steps = None

rows, _, _ = summary_from_profile(global_profile, combustion=combustion, drum_pressure=drum_pressure)

return {
    "gas_in": gas,
    "water_in": water,
    "gas_out": gas_out,
    "water_out": water_out,
    "stage_results": stage_results,
    "global_profile": global_profile,
    "steps_df": df_steps,
    "summary_rows": rows,
    "steps_csv": None,
    "summary_csv": None,
    "run_id": run_id,
    "outdir": str(outdir),
    "combustion": combustion,
}
```

==================================== FILE: heat/solver.py ========================
from **future** import annotations from typing import List, Tuple, Optional from math import ceil, log10 import logging

from common.units import Q_ from common.models import HXStage, GasStream, WaterStream from common.results import StepResult, StageResult from heat.step_solver import solve_step from common.props import GasProps, WaterProps from common.logging_utils import setup_logging

_gasprops = GasProps() _gas = GasProps()

def _clamp(v: int, lo: int, hi: int) -> int: return max(lo, min(hi, v))

def *median_length(stages: List[HXStage]) -> Q*: Ls = sorted([st.spec["inner_length"].to("m") for st in stages], key=lambda x: x.magnitude) return Ls[len(Ls)//2] def *make__grid(L: Q*, n__steps: int) -> tuple[List[Q__], Q__]: dx = (L / n__steps).to(“m”) xs = [(i * dx).to(“m”) for i in range(n__steps)] return xs, dx

def __copy__step__with__stage(sr: StepResult, stage__name: str, stage__index: int, *, dP__fric: Q__ | None = None, dP__minor: Q__ | None = None, dP__total: Q__ | None = None, w__dP__fric: Q__ | None = None, w__dP__minor: Q__ | None = None, w__dP__tot: Q__ | None = None,) -> StepResult: return StepResult(i=sr.i, x=sr.x, dx=sr.dx, gas=sr.gas, water=sr.water, Tgw=sr.Tgw, Tww=sr.Tww, UA__prime=sr.UA__prime, qprime=sr.qprime, boiling=sr.boiling, h__g=sr.h__g, h__c=sr.h__c, qprime__conv=sr.qprime__conv, qprime__rad=sr.qprime__rad,
stage__name=stage__name, stage__index=stage__index, dP__fric=dP__fric if dP__fric is not None else Q__(0.0, “Pa”), dP__minor=dP__minor if dP__minor is not None else Q__(0.0, “Pa”), dP__total=dP__total if dP__total is not None else Q__(0.0, “Pa”), w__dP__fric=w__dP__fric if w__dP__fric is not None else Q__(0.0, “Pa”), w__dP__minor=w__dP__minor if w__dP__minor is not None else Q__(0.0, “Pa”), w__dP__tot=w__dP__tot if w__dP__tot is not None else Q__(0.0, “Pa”),)

def initial__wall__guesses(g: GasStream, w: WaterStream, stage: HXStage) -> tuple[Q__, Q__, Q__]: Tg = g.T.to(“K”) if stage.spec.get(“pool__boiling”, False): Tw = WaterProps.Tsat(w.P).to(“K”) else: Tw = WaterProps.T__from__Ph(w.P, w.h).to(“K”) qprime = Q__(1e4, “W/m”) return Tg, Tw, qprime

def __colebrook__white__f(Re: float, eps__over__D: float) -> float: Re = max(Re, 1e-6) if Re < 2300.0: return 64.0 / max(Re, 1e-12)

```
f = 0.25 / (log10(eps_over_D/3.7 + 5.74/(Re**0.9)) ** 2)

for _ in range(30):
    invsqrtf_old = 1.0 / (f ** 0.5)
    rhs = -2.0 * log10(eps_over_D/3.7 + 2.51/(Re * (f ** 0.5)))
    invsqrtf = rhs
    if abs(invsqrtf - invsqrtf_old) < 1e-6:
        break
    f = 1.0 / (invsqrtf ** 2)
return float(f)
```

def __friction__factor(Re: float, eps__over__D: float) -> float: if Re < 2300.0: return 64.0 / max(Re, 1e-12) if Re >= 4000.0: return __colebrook__white__f(Re, eps__over__D) f__lam = 64.0 / max(Re, 1e-12) f__turb = __colebrook__white__f(4000.0, eps__over__D) w = (Re - 2300.0) / (4000.0 - 2300.0) return float((1-w) * f__lam + w * f__turb)

def *stage__minor__K__sum(stage: HXStage) -> Q*: spec = stage.spec kind = stage.kind.lower()

```
K_hot_bend = spec.get("K_hot_bend", None)
if K_hot_bend is not None:
    return K_hot_bend.to("")

if kind == "reversal_chamber":
    Rc = spec.get("curvature_radius", None)
    Do = spec.get("outer_diameter", None)
    if Rc is not None and Do is not None and Rc.to("m").magnitude > 0 and Do.to("m").magnitude > 0:
        r = (Rc / Do).to("").magnitude
        Kbend = max(0.2, min(2.0, 0.9 / max(r, 1e-6)))
    else:
        Kbend = 0.5
    return Q_(Kbend, "")

return Q_(0.0, "")
```

def *gas__dp__economiser__crossflow(g: GasStream, stage: HXStage, dx: Q*, i__step: int, n__steps: int,) -> tuple[Q__, Q__, Q__]: spec = stage.spec

```
A_hot = spec["hot_flow_A"].to("m^2")

Dh  = spec["hot_Dh"].to("m")
eps = spec.get("roughness_out", Q_(0.0, "m")).to("m")

rho = _gas.rho(g.T, g.P, g.comp)
mu  = _gas.mu(g.T, g.P, g.comp)

V_bulk = (g.mass_flow / (rho * A_hot)).to("m/s")
umax_factor_q = spec.get("umax_factor", Q_(1.0, ""))
umax_factor = umax_factor_q.to("").magnitude
V_char = (V_bulk * max(umax_factor, 1.0)).to("m/s")

Re = max((rho * V_char * Dh / mu).to("").magnitude, 1e-6)
eps_over_D = (eps / Dh).to("").magnitude

f = _friction_factor(Re, eps_over_D)

q_dyn = (rho * V_char**2 / 2.0).to("Pa")

dP_fric = (-Q_(f, "") * (dx / Dh) * q_dyn).to("Pa")

K_bend_per_step = spec.get("_K_bend_per_step", Q_(0.0, "")).to("")
K_inlet  = spec.get("K_hot_inlet",  Q_(0.0, "")).to("")
K_outlet = spec.get("K_hot_outlet", Q_(0.0, "")).to("")

K_minor = K_bend_per_step
if i_step == 0:
    K_minor = (K_minor + K_inlet).to("")
if i_step == max(n_steps - 1, 0):
    K_minor = (K_minor + K_outlet).to("")

dP_minor = (-K_minor * q_dyn).to("Pa")
dP_total = (dP_fric + dP_minor).to("Pa")

return dP_fric, dP_minor, dP_total
```

def *gas_dp_components(g: GasStream, stage: HXStage, dx: Q*, i_step: int, n_steps: int,) -> tuple[Q_, Q_, Q_]: kind = stage.kind.lower()

```
if kind == "economiser":
    return _gas_dp_economiser_crossflow(g, stage, dx, i_step, n_steps)

spec = stage.spec
A = spec["hot_flow_A"].to("m^2")
Dh = spec["hot_Dh"].to("m")
eps = spec.get("roughness_in", Q_(0.0, "m")).to("m")
rho = _gas.rho(g.T, g.P, g.comp)
mu  = _gas.mu(g.T, g.P, g.comp)

V = (g.mass_flow / (rho * A)).to("m/s")
Re = max((rho * V * Dh / mu).to("").magnitude, 1e-6)
eps_over_D = (eps / Dh).to("").magnitude

f = _friction_factor(Re, eps_over_D)
```

```
q = (rho * V**2 / 2.0).to("Pa")

dP_fric = (-f * (dx / Dh) * q).to("Pa")

K_bend_per_step = spec.get("_K_bend_per_step", Q_(0.0, "")).to("")

K_inlet = spec.get("K_hot_inlet", Q_(0.0, "")).to("")
K_outlet = spec.get("K_hot_outlet", Q_(0.0, "")).to("")

K_minor = K_bend_per_step

if i_step == 0:
    K_minor = (K_minor + K_inlet).to("")

if i_step == max(n_steps - 1, 0):
    K_minor = (K_minor + K_outlet).to("")

dP_minor = (-K_minor * q).to("Pa")
dP_total = (dP_fric + dP_minor).to("Pa")
return dP_fric, dP_minor, dP_total
```

def pressure__drop__gas(g: GasStream, stage: HXStage, i: int, dx: Q__, n_steps: int) -> Q__: , , dP__total = __gas__dp__components(g, stage, dx, i, n_steps) return dP__total

def *solve__T__for__h(P, X, h__target, T0, maxit=30): T = T0.to("K"); h__target = h__target.to("J/kg") for* in range(maxit): h = __gasprops.h(T, P, X) dh = (h__target - h).to("J/kg") if abs(dh).magnitude < 1e-3: return T cp = __gasprops.cp(T, P, X) dT = (dh / cp).to("K") T = (T + 0.8*dT).to("K") return T

def update__gas__after__step(g, qprime, dx, stage, i: int, n_steps: int) -> GasStream: Q__step = (qprime * dx).to("W") dh = (-Q__step / g.mass__flow).to("J/kg") h__old = __gasprops.h(g.T, g.P, g.comp) h__new = (h__old + dh).to("J/kg") T__new = __solve__T__for__h(g.P, g.comp, h__new, g.T) P__new = (g.P + pressure__drop__gas(g, stage, i=i, dx=dx, n__steps=n__steps)).to("Pa") return GasStream(mass__flow=g.mass__flow, T=T__new, P=P__new, comp=g.comp)

def *water__dp__components(w: WaterStream, stage: HXStage, dx: Q*, i__step: int, n__steps: int) -> tuple[Q__, Q__, Q__]: spec = stage.spec kind = stage.kind.lower()

```
if spec.get("pool_boiling", False):
    z = Q_(0.0, "Pa")
    return z, z, z

xq = WaterProps.quality_from_Ph(w.P, w.h)
if xq is not None:
    z = Q_(0.0, "Pa")
    return z, z, z


dP_fric = Q_(0.0, "Pa")
dP_minor = Q_(0.0, "Pa")

if kind == "economiser":
    A = spec["cold_flow_A"].to("m^2")
    Dh = spec["cold_Dh"].to("m")
    eps = spec.get("roughness_cold_surface", Q_(0.0, "m")).to("m")

    rho = WaterProps.rho_from_Ph(w.P, w.h)
```

```
    mu  = WaterProps.mu_from_Ph(w.P, w.h)

    V = (w.mass_flow / (rho * A)).to("m/s")
    Re = max((rho * V * Dh / mu).to("").magnitude, 1e-6)
    eps_over_D = (eps / Dh).to("").magnitude

    f = _friction_factor(Re, eps_over_D)
    q = (rho * V**2 / 2.0).to("Pa")

    f_dx = spec.get("water_dx_factor", Q_(1.0, "")).to("").magnitude
    dx_w = (dx * Q_(f_dx, "")).to("m")

    dP_fric = (-f * (dx_w / Dh) * q).to("Pa")
else:
    A = spec.get("cold_flow_A", None)
    if A is not None:
        A = A.to("m^2")
        rho = WaterProps.rho_from_Ph(w.P, w.h)
        V = (w.mass_flow / (rho * A)).to("m/s")
        q = (rho * V**2 / 2.0).to("Pa")
    else:
        rho = WaterProps.rho_from_Ph(w.P, w.h)
        q = Q_(0.0, "Pa")

K_cold_bend_total = spec.get("K_cold_bend", Q_(0.0, "")).to("")
K_cold_bend_per_step = (K_cold_bend_total / max(n_steps, 1)).to("")

K_cold_inlet = spec.get("K_cold_inlet", Q_(0.0, "")).to("")
K_cold_outlet = spec.get("K_cold_outlet", Q_(0.0, "")).to("")

K_minor = K_cold_bend_per_step

if i_step == 0:
    K_minor = (K_minor + K_cold_inlet).to("")

if i_step == max(n_steps - 1, 0):
    K_minor = (K_minor + K_cold_outlet).to("")

if "q" not in locals():
    A = spec.get("cold_flow_A", None)
    if A is not None:
        A = A.to("m^2")
        rho = WaterProps.rho_from_Ph(w.P, w.h)
        V = (w.mass_flow / (rho * A)).to("m/s")
        q = (rho * V**2 / 2.0).to("Pa")
    else:
        q = Q_(0.0, "Pa")

dP_minor = (-K_minor * q).to("Pa")

dP_total = (dP_fric + dP_minor).to("Pa")
return dP_fric, dP_minor, dP_total
```

def update_water_after_step(w: WaterStream, qprime: Q_, dx: Q_, stage: HXStage, i: int, n_steps: int)

-> WaterStream: Q_step = (qprime * dx).to("W") dh = (Q_step / w.mass_flow).to("J/kg") h_new = (w.h + dh).to("J/kg")

```
if stage.spec.get("pool_boiling", False):
    hf = WaterProps.h_f(w.P).to("J/kg")
    return WaterStream(mass_flow=w.mass_flow, h=hf, P=w.P)

dP_fric, dP_minor, dP_tot = _water_dp_components(w, stage, dx, i, n_steps)
P_new = (w.P + dP_tot).to("Pa")
return WaterStream(mass_flow=w.mass_flow, h=h_new, P=P_new)
```

def solve_stage(g_in: GasStream, w_in: WaterStream, stage: HXStage, n_steps: int, *, stage_index: int, logger_name: str = "solver",) -> tuple[GasStream, WaterStream, StageResult]: log = logging.getLogger(logger_name) if stage.kind.lower() == "economiser": L = stage.spec["hot_flow_length"].to("m") else: L = stage.spec["inner_length"].to("m") xs, dx = _make_grid(L, n_steps)

```
K_sum = _stage_minor_K_sum(stage).to("")
K_per_step = (K_sum / max(n_steps, 1)).to("")
stage.spec["_K_bend_per_step"] = K_per_step

steps: List[StepResult] = []

g = g_in
w = w_in
Tgw_guess, Tww_guess, qprime_guess = initial_wall_guesses(g, w, stage)

Q_sum = Q_(0.0, "W")
UA_sum = Q_(0.0, "W/K")
dP_fric_sum = Q_(0.0, "Pa")
dP_minor_sum = Q_(0.0, "Pa")
dP_total_sum = Q_(0.0, "Pa")
w_dP_fric_sum  = Q_(0.0, "Pa")
w_dP_minor_sum  = Q_(0.0, "Pa")
w_dP_tot_sum  = Q_(0.0, "Pa")

for i, x in enumerate(xs):
    dP_fric_step, dP_minor_step, dP_tot_step = _gas_dp_components(g, stage, dx, i, n_steps)
   w_dP_fric_step, w_dP_minor_step, w_dP_tot_step = _water_dp_components(w, stage, dx, i, n_steps)

    sr = solve_step(
        g=g, w=w, stage=stage,
        Tgw_guess=Tgw_guess, Tww_guess=Tww_guess, qprime_guess=qprime_guess,
        i=i, x=x, dx=dx
    )

    sr = _copy_step_with_stage(
        sr, stage.name, stage_index,
        dP_fric=dP_fric_step,
        dP_minor=dP_minor_step,
        dP_total=dP_tot_step,
        w_dP_fric=w_dP_fric_step,
        w_dP_minor=w_dP_minor_step,
        w_dP_tot=w_dP_tot_step,
    )
    steps.append(sr)
```

```
        Q_sum = (Q_sum + (sr.qprime * dx)).to("W")
        UA_sum = (UA_sum + (sr.UA_prime * dx)).to("W/K")

        dP_fric_sum  = (dP_fric_sum  + dP_fric_step).to("Pa")
        dP_minor_sum = (dP_minor_sum + dP_minor_step).to("Pa")
        dP_total_sum = (dP_total_sum + dP_tot_step).to("Pa")

        w_dP_fric_sum  = (w_dP_fric_sum  + w_dP_fric_step).to("Pa")
        w_dP_minor_sum = (w_dP_minor_sum + w_dP_minor_step).to("Pa")
        w_dP_tot_sum   = (w_dP_tot_sum   + w_dP_tot_step).to("Pa")

        Tgw_guess, Tww_guess, qprime_guess = sr.Tgw, sr.Tww, sr.qprime
        g = update_gas_after_step(g, sr.qprime, dx, stage, i, n_steps)
        w = update_water_after_step(w, sr.qprime, dx, stage, i, n_steps)

        log.debug(
            "step",
            extra={"stage": stage.name, "step": f"{i+1}/{n_steps}"}
        )

    g_out = g
    w_out = w

    if steps:
        last = steps[-1]
        steps.append(
            StepResult(
                i=n_steps,
                x=L.to("m"),
                dx=Q_(0.0, "m"),
                gas=g_out,
                water=w_out,
                Tgw=last.Tgw,
                Tww=last.Tww,
                UA_prime=Q_(0.0, "W/K/m"),
                qprime=Q_(0.0, "W/m"),
                boiling=last.boiling,
                h_g=Q_(0.0, "W/m^2/K"),
                h_c=Q_(0.0, "W/m^2/K"),
                stage_name=stage.name,
                stage_index=stage_index,
                dP_fric=Q_(0.0, "Pa"),
                dP_minor=Q_(0.0, "Pa"),
                dP_total=Q_(0.0, "Pa"),
                w_dP_fric=Q_(0.0, "Pa"),
                w_dP_minor=Q_(0.0, "Pa"),
                w_dP_tot=Q_(0.0, "Pa"),
                qprime_conv=Q_(0.0, "W/m"),
                qprime_rad=Q_(0.0, "W/m"),
            )
        )
```

```
stage_res = StageResult(
    stage_name=stage.name,
    stage_kind=stage.kind,
    steps=steps,
    Q_stage=Q_sum,
    UA_stage=UA_sum,
    dP_stage_fric=dP_fric_sum,
    dP_stage_minor=dP_minor_sum,
    dP_stage_total=dP_total_sum,
    dP_water_stage_fric=w_dP_fric_sum,
    dP_water_stage_minor=w_dP_minor_sum,
    dP_water_stage_total=w_dP_tot_sum,
    hot_flow_A=stage.spec["hot_flow_A"],
    cold_flow_A=stage.spec["cold_flow_A"],
    hot_Dh=stage.spec["hot_Dh"],
    cold_Dh=stage.spec["cold_Dh"],
)

recon = sum([(s.qprime * s.dx).to("W") for s in steps if s.dx.to("m").magnitude > 0], Q_(0.0, "W"))
if abs((stage_res.Q_stage - recon) / (stage_res.Q_stage + Q_(1e-12, "W"))) > 0.005:
    raise RuntimeError(f"Stage energy accumulation mismatch >0.5% in {stage.name}")

log.debug(
    f"{stage.name}: dP_fric={stage_res.dP_stage_fric:~P}, "
    f"dP_minor={stage_res.dP_stage_minor:~P}, dP_total={stage_res.dP_stage_total:~P}",
    extra={"stage": stage.name, "step": "ΔP"},
)

log.debug(
    f"{stage.name}: gas_in(T={g_in.T:~P},P={g_in.P:~P}) gas_out(T={g_out.T:~P},P={g_out.P:~P}) "
   f"water_in(h={w_in.h:~P},P={w_in.P:~P}) water_out(h={w_out.h:~P}) Q_stage={stage_res.Q_stage:~P}",
    extra={"stage": stage.name, "step": f"{len(steps)}/{n_steps}"},
)



return g_out, w_out, stage_res
```

def solve_exchanger(stages: List[HXStage], gas_in: GasStream, water_in: WaterStream, *, drum_pool: WaterStream | None = None, drum_pool_stage_count: int = 5, target_dx: Q_ | None = None, min_steps_per_stage: int = 20, max_steps_per_stage: int = 400, max_passes: int = 20, tol_Q: Q_ = Q_(1e-3, "W"), tol_end: Q_ = Q_(1e-3, "J/kg"), log_level: str = "INFO",) -> tuple[List[StageResult], GasStream, WaterStream]: setup_logging(level=log_level) log = logging.getLogger("solver")

```
if len(stages) != 6:
    raise ValueError(f"Expected 6 stages. Got {len(stages)}.")

if target_dx is None:
    dx_target = (_median_length(stages) / 100).to("m")
else:
    dx_target = target_dx.to("m")

n_steps_by_stage: List[int] = []

for st in stages:
```

```
    if st.kind.lower() == "economiser":
        if "hot_flow_length" in st.spec:
            L = st.spec["hot_flow_length"].to("m")
        elif "inner_length" in st.spec:
            L = st.spec["inner_length"].to("m")
        else:
            raise KeyError(f"{st.name}: economiser missing both 'hot_flow_length' and 'inner_length'")
    else:
        L = st.spec["inner_length"].to("m")


    n = _clamp(
        int(ceil((L / dx_target).to("").magnitude)),
        min_steps_per_stage,
        max_steps_per_stage
    )
    n_steps_by_stage.append(n)

prev_Q_total: Optional[Q_] = None
prev_end_h: Optional[Tuple[Q_, Q_, Q_, Q_]] = None
final_stage_results: List[StageResult] = []

h_g_in = _gasprops.h(gas_in.T, gas_in.P, gas_in.comp)
h_w_in = water_in.h

for p in range(max_passes + 1):
    gas_stage_results: List[StageResult] = []
    gas_at_stage_in: List[GasStream] = []
    water_for_stage_boundary: List[WaterStream] = []

    g = gas_in
    for i, st in enumerate(stages):
        if drum_pool is not None and i < drum_pool_stage_count:
            w_boundary = drum_pool
        else:
            if p == 0:
                w_boundary = water_in
            else:
                w_boundary = final_stage_results[i].steps[0].water

        gas_at_stage_in.append(g)
        water_for_stage_boundary.append(w_boundary)

        g, w_tmp, st_res = solve_stage(g, w_boundary, st, n_steps_by_stage[i], stage_index=i)
        gas_stage_results.append(st_res)

    water_stage_results: List[StageResult] = []
    g_fields_for_water: List[GasStream] = [gs for gs in gas_at_stage_in]

    w = water_in
    w_econ_out: WaterStream | None = None
    for i_rev, st in enumerate(reversed(stages)):
        idx = 5 - i_rev
        g_for_stage = g_fields_for_water[idx]
```

```
    if drum_pool is not None and idx < drum_pool_stage_count:
    g_new, _w_dummy, st_res = solve_stage(g_for_stage, drum_pool, st, n_steps_by_stage[idx], stage_index=idx)
    else:
     g_new, w, st_res = solve_stage(g_for_stage, w, st, n_steps_by_stage[idx], stage_index=idx)
        w_econ_out = w

    g_fields_for_water[idx] = g_new
    water_stage_results.append(st_res)

water_stage_results = list(reversed(water_stage_results))

Q_total = sum([sr.Q_stage.to("W") for sr in water_stage_results], Q_(0.0, "W")).to("W")

g_out = gas_stage_results[-1].steps[-1].gas
w_out = w_econ_out if w_econ_out is not None else water_in

h_g_out = _gasprops.h(g_out.T, g_out.P, g_out.comp)
h_w_out = w_out.h

end_tuple = (h_g_in, h_g_out, h_w_in, h_w_out)

duty_ok = prev_Q_total is not None and abs(Q_total - prev_Q_total) < tol_Q
end_ok = prev_end_h is not None and max(
    abs(end_tuple[0] - prev_end_h[0]),
    abs(end_tuple[1] - prev_end_h[1]),
    abs(end_tuple[2] - prev_end_h[2]),
    abs(end_tuple[3] - prev_end_h[3]),
) < tol_end

log.info(
    f"pass {p}: Q_total={Q_total:~P} "
    f"ΔQ={(Q_total - (prev_Q_total or Q_(0,'W'))):~P} "
  f"max Δends={max( (abs(end_tuple[i] - (prev_end_h[i] if prev_end_h else end_tuple[i])) for i in range(4)), defa
    f"converged={'yes' if (duty_ok and end_ok) else 'no'}",
    extra={"stage": "ALL", "step": f"pass {p}"},
)

if duty_ok and end_ok:
    water_boundaries = [sr.steps[0].water for sr in water_stage_results]
    g = gas_in
    final_forward_results: List[StageResult] = []

    for i, st in enumerate(stages):
        w_boundary = water_boundaries[i]
        g, w_tmp, st_res = solve_stage(g, w_boundary, st, n_steps_by_stage[i], stage_index=i)
        final_forward_results.append(st_res)
        if i == (len(stages) - 1):
            w_out_sync = w_tmp

    g_out_sync = g

    h_g_out = _gasprops.h(g_out_sync.T, g_out_sync.P, g_out_sync.comp)
    h_w_out = w_out_sync.h
```

```
        Q_gas = (gas_in.mass_flow * (h_g_in - h_g_out)).to("W")
        mismatch = abs(Q_gas - Q_total) / (abs(Q_total) + Q_(1e-12, "W"))
        log.info(
        f"FINAL forward: Q_total={sum((sr.Q_stage for sr in final_forward_results), Q_(0,'W')):~P} "
            f"Q_gas={Q_gas:~P} rel_err={mismatch:~P}",
            extra={"stage": "ALL", "step": "final_forward"},
        )

        if mismatch.magnitude > 0.005:
            raise RuntimeError(
                f"Energy mismatch >0.5% on final sweep. "
                f"Q_gas={Q_gas:~P}, rel_err={mismatch:~P}"
            )

        return final_forward_results, g_out_sync, w_out_sync

    prev_Q_total = Q_total
    prev_end_h = end_tuple
    final_stage_results = water_stage_results

worst_idx = max(range(6), key=lambda k: abs(final_stage_results[k].Q_stage).to("W").magnitude if final_stage_result
raise RuntimeError(
    f"Did not converge in {max_passes} passes. "
    f"last_Q_total={prev_Q_total:~P if prev_Q_total else 'n/a'} "
    f"last_end_delta={prev_end_h if prev_end_h else 'n/a'} "
    f"worst_stage_index={worst_idx}"
)
```

==================================== FILE: heat/step_solver.py ====================

from common.results import StepResult from common.units import Q_ from common.models import HXStage, GasStream, WaterStream from heat.physics import wall_resistance, fouling_resistances from heat.water_htc import water_htc from common.props import WaterProps from heat.gas_htc import gas_htc, gas_htc_parts

def solve_step(g: GasStream, w: WaterStream, stage: HXStage, Tgw_guess: Q_, Tww_guess: Q_, qprime_guess: Q_, i: int, x: Q_, dx: Q_) -> StepResult: spec = stage.spec Pg = spec[“hot_wet_P”] Pw = spec[“cold_wet_P”] Tg = g.T if stage.spec[“pool_boiling”]: Tw = WaterProps.Tsat(w.P) else: Tw = WaterProps.T_from_Ph(w.P, w.h) Tgw = Tgw_guess Tww = Tww_guess qprime = qprime_guess alpha = 0.25 tolT = Q_(1e-3,“K”); tolq = Q_(1e-3,“W/m”); maxit = 10

```
for _ in range(maxit):
    h_g = gas_htc(g, spec, Tgw, stage_kind=stage.kind)
    qpp_cold = (qprime / Pw).to("W/m^2")
    h_c, boiling = water_htc(w, stage, Tww, qpp_cold)
    Rfg, Rfc = fouling_resistances(spec)
    Rw = wall_resistance(spec)
    Rg = (1/(h_g*Pg)).to("K*m/W")
    Rc = (1/(h_c*Pw)).to("K*m/W")

    UA_prime = (1/(Rg + Rfg + Rw + Rfc + Rc)).to("W/K/m")

    qprime_new = (UA_prime * (Tg - Tw)).to("W/m")

    qpp_hot  = (qprime_new / Pg).to("W/m^2")
    qpp_cold = (qprime_new / Pw).to("W/m^2")
```

```
    Tgw_new = (Tg - qpp_hot/h_g - qpp_hot*Rfg*Pg).to("K")
    Tww_new = (Tw + qpp_cold*Rw*Pw + qpp_cold*Rfc*Pw + qpp_cold/h_c).to("K")

    dTgw = abs(Tgw_new - Tgw); dTww = abs(Tww_new - Tww); dq = abs(qprime_new - qprime)
    if dTgw < tolT and dTww < tolT and dq < tolq:
        Tgw, Tww, qprime = Tgw_new, Tww_new, qprime_new
        break

    Tgw = (alpha*Tgw_new + (1-alpha)*Tgw).to("K")
    Tww = (alpha*Tww_new + (1-alpha)*Tww).to("K")
    qprime = (alpha*qprime_new + (1-alpha)*qprime).to("W/m")
h_conv, h_rad = gas_htc_parts(g, spec, Tgw, stage_kind=stage.kind)
h_g = (h_conv + h_rad).to("W/m^2/K")

h_tot_mag = h_g.to("W/m^2/K").magnitude
if h_tot_mag > 0:
    frac_conv = (h_conv / h_g).to("").magnitude
    frac_conv = max(0.0, min(1.0, frac_conv))
else:
    frac_conv = 0.0

qprime_conv = (qprime * frac_conv).to("W/m")
qprime_rad  = (qprime - qprime_conv).to("W/m")

return StepResult(
    i=i, x=x, dx=dx,
    gas=g, water=w,
    Tgw=Tgw, Tww=Tww,
    UA_prime=UA_prime,
    qprime=qprime,
    boiling=boiling,
    h_g=h_g,
    h_c=h_c,
    qprime_conv=qprime_conv,
    qprime_rad=qprime_rad,
)
```

==================================== FILE: heat/water_htc.py ====================
from math import log, sqrt, exp from common.units import Q_ from common.models import WaterStream, HXStage from common.props import WaterProps

P_CRIT_WATER = Q_(22.064, "MPa") MW_WATER = 18.01528

def velocity(w: WaterStream, Aflow, umax_factor=None) -> Q_: rho = WaterProps.rho_from_Ph(w.P, w.h) u = (w.mass_flow / (rho * Aflow)).to("m/s") if umax_factor is not None: return (umax_factor * u).to("m/s") return u

def reynolds_number(w:WaterStream, Aflow, char_len, umax_factor=None): rho = WaterProps.rho_from_Ph(w.P, w.h) v = velocity(w, Aflow, umax_factor) mu = WaterProps.mu_from_Ph(w.P, w.h) return (rho * v * char_len) / mu

def prandtl_number(cp: Q_, mu: Q_, k: Q_) -> Q_: return cp * mu / k

def film_temp(T_bulk: Q_, T_wall: Q_) -> Q_: return 0.5 * (T_bulk + T_wall)

def *is_boiling(P, h, T_wall: Q* | None = None) -> bool: hf = WaterProps.h_f(P) hg = WaterProps.h_g(P) if hf <= h <= hg: return True if (h < hf) and (T_wall is not None): return T_wall > WaterProps.Tsat(P) + Q_(3, "K") return False

def pr(w:WaterStream) -> Q_: cp = WaterProps.cp_from_Ph(w.P, w.h) mu = WaterProps.mu_from_Ph(w.P, w.h) k = WaterProps.k_from_Ph(w.P, w.h) return prandtl_number(cp, mu, k)

def pr_s(w: WaterStream, T_wall: Q_) -> Q_: cp_s = WaterProps.cp_from_PT(w.P, T_wall) mu_s = WaterProps.mu_from_PT(w.P, T_wall) k_s = WaterProps.k_from_PT(w.P, T_wall) return prandtl_number(cp_s, mu_s, k_s)

def nu_zukauskas_bank(Re: Q_, Pr: Q_, Pr_s: Q_, arrangement: str) -> tuple[Q_, Q_]: Re = Re.to(" ").magnitude Pr = Pr.to("").magnitude Pr_s = Pr_s.to(" ").magnitude

```
bands = [
    (1e3, 2e3, {"inline": (0.90, 0.40), "staggered": (1.04, 0.40)}),
    (2e3, 4e3, {"inline": (0.52, 0.50), "staggered": (0.71, 0.50)}),
    (4e3, 1e5, {"inline": (0.27, 0.63), "staggered": (0.35, 0.60)}),
    (1e5, 2e6, {"inline": (0.021, 0.84), "staggered": (0.022, 0.84)}),
]
C = None; m = None
for Re_min, Re_max, table in bands:
    if Re_min <= Re < Re_max:
        C, m = table["staggered" if arrangement == "staggered" else "inline"]
        break
if C is None:
    raise ValueError(f"Re={Re:.0f} outside Zukauskas bands")
n = 0.36 if Pr <= 10.0 else 0.25
s = 0.25
nu = C * (Re**m) * (Pr**n) * ((Pr / max(Pr_s, 1e-12))**s)
return Q_(nu, ""),Q_(m, "")
```

def nu_churchill_bernstein(Re: Q_, Pr: Q_) -> Q_: Re = Re.to(" ").magnitude Pr = Pr.to("").magnitude a = 0.3 b = (0.62 * Re**0.5 * Pr**(1/3)) / (1 + (0.4/Pr)**(2/3))**0.25 c = (1 + (Re/282000.0)**(5/8))**(4/5) return Q_(a + b * c," ")

def nu_gnielinski(Re: Q_, Pr: Q_, mu_ratio: Q_, L: Q_, D: Q_) -> Q_: Re = Re.to(" ").magnitude Pr = Pr.to("").magnitude L = L.to("meter").magnitude D = D.to("meter").magnitude if Re < 2300.0: Gz = Re * Pr * (D / max(L, 1e-12)) return Q_(3.66 + (0.0668 * Gz) / (1 + 0.04 * Gz**(2/3)), " ") f = (0.79 * log(Re) - 1.64) ** -2 num = (f/8) * (Re - 1000.0) * Pr den = 1 + 12.7 * (f/8)**0.5 * (Pr**(2/3) - 1) Nu = num / max(den, 1e-12) mu_ratio = mu_ratio.to("").magnitude return Q_(Nu * (mu_ratio ** 0.11)," ")

def compute_nusselt(w:WaterStream, stage: HXStage, T_wall: Q_) -> Q_:

```
if stage.kind == "single_tube":
    L = stage.spec["outer_diameter"]
    Aflow = stage.spec["cold_flow_A"]
    umax = stage.spec.get("umax_factor")
    Re = reynolds_number(w, Aflow, L, umax)
    Pr = pr(w)
    return nu_churchill_bernstein(Re, Pr)

if stage.kind == "tube_bank":
    L = stage.spec["outer_diameter"]
    Aflow = stage.spec["cold_flow_A"]
    umax = stage.spec.get("umax_factor")
    Re = reynolds_number(w, Aflow, L, umax)
    Pr = pr(w)
    Pr_s = pr_s(w, T_wall)
```

```
        N_rows = stage.spec["N_rows"]
        ST = stage.spec["ST"]
        SL = stage.spec["SL"]
        arrangement = stage.spec["arrangement"]
        Nu, m = nu_zukauskas_bank(Re, Pr, Pr_s, arrangement)
        Nu *= bank_row_factor(N_rows)
        Nu *= spacing_factor(L, ST, SL, arrangement, m)
        return Nu

    if stage.kind == "reversal_chamber":
        L = stage.spec["outer_diameter"]
        Aflow = stage.spec["cold_flow_A"]
        umax = stage.spec.get("umax_factor")
        Re = reynolds_number(w, Aflow, L, umax)
        Pr = pr(w)
        Rc = stage.spec["curvature_radius"]
        Nu = nu_churchill_bernstein(Re, Pr)
        return Nu * bend_factor_external(L, Rc)

    if stage.kind == "economiser":
        D = stage.spec["inner_diameter"]
        if "tube_length" in stage.spec:
            L = stage.spec["tube_length"]
        elif "inner_length" in stage.spec:
            L = stage.spec["inner_length"]
        else:
            raise KeyError(f"{stage.name}: economiser missing both 'tube_length' and 'inner_length'")
        Aflow = stage.spec["cold_flow_A"]
        T_bulk = WaterProps.T_from_Ph(w.P, w.h)
        umax = stage.spec.get("umax_factor")
        Re = reynolds_number(w, Aflow, D, umax)
        Pr = pr(w)
        mu_ratio = _mu_ratio(w, T_bulk, T_wall)
        return nu_gnielinski(Re, Pr, mu_ratio, L, D)

    raise ValueError(f"unknown stage kind: {stage.kind}")
```

def *mu_ratio(w: WaterStream, T_bulk: Q*, T_wall: Q_) -> Q_: mu_b = WaterProps.mu_from_PT(w.P, T_bulk) mu_w = WaterProps.mu_from_PT(w.P, T_wall) return mu_b / mu_w

def bend_factor_external(D: Q_, Rc: Q_) -> Q_:

```
if Rc <= 0 or D <= 0:
    return 1.0
phi = 1.0 + 0.10 * sqrt(D / Rc)
return Q_(phi, "")
```

def spacing_factor(D: Q_, ST: Q_, SL: Q_, arrangement: str, m_exp: Q_) -> Q_: if arrangement == "staggered": denom_T = ST - D denom_L = SL - (0.5 * D) vmax_ratio = (ST / denom_T) * (SL / denom_L) vmax_ratio = vmax_ratio**0.5 else: vmax_ratio = ST / (ST - D) m_exp = m_exp.to("").magnitude phi = vmax_ratio** m_exp return phi

def bank_row_factor(N_rows: Q_) -> Q_: n = N_rows.to(" ").magnitude f = 1.0 - 0.30 * exp(-0.30 * n) return Q_(f," ")

def *h_water_singlephase(w: WaterStream, stage: HXStage, T_wall) -> Q*: Nu = compute_nusselt(w, stage, T_wall) k = WaterProps.k_from_Ph(w.P, w.h) if stage.kind in ("single_tube", "tube_bank", "reversal_chamber"): Dh = stage.spec["outer_diameter"] else: Dh = stage.spec["inner_diameter"] return (Nu * k / Dh).to("W/m^2/K")

def *h_water_boil_cooper(P: Q*, qpp: Q_, Rp: Q_) -> Q_: p_r = (P.to("MPa") / P_CRIT_WATER).magnitude Rp_um = Rp.to("micrometer").magnitude q_kWm2 = qpp.to("kW/m^2").magnitude h_kWm2K = 55.0 * (p_r**0.12) * (((Rp_um))**-0.55) * (MW_WATER**-0.5) * (q_kWm2**0.67) return Q_(h_kWm2K, "kW/m^2/K").to("W/m^2/K")

def water_htc(w: WaterStream, stage: HXStage, T_wall: Q_, qpp: Q_) -> tuple[Q_, bool]: if stage.spec["pool_boiling"]: Rp = stage.spec["roughness_cold_surface"] h_nb = _h_water_boil_cooper(w.P, qpp, Rp) return h_nb, True

```
boiling = _is_boiling(w.P, w.h, T_wall)
if boiling:
    h_lo = _h_liquid_only(w, stage, T_wall)
    h_nb = _h_water_boil_cooper(w.P, qpp, stage.spec["roughness_cold_surface"])
    T_sat = WaterProps.Tsat(w.P)
    mu_l  = WaterProps.mu_from_PT(w.P, T_sat)
    A = stage.spec["cold_flow_A"]
    Dh = stage.spec["cold_Dh"]
    G = _mass_flux(w, A)
    h_lv = WaterProps.h_g(w.P) - WaterProps.h_f(w.P)
    x = WaterProps.quality_from_Ph(w.P, w.h)
    Re_lo = (G * Dh / mu_l).to("")
    S = _chen_S_factor(qpp, G, h_lv, Re_lo)
    if x is not None:
        F = _chen_F_factor(w.P, x)
    else:
        F = 1
    h_c = F * h_lo + S * h_nb
else:
    h_c = _h_water_singlephase(w, stage, T_wall)
return h_c, boiling
```

def *mass_flux(w: WaterStream, Aflow: Q*) -> Q_: return (w.mass_flow / Aflow).to("kg/m^2/s")

def *h_liquid_only(w: WaterStream, stage: HXStage, T_wall: Q*) -> Q_: D_h = stage.spec["cold_Dh"] if stage.kind == "economiser": if "tube_length" in stage.spec: L = stage.spec["tube_length"] else: raise KeyError(f"{stage.name}: missing 'tube_length' or 'inner_length' for liquid-only boiling model") else: L = stage.spec["inner_length"] A = stage.spec["cold_flow_A"] T_sat = WaterProps.Tsat(w.P) mu_l = WaterProps.mu_from_PT(w.P, T_sat) k_l = WaterProps.k_from_PT(w.P, T_sat) cp_l = WaterProps.cp_from_PT(w.P, T_sat) G = _mass_flux(w, A) Re_lo = (G * D_h / mu_l).to(" ") Pr = prandtl_number(cp_l, mu_l, k_l).to("") mu_ratio = (mu_l / WaterProps.mu_from_PT(w.P, T_wall)).to(" ")

```
if stage.kind == "economiser":
    Nu = nu_gnielinski(Re_lo, Pr, mu_ratio, L, D_h)
elif stage.kind == "tube_bank":
   Pr_s = prandtl_number(cp_l, WaterProps.mu_from_PT(w.P, T_wall), WaterProps.k_from_PT(w.P, T_wall))
    Nu, m = nu_zukauskas_bank(Re_lo, Pr, Pr_s, stage.spec["arrangement"])
    Nu *= bank_row_factor(stage.spec["N_rows"])
    Nu *= spacing_factor(D_h, stage.spec["ST"], stage.spec["SL"], stage.spec["arrangement"], m)
else:
    Nu = nu_churchill_bernstein(Re_lo, Pr)

return (Nu * k_l / D_h).to("W/m^2/K")
```

def *martinelli_Xtt(P: Q*, x: float) -> float: T_sat = WaterProps.Tsat(P) rho_l = WaterProps.rho_from_Px(P, Q_(0.0, "")) rho_g = WaterProps.rho_from_Px(P, Q_(1.0,"")) mu_l = WaterProps.mu_from_PT(P, T_sat) mu_g = WaterProps.mu_from_PT(P, T_sat) mu_ratio = (mu_l / mu_g).to("").magnitude rho_ratio = (rho_g / rho_l).to("").magnitude return ((1 - x) / x) ** 0.9 * (mu_ratio ** 0.1) * (rho_ratio ** 0.5)

def *chen_S_factor(qpp: Q*, G: Q_, h_lv: Q_, Re_lo: Q_) -> Q_: Re = max(1.0, Re_lo.to("").magnitude) S = 1.0 / (1.0 + 2.53e-6 * (Re ** 1.17)) return Q_(max(0.1, min(S, 1.0)),"")

def *chen_F_factor(P: Q*, x: float) -> Q_: Xtt = *martinelli_Xtt(P, x) F = 1.0 + 0.12 * (max(1e-6, 1.0 / Xtt) ** 0.8) return Q*(min(5.0, max(1.0, F)), "")

== FILE: config/air.yaml ==

T: { value: 300.0, unit: kelvin } P: { value: 101325, unit: Pa } composition: O2: { value: 0.23067, unit: dimensionless } N2: { value: 0.755866, unit: dimensionless } Ar: { value: 0.01287, unit: dimensionless } CO2: { value: 0.000594, unit: dimensionless } H2O: { value: 0.0, unit: dimensionless }

== FILE: config/drum.yaml ==

inner_diameter: { value: 4.5, unit: m } length: { value: 5.0, unit: m } wall: thickness: { value: 0.05, unit: m } conductivity: { value: 40, unit: W/m/K } surfaces: inner: roughness: { value: 5, unit: micrometer } emissivity: { value: 0.80, unit: dimensionless } fouling_thickness: { value: 0.0001, unit: m } fouling_conductivity: { value: 0.2, unit: W/m/K }

== FILE: config/flue_cantera.yaml ==

units: length: cm time: s quantity: mol activation-energy: cal/mol

phases: - name: gas_mix thermo: ideal-gas transport: mixture-averaged kinetics: none elements: [C, H, O, N, S, Ar] species: [CO2, H2O, SO2, O2, N2, Ar, CH4, C2H6, C3H8, C4H10, H2S] state: T: 300 K P: 1 atm X: "N2:1.0"

species: # ——————— CO2 ——————— - name: CO2 composition: { C: 1, O: 2 } thermo: model: NASA7 temperature-ranges: [200.0, 1000.0, 6000.0] data: # 200–1000 K - [2.35677352, 8.98459677e-03, -7.12356269e-06, 2.45919022e-09, -1.43699548e-13, -4.83719697e+04, 9.90105222,] # 1000–6000 K - [3.85796028, 4.41437026e-03, -2.21481404e-06, 5.23490188e-10, -4.72084164e-14, -4.87591660e+04, 2.27163806,]

```
transport:
  model: gas
  geometry: linear
  well-depth: 244.0 # K
  diameter: 3.763 # angstrom
  dipole: 0.0 # Debye
  polarizability: 2.650 # angstrom^3
  rotational-relaxation: 2.1 # Zrot at 298 K
```

——————— H2O ——————— - name: H2O composition: { H: 2, O: 1 } thermo: model: NASA7 temperature-ranges: [200.0, 1000.0, 6000.0] data: - [4.19864056, -2.03643410e-03, 6.52040211e-06, -5.48797062e-09, 1.77197817e-12, -3.02937267e+04, -8.49032208e-01,] - [3.03399249, 2.17691804e-03, -1.64072518e-07, -9.70419870e-11, 1.68200992e-14, -3.00042971e+04, 4.96677010,] transport: model: gas geometry: nonlinear diameter: 2.641 well-depth: 809.1 polarizability: 0.0 rotational-relaxation: 4.0 acentric-factor: 0.344

——————— SO2 ——————— - name: SO2 composition: { S: 1, O: 2 } thermo: model: NASA7 temperature-ranges: [200.0, 1000.0, 6000.0] data: - [4.88475400, -2.17239500e-03, 6.74313500e-06, -5.71253000e-09, 1.67180000e-12, -3.05629240e+04, 2.29262800,] - [4.88475400, -2.17239500e-03, 6.74313500e-06, -5.71253000e-09, 1.67180000e-12, -3.05629240e+04, 2.29262800,] transport: model: gas geometry: nonlinear diameter: 4.11 well-depth: 335.0 polarizability: 0.0 rotational-relaxation: 1.0 acentric-factor: 0.256

——————— O2 ——————— - name: O2 composition: { O: 2 } thermo: model: NASA7 temperature-ranges: [200.0, 1000.0, 6000.0] data: - [3.78245636, -2.99673416e-03, 9.84730201e-06, -9.68129509e-09, 3.24372837e-12, -1.06394356e+03, 3.65767573,] - [3.28253784, 1.48308754e-03, -7.57966669e-07, 2.09470555e-10, -2.16717794e-14, -1.08845772e+03, 5.45323129,] transport: model: gas geometry: linear diameter: 3.458 well-depth: 107.4 polarizability: 1.6 rotational-relaxation: 3.8 acentric-factor: 0.0222

——————— N2 ——————— - name: N2 composition: { N: 2 } thermo: model: NASA7 temperature-ranges: [200.0, 1000.0, 6000.0] data: - [3.53100528, -1.23660987e-04, -5.02999433e-07, 2.43530612e-09, -1.40881235e-12, -1.04697628e+03, 2.96747468,] - [2.95257626, 1.39690040e-03, -4.92631603e-07, 7.86010367e-11, -4.60755321e-15, -9.23948688e+02, 5.87188762,] transport: model: gas geometry: linear diameter: 3.621 well-depth: 97.53 polarizability: 1.76 rotational-relaxation: 4.0 acentric-factor: 0.040

——————— Ar ——————— - name: Ar composition: { Ar: 1 } thermo: model: NASA7 temperature-ranges: [200.0, 1000.0, 6000.0] data: - [2.50000000, 0.00000000, 0.00000000, 0.00000000, 0.00000000, -7.45375000E+02, 4.37967491,] - [2.50000000, 0.00000000, 0.00000000, 0.00000000, 0.00000000, -7.45375000E+02, 4.37967491,] transport: model: gas geometry: atom diameter: 3.330 well-depth: 136.5 polarizability: 1.641 rotational-relaxation: 0.0 acentric-factor: 0.000

——————— CH4 ——————— - name: CH4 composition: { C: 1, H: 4 } thermo: model: NASA7 temperature-ranges: [200.0, 1000.0, 6000.0] data: - [5.14987613e+00, -1.36709788e-02, 4.91800599e-05, -4.84743026e-08, 1.66693956e-11, -1.02466476e+04, -4.64130376e+00,] - [7.48514950e-02, 1.33909467e-02, -5.73285809e-06, 1.22292535e-09, -1.01815230e-13, -1.00095936e+04, 1.84373180e+01,] transport: model: gas geometry: nonlinear diameter: 3.746 well-depth: 141.4 polarizability: 2.6 rotational-relaxation: 13.0 acentric-factor: 0.011

——————— C2H6 ——————— - name: C2H6 composition: { C: 2, H: 6 } thermo: model: NASA7 temperature-ranges: [200.0, 1000.0, 3500.0] data: # 200–1000 K - [4.29142492, -5.50154270e-03, 5.99438288e-05, -7.08466285e-08, 2.68685771e-11, -1.15222055e+04, 2.66682316,] # 1000–3500 K - [1.07188150, 2.16852677e-02, -1.00256067e-05, 2.21412001e-09, -1.90002890e-13, -1.14263932e+04, 15.11561070,] transport: model: gas geometry: nonlinear diameter: 4.302 # Å well-depth: 252.3 # K polarizability: 4.226 # Å^3 rotational-relaxation: 1.5 acentric-factor: 0.099

——————— C3H8 ——————— - name: C3H8 composition: { C: 3, H: 8 } thermo: model: NASA7 temperature-ranges: [200.0, 1000.0, 6000.0] data: - [5.40872872, -8.55221825e-03, 8.42178491e-05, -1.00942683e-07, 3.86914479e-11, 9.42600956e+03, 3.62322504,] - [5.75125882, 1.87605762e-02, -6.70191976e-06, 1.07751871e-09, -6.43090885e-14, 7.97977293e+03, -4.91359355,] transport: model: gas geometry: nonlinear diameter: 4.982 # Å well-depth: 266.8 # K polarizability: 5.921 # Å^3 rotational-relaxation: 1.0 acentric-factor: 0.1521

——————— C4H10 ——————— - name: C4H10 composition: { C: 4, H: 10 } thermo: model: NASA7 temperature-ranges: [200.0, 1000.0, 6000.0] data: - [6.14474013, 1.64500242e-04, 9.67848789e-05, -1.25486208e-07, 4.97846257e-11, -1.75989467e+04, -1.08058878,] - [9.44547835, 2.57856620e-02, -9.23613194e-06, 1.48631762e-09, -8.87891206e-14, -2.01383773e+04, -2.63477585e+01,] transport: model: gas geometry: nonlinear diameter: 5.206 well-depth: 350.9 polarizability: 0.0 rotational-relaxation: 1.0 acentric-factor: 0.20081

——————— H2S ——————— - name: H2S composition: { H: 2, S: 1 } thermo: model: NASA7 temperature-ranges: [200.0, 1000.0, 6000.0] data: - [3.93234760, -5.02609050e-04, 4.59284730e-06, -3.18072140e-09, 6.64975610e-13, -3.65053590e+03, 2.31579050,] - [2.74521990, 4.04346070e-03, -1.63845100e-06, 2.75202490e-10, -1.85920950e-14, -3.41994440e+03, 8.05467450,] transport: model: gas geometry: nonlinear diameter: 3.60 # Å well-depth: 301.0 # K polarizability: 3.76 # Å^3 rotational-relaxation: 4.0 acentric-factor: 0.10

== FILE: config/fuel.yaml ==
T: { value: 300.0, unit: kelvin } P: { value: 101325, unit: Pa } mass_flow: { value: 0.1, unit: kg/s } composition: CH4: { value: 0.849546, unit: dimensionless } C2H6: { value: 0.061889, unit: dimensionless

} C3H8: { value: 0.020597, unit: dimensionless } C4H10: { value: 0.005154, unit: dimensionless } H2S: { value: 0.000103, unit: dimensionless } N2: { value: 0.041293, unit: dimensionless } CO2: { value: 0.016418, unit: dimensionless } H2O: { value: 0.005, unit: dimensionless } Ar: { value: 0.0, unit: dimensionless }

======================================= FILE: config/operation.yaml ======================================= excess_air_ratio: { value: 1.05, unit: dimensionless } drum_pressure: { value: 10, unit: bar }

======================================= FILE: config/stages.yaml ======================================= HX_1: kind: "single_tube" pool_boiling: true inner_diameter: { value: 1.4, unit: m } inner_length: { value: 5.276, unit: m } wall: thickness: { value: 0.02, unit: m } conductivity: { value: 50, unit: W/m/K } surfaces: inner: roughness: { value: 50, unit: micrometer } emissivity: { value: 0.80, unit: dimensionless } fouling_thickness: { value: 0.0001, unit: m } fouling_conductivity: { value: 0.20, unit: W/m/K } outer: roughness: { value: 20, unit: micrometer } emissivity: { value: 0.80, unit: dimensionless } fouling_thickness: { value: 0.0001, unit: m } fouling_conductivity: { value: 0.20, unit: W/m/K }

K: hot_inlet: 0.5 hot_outlet: 0.0 hot_bend: 0.0

```
cold_inlet: 0.0
cold_outlet: 0.0
cold_bend: 0.0
```

HX_2: kind: "reversal_chamber" pool_boiling: true inner_diameter: { value: 1.6, unit: m } inner_length: { value: 0.8, unit: m } curvature_radius: { value: 0.8, unit: m } wall: thickness: { value: 0.02, unit: m } conductivity: { value: 50, unit: W/m/K } surfaces: inner: roughness: { value: 50, unit: micrometer } emissivity: { value: 0.80, unit: dimensionless } fouling_thickness: { value: 0.0001, unit: m } fouling_conductivity: { value: 0.20, unit: W/m/K } outer: roughness: { value: 20, unit: micrometer } emissivity: { value: 0.80, unit: dimensionless } fouling_thickness: { value: 0.0001, unit: m } fouling_conductivity: { value: 0.20, unit: W/m/K }

K: hot_inlet: 0.0 hot_outlet: 0.0 hot_bend: 0.3

```
cold_inlet: 0.0
cold_outlet: 0.0
cold_bend: 0.0
```

HX_3: kind: "tube_bank" pool_boiling: true inner_diameter: { value: 0.076, unit: m } inner_length: { value: 4.975, unit: m } tubes_number: { value: 118, unit: dimensionless } arrangement: "staggered" N_rows: { value: 6, unit: dimensionless } ST: { value: 0.11, unit: m } SL: { value: 0.11, unit: m } baffle_spacing: { value: 0.45, unit: m } baffle_cut: { value: 0.25, unit: dimensionless } bundle_clearance: { value: 0.010, unit: m } wall: thickness: { value: 0.0029, unit: m } conductivity: { value: 50, unit: W/m/K } surfaces: inner: roughness: { value: 50, unit: micrometer } emissivity: { value: 0.80, unit: dimensionless } fouling_thickness: { value: 0.0001, unit: m } fouling_conductivity: { value: 0.20, unit: W/m/K } outer: roughness: { value: 20, unit: micrometer } emissivity: { value: 0.80, unit: dimensionless } fouling_thickness: { value: 0.0001, unit: m } fouling_conductivity: { value: 0.20, unit: W/m/K }

K: hot_inlet: 0.5 hot_outlet: 1.0 hot_bend: 0.0

```
cold_inlet: 0.0
cold_outlet: 0.0
cold_bend: 0.0
```

HX_4: kind: "reversal_chamber" pool_boiling: true inner_diameter: { value: 1.6, unit: m } inner_length: { value: 0.8, unit: m } curvature_radius: { value: 0.8, unit: m } wall: thickness: { value: 0.02, unit: m } conductivity: { value: 50, unit: W/m/K } surfaces: inner: roughness: { value: 50, unit: micrometer } emissivity: { value: 0.80, unit: dimensionless } fouling_thickness: { value: 0.0001, unit: m } fouling_conductivity: { value: 0.20, unit: W/m/K } outer: roughness: { value: 20, unit: micrometer } emissivity: { value: 0.80, unit: dimensionless } fouling_thickness: { value: 0.0001, unit: m } fouling_conductivity: { value: 0.20, unit: W/m/K }

K: hot_inlet: 0.0 hot_outlet: 0.0 hot_bend: 0.3

```
cold_inlet: 0.0
cold_outlet: 0.0
cold_bend: 0.0
```

HX_5: kind: “tube_bank” pool_boiling: true inner_diameter: { value: 0.076, unit: m } inner_length: { value: 5.620, unit: m } tubes_number: { value: 100, unit: dimensionless } arrangement: “staggered” N_rows: { value: 6, unit: dimensionless } ST: { value: 0.11, unit: m } SL: { value: 0.11, unit: m } baffle_spacing: { value: 0.45, unit: m } baffle_cut: { value: 0.25, unit: dimensionless } bundle_clearance: { value: 0.010, unit: m } wall: thickness: { value: 0.0029, unit: m } conductivity: { value: 50, unit: W/m/K } surfaces: inner: roughness: { value: 50, unit: micrometer } emissivity: { value: 0.80, unit: dimensionless } fouling_thickness: { value: 0.0001, unit: m } fouling_conductivity: { value: 0.20, unit: W/m/K } outer: roughness: { value: 20, unit: micrometer } emissivity: { value: 0.80, unit: dimensionless } fouling_thickness: { value: 0.0001, unit: m } fouling_conductivity: { value: 0.20, unit: W/m/K }

K: hot_inlet: 0.5 hot_outlet: 1.0 hot_bend: 0.0

```
cold_inlet: 0.0
cold_outlet: 0.0
cold_bend: 0.0
```

HX_6: kind: “economiser” pool_boiling: false inner_diameter: { value: 0.0250, unit: m } tube_length: { value: 80, unit: m } n_tubes: { value: 120, unit: dimensionless } arrangement: “staggered” N_rows: { value: 26, unit: dimensionless } ST: { value: 0.075, unit: m } SL: { value: 0.08, unit: m } baffle_spacing: { value: 0.15, unit: m } baffle_cut: { value: 0.25, unit: dimensionless } bundle_clearance: { value: 0.010, unit: m } wall: thickness: { value: 0.0026, unit: m } conductivity: { value: 50, unit: W/m/K } surfaces: inner: roughness: { value: 20, unit: micrometer } emissivity: { value: 0.80, unit: dimensionless } fouling_thickness: { value: 0.0, unit: m } fouling_conductivity: { value: 0.20, unit: W/m/K } outer: roughness: { value: 50, unit: micrometer } emissivity: { value: 0.80, unit: dimensionless } fouling_thickness: { value: 0.0, unit: m } fouling_conductivity: { value: 0.20, unit: W/m/K } shell_inner_diameter: { value: 0.60, unit: m } # Gas side

K: hot_inlet: 0.5 hot_outlet: 1.0 hot_bend: 0.0

```
cold_inlet: 0.5
cold_outlet: 1.0
cold_bend: 0.3
```

==================================== FILE: config/water.yaml ==================== enthalpy: { value: 440000, unit: J/kg } composition: H2O: { value: 1.0, unit: dimensionless }