Building Code with AI

This document outlines the changes that were made to the provided code as an alternative method than screenshots. To get my code run:

```
git clone https://github.com/X-man34/ME401-project1.git
```

To see the differencess between my code and Max's run:

```
git fetch upstream; git diff upstream/main...origin/main
```

You can also use the GUI and view the commits on the website.

The AI generated a rankine validation script which is the following

```python
"""
Rankine cycle optimization validation utilities.

This module validates optimized cycle points against first-law consistency and
basic physical constraints, then writes a machine-readable report.
"""

from __future__ import annotations

import json
from datetime import datetime, timezone
from pathlib import Path
from typing import Any, Dict, List, Optional

import numpy as np
from CoolProp.CoolProp import PropsSI

from rankine_cycle import RankineCycle


DEFAULT_TOLERANCES = {
    "energy_balance_abs_tol": 1e-3,  # J/kg
    "energy_balance_rel_tol": 1e-6,  # relative to max(|q_in|,|q_out|,|w_net|)
    "efficiency_tol": 1e-6,
    "constraint_tol": 1e-8,
    "minimum_superheat": 50.0,  # K
    "minimum_quality": 0.88,  # vapor quality at turbine outlet
}


def _as_bool(value: bool) -> bool:
    return bool(value)
```

```
def _to_float(value: Any) -> Optional[float]:
    try:
        out = float(value)
        if np.isfinite(out):
            return out
        return None
    except Exception:
        return None


def _superheat_margin(x_opt: np.ndarray, minimum_superheat: float) -> Optional[float]:
    try:
        p_boiler, t_boiler, _ = x_opt
        t_sat = PropsSI("T", "P", float(p_boiler), "Q", 1, "Water")
        return float(t_boiler - t_sat - minimum_superheat)
    except Exception:
        return None


def _quality_margin(
    cycle: RankineCycle,
    p_condenser: float,
    minimum_quality: float,
) -> tuple[Optional[float], Optional[float], Optional[str]]:
    try:
        quality = PropsSI("Q", "P", float(p_condenser), "H", cycle.states[4]["h"], cycle.flu
        quality = float(quality)
        if np.isfinite(quality) and (0.0 <= quality <= 1.0):
            return quality - minimum_quality, quality, None
        return None, quality, (
            "CoolProp returned non-two-phase quality; turbine exit may be "
            "superheated/subcooled at this state."
        )
    except Exception as exc:
        return None, None, f"Unable to evaluate turbine exit quality: {exc}"


def validate_rankine_solution(
    method_name: str,
    x_opt: np.ndarray,
    objective_value: float,
    success: bool,
    message: str,
    tolerances: Optional[Dict[str, float]] = None,
) -> Dict[str, Any]:
    """
    Validate one optimizer solution against physics and constraints.
    """
    tol = dict(DEFAULT_TOLERANCES)
    if tolerances:
        tol.update(tolerances)

    p_boiler, t_boiler, p_condenser = [float(v) for v in x_opt]
    warnings: List[str] = []
    checks: Dict[str, Dict[str, Any]] = {}

    objective_efficiency = _to_float(-objective_value)
    cycle: Optional[RankineCycle] = None
    performance: Dict[str, Any] = {}

    try:
        cycle = RankineCycle(p_boiler, t_boiler, p_condenser, eta_pump=1.0, eta_turbine=1.0)
        performance = cycle.calculate_performance()
    except Exception as exc:
        warnings.append(f"Failed to construct cycle at optimized point: {exc}")

    def add_check(name: str, passed: bool, value: Any, threshold: Any, detail: str) -> None
        checks[name] = {
            "pass": _as_bool(passed),
            "value": value,
            "threshold": threshold,
            "detail": detail,
        }

    # Basic pressure and finiteness sanity.
    finite_ok = all(np.isfinite([p_boiler, t_boiler, p_condenser]))
    add_check(
        "finite_inputs",
        finite_ok,
        {"P_boiler_Pa": p_boiler, "T_boiler_K": t_boiler, "P_condenser_Pa": p_condenser},
        "all finite",
        "Decision variables must be finite scalars.",
    )
    add_check(
        "positive_inputs",
        finite_ok and (p_boiler > 0.0 and t_boiler > 0.0 and p_condenser > 0.0),
        {"P_boiler_Pa": p_boiler, "T_boiler_K": t_boiler, "P_condenser_Pa": p_condenser},
        "> 0",
        "Pressure and temperature must be positive.",
    )
    add_check(
        "pressure_hierarchy",
        finite_ok and (p_boiler > p_condenser),
        {"P_boiler_Pa": p_boiler, "P_condenser_Pa": p_condenser},
        "P_boiler > P_condenser",
        "Rankine high pressure should exceed condenser pressure.",
    )

    if cycle is not None:
        q_in = float(performance["q_in"])
        q_out = float(performance["q_out"])
        w_net = float(performance["w_net"])
        w_turbine = float(performance["w_turbine"])
        w_pump = float(performance["w_pump"])
        eta_cycle = float(performance["eta_thermal"])

        # First-law residual.
        residual = q_in - q_out - w_net
        scale = max(abs(q_in), abs(q_out), abs(w_net), 1.0)
        residual_tol = max(float(tol["energy_balance_abs_tol"]), float(tol["energy_balance_r
        add_check(
            "energy_balance",
            abs(residual) <= residual_tol,
            residual,
            residual_tol,
            "q_in - q_out - w_net should be approximately zero.",
        )

        # Efficiency consistency.
        eta_from_components = w_net / q_in if q_in != 0.0 else np.nan
        eta_ok = False
        if objective_efficiency is not None and np.isfinite(eta_from_components):
            eta_ok = abs(eta_from_components - objective_efficiency) <= float(tol["efficienc
        add_check(
            "efficiency_consistency",
            eta_ok,
            {
                "eta_from_components": float(eta_from_components),
                "eta_from_objective": objective_efficiency,
            },
            float(tol["efficiency_tol"]),
            "Efficiency from states should match optimizer objective.",
        )

        # Sign checks.
        add_check("q_in_positive", q_in > 0.0, q_in, "> 0", "Boiler heat input must be posit
        add_check("q_out_positive", q_out > 0.0, q_out, "> 0", "Condenser heat rejection mus
        add_check("w_turbine_positive", w_turbine > 0.0, w_turbine, "> 0", "Turbine work mus
        add_check("w_pump_nonnegative", w_pump >= 0.0, w_pump, ">= 0", "Pump work should not
        add_check("w_net_positive", w_net > 0.0, w_net, "> 0", "Net cycle work should be pos

        # Bounds on thermal efficiency.
        add_check(
            "efficiency_bounds",
            (eta_cycle > 0.0) and (eta_cycle < 1.0),
            eta_cycle,
            "0 < eta < 1",
            "Thermal efficiency should be between 0 and 1 for this cycle model.",
        )

        # Superheat constraint margin.
        superheat_margin = _superheat_margin(x_opt, float(tol["minimum_superheat"]))
        superheat_ok = superheat_margin is not None and (superheat_margin >= -float(tol["con
        add_check(
            "superheat_margin",
            superheat_ok,
            superheat_margin,
            f">= {-float(tol['constraint_tol'])}",
            "Superheat margin after minimum required superheat.",
        )

        # Turbine-exit quality constraint margin.
        quality_margin, quality_value, quality_note = _quality_margin(
            cycle, p_condenser, float(tol["minimum_quality"])
        )
        quality_ok = quality_margin is not None and (quality_margin >= -float(tol["constrain
        add_check(
            "turbine_exit_quality_margin",
            quality_ok,
            {"margin": quality_margin, "quality": quality_value},
            f">= {-float(tol['constraint_tol'])}",
            "Quality margin relative to minimum acceptable turbine exit quality.",
        )
        if quality_note:
            warnings.append(quality_note)
    else:
        add_check(
            "cycle_constructed",
            False,
            None,
            "cycle created",
            "Cycle could not be evaluated at this point; dependent checks skipped.",
        )
    overall_validation_pass = bool(success) and all(item["pass"] for item in checks.values()

    return {
        "method": method_name,
        "optimizer_success": bool(success),
        "optimizer_message": str(message),
        "x_opt": {
            "P_boiler_Pa": p_boiler,
            "T_boiler_K": t_boiler,
            "P_condenser_Pa": p_condenser,
            "P_boiler_MPa": p_boiler / 1e6,
            "T_boiler_C": t_boiler - 273.15,
            "P_condenser_kPa": p_condenser / 1e3,
        },
        "objective_efficiency": objective_efficiency,
        "checks": checks,
        "warnings": warnings,
        "overall_validation_pass": overall_validation_pass,
        "tolerances": tol,
        "timestamp_utc": datetime.now(timezone.utc).isoformat(),
    }


def print_validation_report(validation_records: List[Dict[str, Any]]) -> None:
    """Print a concise validation summary and check details."""
    print("\n" + "=" * 80)
    print("VALIDATION SUMMARY")
    print("=" * 80)
    print(f"{'Method':<25} {'Optimizer Success':<20} {'Validation Pass':<18} {'Energy Resid
    print("-" * 80)

    for record in validation_records:
        method = record["method"]
        success = "Yes" if record["optimizer_success"] else "No"
        validation_pass = "Yes" if record["overall_validation_pass"] else "No"
        energy_value = record["checks"].get("energy_balance", {}).get("value", None)
        if isinstance(energy_value, (int, float)) and np.isfinite(float(energy_value)):
            energy_text = f"{float(energy_value):.3e}"
        else:
            energy_text = "N/A"
        print(f"{method:<25} {success:<20} {validation_pass:<18} {energy_text:<20}")

    print("=" * 80)

    for record in validation_records:
        print(f"\n{record['method']} detailed checks:")
        for check_name, check_data in record["checks"].items():
            status = "PASS" if check_data["pass"] else "FAIL"
            print(f"  [{status}] {check_name}: {check_data['detail']}")
            print(f"        value={check_data['value']} threshold={check_data['threshold']}'
        if record["warnings"]:
            print("  Warnings:")
            for warning in record["warnings"]:
                print(f"    - {warning}")


def build_validation_report(
    validation_records: List[Dict[str, Any]],
    output_path: str = "rankine_validation_report.json",
) -> Dict[str, Any]:
    """Build and write aggregate validation report JSON."""
    report = {
        "created_utc": datetime.now(timezone.utc).isoformat(),
        "record_count": len(validation_records),
        "all_valid": all(r.get("overall_validation_pass", False) for r in validation_records
        "methods": validation_records,
    }

    output_file = Path(output_path)
    output_file.write_text(json.dumps(report, indent=2), encoding="utf-8")
    report["output_path"] = str(output_file.resolve())
    return report
```

There were some modifications made to the main analysis script to maintain compatibility with this new code and notable, to fix the existing bug in max's code related to the constraints on the differential evolution solver. The bug fix was to simply add another set of constraints and pass them to the DE solver.

```
constraints = [
    {'type': 'ineq', 'fun': constraint_superheat},
    {'type': 'ineq', 'fun': constraint_quality_turbine_exit}
]
de_constraints = (
    NonlinearConstraint(constraint_superheat, 0.0, np.inf),
    NonlinearConstraint(constraint_quality_turbine_exit, 0.0, np.inf),
)
```

When run the validation script produced the following json output

```
{
  "created_utc": "2026-02-13T21:10:11.986993+00:00",
  "record_count": 2,
  "all_valid": true,
"methods": [
  {
    "method": "SLSQP",
    "optimizer_success": true,
    "optimizer_message": "Optimization terminated successfully",
    "x_opt": {
      "P_boiler_Pa": 8247941.199683601,
      "T_boiler_K": 893.1499999422433,
      "P_condenser_Pa": 20000.0,
      "P_boiler_MPa": 8.2479411996836,
      "T_boiler_C": 619.9999999422433,
      "P_condenser_kPa": 20.0
    },
    "objective_efficiency": 0.394678514507695,
    "checks": {
      "finite_inputs": {
        "pass": true,
        "value": {
          "P_boiler_Pa": 8247941.199683601,
          "T_boiler_K": 893.1499999422433,
          "P_condenser_Pa": 20000.0
        },
        "threshold": "all finite",
        "detail": "Decision variables must be finite scalars."
      },
      "positive_inputs": {
        "pass": true,
        "value": {
          "P_boiler_Pa": 8247941.199683601,
          "T_boiler_K": 893.1499999422433,
          "P_condenser_Pa": 20000.0
        },
        "threshold": "> 0",
        "detail": "Pressure and temperature must be positive."
      },
      "pressure_hierarchy": {
        "pass": true,
        "value": {
          "P_boiler_Pa": 8247941.199683601,
          "P_condenser_Pa": 20000.0
        },
        "threshold": "P_boiler > P_condenser",
        "detail": "Rankine high pressure should exceed condenser pressure."
      },
      "energy_balance": {
        "pass": true,
    "value": 0.0,
    "threshold": 3.4286691745189404,
    "detail": "q_in - q_out - w_net should be approximately zero."
  },
  "efficiency_consistency": {
    "pass": true,
    "value": {
      "eta_from_components": 0.394678514507695,
      "eta_from_objective": 0.394678514507695
    },
    "threshold": 1e-06,
    "detail": "Efficiency from states should match optimizer objective."
  },
  "q_in_positive": {
    "pass": true,
    "value": 3428669.1745189405,
    "threshold": "> 0",
    "detail": "Boiler heat input must be positive."
  },
  "q_out_positive": {
    "pass": true,
    "value": 2075447.1179814802,
    "threshold": "> 0",
    "detail": "Condenser heat rejection must be positive."
  },
  "w_turbine_positive": {
    "pass": true,
    "value": 1361576.749702942,
    "threshold": "> 0",
    "detail": "Turbine work must be positive."
  },
  "w_pump_nonnegative": {
    "pass": true,
    "value": 8354.693165481818,
    "threshold": ">= 0",
    "detail": "Pump work should not be negative."
  },
  "w_net_positive": {
    "pass": true,
    "value": 1353222.0565374603,
    "threshold": "> 0",
    "detail": "Net cycle work should be positive."
  },
  "efficiency_bounds": {
    "pass": true,
    "value": 0.394678514507695,
      "threshold": "0 < eta < 1",
      "detail": "Thermal efficiency should be between 0 and 1 for this cycle model."
    },
    "superheat_margin": {
      "pass": true,
      "value": 272.85362154846746,
      "threshold": ">= -1e-08",
      "detail": "Superheat margin after minimum required superheat."
    },
    "turbine_exit_quality_margin": {
      "pass": true,
      "value": {
        "margin": 0.0003545402648865714,
        "quality": 0.8803545402648866
      },
      "threshold": ">= -1e-08",
      "detail": "Quality margin relative to minimum acceptable turbine exit quality."
    }
  },
  "warnings": [],
  "overall_validation_pass": true,
  "tolerances": {
    "energy_balance_abs_tol": 0.001,
    "energy_balance_rel_tol": 1e-06,
    "efficiency_tol": 1e-06,
    "constraint_tol": 1e-08,
    "minimum_superheat": 50.0,
    "minimum_quality": 0.88
  },
  "timestamp_utc": "2026-02-13T21:10:11.976480+00:00"
},
{
  "method": "Differential Evolution",
  "optimizer_success": true,
  "optimizer_message": "Optimization terminated successfully.",
  "x_opt": {
    "P_boiler_Pa": 6054123.135827588,
    "T_boiler_K": 893.149354046388,
    "P_condenser_Pa": 11178.07411262336,
    "P_boiler_MPa": 6.054123135827588,
    "T_boiler_C": 619.999354046388,
    "P_condenser_kPa": 11.17807411262336
  },
  "objective_efficiency": 0.3995908431078804,
  "checks": {
    "finite_inputs": {
```

```
    "pass": true,
    "value": {
      "P_boiler_Pa": 6054123.135827588,
      "T_boiler_K": 893.149354046388,
      "P_condenser_Pa": 11178.07411262336
    },
    "threshold": "all finite",
    "detail": "Decision variables must be finite scalars."
  },
  "positive_inputs": {
    "pass": true,
    "value": {
      "P_boiler_Pa": 6054123.135827588,
      "T_boiler_K": 893.149354046388,
      "P_condenser_Pa": 11178.07411262336
    },
    "threshold": "> 0",
    "detail": "Pressure and temperature must be positive."
  },
  "pressure_hierarchy": {
    "pass": true,
    "value": {
      "P_boiler_Pa": 6054123.135827588,
      "P_condenser_Pa": 11178.07411262336
    },
    "threshold": "P_boiler > P_condenser",
    "detail": "Rankine high pressure should exceed condenser pressure."
  },
  "energy_balance": {
    "pass": true,
    "value": -2.3283064365386963e-10,
    "threshold": 3.498208290918267,
    "detail": "q_in - q_out - w_net should be approximately zero."
  },
  "efficiency_consistency": {
    "pass": true,
    "value": {
      "eta_from_components": 0.3995908431078804,
      "eta_from_objective": 0.3995908431078804
    },
    "threshold": 1e-06,
    "detail": "Efficiency from states should match optimizer objective."
  },
  "q_in_positive": {
    "pass": true,
    "value": 3498208.290918267,
    "threshold": "> 0",
    "detail": "Boiler heat input must be positive."
  },
  "q_out_positive": {
    "pass": true,
    "value": 2100356.2905832594,
    "threshold": "> 0",
    "detail": "Condenser heat rejection must be positive."
  },
  "w_turbine_positive": {
    "pass": true,
    "value": 1403955.039119502,
    "threshold": "> 0",
    "detail": "Turbine work must be positive."
  },
  "w_pump_nonnegative": {
    "pass": true,
    "value": 6103.038784494478,
    "threshold": ">= 0",
    "detail": "Pump work should not be negative."
  },
  "w_net_positive": {
    "pass": true,
    "value": 1397852.0003350077,
    "threshold": "> 0",
    "detail": "Net cycle work should be positive."
  },
  "efficiency_bounds": {
    "pass": true,
    "value": 0.3995908431078804,
    "threshold": "0 < eta < 1",
    "detail": "Thermal efficiency should be between 0 and 1 for this cycle model."
  },
  "superheat_margin": {
    "pass": true,
    "value": 293.82777542261783,
    "threshold": ">= -1e-08",
    "detail": "Superheat margin after minimum required superheat."
  },
  "turbine_exit_quality_margin": {
    "pass": true,
    "value": {
      "margin": 3.145976806839812e-07,
      "quality": 0.8800003145976807
    },
    "threshold": ">= -1e-08",
          "detail": "Quality margin relative to minimum acceptable turbine exit quality."
        }
      },
      "warnings": [],
      "overall_validation_pass": true,
      "tolerances": {
        "energy_balance_abs_tol": 0.001,
        "energy_balance_rel_tol": 1e-06,
        "efficiency_tol": 1e-06,
        "constraint_tol": 1e-08,
        "minimum_superheat": 50.0,
        "minimum_quality": 0.88
      },
      "timestamp_utc": "2026-02-13T21:10:11.986993+00:00"
    }
  ]
}
```

These code changes were used to provide the information nessacary to write the report.