teqpflsh

Release 0.0.3

Ian Bell

Jan 09, 2025

CONTENTS:

1 Background **1**

2 SuperAncillary **3**
- 2.1 Superancillary functions 3
- 2.2 Superancillary iterations 13

3 Polygons and Regions **21**
- 3.1 Polygon operations 21
- 3.2 Polygon validation 27

4 K-D tree lookup **29**
- 4.1 K-D tree fundamentals 29

5 Regioned Flasher **33**

6 Complete Example **39**

7 Outlook **43**

8 teqpflsh **45**
- 8.1 teqpflsh Package 45

9 Indices and tables **59**

Index **61**

CHAPTER ONE

BACKGROUND

The approach in this library has the goal of making iterative calculations with thermodynamic models much more reliable. In some cases, more than 1000x faster without sacrificing any accuracy. The greatest speedups are possible for two-phase inputs where at least one of the variables is temperature or pressure

The core question to answer is: suppose I know two thermodynamic variables that are not temperature and density (say, pressure and enthalpy), how can I most efficiently determine the temperature and density, the independent variables of the equation of state?

Two enabling technologies are:

- **Superancillary equations**: A set of mathematical approximation functions, pre-calculated based on phase equilibrium calculations in extended precision arithmetic. These numerical functions are selected because they can represent the results of the phase equilibrium so accurately that the conventional phase equilibrium calculation can be replaced with a single functional evaluation, which is exceptionally fast (order of 10 ns instead of 10 us)
- **K-D trees**: An efficient numerical structure that allows for nearest neighbor lookup

For pure fluids, for a given set x, y of thermodynamic variables, the general approach for iterative calculations can be expressed as two simpler subproblems:

A. *Is the input single phase or does it correspond to equilibrium of two phases?*

B. *For a region of interest of the phase diagram, what is a good guess value for the temperature and density close to the final solution?*

Subproblem A is handled with superancillary equations. Internal iteration is required for some input pairs, but the iterations are based on the use of superancillary functions, so the subiterations are reliable and fast. This represents a generational improvement in the reliability and speed of phase determination from thermodynamic models

After subproblem A is complete, the phase of the inputs is known. If the inputs are two-phase, the overall problem is finished.

Next comes subproblem B. When the thermodynamic model was loaded, “lighthouse points” were densely populated throughout the entire single-phase portion of the phase diagram. These points were then passed into a K-D tree nearest-neighbor lookup structure. The entire single-phase region is partitioned into non-overlapping and non-intersecting regions.

In subproblem B, the regions are queried for their nearest point closest to the values satisfying the constraint equations. The lookup is very fast (order of ns), and once this guess value is known, a traditional Newton iteration is carried out to obtain the solution. If the routines are not reliable enough, the algorithm can be made arbitrarily reliable by increasing the number of points in the K-D tree. Of course, that reliability comes at the cost of additional memory required for the storage.

CHAPTER

TWO

SUPERANCILLARY

2.1 Superancillary functions

TODO: object hierarchy

TODO: rough description of algorithms

[1]:
```
import json
import timeit
import tarfile
import functools
import itertools
from dataclasses import dataclass

import numpy as np
import matplotlib.pyplot as plt

import ChebTools
import teqpflsh
import CoolProp.CoolProp as CP
```

Build a superancillary function for water

To begin, load from the provided files:

- $\rho'(T)$
- $\rho''(T)$
- $p(T)$

And then use the EOS (as implemented in CoolProp, but REFPROP would be fine too) to add

- $h'(T)$, $h''(T)$
- $s'(T)$, $s''(T)$
- $u'(T)$, $u''(T)$

[2]:
```
FLD = 'WATER'
with tarfile.open('superancillaryJSON.tar.xz', mode='r:xz') as tar:
    # for member in tar.getmembers(): print(member)
    j = json.load(tar.extractfile(f'./{FLD}_exps.json'))
sa = teqpflsh.SuperAncillary(json.dumps(j))

ca = sa.get_approx1d(k='D', q=0)
print('Water has non-monotonic rho\'(T). The monotonic intervals are:')
```

(continues on next page)

(continued from previous page)

```
for inter in ca.monotonic_intervals:
    print(f'({inter.xmin}, {inter.xmax}) K')
```

```
Water has non-monotonic rho'(T). The monotonic intervals are:
(273.16, 277.15003423906836) K
(277.15003423906836, 647.0959999999867) K
```

[3]:
```
AS = CP.AbstractState('HEOS', 'Water')
def calc(T, rho, AS, key):
    AS.specify_phase(CP.iphase_gas)
    AS.update(CP.DmolarT_INPUTS, rho, T)
    val = AS.keyed_output(key)
    AS.unspecify_phase()
    return val

# Add another thermodynamic variable to the superancillary
# Speed is order of ms per variable, likely MUCH faster in C++
# caller is a callable function that takes temperature and density and returns a␣
↪value of a given property type
# here we can avoid flash calculations because we take T,rho value from the␣
↪superancillary and
# get values for the "other" variable
sa.add_variable(k='H', caller=functools.partial(calc, AS=AS, key=CP.iHmolar))
sa.add_variable(k='S', caller=functools.partial(calc, AS=AS, key=CP.iSmolar))
sa.add_variable(k='U', caller=functools.partial(calc, AS=AS, key=CP.iUmolar))

# Here is the call signature for the method
print(sa.add_variable.__doc__)
```

```
add_variable(self, *, k: str, caller: collections.abc.Callable[[float, float],␣
↪float]) -> None
```

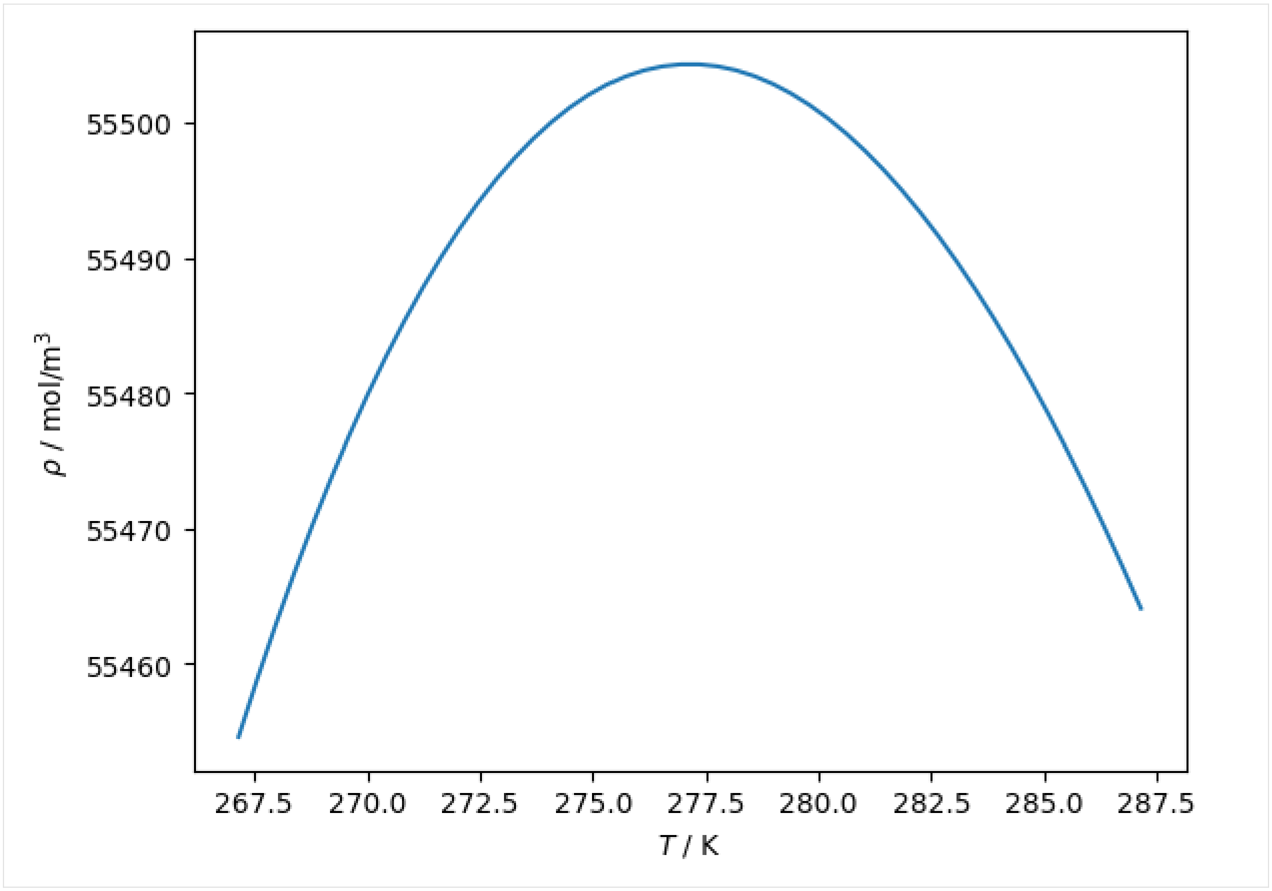
[4]:
```
# Solving for temperature given saturated liquid density, around the density maximum
# as a challenging test for the rootfinding. The approach works well, even very␣
↪close(!)
# to the extremum

ca = sa.get_approx1d(k='D', q=0)
print('The monotonic intervals are:')
for inter in ca.monotonic_intervals:
    print(f'({inter.xmin}, {inter.xmax}) K')

print('T at extrema in rho(T):', ca.x_at_extrema, 'K')
print('and corresponding value', ca.eval(ca.x_at_extrema[0]), 'mol/m³')

plt.figure()
Trange = np.linspace(-10, 10) + ca.x_at_extrema[0]
plt.plot(Trange, [ca.eval(T) for T in Trange])
plt.gca().set(xlabel=r'$T$ / K', ylabel=r'$\rho$ / mol/m$^3$')

# Starting at a density below the extremum, test getting
# very close to the extremum and solving for temperature

plt.figure()
y_extremum = ca.eval(ca.x_at_extrema[0])
delta = y_extremum - 0.9999*y_extremum
```

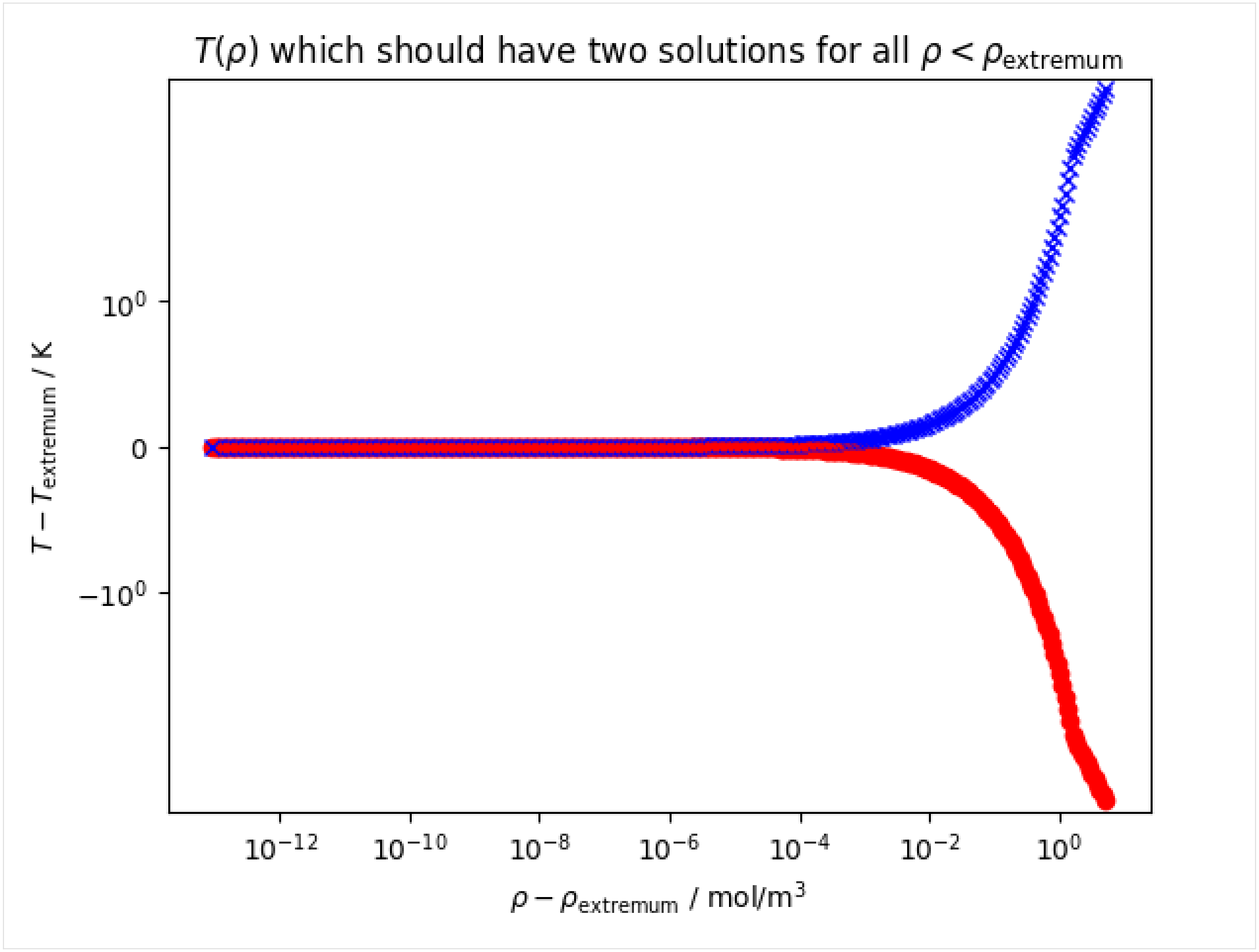
(continues on next page)

(continued from previous page)

```
while delta > 1e-13:
    # First output argument is the solution, second is the number of iterations␣
↪required
    Tsoln = ca.get_x_for_y(y=y_extremum-delta)

    if len(Tsoln) != 2:
        break
    if Tsoln[0][0] > Tsoln[1][0]:
        break

    delta /= 1.1
    plt.plot(delta, Tsoln[0][0]-ca.x_at_extrema[0], 'ro')
    plt.plot(delta, Tsoln[1][0]-ca.x_at_extrema[0], 'bx')

plt.xscale('log')
plt.yscale('symlog')
plt.gca().set(xlabel=r'$\rho-\rho_{\rm extremum}$ / mol/m$^3$', ylabel=r'$T-T_{\rm␣
↪extremum}$ / K')
plt.title(r'$T(\rho)$ which should have two solutions for all $\rho < \rho_{\rm␣
↪extremum}$')

plt.show()
```

```
The monotonic intervals are:
(273.16, 277.15003423906836) K
(277.15003423906836, 647.0959999999867) K
T at extrema in rho(T): [277.15003423906836] K
and corresponding value 55504.316178396366 mol/m³
```


55500
55490
55480
55470
55460
ρ / mol/m^3
267.5
270.0
272.5
275.0
277.5
280.0
282.5
285.0
287.5
T / K



```
[5]: # Time forward evalution of a chebyshev expansion
T = np.linspace(273.16, 290, 1000000)
ybuf = np.zeros_like(T)
tic = timeit.default_timer()
ca.eval_many(T, ybuf)
toc = timeit.default_timer()
print('rho(T) takes', (toc-tic)/len(T)*1e6, 'μs/call')

# Time the rootfinding in the superancillary
tic = timeit.default_timer()
ybuf = np.linspace(55400, 55503, 1000000)
xbuf = np.zeros_like(ybuf)
ca.count_x_for_y_many(ybuf, 64, 100, 1e-10, xbuf)
toc = timeit.default_timer()
print('T(rho) takes', (toc-tic)/len(xbuf)*1e6, 'μs/call')
print('so the inversion is much slower, and here there are two candidate regions, so␣
↪it is again two times worse than normal fluids, which would be more like:')

tic = timeit.default_timer()
ybuf = np.linspace(20400, 20503, 1000000)
xbuf = np.zeros_like(ybuf)
ca.count_x_for_y_many(ybuf, 64, 100, 1e-10, xbuf)
toc = timeit.default_timer()
print('T(rho) takes', (toc-tic)/len(xbuf)*1e6, 'μs/call when there is only one␣
```

(continues on next page)

(continued from previous page)

```
↪solution')
```

```
rho(T) takes 0.016125665977597237 µs/call
T(rho) takes 0.6450482499785721 µs/call
so the inversion is much slower, and here there are two candidate regions, so it is␣
↪again two times worse than normal fluids, which would be more like:
T(rho) takes 0.45679099997505546 µs/call when there is only one solution
```

Rootfinding is based on the TOMS748 method, which is an advanced version of the Brent method that is bounded and uses the optimal combination of secant, quadratic, and cubic interpolation mixed with bisection.

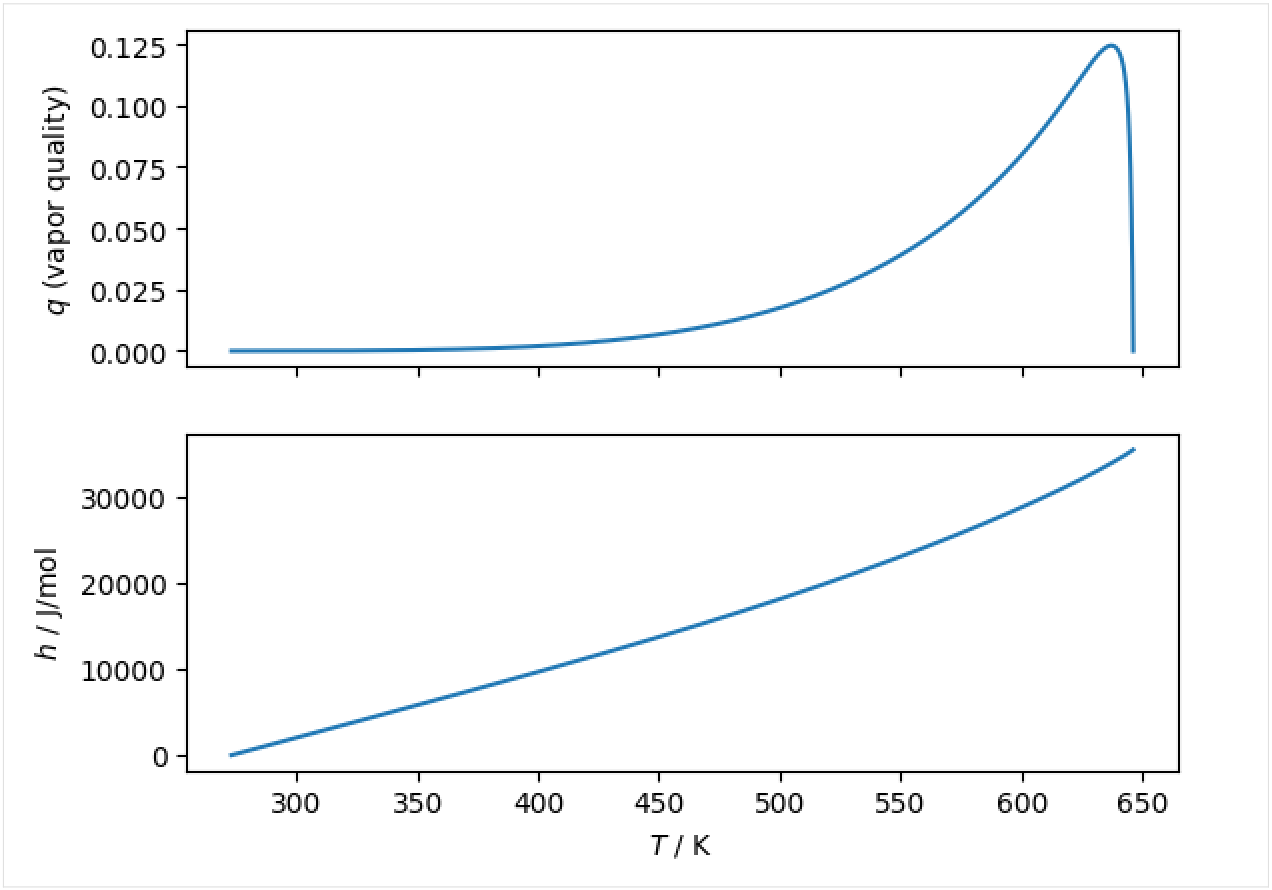
```
[6]: T_K, steps = ca.get_x_for_y(y=20250)[0]
     print(f'{steps} iterations were required')
```

```
7 iterations were required
```

which is about that much higher than the forward evaluation of a superancillary itself

```
[7]: ca = sa.get_approx1d(k='D', q=0)
     rhoc = ca.monotonic_intervals[1].ymin

     # Pick a density where there is only one possible solution for temperature
     for D in [22082.571366851185]:

         # Get the saturation temperature, if possible
         Tlims = [_ for _ in ca.get_x_for_y(y=D)]
         if len(Tlims) == 1:
             Trange = [ca.expansions[0].xmin, Tlims[0][0]]
         else:
             Trange = [Tlims[0][0], Tlims[1][0]]

         Ts = np.linspace(*Trange, 100000)
         # Non-iteratively solve for q for value of density
         q = np.array([sa.get_vaporquality(T=T_, propval=D, k='D') for T_ in Ts])

         fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)
         # And then plot another property
         y = np.zeros_like(q)
         tic = timeit.default_timer()
         sa.get_yval_many(q=q, T=Ts, k='H', y=y)
         toc = timeit.default_timer()
         print((toc-tic)/len(T)*1e6, 'µs/call')
         ax1.plot(Ts, q,)
         ax2.plot(Ts, y, label=D)
         ax1.set(ylabel='$q$ (vapor quality)')
         ax2.set(xlabel='$T$ / K', ylabel='$h$ / J/mol')
```

```
0.005306542036123574 µs/call
```



Plot lines of constant quality in the two-phase region according to the superancillary functions

```
[8]: eps = 1e-6
Tt = 273.16
Tc = 647.0959999999867

for T in np.linspace(Tt, Tc-eps, 10000):
    p = sa.get_yval(T=T, q=1, k='P')
    Tnew = sa.get_T_from_p(p=p)
    DELTAT = Tnew-T
    if abs(DELTAT) > 0.001:
        print(T, p, Tnew-T)
```

```
[9]: caV = sa.get_approx1d(k='D', q=1)
# for interval in caV.monotonic_intervals:
#     print(interval)
#     print({k: getattr(interval,k) for k in dir(interval) if not k.startswith('__')})
#     for m in interval.expansioninfo:
#         print({k: getattr(m,k) for k in dir(m) if not k.startswith('__')})
#         print(m.xmin, m.xmax)
print(caV.eval(273.16), caV.eval(647.096), caV.get_x_for_y(y=200))

# print([0].ymin, caV.monotonic_intervals[0].ymax)
# print(caV.get_x_for_y(y=200))

ca = sa.get_approx1d(k='D', q=0)
```

(continues on next page)

(continued from previous page)

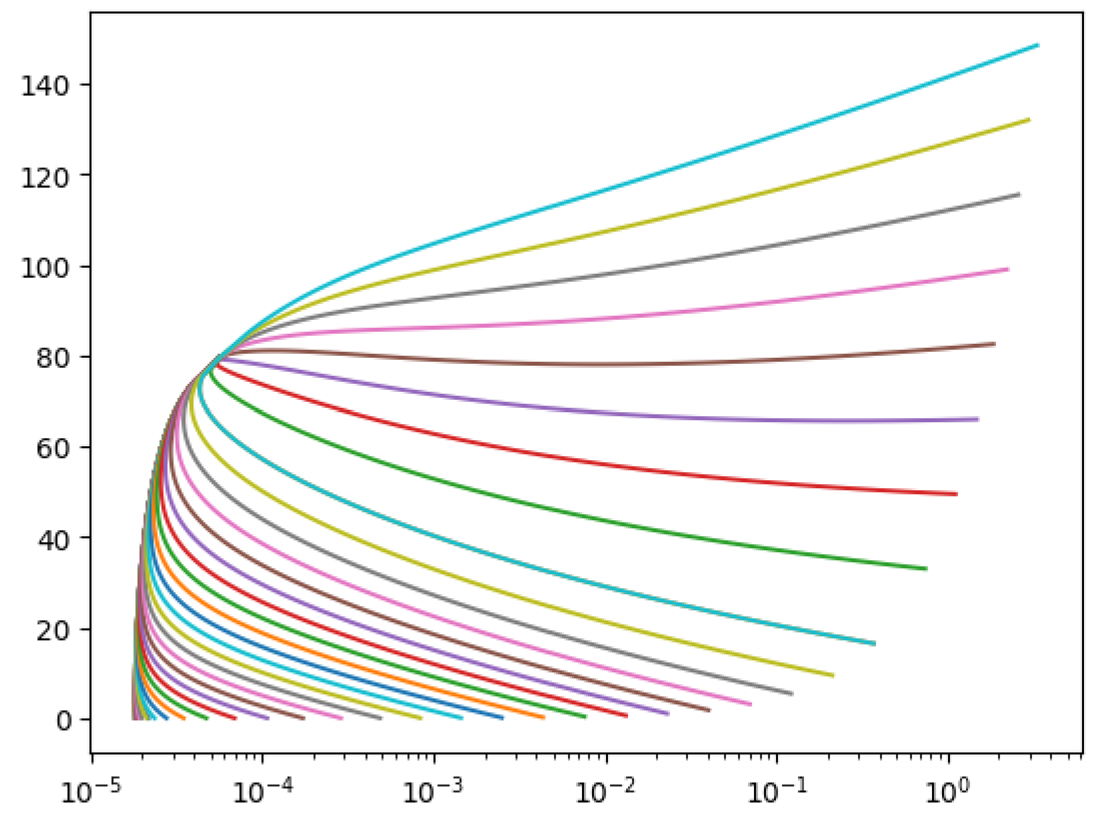
```
y = ca.eval(ca.x_at_extrema[0])*0.9999
Tlims = [_[0] for _ in ca.get_x_for_y(y=y)]

Ts = np.linspace(ca.expansions[0].xmin+1e-6, ca.expansions[-1].xmax-1e-6, 100)
# Ts = np.linspace(273.1600001, 280, 10000)

for Q in np.arange(1e-6, 1.0000, 0.1, dtype=float).tolist() + np.logspace(-8, -1, 30).
↪tolist():
    Qs = Q*np.ones_like(Ts)

    rho = np.zeros_like(Ts)
    sa.get_yval_many(T=Ts, q=Qs, k='D', y=rho)

    other = np.zeros_like(Ts)
    kother = 'S'
    sa.get_yval_many(T=Ts, q=Qs, k=kother, y=other)
    plt.plot(1/rho, other)

    Tbuf = np.zeros_like(Ts)
    qbuf = np.zeros_like(Ts)
    countbuf = np.zeros_like(Ts)
    tic = timeit.default_timer()
    sa.solve_for_Tq_DX_many(rho, other, kother, 64, 100, 1e-10, Tbuf, qbuf, countbuf)
    toc = timeit.default_timer()
    print((toc-tic)/len(Tbuf)*1e6, 'µs/call from', np.mean(countbuf), 'steps on␣
↪average')

    for T_goal_, rho_, other_ in zip(Ts, rho, other):
        soln = sa.solve_for_Tq_DX(rho_, other_, kother, 64, 100, 1e-10)
        try:
            T_ = soln.T; q_ = soln.q; count_ = soln.counter
        except BaseException as be:
            print(rho_, other_)
            print(be, T_goal_, Q)
            plt.plot(1/rho_, other_, 'o')

# plt.yscale('log')
plt.xscale('log')
```

```
0.2694700808656397 17873.7282300601 [(437.3594696512889, 10)]
1.4787499094381928 µs/call from 5.3 steps on average
1.970840385183692 µs/call from 8.17 steps on average
2.230409882031381 µs/call from 9.6 steps on average
2.2991601144894958 µs/call from 10.12 steps on average
2.3312499979510903 µs/call from 10.27 steps on average
2.40708002820611 µs/call from 10.58 steps on average
2.4075002875179052 µs/call from 10.59 steps on average
2.4258403573185205 µs/call from 10.79 steps on average
2.9266695491969585 µs/call from 10.46 steps on average
2.4391600163653493 µs/call from 9.92 steps on average
1.431250129826367 µs/call from 5.23 steps on average
1.4441600069403648 µs/call from 5.32 steps on average
1.497499761171639 µs/call from 5.2 steps on average
1.4208396896719933 µs/call from 5.19 steps on average
1.4154094969853759 µs/call from 5.2 steps on average
1.4266703510656953 µs/call from 5.25 steps on average
1.413749996572733 µs/call from 5.23 steps on average
```

(continues on next page)

(continued from previous page)

```
1.5320797683671117 µs/call from 5.3 steps on average
1.4416599879041314 µs/call from 5.36 steps on average
1.4595797983929515 µs/call from 5.56 steps on average
1.5179102774709463 µs/call from 5.65 steps on average
1.5229103155434132 µs/call from 5.7 steps on average
1.5283300308510661 µs/call from 5.79 steps on average
1.5508296201005578 µs/call from 5.95 steps on average
1.604580320417881 µs/call from 6.03 steps on average
1.5791598707437515 µs/call from 6.08 steps on average
1.590839819982648 µs/call from 6.13 steps on average
1.5712500317022204 µs/call from 6.05 steps on average
1.6354199033230543 µs/call from 6.2 steps on average
1.5858298866078258 µs/call from 6.12 steps on average
1.5791598707437515 µs/call from 6.11 steps on average
1.5845795860514045 µs/call from 6.15 steps on average
1.5899998834356666 µs/call from 6.2 steps on average
1.6104202950373292 µs/call from 6.3 steps on average
1.736659905873239 µs/call from 6.71 steps on average
1.7420802032575011 µs/call from 6.94 steps on average
1.8787500448524952 µs/call from 7.27 steps on average
1.9300001440569758 µs/call from 7.41 steps on average
1.8570898100733757 µs/call from 7.45 steps on average
1.9204203272238374 µs/call from 8.04 steps on average
```



```
[10]: eps = 1e-6

Tt = 273.16
```

(continues on next page)

(continued from previous page)

```
Tc = 647.0959999999867

# A class storing the info for a single two-phase point
@dataclass
class TwoPhasePoint:
    T: float
    Q: float
    D: float
    H: float
    S: float
    U: float
    P: float

@dataclass
class TwoPhaseResult:
    Terr: float
    Qerr: float
    elap_us: float
    count: float
    proppair: list[str]

# Build up database of points for two-phase data
points = []
for T in np.linspace(Tt, Tc-eps, 300):
    for q in np.linspace(eps, 1-eps, 500):
        pt = TwoPhasePoint(
            T = T,
            Q = q,
            P = sa.get_yval(T=T, q=q, k='P'),
            D = sa.get_yval(T=T, q=q, k='D'),
            H = sa.get_yval(T=T, q=q, k='H'),
            S = sa.get_yval(T=T, q=q, k='S'),
            U = sa.get_yval(T=T, q=q, k='U')
        )
        points.append(pt)

keys = ['D', 'H', 'S', 'U', 'P', 'T']

results = []
for proppair in itertools.combinations(keys, 2):
    proppair = sorted(proppair)
    if proppair == ['H', 'U']: continue
    if proppair == ['P', 'T']: continue

    val1 = np.array([getattr(pt, proppair[0]) for pt in points])
    val2 = np.array([getattr(pt, proppair[1]) for pt in points])

    flash_key = teqpflsh.get_pair_from_chars(*proppair)
    tic = timeit.default_timer()
    T = np.zeros_like(val1)
    q = np.zeros_like(val2)
    count = np.zeros_like(val2, dtype=int)
    sa.flash_many(flash_key, val1, val2, T, q, count)
    toc = timeit.default_timer()

    valQ = np.array([getattr(pt, 'Q') for pt in points])
    valT = np.array([getattr(pt, 'T') for pt in points])
```

(continues on next page)

(continued from previous page)

```
    DELTAT = np.abs((valT-T))
    badsolns = sum(T < 0) #

    results.append(TwoPhaseResult(
        elap_us=(toc-tic)/len(val1)*1e6,
        Terr=float(np.mean(np.abs((valT-T)))),
        Qerr=float(np.mean(np.abs((valQ-q)))),
        count=float(np.mean(count)),
        proppair=proppair
    ))

for el in sorted(results, key=lambda x: x.elap_us):
    print(el)
```

```
TwoPhaseResult(Terr=0.0, Qerr=2.795420235938882e-16, elap_us=0.05967111326754093,␣
↪count=0.0, proppair=['D', 'T'])
TwoPhaseResult(Terr=0.0, Qerr=8.929621463779706e-16, elap_us=0.06645833336127302,␣
↪count=0.0, proppair=['T', 'U'])
TwoPhaseResult(Terr=0.0, Qerr=1.1279329866925603e-15, elap_us=0.06656027981080115,␣
↪count=0.0, proppair=['S', 'T'])
TwoPhaseResult(Terr=0.0, Qerr=7.645675477278846e-16, elap_us=0.06656500006405015,␣
↪count=0.0, proppair=['H', 'T'])
TwoPhaseResult(Terr=7.164476301113609e-13, Qerr=7.684752486105178e-10, elap_us=0.
↪12980527981805304, count=0.0, proppair=['D', 'P'])
TwoPhaseResult(Terr=7.164476301113609e-13, Qerr=7.684694347776641e-10, elap_us=0.
↪1314924998829762, count=0.0, proppair=['P', 'S'])
TwoPhaseResult(Terr=7.164476301113609e-13, Qerr=7.684689638248323e-10, elap_us=0.
↪13159055340414247, count=0.0, proppair=['P', 'U'])
TwoPhaseResult(Terr=7.164476301113609e-13, Qerr=7.684687278664324e-10, elap_us=0.
↪13510666671209037, count=0.0, proppair=['H', 'P'])
TwoPhaseResult(Terr=9.051291272044182e-14, Qerr=2.2156901951207798e-11, elap_us=2.
↪0759752734253807, count=0.0, proppair=['S', 'U'])
TwoPhaseResult(Terr=1.6613512343610638e-13, Qerr=1.0721330329692832e-10, elap_us=2.
↪202141946569706, count=0.0, proppair=['H', 'S'])
TwoPhaseResult(Terr=2.3948511322184155e-14, Qerr=2.2876501559810492e-11, elap_us=2.
↪232273886911571, count=0.0, proppair=['D', 'S'])
TwoPhaseResult(Terr=2.379654991576293e-14, Qerr=1.6898106011334103e-11, elap_us=2.
↪3842438865297786, count=0.0, proppair=['D', 'U'])
TwoPhaseResult(Terr=2.4474881380835237e-14, Qerr=1.7895061607415564e-11, elap_us=2.
↪4198447268766663, count=0.0, proppair=['D', 'H'])
```

2.2 Superancillary iterations

The state of a two-phase point for a pure fluid can be fully specified by the temperature and vapor quality.

When temperature T is known, the vapor quality q can be calculated as

$$q = \frac{y - y'_{\rm ch}(T)}{y''_{\rm ch}(T) - y'_{\rm ch}(T)}$$

where y is a property of interest, one of $\{h, s, u\}$. In the case of density, one uses $v = 1/\rho$. If $0 \leq q \leq 1$ the state point is considered to be single-phase.

For a pure fluid there is no distinction between quality on a mass or molar basis because the molar mass of both phases are identical.

If p is given, one obtains $T(p)$ and uses the calculated T to get the quality.

When neither T nor p are given, one must work harder to determine the temperature and quality. That is the subject of this section.

The residual function to be driven to zero can be implemented as something like:

```
double resid(double T){
 double q_fromv1 = get_vaporquality(T, val1, ch1);
 return get_yval(T, q_fromv1, ch2) - val2;
};
```

One calculates the vapor quality with one of the imposed variables, calculates the other variable's value, and the residual to be driven to zero is then the difference between the given and calculated values of the second variable.

What appears on its face to be a simple residual function is complicated in practice. The complication occurs because one must either

1. Get a good guess value for the temperature to launch the iteration from and do some sort of unbounded Newton solver from this temperature
2. Develop reliable bounds on the temperature that bounds the solution (if such a solution exists).

A solution exists if you can find a T, q pair that gives the specified values of both variables. The solution should be unique in most cases, h, u being an exception.

[1]:
```
import json
import functools
import tarfile

import numpy as np
import matplotlib.pyplot as plt

import teqpflsh
import CoolProp.CoolProp as CP
```

[2]:
```
FLD = 'PENTANE'
# Extract the JSON from the archive in LZMA compressed format, name is the REFPROP␣
↪standard name
with tarfile.open('superancillaryJSON.tar.xz', mode='r:xz') as tar:
    # for member in tar.getmembers(): print(member)
    j = json.load(tar.extractfile(f'./{FLD}_exps.json'))
```

[3]:
```
sa = teqpflsh.SuperAncillary(json.dumps(j))

AS = CP.AbstractState('HEOS',FLD)
def calc(T, rho, AS, key):
    AS.specify_phase(CP.iphase_gas)
    AS.update(CP.DmolarT_INPUTS, rho, T)
    val = AS.keyed_output(key)
    AS.unspecify_phase()
    return val

# Order of ms per variable, likely MUCH faster in C++
sa.add_variable(k='S', caller=functools.partial(calc, AS=AS, key=CP.iSmolar))
sa.add_variable(k='H', caller=functools.partial(calc, AS=AS, key=CP.iHmolar))

def plot_base(ax):
    def plot_sat(q):
```

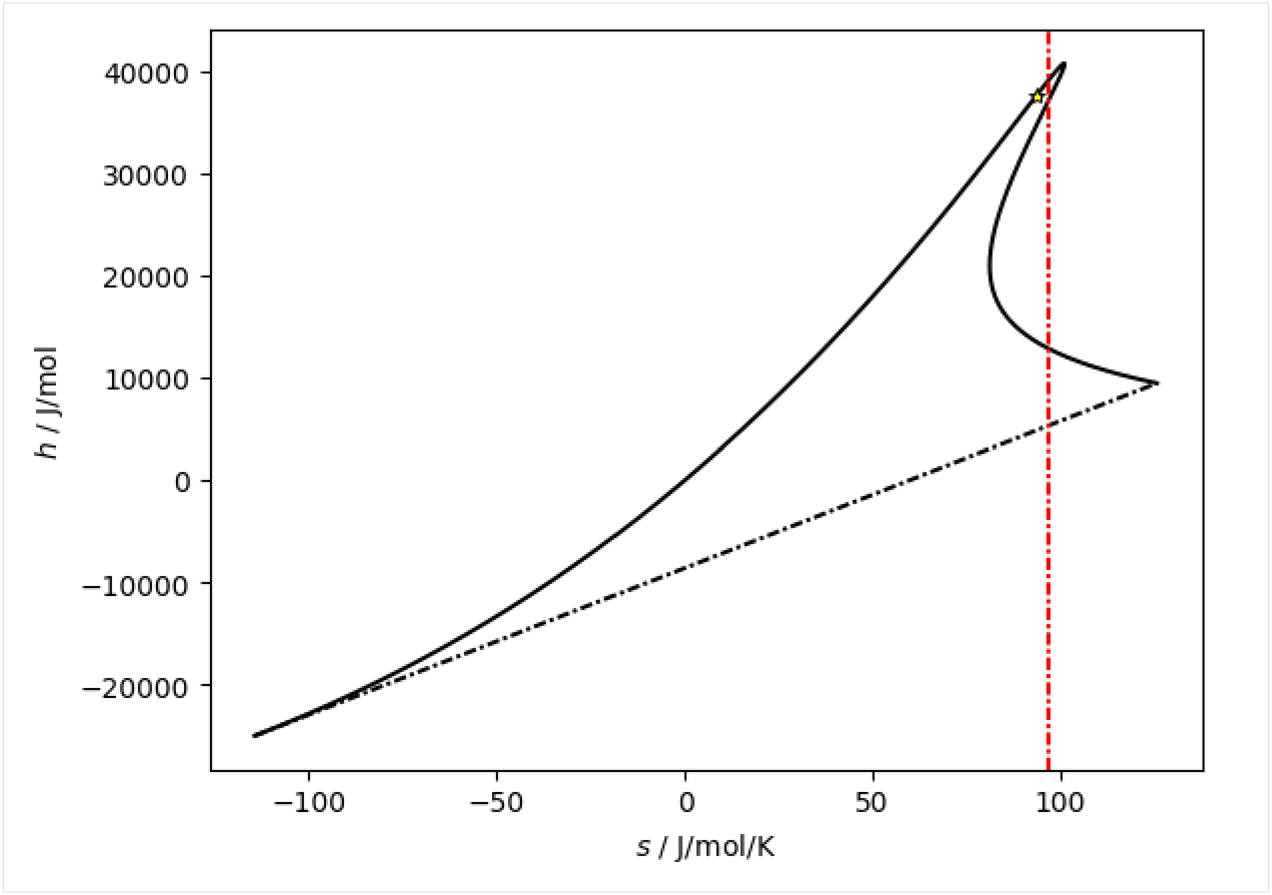
(continues on next page)

(continued from previous page)

```
        approx1dh = sa.get_approx1d(k='H', q=q)
        approx1ds = sa.get_approx1d(k='S', q=q)

        Ts = np.linspace(approx1dh.xmin, approx1dh.xmax, 10000)
        S = np.zeros_like(Ts)
        sa.eval_sat_many(k='S', T=Ts, q=q, y=S)
        H = np.zeros_like(Ts)
        sa.eval_sat_many(k='H', T=Ts, q=q, y=H)

        ax.plot(S, H, color='k')
        return S[0], H[0]

    s0, h0 = plot_sat(q=0)
    s1, h1 = plot_sat(q=1)
    ax.plot([s0,s1], [h0,h1], dashes=[3,1,1,1], color='k')

    ax.set(xlabel='$s$ / J/mol/K', ylabel='$h$ / J/mol')

Tc = sa.get_approx1d(k='S', q=1).xmax
```

The problems begin with entropy as an input. Let's suppose that we consider the following h, s coordinates for n-pentane. Along the saturated vapor curve, there are three intersections at the given value of entropy

```
[4]: fig, ax = plt.subplots(1,1)
     plot_base(ax)
     hc = sa.get_approx1d(k='H', q=1).eval(Tc)
     sc = sa.get_approx1d(k='S', q=1).eval(Tc)
     plt.plot(sc, hc, '*', color='yellow', mew=0.7, mec='k')
     ax.axvline(97, color='red', dashes=[3,1,1,1]);
```



[5]:
```python
Tpt = 235; q = 0.3
hptA = sa.get_yval(k='H', T=Tpt, q=q)
sptA = sa.get_yval(k='S', T=Tpt, q=q)

Tpt = 0.9*Tc; q = 0.95
hptB = sa.get_yval(k='H', T=Tpt, q=q)
sptB = sa.get_yval(k='S', T=Tpt, q=q)

hptC = hptB-20000
sptC = sptB

def overlay_points(points, ofname, *, suptitle=None, ylabels=None):

    figg, axes = plt.subplots(1+len(points), 1, sharex=True)

    approx1dh = sa.get_approx1d(k='H', q=0)
    approx1ds = sa.get_approx1d(k='S', q=0)
    Ts = (np.geomspace(approx1dh.xmin, 0.9*approx1dh.xmax, 1000).tolist()
    + np.geomspace(0.9*approx1dh.xmax, approx1dh.xmax*0.999999, 100000).tolist())

    for ipt, (hpt, spt) in enumerate(points):

        axmain.plot(spt, hpt, 'o')

        r, Qcalc = [], []
        for T in Ts:
```

(continues on next page)

(continued from previous page)

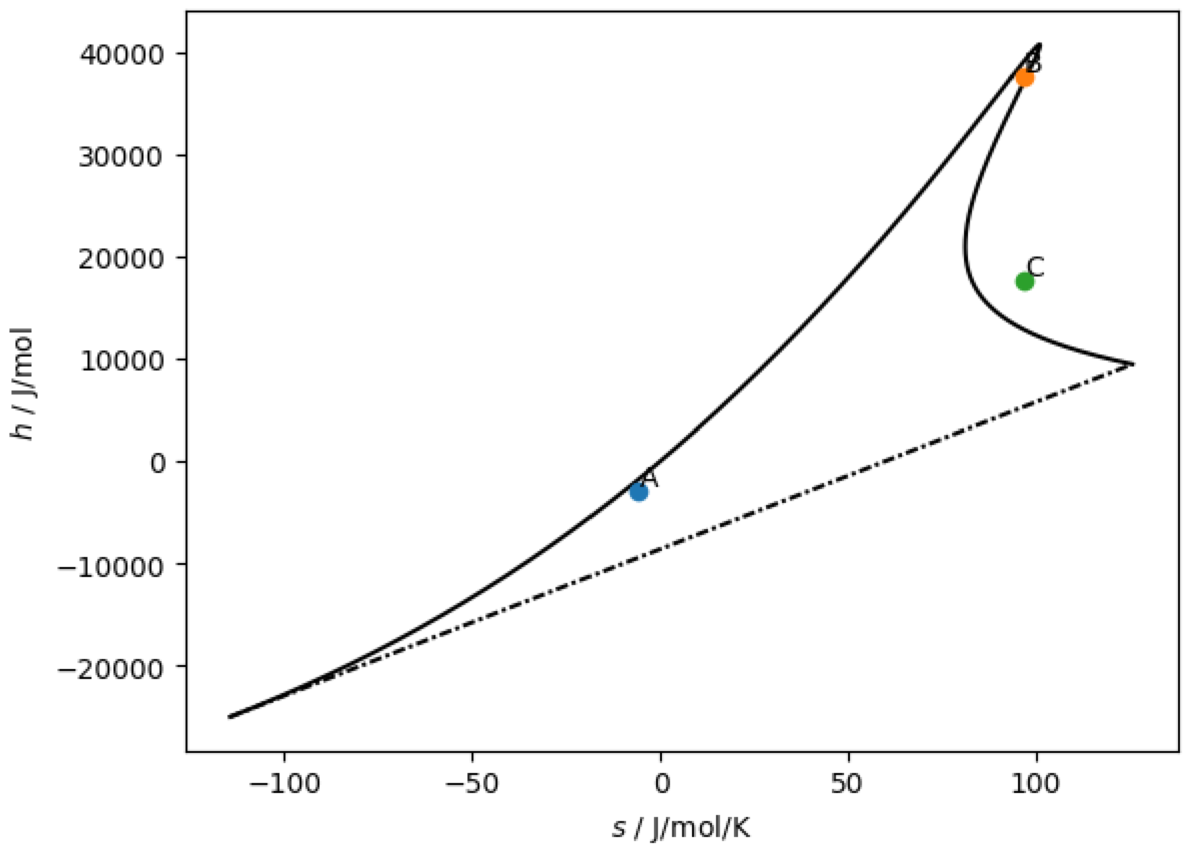
```
            qq = sa.get_vaporquality(T=T, k='S', propval=spt)
            qh = sa.get_vaporquality(T=T, k='H', propval=hpt)
            r.append(sa.get_yval(T=T, q=qq, k='H')-hpt)
            Qcalc.append(qq)

        solns = (
            sa.solve_for_T(propval=spt, k='S', q=True, bits=64, max_iter=100,␣
↪boundsftol=1e-13)
            + sa.solve_for_T(propval=spt, k='S', q=False, bits=64, max_iter=100,␣
↪boundsftol=1e-13)
        )
        if len(solns) == 3:
            bands = [(approx1dh.xmin, solns[0][0]),(solns[1][0], solns[2][0])]
        elif len(solns) == 2:
            bands = [(solns[0][0], solns[1][0])]
        elif len(solns) == 1:
            bands = [(approx1dh.xmin, solns[0][0])]
        else:
            raise ValueError(len(solns))

        if ipt == 0:
            ax1 = axes[0]
            ax1.plot(Ts, np.array(Qcalc))
            ax1.set_ylim(-0.5,1.5)
            ax1.set(ylabel=r'$q(s)$')
            ax1.axhspan(1,1.5, color='red', zorder=-1)
            ax1.axhspan(-0.5, 0, color='red', zorder=-1)
            for band in bands:
                ax1.axvspan(*band, color='lightgrey')
                ax1.text(np.mean(band), -0.25, '[interval]',ha='center', va='center')

        for band in bands:
            for T in band:
                qq = sa.get_vaporquality(T=T, k='S', propval=spt)
                qh = sa.get_vaporquality(T=T, k='H', propval=hpt)
                rr = sa.get_yval(T=T, q=qq, k='H')-hpt
                axes[ipt+1].plot(T, rr, 'ko')

        axx = axes[ipt+1]
        axx.plot(Ts, r)
        axx.axhline(0, dashes=[2, 2])
        axx.set_ylabel('$r$' + (f' {ylabels[ipt]}' if ylabels else ''))
        if ipt == len(points)-1:
            axx.set_xlabel('$T$ / K')
        if suptitle:
            figg.suptitle(suptitle)
    figg.tight_layout(pad=0.2)
    # figg.savefig(ofname)

figmain, axmain = plt.subplots(1)
plot_base(axmain)
points = [(hptA, sptA), (hptB, sptB), (hptC, sptC)]
overlay_points(points[0:1], ofname='liquid_side.pdf', suptitle = 'point A', ylabels=[
↪'A'])
overlay_points(points[1::], ofname='vapor_side.pdf', suptitle = 'point B&C', ylabels=[
↪'B','C'])
print('label (h,s coordinates)')
```

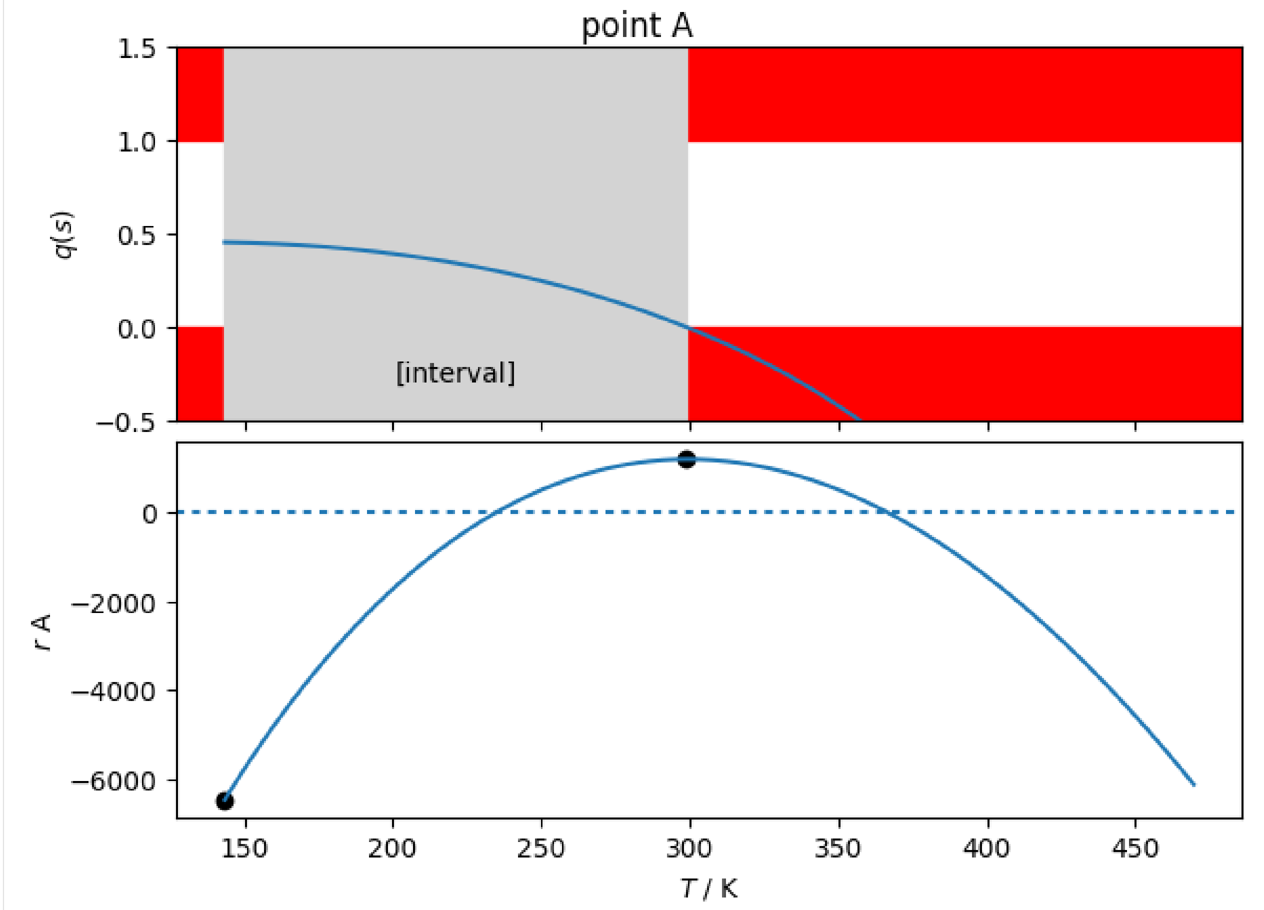
(continues on next page)

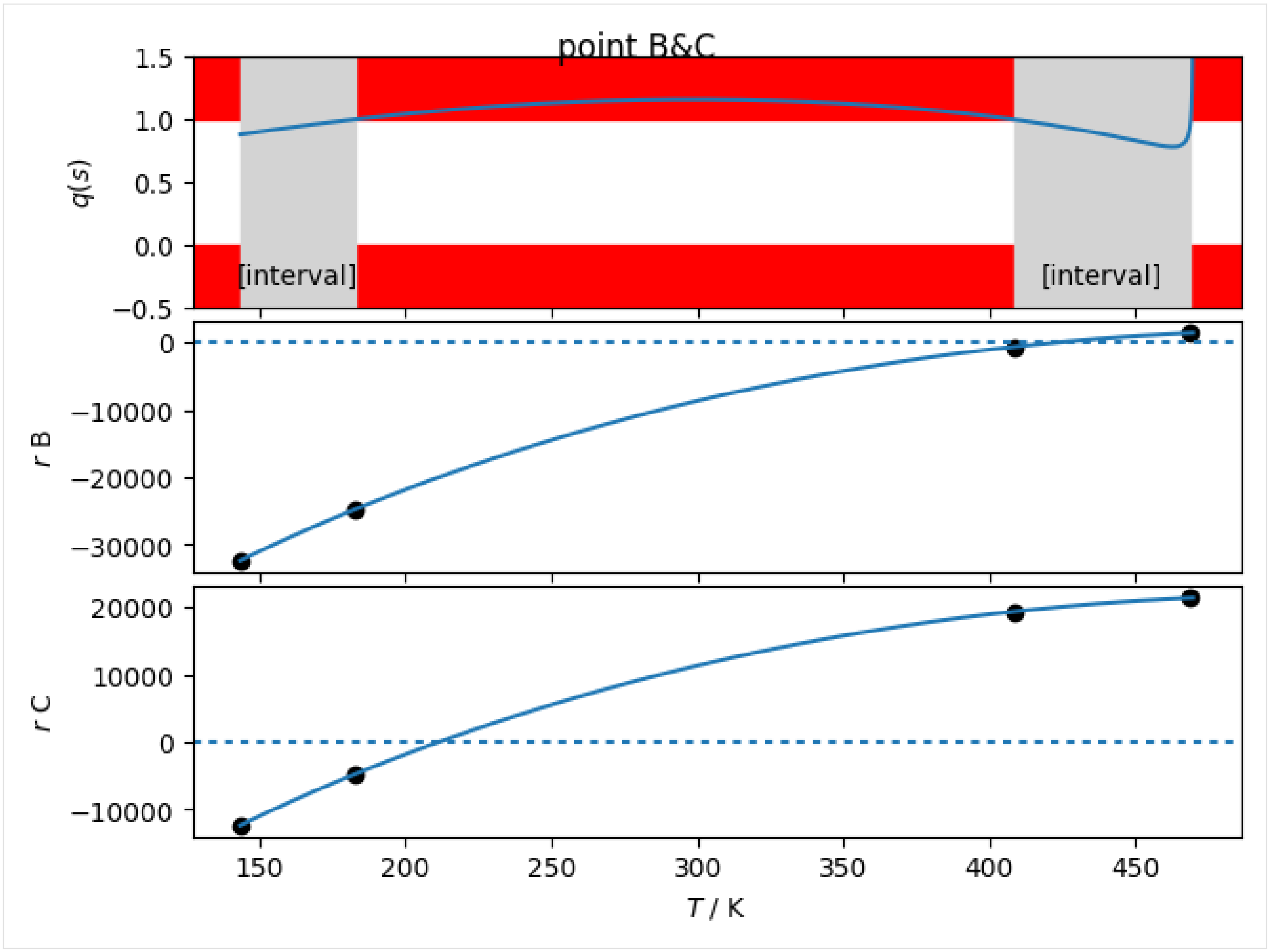
(continued from previous page)

```python
for point, label in zip(points, ['A','B','C']):
    print(label, list(reversed(point)))
    axmain.text(*reversed(point), label, ha='left', va='bottom')
# figmain.savefig(f'{FLD}_HS_main.pdf')
```

```
label (h,s coordinates)
A [-5.743214751940169, -2941.112588584603]
B [96.72730245435365, 37648.0894811022]
C [96.72730245435365, 17648.0894811022]
```




point A
1.5
1.0
0.5
0.0
−0.5
q(s)
[interval]
0
−2000
−4000
−6000
r A
150
200
250
300
350
400
450
T / K



We have three points: A, B, and C.

Point A is on the liquid side. When searching for values of entropy yielding $q = 0$ or $q = 1$ (breakpoints in the possible solution interval), only one solution is found. According to the intermediate value theorem, we know that the interval from $T_{\rm min}$ to the saturated liquid entropy contains one solution because the value of the residual function changes sign in this interval. Thus a bounded solver based on this solution interval can be practically guaranteed to converge. The superancillary routines are used to evaluate the residual function.

Point B and C are on on the vapor side at the same value of entropy. Along the given value of entropy, there are three values of saturated vapor entropy corresponding to the given value of entropy. Thus there are two possible search intervals. In each search interval, if the value of the residual function has the same sign on both edges, the solution can be guaranteed to not exist and the state point must be single-phase. Such is the case for point C. There are two candidate intervals, and in each interval the sign of the residual function is either both positive or both negative at the edges of the intervals. For point B, in one interval the residual function changes sign, and thus, a solution can be found using a bounded solver.

The TOMS748 algorithm is used within teqpflsh to do all bounded rootfinding of 1D residual functions.

CHAPTER

THREE

POLYGONS AND REGIONS

3.1 Polygon operations

Polygons are essential to the tools developed in teqpflsh. The GEOS C++ library is used for all the polygon operations. Profiling is provided in this file to indicate the computational speed of operations with this library.

One of the key operations in `teqpflsh` is to take an arbitrary polygon and sample it evenly. This is for instance how the single-phase points are distributed within the single-phase polygon(s). For efficiency, this process is done by first breaking up the non-intersecting polygon into a number of triangles. The reason for that process is that sampling a triangle evenly is easy, and you weight the samples of the triangles by their area to get the evenly sampled polygon. This generates samples within the polygon. To get there, first examples are shown of operations on polygons powered by the tools in `teqpflsh` followed by the even sampling.

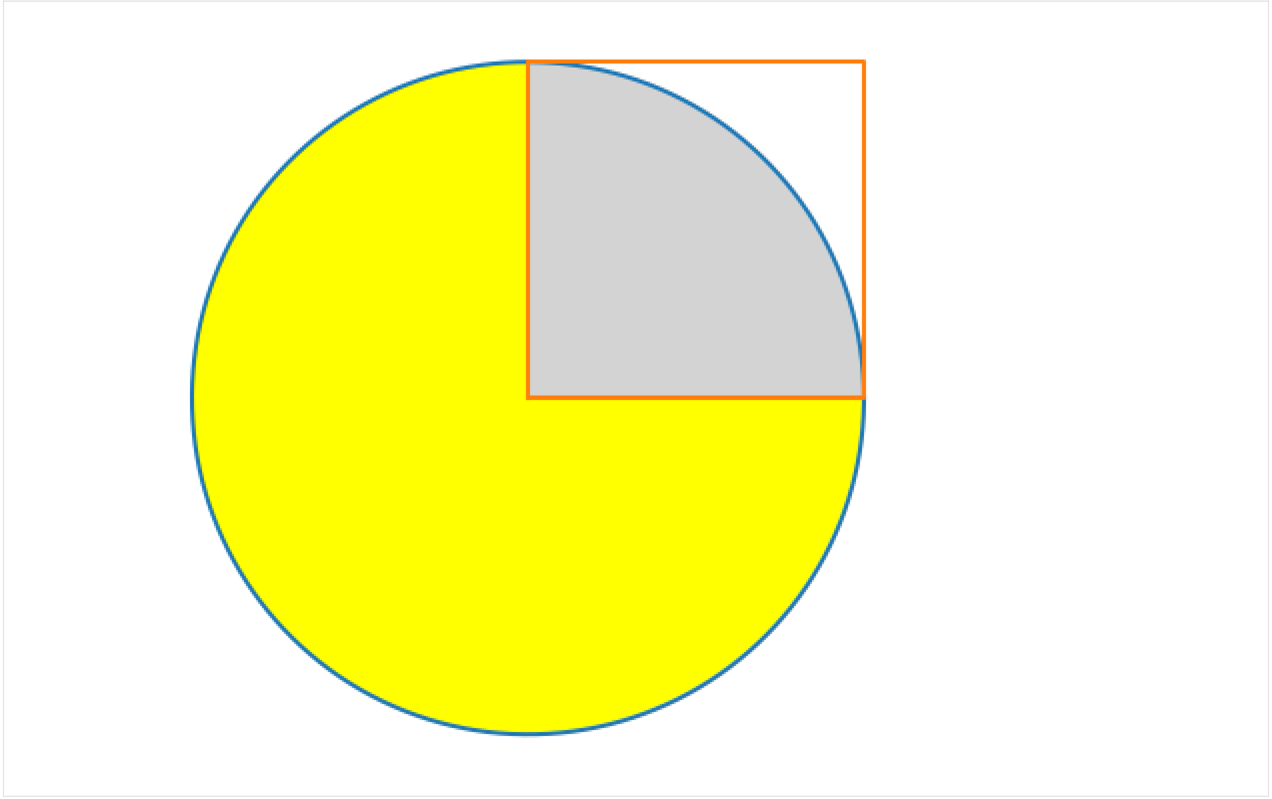
[1]:
```
import numpy as np
import teqpflsh
import matplotlib.pyplot as plt
import timeit

def getcircle(r, N, *, ptr):
    t = np.linspace(0, 2*np.pi, 10000)
    X,Y = np.cos(t), np.sin(t)
    return ptr.makeclosedpolygon(X,Y)
```

[2]:
```
# Here we do intersection and difference operations on simple
# polygons to demonstrate how polygon operations work
# in teqpflsh

ptr = teqpflsh.GeometryFactoryHolder()

# Polygon for a circle
t = np.linspace(0, 2*np.pi, 10000)
X,Y = np.cos(t), np.sin(t)
poly1 = ptr.makeclosedpolygon(X, Y)
plt.plot(X, Y)

# Polygon for a square
x = np.array([0,1,1,0,0])
y = np.array([0,0,1,1,0])
poly2 = ptr.makeclosedpolygon(x, y)
plt.plot(x, y)

# Intersection of the circle and the square
Xi, Yi = poly1.intersection(poly2).getXY()
```

(continues on next page)

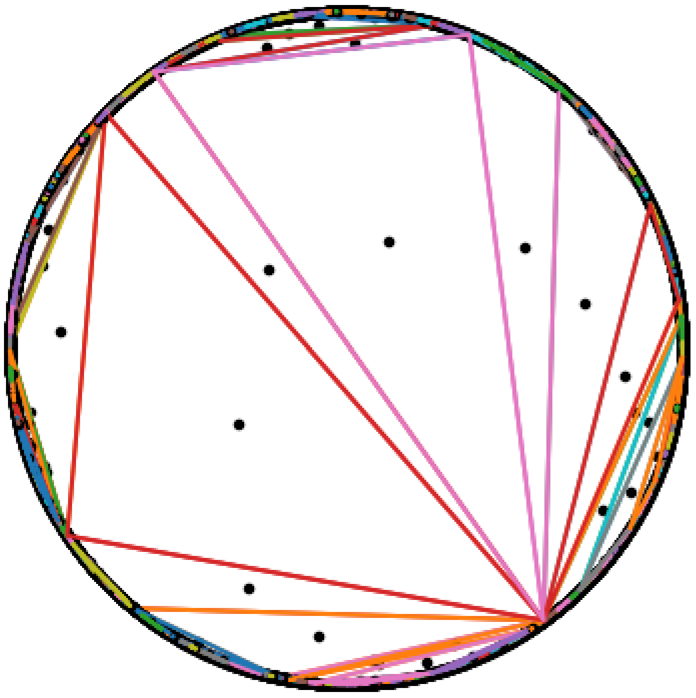
(continued from previous page)

```
plt.fill(Xi, Yi, 'lightgrey')

# Difference of the circle and the square
Xd, Yd = poly1.difference(poly2).getXY()
plt.fill(Xd, Yd, 'yellow')

plt.axis('equal')
plt.axis('off');
```

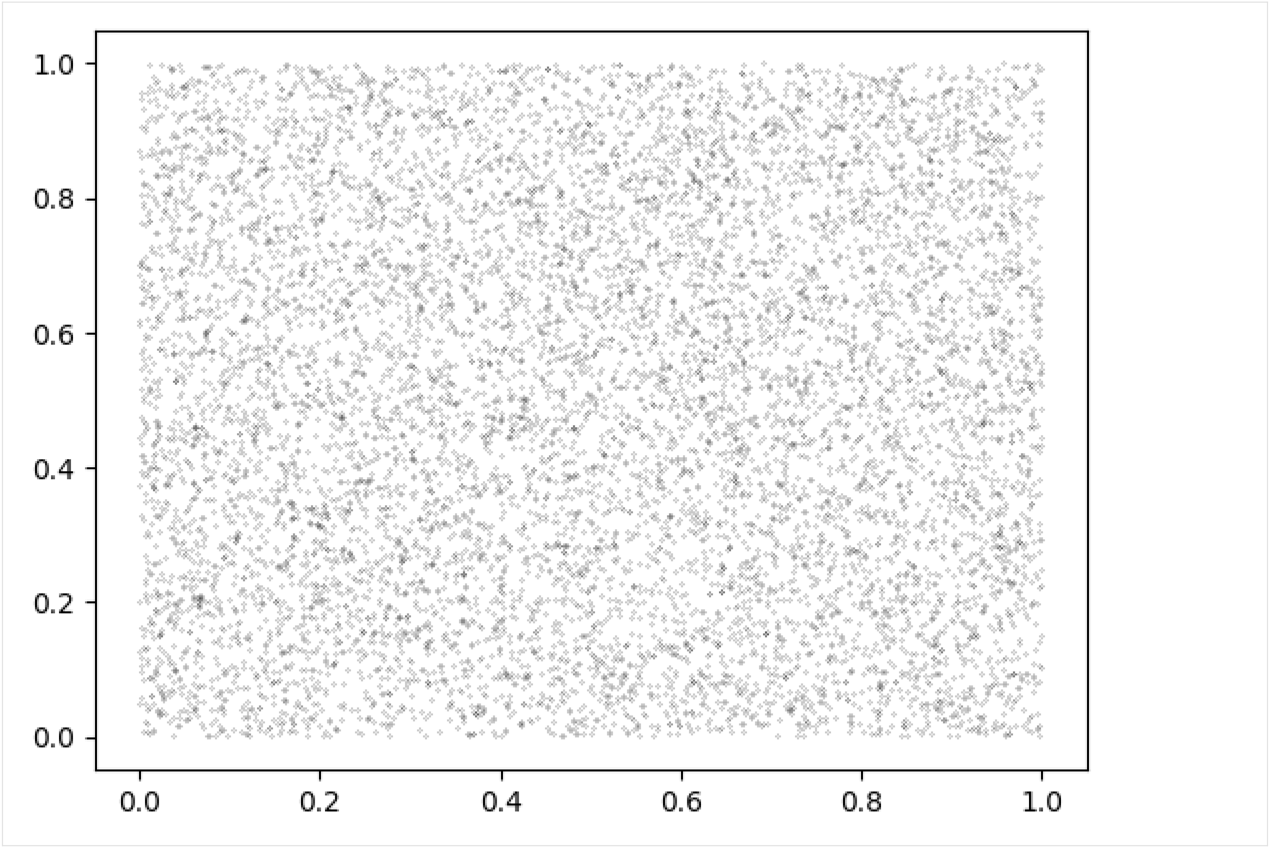


[3]:

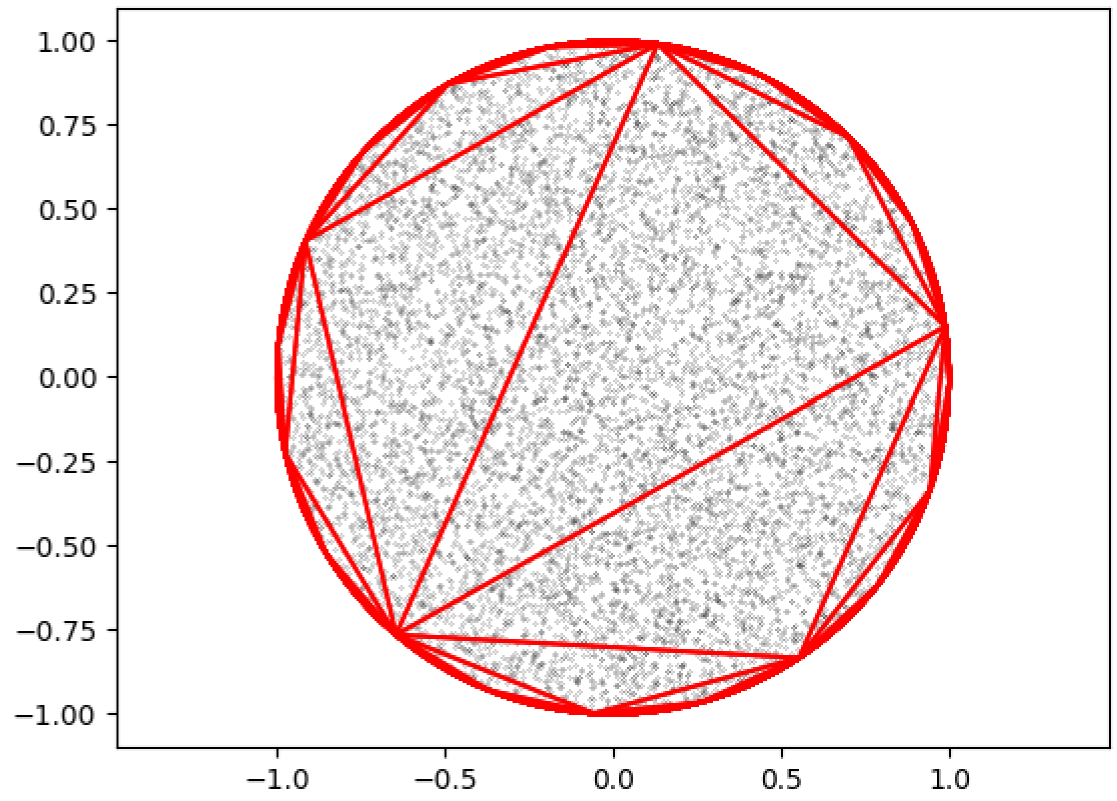
```
ptr = teqpflsh.GeometryFactoryHolder()

# Polygon for a circle
poly1 = getcircle(1, 10000, ptr=ptr)
plt.plot(*poly1.getXY())

# Delaunay triangulation of the circle into triangles that fully cover the original␣
↪polygon
tri = poly1.DelaunayTriangulate()
Ngeo = tri.getNumGeometries()
for i in range(Ngeo):
    geo = tri.getGeometryN(i)
    X, Y = geo.getXY()
    cen = geo.getCentroid()
    x, y = cen.getX(), cen.getY()
    plt.plot(x, y, 'k.')
    plt.plot(X, Y)
del ptr
plt.axis('equal')
plt.axis('off');
```



```
[4]: # A box, randomly sampled using the code in
     # teqpflsh
     X = np.array([0,1,1,0,0.0])
     Y = np.array([0,0,1,1,0.0])
     N = 10000
     reg = teqpflsh.QuadRegion2D(x=X,y=Y)
     x, y = np.zeros(N,), np.zeros(N)
     reg.sample_random(len(x), x, y)
     plt.plot(x, y, 'k.', ms=0.3);
     del reg
```



```
[5]: # A circle, randomly sampled by first triangulation and then sampling
ptr = teqpflsh.GeometryFactoryHolder()
circ = getcircle(1, 10000, ptr=ptr)
X, Y = circ.getXY()

# The "region", which here is a circle
reg = teqpflsh.QuadRegion2D(x=X,y=Y)
x, y = np.zeros(N,), np.zeros(N) # allocate buffers
reg.sample_random(len(x), x, y)
plt.plot(x, y, 'k.', ms=0.3)
plt.axis('equal')

# And here are the triangles
tri = reg.do_fast_triangulation()
for i in range(tri.getNumGeometries()):
    coords = tri.getGeometryN(i).getCoordinates()
    # get the vertices of the triangle
    Ncoords = coords.getSize()
    x = [coords.getX(_) for _ in range(Ncoords)]
    y = [coords.getY(_) for _ in range(Ncoords)]
    plt.plot(x, y, 'r')
del reg
```



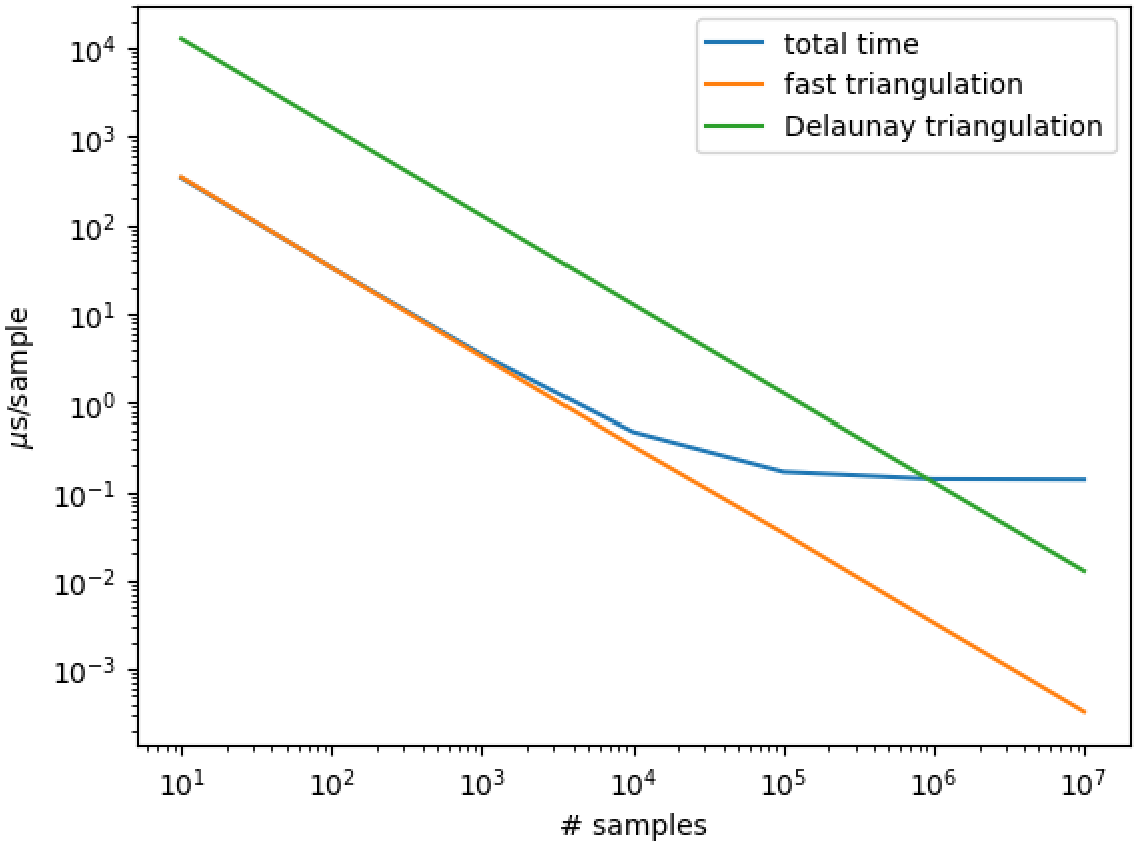
```
[6]: # For few samples, you are dominated by triangulation cost
     # and then for more samples, the cost is dominated by the sampling
     # itself. The more refined the polygon, the slower the triangulation

     reg = teqpflsh.QuadRegion2D(x=X,y=Y)
     Deltimes, tritimes, exponents, times = [],[],[],[]
     for exponent in range(1, 8):
         N = 10**exponent
         x, y = np.zeros(N,), np.zeros(N)

         tic = timeit.default_timer()
         reg.do_fast_triangulation()
         toc = timeit.default_timer()
         tritimes.append((toc-tic)*1e6/N)

         tic = timeit.default_timer()
         reg.do_Delaunay_triangulation()
         toc = timeit.default_timer()
         Deltimes.append((toc-tic)*1e6/N)

         tic = timeit.default_timer()
         reg.sample_random(len(x), x, y)
         toc = timeit.default_timer()
         times.append((toc-tic)*1e6/N)
         print(exponent, times[-1])
         exponents.append(exponent)

     plt.plot(10**np.array(exponents), times, label='total time')
```

(continues on next page)

(continued from previous page)

```
plt.plot(10**np.array(exponents), tritimes, label='fast triangulation')
plt.plot(10**np.array(exponents), Deltimes, label='Delaunay triangulation')
plt.xscale('log')
plt.yscale('log')
plt.gca().set(xlabel=r'# samples', ylabel=r'$\mu$s/sample')
plt.legend()
plt.show()
del reg
```

```
1 343.95409747958183
2 33.52916974108666
3 3.48491600016132
4 0.47287080087698996
5 0.1693791605066508
6 0.14071825001155958
7 0.13905381249496712
```



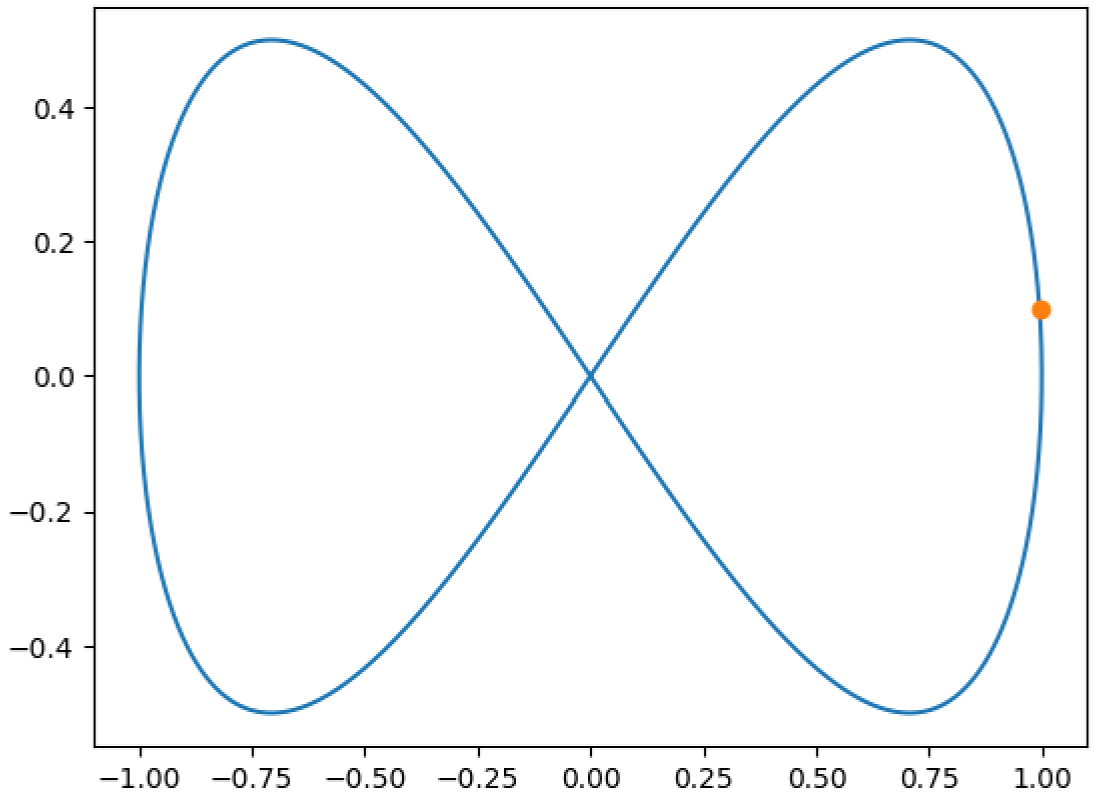
3.2 Polygon validation

Making the polygons be non-self-intersecting is critical. There are routines in GEOS (and exposed in teqpflsh) to break up self-intersecting polygons into non-self-intersecting polygons

[1]:
```
import numpy as np
import teqpflsh
import matplotlib.pyplot as plt
```

[2]:
```
ptr = teqpflsh.GeometryFactoryHolder()

# A bowtie curve that is periodic and self-intersecting
t = np.linspace(0+0.1, 2*np.pi+0.1, 10000)
X,Y = np.cos(t), np.cos(t)*np.sin(t)
poly1 = ptr.makeclosedpolygon(X, Y)
plt.plot(X, Y)
plt.plot(X[0], Y[0], 'o')
poly1.isValid # False since self-intersecting
```

[2]:
```
False
```



Now we need to break up the polygon into portions that are simple (non self-intersecting) with the MakeValid class of geos : https://libgeos.org/doxygen/classgeos_1_1operation_1_1valid_1_1MakeValid.html

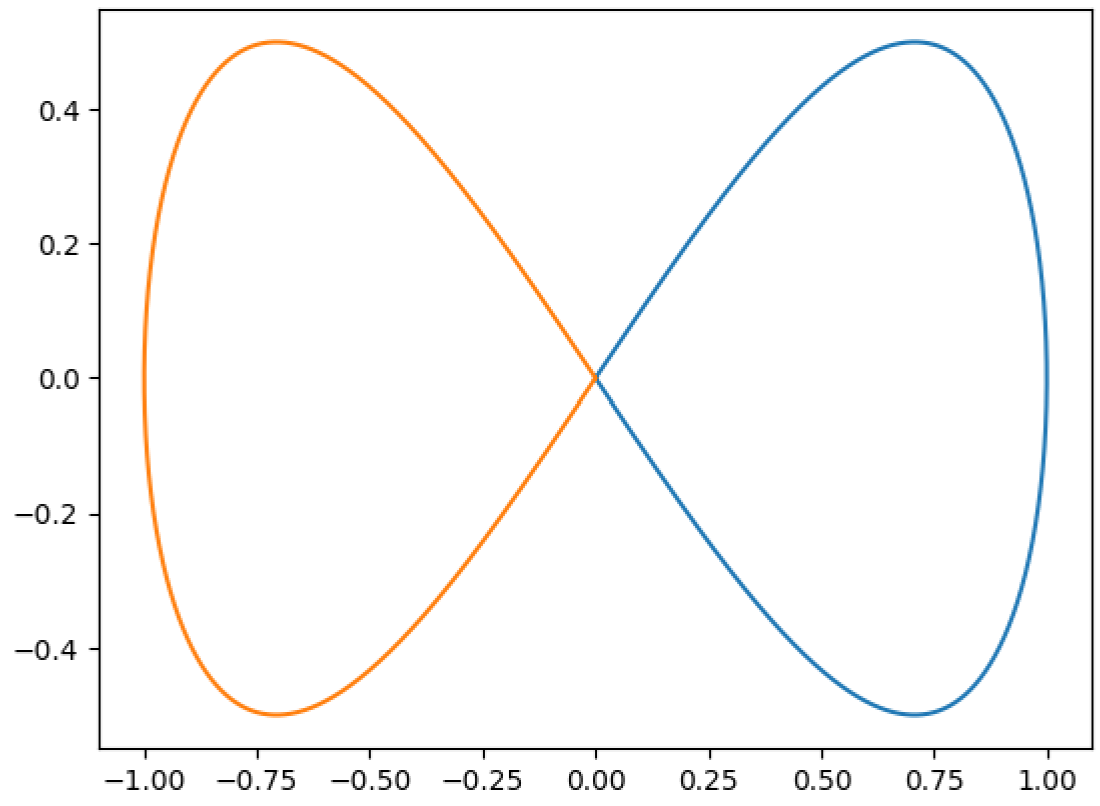
[3]:
```
simpl = poly1.make_valid()
print(f'N: {simpl.getNumGeometries()}')
print(f'simple: {simpl.isSimple}')
print(f'valid: {simpl.isValid}')
```

(continues on next page)

(continued from previous page)

```python
for i in range(simpl.getNumGeometries()):
    pI = simpl.getGeometryN(i)
    plt.plot(*pI.getXY())
    print(f'N: {pI.getNumGeometries()}')
    print(f'simple: {pI.isSimple}')
    print(f'valid: {pI.isValid}')
```

```
N: 2
simple: True
valid: True
N: 1
simple: True
valid: True
N: 1
simple: True
valid: True
```



CHAPTER

FOUR

K-D TREE LOOKUP

4.1 K-D tree fundamentals

TODO: what is a K-D tree

The nanoflann library is used within teqpflsh due to its computational efficiency.

The `L2TreeHolder` class makes a copy of the data for the tree to ensure that the lifetime of the data copied into the holder is longer than the tree itself. The underlying `L2Tree` object obtained via the `.tree` attribute then makes a reference to the data held in the holder class.

[1]:
```
import numpy as np
import teqpflsh
import matplotlib.pyplot as plt

def boxpoly(*, top, bottom, left, right, ptr):
    X = np.array([left, right, right, left, left])
    Y = np.array([bottom, bottom, top, top, bottom])
    return ptr.makeclosedpolygon(X, Y)
```

[2]:
```
ptr = teqpflsh.GeometryFactoryHolder()

# Polygon for the shifted circle
t = np.linspace(0, 2*np.pi, 10000)
X = 0.5 + 0.3*np.cos(t)
Y = 0.3*np.sin(t)
poly1 = ptr.makeclosedpolygon(X, Y)
poly2 = boxpoly(left=0, right=1, bottom=0, top=1, ptr=ptr)

# Polygon for the square [0,1]x[0,1] minus small circle
poly = poly2.difference(poly1)
X, Y = poly.getXY()

def do_one(*, NKD, Nsample, plot=False, close=True):
    def get_random(NKD):
        """ Random points for the tree """
        XX, YY = [], []
        while len(XX) < NKD:
            x_, y_ = np.random.random(2)
            pt = ptr.createPoint(float(x_), float(y_))
            if poly.containsPoint(pt):
                XX.append(x_)
                YY.append(y_)
        return XX, YY
```

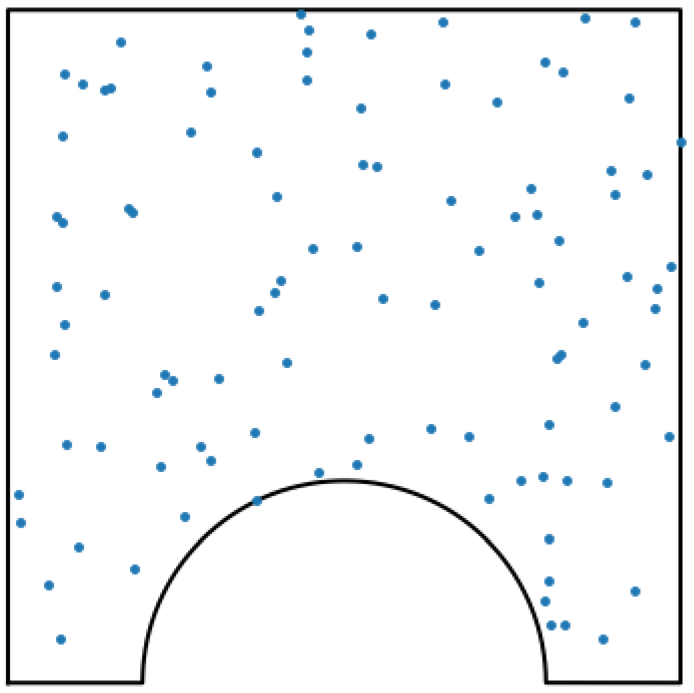
(continues on next page)

(continued from previous page)

```
    XX, YY = get_random(NKD)

    if plot:
        plt.plot(X, Y, 'k')
        plt.plot(XX, YY, '.', ms=5)

    holder = teqpflsh.L2TreeHolder(np.array(XX), np.array(YY), 10)
    tree = holder.tree

    xsample, ysample = get_random(Nsample)
    d2 = [tree.get_nearest_indexd2(x_, y_)[1] for x_, y_ in zip(xsample, ysample)]

    if plot:
        plt.axis('off')
        plt.axis('equal');
        if close:
            plt.close()

    return np.mean(np.array(d2)**0.5), tree.get_used_bytes()
```

Here is a small number of “lighthouse” points randomly distributed in the domain. A random point is first pulled from [0,1]x[0,1] and checked whether it is within the domain or not. This so-called point-in-polygon problem is quite slow (relatively).

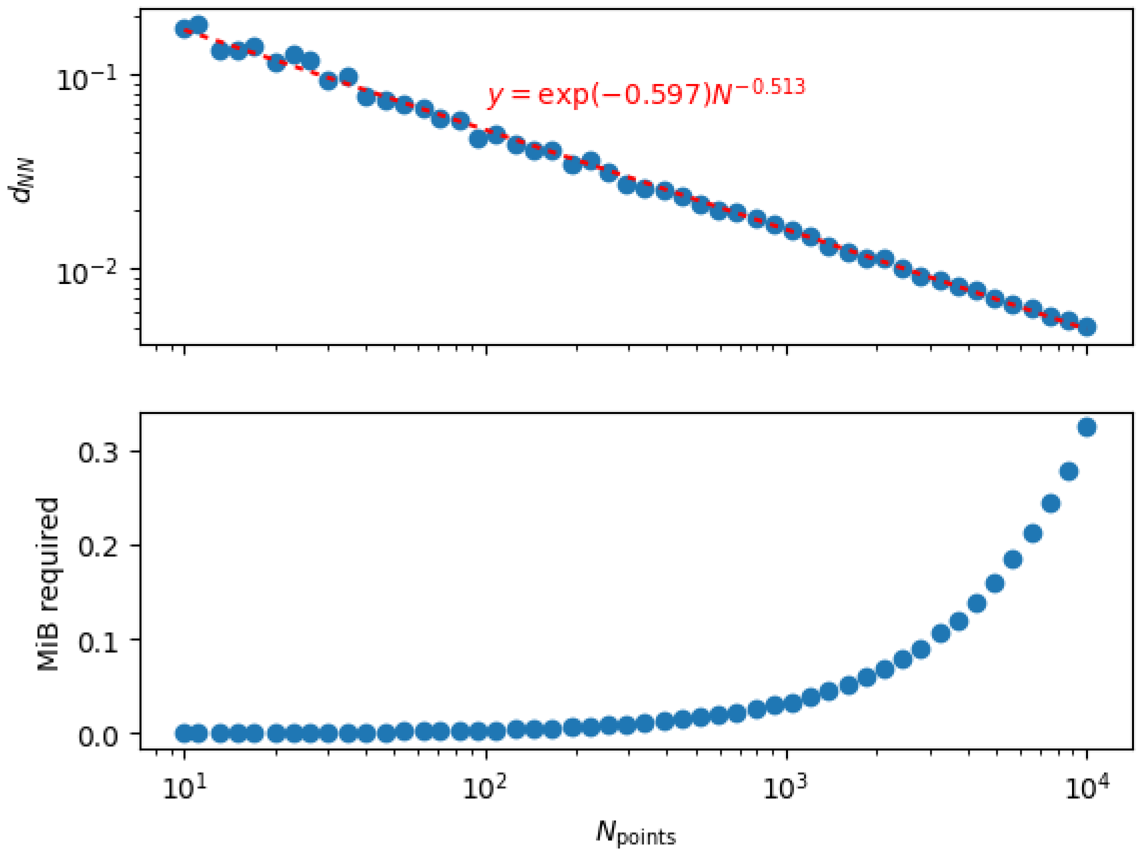
```
[3]: do_one(NKD=100, Nsample=100, plot=True, close=False);
```



As you increase the number of points N inside the domain, the distance to the nearest point goes down like $N^{-1/2}$ and in general the scaling should be like $N^{-1/D}$ where D is the number of spatial dimensions (I think).

The required memory is linear with the number of points in the K-D tree(!)

```
[4]: Ntrees = np.geomspace(10, 10**4, dtype=int)

     fig, (ax1, ax2) = plt.subplots(2, 1, sharex=True)

     d2s = [do_one(NKD=NKD_, Nsample=10**3)[0] for NKD_ in Ntrees]
     ax1.plot(Ntrees, d2s, 'o')

     pf = np.polyfit(np.log(Ntrees), np.log(d2s), 1)
     xx = np.geomspace(np.min(Ntrees), np.max(Ntrees), 1000)
     ax1.plot(xx, np.exp(np.polyval(pf, np.log(xx))), dashes=[2,2], color='r')
     ax1.text(100, 0.07, rf'$y=\exp({pf[1]:0.3f})N^{{{pf[0]:0.3f}}}$', color='r')

     ax1.set_xscale('log')
     ax1.set_yscale('log')
     ax1.set(ylabel=r'$d_{NN}$')

     MiBs = [do_one(NKD=NKD_, Nsample=10**3)[1]/1024**2 for NKD_ in Ntrees]
     ax2.plot(Ntrees, MiBs, 'o')
     ax2.set(xlabel=r'$N_{\rm points}$', ylabel='MiB required');

     # del ptr
```



CHAPTER

FIVE

REGIONED FLASHER

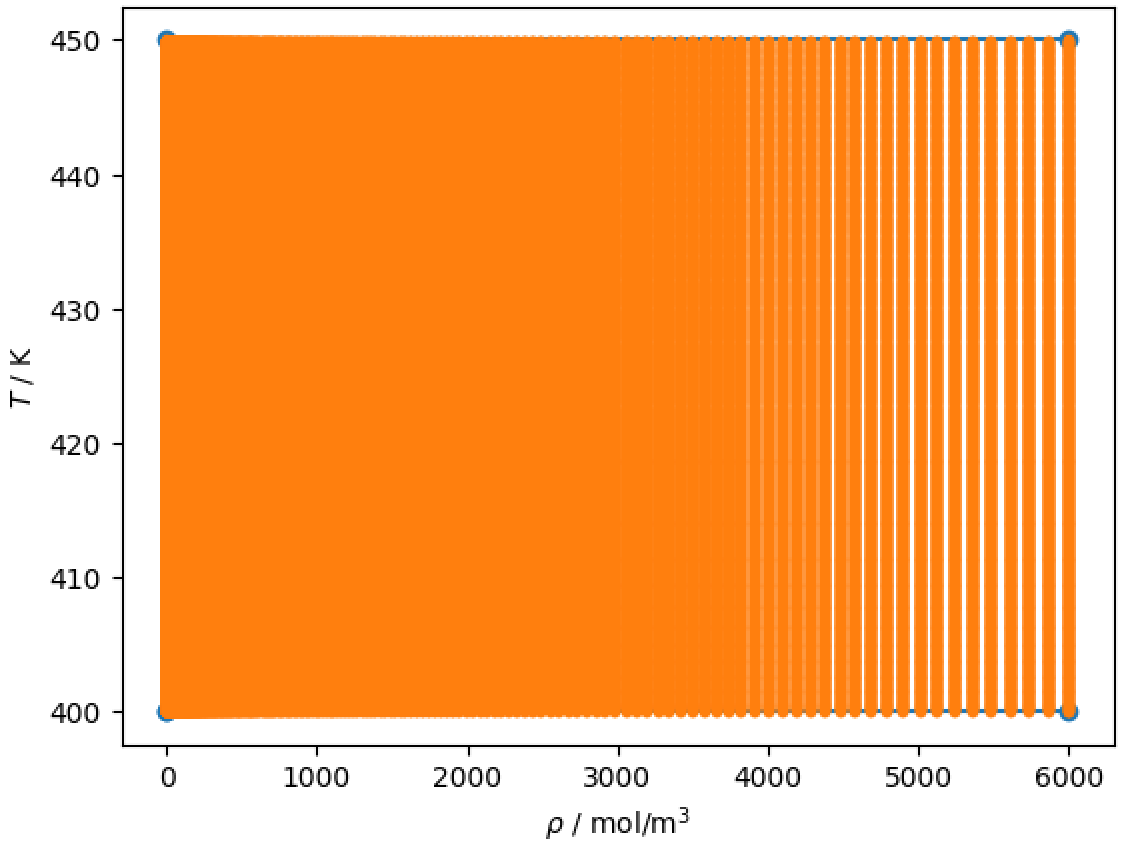
The regioned flasher breaks up the temperature-density plane into multiple region. In each region, a K-D tree is constructed for points distributed within the region in T-D coordinates (because they are the coordinates of the EOS and require no iteration to obtain them). Thus distributing the points cannot fail (like iterative calculations in P-H coordinates might). To start, lets begin with a rectangular region in the supercritical region as a demonstration. The real regions are much more complex as they need to handle the complete fluid domain, deal with solid-liquid phase equilibria, etc..

[1]:
```
import timeit, json

import teqpflsh, teqp

import numpy as np
import matplotlib.pyplot as plt
```

[2]:
```
name = "n-Propane"

path = f'{teqp.get_datapath()}/dev/fluids/{name}.json'
jresid = {"kind": "multifluid", "model": {"components": [name], "root": teqp.get_
↪datapath()}}
jidealgas = {"kind": "IdealHelmholtz", "model": [teqp.convert_CoolProp_idealgas(path,␣
↪0)]}

rf = teqpflsh.RegionedFlasher(
    ideal_gas=json.dumps(jidealgas),
    resid=json.dumps(jresid),
    mole_fractions=np.array([1.0])
)
# To start off there are no regions in the regioned flasher
print('# of regions:', len(rf.get_regions_rw()))

# Now we make a region with rectangular shape in T, rho coordinates
# As we will see, a rectangular shape with only the corners defined doesn't work so␣
↪well when transformed into
# other coordinates
Tmin = 400 # K
Tmax = 450 # K
rhomin = 1e-6 # mol/m³
rhomax = 6000 # mol/m³
Tpoly = np.array([Tmin, Tmin, Tmax, Tmax, Tmin])
rhopoly = np.array([rhomin, rhomax, rhomax, rhomin, rhomin])
NT = 1000
Nrho = 1000

rf.add_region(T=Tpoly, rho=rhopoly, NT=NT, Nrho=Nrho)
print('# of regions:', len(rf.get_regions_ro()))
```

```
# of regions: 0
# of regions: 1
```

[3]:
```
# Here is the bounding region and the points used for construction of the K-D tree
# This all looks nice
for reg in rf.get_regions_rw():
    reg.add_pair(proppair=teqpflsh.PropertyPairs.DT, Nsplit=5)

    pset = reg.propset_bounding
    plt.plot(pset.rho, pset.T, 'o-')

    pset = reg.propset_Trhogrid
    plt.plot(pset.rho, pset.T, '.')
plt.gca().set(xlabel=r'$\rho$ / mol/m$^3$', ylabel='$T$ / K');
```



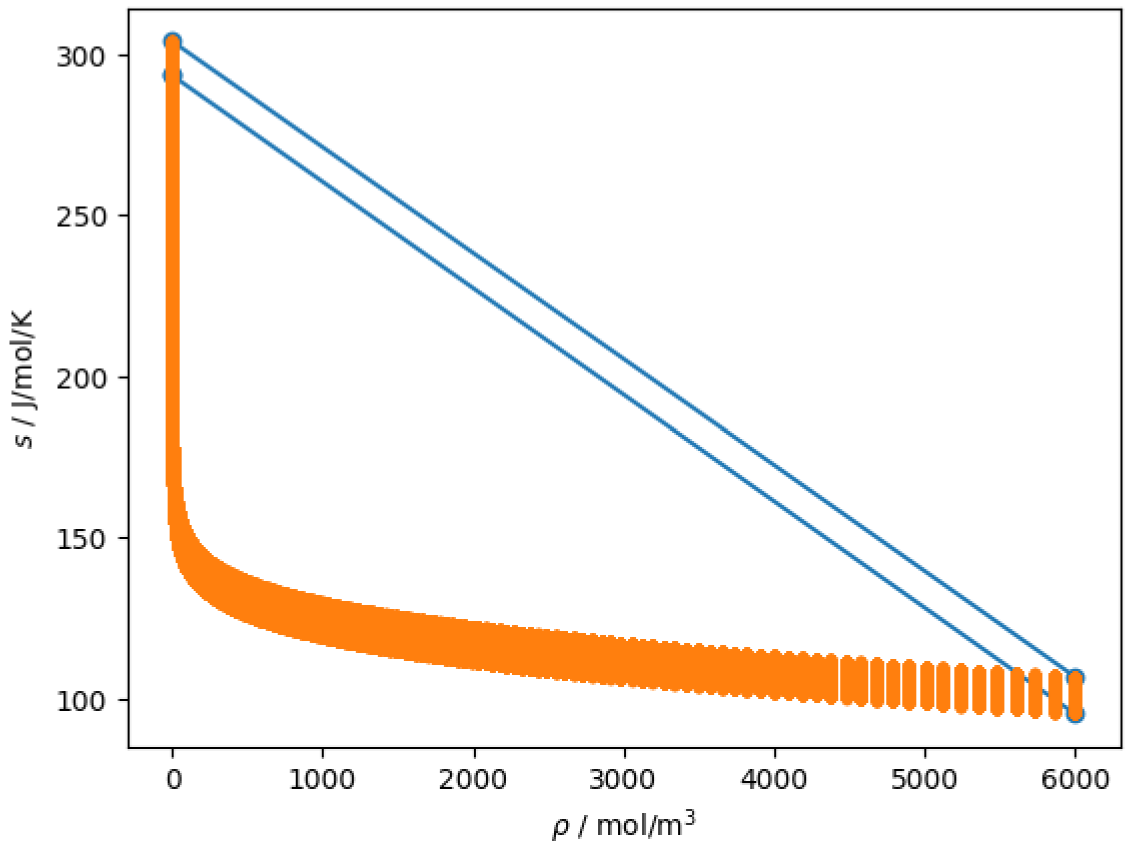
But when you shift to another variable pair, here density and entropy, the rectangular box and its sampled (in $T$, $\rho$) points do not all map together

[4]:
```
for reg in rf.get_regions_rw():
    reg.add_pair(proppair=teqpflsh.PropertyPairs.DS, Nsplit=5)

    pset = reg.propset_bounding
    plt.plot(pset.rho, pset.s, 'o-')

    pset = reg.propset_Trhogrid
    plt.plot(pset.rho, pset.s, '.')
```

(continues on next page)

(continued from previous page)

```
plt.gca().set(xlabel=r'$\rho$ / mol/m$^3$', ylabel='$s$ / J/mol/K');
```



In order to get around this problem, the box needs to be sampled at more than 5 points

Exercise for reader: build a more dense polygon defining the boundary of the box, with many more points along each side

To better understand the timing of each step, it can be useful to profile each step independently. The `_many` methods has been written for this purpose. The overhead in nanobind with pre-allocated buffers being passed to the function is functionally zero.

```
[5]: N = 500000

for reg in rf.get_regions_rw():
    reg.add_pair(proppair=teqpflsh.PropertyPairs.DP, Nsplit=5)

    tree = reg.get_kdtree(teqpflsh.PropertyPairs.DP)
    X = np.linspace(2000, 2001, N)
    Y = np.linspace(0.25e7, 0.251e7, N)
    idx = np.zeros_like(X, dtype=int)
    d2 = np.zeros_like(Y)
    tic = timeit.default_timer()
    tree.get_nearest_indexd2_many(X, Y, idx, d2)
    toc = timeit.default_timer()
    print((toc-tic)/N*1e6, 'µs to look up a point from the K-D tree')
    print(f'The K-D tree consumes', tree.get_used_bytes()/1024**2, "MiB")
```

(continues on next page)

(continued from previous page)

```
    TT = np.zeros_like(X)
    DD = np.zeros_like(X)
    tic = timeit.default_timer()
    reg.get_starting_Trho_many(teqpflsh.PropertyPairs.DP, X, Y, TT, DD, d2)
    toc = timeit.default_timer()
    print((toc-tic)/N*1e6, 'µs to look up a point from the K-D tree and return the (T,
↪ rho) point and its '
          'distance. This should be a smidge slower than the above calculation')
```

```
0.19432550005149096 µs to look up a point from the K-D tree
The K-D tree consumes 32.41525650024414 MiB
0.1970805840101093 µs to look up a point from the K-D tree and return the (T, rho)␣
↪point and its distance. This should be a smidge slower than the above calculation
```

Putting it all together, here is an example of using the entire flash calculation. The steps are:

1. Find the nearest point in the K-D tree to get a starting value for T and ρ for further iteration
2. Do the iteration to find the right T, ρ satisfying the problem statement (trivial in this case because T, ρ are input variables)

[6]:
```
for reg in rf.get_regions_rw():
    reg.add_pair(proppair=teqpflsh.PropertyPairs.DT, Nsplit=5)
N = 50000
X = np.linspace(2000, 2001, N)
Y = np.linspace(250, 251, N)
TT = np.zeros_like(X)
DD = np.zeros_like(X)
steps = np.zeros_like(X, dtype=int)
maxabsr = np.zeros_like(Y)
newtontime = np.zeros_like(Y)
candtime = np.zeros_like(Y)
tic = timeit.default_timer()
rf.flash_many(teqpflsh.PropertyPairs.DT, X, Y, TT, DD, steps, maxabsr, newtontime,␣
↪candtime)
toc = timeit.default_timer()
print((toc-tic)/N*1e6, np.mean(newtontime), np.mean(candtime), np.mean(steps))
```

```
1.3062583201099187 0.9456310399999999 0.25357912 0.0
```

The timing is carried out at a fairly granular level. The `candtime` argument is the time required (in µs) to do preparation of the candidates from the K-D tree values. The `newtontime` is the time spent (in µs) preparing the iteration object and actually doing the iteration. In this case the inputs do not require any iteration, but the newton iterator is still constructed.

[7]:
```
# Here we iterate for two variables, it is much slower
for reg in rf.get_regions_rw():
    reg.add_pair(proppair=teqpflsh.PropertyPairs.PS, Nsplit=5)

# Take the points in the K-D tree to do calculations
# They trivially satisfy the stopping conditions!
propset = reg.propset_Trhogrid
X = propset.p
Y = propset.s
o = rf.flash(teqpflsh.PropertyPairs.PS, X[0], Y[0])
print(o.T, o.rho, o.candidate_duration_us, o.total_duration_us, o.newton_duration_us,␣
↪o.step_count)
```

```
400.04716315487264 1.0000000000000004e-06 5.041 16.0 10.834 1
```

```
[8]: # Now p, s inputs but with a bit of noise in entropy to force the
     # Newton iterator to actually do something

     # Input variables
     X = propset.p
     Y = propset.s + np.random.random(X.shape)

     # Output buffers
     TT = np.zeros_like(X)
     DD = np.zeros_like(X)
     steps = np.zeros_like(X)
     maxabsr = np.zeros_like(Y)
     newtontime = np.zeros_like(Y)
     candtime = np.zeros_like(Y)
     tic = timeit.default_timer()

     rf.flash_many(teqpflsh.PropertyPairs.PS, X, Y, TT, DD, steps, maxabsr, newtontime,␣
      ↪candtime)
     toc = timeit.default_timer()
     print((toc-tic)/len(X)*1e6, np.mean(newtontime), np.mean(candtime), np.mean(steps),␣
      ↪np.mean(maxabsr), np.sum(TT<0))
```

```
3.2074755386728637 2.643995782586261 0.465123295975471253 3.0077775161447087 1.
 ↪4765034340778192e-10 0
```

Ok, that's good. Iteration was carried out, and the deviations between the specified and iterated entropy and pressure were good. It took approximately 1.1 µs per iteration step of the Newton iterator, which isn't too bad, but we did start not too far from the actual solution.

CHAPTER

SIX

COMPLETE EXAMPLE

The subcomponents are all joined together into the main flasher class, which combines the superancillary to first check the phase, followed by K-D tree lookup for the best starting point, and finally Newton iteration to get the correct temperature and density

[1]:
```
import timeit, json, functools

import teqpflsh, teqp
import CoolProp

import numpy as np
import matplotlib.pyplot as plt
```

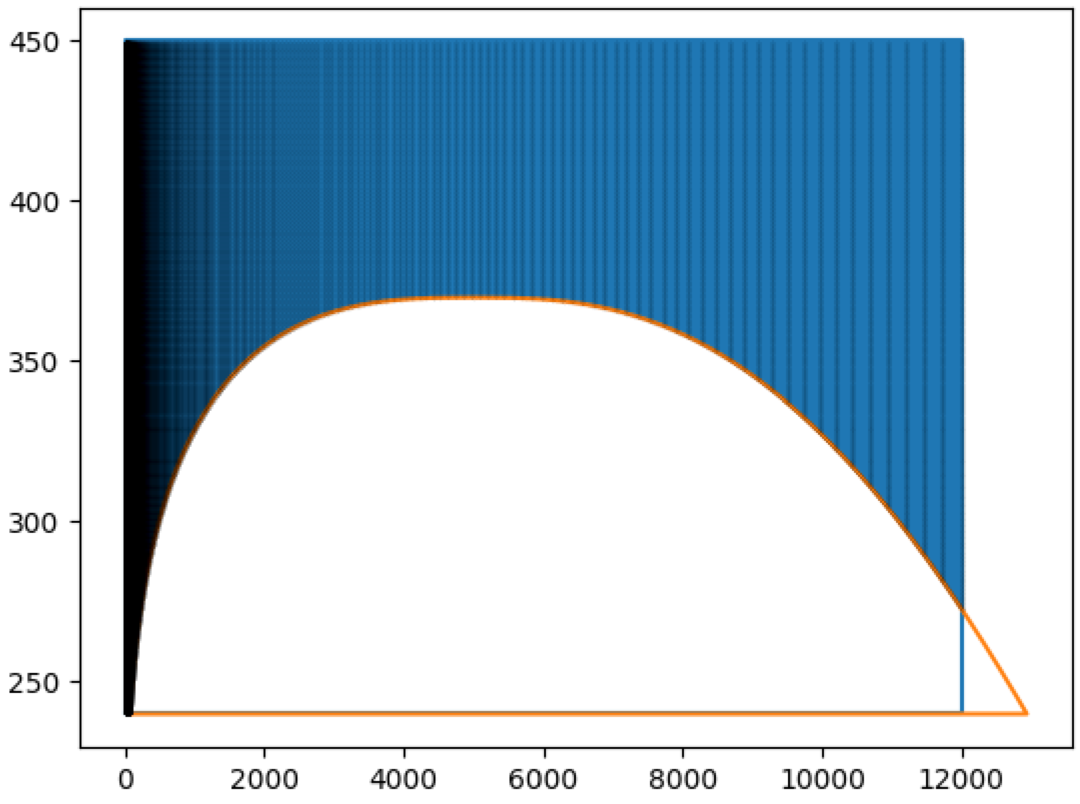
[2]:
```
name = "n-Propane"

j = json.load(open('PROPANE_exps.json'))
sa = teqpflsh.SuperAncillary(json.dumps(j))
import CoolProp.CoolProp as CP
AS = CP.AbstractState('HEOS', 'n-Propane')
def calc(T, rho, AS, key):
    AS.specify_phase(CP.iphase_gas)
    AS.update(CP.DmolarT_INPUTS, rho, T)
    val = AS.keyed_output(key)
    AS.unspecify_phase()
    return val
sa.add_variable(k='H', caller=functools.partial(calc, AS=AS, key=CP.iHmolar))
sa.add_variable(k='S', caller=functools.partial(calc, AS=AS, key=CP.iSmolar))
sa.add_variable(k='U', caller=functools.partial(calc, AS=AS, key=CP.iUmolar))
Tcrit = sa.get_approx1d(k='D', q=1).xmax

path = f'{teqp.get_datapath()}/dev/fluids/{name}.json'
jresid = {"kind": "multifluid", "model": {"components": [name], "root": teqp.get_
↪datapath()}}
jidealgas = {"kind": "IdealHelmholtz", "model": [teqp.convert_CoolProp_idealgas(path,␣
↪0)]}
rf = teqpflsh.RegionedFlasher(
    ideal_gas=json.dumps(jidealgas),
    resid=json.dumps(jresid),
    mole_fractions=np.array([1.0])
)

# Now we make a region with rectangular shape in T, rho coordinates with only its 5␣
↪corners
Tmin = 240 # K
```

(continues on next page)

(continued from previous page)

```
Tmax = 450 # K
rhomin = 1e-6 # mol/m³
rhomax = 12000 # mol/m³
Tpoly = np.array([Tmin, Tmin, Tmax, Tmax, Tmin])
rhopoly = np.array([rhomin, rhomax, rhomax, rhomin, rhomin])

def makeVLE(Trange):
    x, y = [], []
    for T in Trange:
        x.append(sa.get_yval(T=float(T), q=1.0, k='D')); y.append(T)
    for T in reversed(Trange):
        x.append(sa.get_yval(T=float(T), q=0.0, k='D')); y.append(T)
    return np.array(x), np.array(y)

ptr = teqpflsh.GeometryFactoryHolder()
box = ptr.makeclosedpolygon(rhopoly, Tpoly)
VLE = ptr.makeclosedpolygon(*makeVLE(np.linspace(Tmin, Tcrit, 1000)))
rhoreg, Treg = box.difference(VLE).getXY()

NT = 1000
Nrho = 1000
rf.add_region(T=Treg, rho=rhoreg, NT=NT, Nrho=Nrho)
for reg in rf.get_regions_rw():
    reg.add_pair(proppair=teqpflsh.PropertyPairs.PS, Nsplit=5)
    reg.add_pair(proppair=teqpflsh.PropertyPairs.ST, Nsplit=5)

# Build the helper class holding the models for ideal gas and residual Helmholtz␣
↪energy
helm = teqpflsh.teqpHelmholtzInterface(ideal_gas=json.dumps(jidealgas), residual=json.
↪dumps(jresid))

# Build the main flasher which includes all the parts
mf = teqpflsh.MainFlasher(regions=rf, superancillary=sa, helm=helm)

# Plot the region that is being mapped
plt.plot(*box.getXY())
plt.plot(*VLE.getXY())
plt.fill(rhoreg, Treg)
reg = mf.regioned_flasher.get_regions_ro()[0]
propset = reg.propset_Trhogrid
plt.plot(propset.rho, propset.T, 'k.', ms=0.05)
```

```
[2]: [<matplotlib.lines.Line2D at 0x10bae2210>]
```



```
[3]: reg = mf.regioned_flasher.get_regions_ro()[0]
     propset = reg.propset_Trhogrid

     val1 = propset.p
     val2 = propset.s + np.random.random(val1.shape)
     T = np.zeros_like(val1)
     rho = np.zeros_like(val1)
     q = np.zeros_like(val1)
     print(len(val1))

     tic = timeit.default_timer()
     mf.flash_many(teqpflsh.PropertyPairs.PS, val1, val2, T, rho, q)
     toc = timeit.default_timer()
     print((toc-tic)/len(val1)*1e6, 'µs per flash call')
```

```
907865
3.4448237535615216 µs per flash call
```

```
[4]: reg = mf.regioned_flasher.get_regions_ro()[0]
     propset = reg.propset_Trhogrid

     val1 = propset.s + np.random.random(val1.shape)
     val2 = propset.T
     T = np.zeros_like(val1)
     rho = np.zeros_like(val1)
     q = np.zeros_like(val1)

     tic = timeit.default_timer()
```

(continues on next page)

(continued from previous page)

```
mf.flash_many(teqpflsh.PropertyPairs.ST, val1, val2, T, rho, q)
toc = timeit.default_timer()
print((toc-tic)/len(val1)*1e6, 'μs per flash call')
```

```
2.295959200958003 μs per flash call
```

CHAPTER

SEVEN

OUTLOOK

The code works pretty well. The iteration approach is quite fast and reliable.

There are some outstanding items:

- Look into the memory used by the K-D trees, see if it can be further optimized because some copies of data are being made and some of these copies can in theory at least be avoided
- Resolve issues where the bounding polygon of a region is not simple (has self-intersection). This happens especially in the case of water at low temperatures. Such self-intersection does not happen in T, ρ coordinates, but does happen in other coordinate sets. The C++ method (https://libgeos.org/doxygen/MakeValid_8h_source.html) is exposed as `Geometry.make_valid`. It has not been fully implemented into the flashing code.
- Automate generation of the bounding polygons in C++. Currently the bounding polygon is generated in Python and passed into C++. To do so will require to implement the melting line models, and additional conversions might be required in REFPROP-interop
- See if there is a better way to define the bounding edges of the regions. Perhaps splines or something like that might be better, enabling a more efficient representation, not requiring as many points to remap into other coordinates more smoothly.

CHAPTER

EIGHT

TEQPFLSH

8.1 teqpflsh Package

class `teqpflsh._teqpflsh_impl.`**`AbstractScaler`**

Bases: `object`

C++ docs: AbstractScaler

class `teqpflsh._teqpflsh_impl.`**`ChebyshevApproximation1D`**(**args*, ***kwargs*)

Bases: `object`

C++ docs: ChebyshevApproximation1D

`count_x_for_y_many`(*self, arg0: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg1: int, arg2: int, arg3: float, arg4: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], /*) → None

`eval`(*self*, *arg: float*, */*) → float

`eval_many`(*self, arg0: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg1: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], /*) → None

property expansions

(self) -> list[teqpflsh._teqpflsh_impl.ChebyshevExpansion]

`get_intervals_containing_y`(*self*, *arg: float*, */*) → list[*teqpflsh._teqpflsh_impl.IntervalMatch*]

`get_x_for_y`(*self*, ***, *y: float*, *bits: int = 64*, *max_iter: int = 100*, *boundsftol: float = 1e-13*) → list[tuple[float, int]]

property monotonic_intervals

(self) -> list[teqpflsh::superancillary::IntervalMatch]

property x_at_extrema

(self) -> list[float]

property xmax

(self) -> float

property xmin

(self) -> float

class `teqpflsh._teqpflsh_impl.`**`ChebyshevExpansion`**(**args*, ***kwargs*)

Bases: `object`

C++ docs: ChebyshevExpansion

property coeff

(self) -> numpy.ndarray[dtype=float64, shape=(*), order='C']

`eval`(*self*, *arg: float*, */*) → float

`eval_Eigen`(*self, arg0: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg1: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], /*) → None

`eval_many`(*self, arg0: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg1: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], /*) → None

`solve_for_x`(*self*, *arg0: float*, *arg1: float*, *arg2: float*, *arg3: int*, *arg4: int*, *arg5: float*, */*) → float

`solve_for_x_count`(*self*, *arg0: float*, *arg1: float*, *arg2: float*, *arg3: int*, *arg4: int*, *arg5: float*, */*) → tuple[float, int]

`solve_for_x_many`(*self, arg0: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg1: float, arg2: float, arg3: int, arg4: int, arg5: float, arg6: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg7: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], /*) → None

property xmax

(self) -> float

property xmin

(self) -> float

class `teqpflsh._teqpflsh_impl.`**`CoordinateSequence`**(**args*, ***kwargs*)

Bases: `object`

C++ docs: CoordinateSequence

`add`(*self*, *arg0: float*, *arg1: float*, */*) → None

`closeRing`(*self*, *arg: bool*, */*) → None

`getSize`(*self*) → int

`getX`(*self*, *arg: int*, */*) → float

`getY`(*self*, *arg: int*, */*) → float

class `teqpflsh._teqpflsh_impl.`**`Envelope`**

Bases: `object`

C++ docs: Envelope

property x_max

(self) -> float

property x_min

(self) -> float

property y_max

(self) -> float

property y_min
(self) -> float

class teqpflsh._teqpflsh_impl.**FlashPhase**
Bases: object

C++ docs: FlashPhase

property mole_fractions
(self) -> numpy.ndarray[dtype=float64, shape=(*), order='C']

property qmolar
(self) -> float

property rho_molm3
(self) -> float

class teqpflsh._teqpflsh_impl.**FlashSolution**
Bases: object

C++ docs: FlashSolution

property Nphases
(self) -> int

property T_K
(self) -> float

property phases
(self) -> list[teqpflsh._teqpflsh_impl.FlashPhase]

class teqpflsh._teqpflsh_impl.**Geometry**
Bases: object

C++ docs: Geometry

DelaunayTriangulate(*self*) → *teqpflsh._teqpflsh_impl.Geometry*

containsPoint(*self*, *arg:* teqpflsh._teqpflsh_impl.Point, /) → bool

difference(*self*, *arg:* teqpflsh._teqpflsh_impl.Geometry, /) → *teqpflsh._teqpflsh_impl.Geometry*

fastTriangulate(*self*) → *teqpflsh._teqpflsh_impl.Geometry*

getCentroid(*self*) → *teqpflsh._teqpflsh_impl.Point*

getCoordinates(*self*) → *teqpflsh._teqpflsh_impl.CoordinateSequence*

getGeometryN(*self*, *arg: int*, /) → *teqpflsh._teqpflsh_impl.Geometry*

getNumGeometries(*self*) → int

getNumPoints(*self*) → int

getXY(*self*) → tuple[numpy.ndarray[dtype=float64, shape=(*), order='C'], numpy.ndarray[dtype=float64, shape=(*), order='C']]
Convenience function to return the X, Y coordinates as numpy arrays in Python

get_PreparedGeometry(*self*) → *teqpflsh._teqpflsh_impl.PreparedGeometry*

intersection(*self*, *arg:* teqpflsh._teqpflsh_impl.Geometry, */*) → *teqpflsh._teqpflsh_impl.Geometry*

property isSimple
(self) -> bool

property isValid
(self) -> bool

make_valid(*self*) → *teqpflsh._teqpflsh_impl.Geometry*

run_DouglasPeuckerSimplifier(*self*, *tolerance: float*) → *teqpflsh._teqpflsh_impl.Geometry*

run_TopologyPreservingSimplifier(*self*, *tolerance: float*) → *teqpflsh._teqpflsh_impl.Geometry*

class teqpflsh._teqpflsh_impl.**GeometryFactory**

Bases: `object`

C++ docs: GeometryFactory

createPolygon(*self*, *arg:* teqpflsh._teqpflsh_impl.Geometry, */*) → *teqpflsh._teqpflsh_impl.Geometry*

class teqpflsh._teqpflsh_impl.**GeometryFactoryHolder**(**args*, ***kwargs*)

Bases: `object`

C++ docs: GeometryFactoryHolder

createPoint(*self*, *arg0: float*, *arg1: float*, */*) → *teqpflsh._teqpflsh_impl.Point*

createPolygon(*self*, *arg:* teqpflsh._teqpflsh_impl.CoordinateSequence, */*) → *teqpflsh._teqpflsh_impl.Geometry*

makeclosedpolygon(*self, x: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], y: ndarray[dtype=float64, shape=(*), order='C', device='cpu']*) → *teqpflsh._teqpflsh_impl.Geometry*

A convenience function to make a closed polygon given numpy arrays

class teqpflsh._teqpflsh_impl.**HelmholtzInterface**

Bases: `object`

C++ docs: HelmholtzInterface

class teqpflsh._teqpflsh_impl.**IntervalMatch**

Bases: `object`

C++ docs: IntervalMatch

property expansioninfo
(self) -> list[teqpflsh._teqpflsh_impl.MonotonicExpansionMatch]

property xmax
(self) -> float

property xmin
(self) -> float

property ymax
(self) -> float

property ymin
(self) -> float

class `teqpflsh._teqpflsh_impl.`**`L2Tree`**

Bases: `object`

C++ docs: L2Tree

`get_nearest_indexd2`(*self*, *x: float*, *y: float*) → tuple[int, float]

`get_nearest_indexd2`(*self, pt: numpy.ndarray[dtype=float64, shape=(2), order='C']*) → tuple[int, float]

`get_nearest_indexd2_many`(*self, x: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], y: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], idx: ndarray[dtype=int32, shape=(*), order='C', device='cpu'], d2: ndarray[dtype=float64, shape=(*), order='C', device='cpu']*) → None

`get_used_bytes`(*self*) → int

class `teqpflsh._teqpflsh_impl.`**`L2TreeHolder`**(**args*, ***kwargs*)

Bases: `object`

C++ docs: L2TreeHolder

property tree

(self) -> teqpflsh._teqpflsh_impl.L2Tree

class `teqpflsh._teqpflsh_impl.`**`LeafContents`**

Bases: `object`

C++ docs: LeafContents

property status

(self) -> teqpflsh::PQTStatus

class `teqpflsh._teqpflsh_impl.`**`MainFlasher`**(**args*, ***kwargs*)

Bases: `object`

C++ docs: MainFlasher

`flash`(*self*, *proppair:* teqpflsh._teqpflsh_impl.PropertyPairs, *val1: float*, *val2: float*) → *teqpflsh._teqpflsh_impl.FlashSolution* | None

`flash_many`(*self, proppair: teqpflsh._teqpflsh_impl.PropertyPairs, val1: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], val2: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], T: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], rho: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], q: ndarray[dtype=float64, shape=(*), order='C', device='cpu']*) → None

property regioned_flasher

(self) -> teqpflsh._teqpflsh_impl.RegionedFlasher

class `teqpflsh._teqpflsh_impl.`**`MaxAbsErrorCondition`**(**args*, ***kwargs*)

Bases: *`StoppingCondition`*

C++ docs: MaxAbsErrorCondition

class `teqpflsh._teqpflsh_impl.`**`MinMaxLogScaler`**(**args*, ***kwargs*)

Bases: *`AbstractScaler`*

C++ docs: MinMaxLogScaler

class teqpflsh._teqpflsh_impl.**MinMaxScaler**(**args*, ***kwargs*)

Bases: *AbstractScaler*

C++ docs: MinMaxScaler

class teqpflsh._teqpflsh_impl.**MonotonicExpansionMatch**

Bases: `object`

C++ docs: MonotonicExpansionMatch

property idx

(self) -> int

property xmax

(self) -> float

property xmin

(self) -> float

property ymax

(self) -> float

property ymin

(self) -> float

class teqpflsh._teqpflsh_impl.**NRIterator**

Bases: `object`

C++ docs: NRIterator

calc_J(*self*, *arg0: float*, *arg1: float*, */*) → numpy.ndarray[dtype=float64, shape=(2, 2), order='F']

calc_just_step(*self*, *arg0: float*, *arg1: float*, */*) → numpy.ndarray[dtype=float64, shape=(2), order='C']

calc_maxabsr(*self*, *arg0: float*, *arg1: float*, */*) → float

calc_r(*self*, *arg0: float*, *arg1: float*, */*) → numpy.ndarray[dtype=float64, shape=(2), order='C']

calc_step(*self*, *arg0: float*, *arg1: float*, */*) → tuple[numpy.ndarray[dtype=float64, shape=(2), order='C'], teqp::cppinterface::IterationMatrices]

calc_vals(*self*, *arg0: float*, *arg1: float*, */*) → numpy.ndarray[dtype=float64, shape=(2), order='C']

get_T(*self*) → float

get_maxabsr(*self*) → float

get_nonconstant_indices(*self*) → list[int]

get_rho(*self*) → float

get_step_count(*self*) → int

get_vals(*self*) → numpy.ndarray[dtype=float64, shape=(2), order='C']

path_integration(*self*, *arg0: float*, *arg1: float*, *arg2: int*, */*) → tuple[float, float, float, float]

reset(*self*, *arg0: float*, *arg1: float*, */*) → None

take_steps(*self*, *N: int*, *apply_stopping: bool*) → *teqpflsh._teqpflsh_impl.StoppingConditionReason*

property verbose
(self) -> bool

class teqpflsh._teqpflsh_impl.**NanXDXErrorCondition**(**args*, ***kwargs*)

Bases: *StoppingCondition*

C++ docs: NanXDXErrorCondition

class teqpflsh._teqpflsh_impl.**PQTStatus**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

PQTStatus

inside = 0

intersection = 2

outside = 1

class teqpflsh._teqpflsh_impl.**Point**

Bases: *Geometry*

C++ docs: Point

DelaunayTriangulate(*self*) → *teqpflsh._teqpflsh_impl.Geometry*

containsPoint(*self*, *arg:* teqpflsh._teqpflsh_impl.Point, /) → bool

difference(*self*, *arg:* teqpflsh._teqpflsh_impl.Geometry, /) → *teqpflsh._teqpflsh_impl.Geometry*

fastTriangulate(*self*) → *teqpflsh._teqpflsh_impl.Geometry*

getCentroid(*self*) → *teqpflsh._teqpflsh_impl.Point*

getCoordinates(*self*) → *teqpflsh._teqpflsh_impl.CoordinateSequence*

getGeometryN(*self*, *arg: int*, /) → *teqpflsh._teqpflsh_impl.Geometry*

getNumGeometries(*self*) → int

getNumPoints(*self*) → int

getX(*self*) → float

getXY(*self*) → tuple[numpy.ndarray[dtype=float64, shape=(*), order='C'], numpy.ndarray[dtype=float64, shape=(*), order='C']]

Convenience function to return the X, Y coordinates as numpy arrays in Python

getY(*self*) → float

get_PreparedGeometry(*self*) → *teqpflsh._teqpflsh_impl.PreparedGeometry*

intersection(*self*, *arg:* teqpflsh._teqpflsh_impl.Geometry, /) → *teqpflsh._teqpflsh_impl.Geometry*

property isSimple
(self) -> bool

property isValid
(self) -> bool

make_valid(*self*) → *teqpflsh._teqpflsh_impl.Geometry*

run_DouglasPeuckerSimplifier(*self*, *tolerance: float*) → *teqpflsh._teqpflsh_impl.Geometry*

run_TopologyPreservingSimplifier(*self*, *tolerance: float*) → *teqpflsh._teqpflsh_impl.Geometry*

class `teqpflsh._teqpflsh_impl.`**PolyQuadNode**

Bases: `object`

C++ docs: PolyQuadNode

property NE

(self) -> teqpflsh._teqpflsh_impl.PolyQuadNode

property NW

(self) -> teqpflsh._teqpflsh_impl.PolyQuadNode

property SE

(self) -> teqpflsh._teqpflsh_impl.PolyQuadNode

property SW

(self) -> teqpflsh._teqpflsh_impl.PolyQuadNode

getNode(*self*, *arg0: float*, *arg1: float*, *arg2: bool*, */*) → *teqpflsh._teqpflsh_impl.PolyQuadNode*

get_contents(*self*) → *teqpflsh._teqpflsh_impl.LeafContents*

property terminal

(self) -> bool

xmax(*self*) → float

xmin(*self*) → float

ymax(*self*) → float

ymin(*self*) → float

class `teqpflsh._teqpflsh_impl.`**PolyQuadTree**(**args*, ***kwargs*)

Bases: `object`

C++ docs: PolyQuadTree

area_stats(*self*) → None

do_splits(*self*, *arg: int*, */*) → None

get_leaves(*self*) → list[*teqpflsh._teqpflsh_impl.PolyQuadNode*]

get_polygon_xy(*self*, *arg:* teqpflsh._teqpflsh_impl.PolyQuadNode, */*) → tuple[list[float], list[float]] | None

get_status(*self*, *arg:* teqpflsh._teqpflsh_impl.PolyQuadNode, */*) → *teqpflsh._teqpflsh_impl.PQTStatus*

is_complete(*self*, *arg:* teqpflsh._teqpflsh_impl.PolyQuadNode, */*) → bool

is_intersection(*self*, *arg:* teqpflsh._teqpflsh_impl.PolyQuadNode, */*) → bool

property tree

(self) -> teqpflsh._teqpflsh_impl.PolyQuadNode

class teqpflsh._teqpflsh_impl.**PreparedGeometry**

Bases: `object`

C++ docs: PreparedGeometry

contains(*self*, *arg:* teqpflsh._teqpflsh_impl.Geometry, /) → bool

nearestPoints(*self*, *arg:* teqpflsh._teqpflsh_impl.Geometry, /) → *teqpflsh._teqpflsh_impl.CoordinateSequence*

class teqpflsh._teqpflsh_impl.**PropertyPairs**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `Enum`

PropertyPairs

DH = 3

DP = 2

DS = 4

DT = 1

DU = 5

HP = 6

HS = 9

HT = 10

HU = 14

PS = 7

PT = 13

PU = 8

ST = 0

SU = 12

TU = 11

class teqpflsh._teqpflsh_impl.**PropertySet**

Bases: `object`

C++ docs: PropertySet

property T

(self) -> numpy.ndarray[dtype=float64, shape=(*), order='C']

get_array(*self*, *arg: str*, /) → numpy.ndarray[dtype=float64, shape=(*), order='C']

get_arrays(*self*, *arg:* teqpflsh._teqpflsh_impl.PropertyPairs, /) → tuple[numpy.ndarray[dtype=float64, shape=(*), order='C'], numpy.ndarray[dtype=float64, shape=(*), order='C']]

property h
(self) -> numpy.ndarray[dtype=float64, shape=(*), order='C']

property p
(self) -> numpy.ndarray[dtype=float64, shape=(*), order='C']

property rho
(self) -> numpy.ndarray[dtype=float64, shape=(*), order='C']

property s
(self) -> numpy.ndarray[dtype=float64, shape=(*), order='C']

property u
(self) -> numpy.ndarray[dtype=float64, shape=(*), order='C']

class `teqpflsh._teqpflsh_impl.`**QuadRegion2D**(**args*, ***kwargs*)

Bases: `object`

C++ docs: QuadRegion2D

property bounding_polygon
(self) -> teqpflsh._teqpflsh_impl.Geometry

do_Delaunay_triangulation(*self*) → *teqpflsh._teqpflsh_impl.Geometry*

do_fast_triangulation(*self*) → *teqpflsh._teqpflsh_impl.Geometry*

do_splits(*self*, *arg: int*, */*) → None

get_coords_xy(*self*) → tuple[list[float], list[float]]

get_envelope(*self*) → *teqpflsh._teqpflsh_impl.Envelope*

get_nonsimple_xy(*self*) → tuple[list[float], list[float]]

get_quadtree_ro(*self*) → *teqpflsh._teqpflsh_impl.PolyQuadTree*

get_quadtree_rw(*self*) → *teqpflsh._teqpflsh_impl.PolyQuadTree*

sample_gridded(*self, arg0: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg1: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg2: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg3: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], /*) → int

sample_gridded_w_tree(*self, arg0: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg1: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg2: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg3: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], /*) → int

sample_random(*self, arg0: int, arg1: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg2: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], /*) → None

class `teqpflsh._teqpflsh_impl.`**RegionedFlashReturn**

Bases: `object`

C++ docs: RegionedFlashReturn

property T
Temperature, K

property candidate_duration_us

How long the candidate determination part took, in microseconds

property maxabsr

Maximum absolute residual

property msg

Message associated with stoppping reason

property newton_duration_us

How long the Newton part took, in microseconds

property reason

Enumerated value for stopping reason

property rho

Molar density, mol/m3

property step_count

How many Newton steps were takenm

property total_duration_us

How long the total calculation took, in microseconds

class teqpflsh._teqpflsh_impl.**RegionedFlasher**(**args*, ***kwargs*)

Bases: `object`

C++ docs: RegionedFlasher

add_region(*self, *, T: numpy.ndarray[dtype=float64, shape=(*), order='C'], rho: numpy.ndarray[dtype=float64, shape=(*), order='C'], NT: int, Nrho: int*) → None

Add a region to the set of regions

flash(*self*, *proppair:* teqpflsh._teqpflsh_impl.PropertyPairs, *val1: float*, *val2: float*) → *teqpflsh._teqpflsh_impl.RegionedFlashReturn*

Do a flash calculation

flash_many(*self, proppair: teqpflsh._teqpflsh_impl.PropertyPairs, val1: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], val2: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], T: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], rho: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], steps: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], maxabs: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], newtontime: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], candtime: ndarray[dtype=float64, shape=(*), order='C', device='cpu']*) → None

Do many flash calculations, for testing in Python

get_NRIterator(*self, arg0: collections.abc.Sequence[str], arg1: numpy.ndarray[dtype=float64, shape=(2), order='C'], arg2: float, arg3: float, arg4: numpy.ndarray[dtype=float64, shape=(*), order='C'], arg5: tuple[bool, bool], arg6: collections.abc.Sequence[teqpflsh._teqpflsh_impl.StoppingCondition], /*) → *teqpflsh._teqpflsh_impl.NRIterator*

Construct a Newton iterator object

get_quadtree_intersections(*self*, *arg0:* teqpflsh._teqpflsh_impl.PropertyPairs, *arg1: float*, *arg2: float*, */*) → list[*teqpflsh._teqpflsh_impl.QuadRegion2D*]

`get_regions_ro`(*self*) → list[*teqpflsh._teqpflsh_impl.ThermodynamicRegion*]

Get a read-only view of the regions

`get_regions_rw`(*self*) → list[*teqpflsh._teqpflsh_impl.ThermodynamicRegion*]

Get read-write access to the regions

`get_starting_Trho`(*self*, *arg0:* teqpflsh._teqpflsh_impl.PropertyPairs, *arg1: float*, *arg2: float*, */*) → list[tuple[*teqpflsh._teqpflsh_impl.ThermodynamicRegion*, *teqpflsh._teqpflsh_impl.TrhoLookup*]]

Get the starting temperature, density pair from the K-D tree

`remove_all_regions`(*self*) → None

Remove all the regions to restore object to its initial state

class `teqpflsh._teqpflsh_impl.`**`StoppingCondition`**

Bases: `object`

C++ docs: StoppingCondition

class `teqpflsh._teqpflsh_impl.`**`StoppingConditionReason`**(*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `Enum`

StoppingConditionReason

`fatal = 3`

`keep_going = 1`

`success = 2`

class `teqpflsh._teqpflsh_impl.`**`SuperAncillary`**(**args*, ***kwargs*)

Bases: `object`

C++ docs: SuperAncillary

`add_variable`(*self*, ***, *k: str*, *caller: collections.abc.Callable[[float, float], float]*) → None

`eval_sat`(*self*, ***, *T: float*, *k: str*, *q: int*) → float

`eval_sat_many`(*self, *, T: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], k: str, q: int, y: ndarray[dtype=float64, shape=(*), order='C', device='cpu']*) → None

`flash`(*self*, *arg0:* teqpflsh._teqpflsh_impl.PropertyPairs, *arg1: float*, *arg2: float*, */*) → *teqpflsh._teqpflsh_impl.SuperAncillaryTwoPhaseSolution* | None

`flash_many`(*self, arg0: teqpflsh._teqpflsh_impl.PropertyPairs, arg1: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg2: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg3: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg4: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg5: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], /*) → None

`get_T_from_p`(*self*, ***, *p: float*) → float

`get_approx1d`(*self*, ***, *k: str*, *q: int*) → *teqpflsh._teqpflsh_impl.ChebyshevApproximation1D*

`get_vaporquality`(*self*, ***, *T: float*, *propval: float*, *k: str*) → float

get_yval (*self*, *, *T: float*, *q: float*, *k: str*) → float

get_yval_many (*self, *, T: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], k: str, q: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], y: ndarray[dtype=float64, shape=(*), order='C', device='cpu']*) → None

property invlnp

(self) -> teqpflsh._teqpflsh_impl.ChebyshevApproximation1D

solve_for_T (*self*, *, *propval: float*, *k: str*, *q: bool*, *bits: int = 64*, *max_iter: int = 100*, *boundsftol: float = 1e-13*) → list[tuple[float, int]]

solve_for_Tq_DX (*self*, *arg0: float*, *arg1: float*, *arg2: str*, *arg3: int*, *arg4: int*, *arg5: float*, */*) → *teqpflsh._teqpflsh_impl.SuperAncillaryTwoPhaseSolution* | None

solve_for_Tq_DX_many (*self, arg0: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg1: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg2: str, arg3: int, arg4: int, arg5: float, arg6: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg7: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg8: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], /*) → None

class teqpflsh._teqpflsh_impl.**SuperAncillaryTwoPhaseSolution**

Bases: `object`

C++ docs: SuperAncillaryTwoPhaseSolution

property T

(self) -> float

property counter

(self) -> int

property q

(self) -> float

class teqpflsh._teqpflsh_impl.**ThermodynamicRegion**

Bases: `object`

C++ docs: ThermodynamicRegion

add_pair (*self*, *, *proppair:* teqpflsh._teqpflsh_impl.PropertyPairs, *Nsplit: int*, *and_kdtree: bool = True*) → None

get_kdtree (*self*, *arg:* teqpflsh._teqpflsh_impl.PropertyPairs, */*) → *teqpflsh._teqpflsh_impl.L2Tree*

get_starting_Trho_many (*self, proppair: teqpflsh._teqpflsh_impl.PropertyPairs, val1: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], val2: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], T: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], rho: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], d2: ndarray[dtype=float64, shape=(*), order='C', device='cpu']*) → None

get_transformed_region (*self*, *arg:* teqpflsh._teqpflsh_impl.PropertyPairs, */*) → *teqpflsh._teqpflsh_impl.QuadRegion2D*

has_pair (*self*, *arg:* teqpflsh._teqpflsh_impl.PropertyPairs, */*) → bool

property propset_Trhogrid

(self) -> teqpflsh::properties::PropertySet<Eigen::Array<double, -1, 1, 0, -1, 1> >

property propset_bounding

(self) -> teqpflsh::properties::PropertySet<Eigen::Array<double, -1, 1, 0, -1, 1> >

property transformed_regions

(self) -> dict[teqpflsh::properties::PropertyPairs, teqpflsh._teqpflsh_impl.QuadRegion2D]

class `teqpflsh._teqpflsh_impl.`**TrhoLookup**

Bases: `object`

C++ docs: TrhoLookup

property T

(self) -> float

property d2

(self) -> float

property rho

(self) -> float

`teqpflsh._teqpflsh_impl.`**add**(*arg0: int*, *arg1: int*, */*) → int

add

`teqpflsh._teqpflsh_impl.`**get_pair_from_chars**(*arg0: str*, *arg1: str*, */*) → *teqpflsh._teqpflsh_impl.PropertyPairs*

get_pair_from_chars

`teqpflsh._teqpflsh_impl.`**get_pair_log_scaling**(*arg:* teqpflsh._teqpflsh_impl.PropertyPairs, */*) → tuple[bool, bool]

get_pair_log_scaling

`teqpflsh._teqpflsh_impl.`**get_property_chars**(*arg:* teqpflsh._teqpflsh_impl.PropertyPairs, */*) → tuple[str, str]

get_property_chars

`teqpflsh._teqpflsh_impl.`**indexer**(*arg0: ndarray[dtype=float64, shape=(*), order='C', device='cpu'], arg1: int, /*) → float

indexer

`teqpflsh._teqpflsh_impl.`**indexer33**(*arg0: ndarray[dtype=float64, shape=(3, 3), order='C', device='cpu'], arg1: int, arg2: int, arg3: str, /*) → float

indexer33

class `teqpflsh._teqpflsh_impl.`**teqpHelmholtzInterface**(**args*, ***kwargs*)

Bases: *`HelmholtzInterface`*

C++ docs: teqpHelmholtzInterface

`teqpflsh._teqpflsh_impl.`**toms748_solve**(*arg0: collections.abc.Callable[[float], float]*, *arg1: float*, *arg2: float*, *arg3: int*, *arg4: int*, */*) → tuple[tuple[float, float], int]

toms748_solve

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

INDEX

A

AbstractScaler (*class in teqpflsh._teqpflsh_impl*), 45
add() (*in module teqpflsh._teqpflsh_impl*), 58
add() (*teqpflsh._teqpflsh_impl.CoordinateSequence method*), 46
add_pair() (*teqpflsh._teqpflsh_impl.ThermodynamicRegion method*), 57
add_region() (*teqpflsh._teqpflsh_impl.RegionedFlasher method*), 55
add_variable() (*teqpflsh._teqpflsh_impl.SuperAncillary method*), 56
area_stats() (*teqpflsh._teqpflsh_impl.PolyQuadTree method*), 52

B

bounding_polygon (*teqpflsh._teqpflsh_impl.QuadRegion2D property*), 54

C

calc_J() (*teqpflsh._teqpflsh_impl.NRIterator method*), 50
calc_just_step() (*teqpflsh._teqpflsh_impl.NRIterator method*), 50
calc_maxabsr() (*teqpflsh._teqpflsh_impl.NRIterator method*), 50
calc_r() (*teqpflsh._teqpflsh_impl.NRIterator method*), 50
calc_step() (*teqpflsh._teqpflsh_impl.NRIterator method*), 50
calc_vals() (*teqpflsh._teqpflsh_impl.NRIterator method*), 50
candidate_duration_us (*teqpflsh._teqpflsh_impl.RegionedFlashReturn property*), 54
ChebyshevApproximation1D (*class in teqpflsh._teqpflsh_impl*), 45
ChebyshevExpansion (*class in teqpflsh._teqpflsh_impl*), 45
closeRing() (*teqpflsh._teqpflsh_impl.CoordinateSequence method*), 46
coeff (*teqpflsh._teqpflsh_impl.ChebyshevExpansion property*), 46
contains() (*teqpflsh._teqpflsh_impl.PreparedGeometry method*), 53
containsPoint() (*teqpflsh._teqpflsh_impl.Geometry method*), 47
containsPoint() (*teqpflsh._teqpflsh_impl.Point method*), 51
CoordinateSequence (*class in teqpflsh._teqpflsh_impl*), 46
count_x_for_y_many() (*teqpflsh._teqpflsh_impl.ChebyshevApproximation1D method*), 45
counter (*teqpflsh._teqpflsh_impl.SuperAncillaryTwoPhaseSolution property*), 57
createPoint() (*teqpflsh._teqpflsh_impl.GeometryFactoryHolder method*), 48
createPolygon() (*teqpflsh._teqpflsh_impl.GeometryFactory method*), 48
createPolygon() (*teqpflsh._teqpflsh_impl.GeometryFactoryHolder method*), 48

D

d2 (*teqpflsh._teqpflsh_impl.TrhoLookup property*), 58
DelaunayTriangulate() (*teqpflsh._teqpflsh_impl.Geometry method*), 47
DelaunayTriangulate() (*teqpflsh._teqpflsh_impl.Point method*), 51
DH (*teqpflsh._teqpflsh_impl.PropertyPairs attribute*), 53
difference() (*teqpflsh._teqpflsh_impl.Geometry method*), 47
difference() (*teqpflsh._teqpflsh_impl.Point method*), 51
do_Delaunay_triangulation() (*teqpflsh._teqpflsh_impl.QuadRegion2D method*), 54
do_fast_triangulation() (*teqpflsh._teqpflsh_impl.QuadRegion2D method*), 54
do_splits() (*teqpflsh._teqpflsh_impl.PolyQuadTree method*), 52
do_splits() (*teqpflsh._teqpflsh_impl.QuadRegion2D method*), 54

DP (*teqpflsh._teqpflsh_impl.PropertyPairs attribute*), 53
DS (*teqpflsh._teqpflsh_impl.PropertyPairs attribute*), 53
DT (*teqpflsh._teqpflsh_impl.PropertyPairs attribute*), 53
DU (*teqpflsh._teqpflsh_impl.PropertyPairs attribute*), 53

E

Envelope (*class in teqpflsh._teqpflsh_impl*), 46
eval() (*teqpflsh._teqpflsh_impl.ChebyshevApproximation1D method*), 45
eval() (*teqpflsh._teqpflsh_impl.ChebyshevExpansion method*), 46
eval_Eigen() (*teqpflsh._teqpflsh_impl.ChebyshevExpansion method*), 46
eval_many() (*teqpflsh._teqpflsh_impl.ChebyshevApproximation1D method*), 45
eval_many() (*teqpflsh._teqpflsh_impl.ChebyshevExpansion method*), 46
eval_sat() (*teqpflsh._teqpflsh_impl.SuperAncillary method*), 56
eval_sat_many() (*teqpflsh._teqpflsh_impl.SuperAncillary method*), 56
expansioninfo (*teqpflsh._teqpflsh_impl.IntervalMatch property*), 48
expansions (*teqpflsh._teqpflsh_impl.ChebyshevApproximation1D property*), 45

F

fastTriangulate() (*teqpflsh._teqpflsh_impl.Geometry method*), 47
fastTriangulate() (*teqpflsh._teqpflsh_impl.Point method*), 51
fatal (*teqpflsh._teqpflsh_impl.StoppingConditionReason attribute*), 56
flash() (*teqpflsh._teqpflsh_impl.MainFlasher method*), 49
flash() (*teqpflsh._teqpflsh_impl.RegionedFlasher method*), 55
flash() (*teqpflsh._teqpflsh_impl.SuperAncillary method*), 56
flash_many() (*teqpflsh._teqpflsh_impl.MainFlasher method*), 49
flash_many() (*teqpflsh._teqpflsh_impl.RegionedFlasher method*), 55
flash_many() (*teqpflsh._teqpflsh_impl.SuperAncillary method*), 56
FlashPhase (*class in teqpflsh._teqpflsh_impl*), 47
FlashSolution (*class in teqpflsh._teqpflsh_impl*), 47

G

Geometry (*class in teqpflsh._teqpflsh_impl*), 47
GeometryFactory (*class in teqpflsh._teqpflsh_impl*), 48
GeometryFactoryHolder (*class in teqpflsh._teqpflsh_impl*), 48
get_approx1d() (*teqpflsh._teqpflsh_impl.SuperAncillary method*), 56
get_array() (*teqpflsh._teqpflsh_impl.PropertySet method*), 53
get_arrays() (*teqpflsh._teqpflsh_impl.PropertySet method*), 53
get_contents() (*teqpflsh._teqpflsh_impl.PolyQuadNode method*), 52
get_coords_xy() (*teqpflsh._teqpflsh_impl.QuadRegion2D method*), 54
get_envelope() (*teqpflsh._teqpflsh_impl.QuadRegion2D method*), 54
get_intervals_containing_y() (*teqpflsh._teqpflsh_impl.ChebyshevApproximation1D method*), 45
get_kdtree() (*teqpflsh._teqpflsh_impl.ThermodynamicRegion method*), 57
get_leaves() (*teqpflsh._teqpflsh_impl.PolyQuadTree method*), 52
get_maxabsr() (*teqpflsh._teqpflsh_impl.NRIterator method*), 50
get_nearest_indexd2() (*teqpflsh._teqpflsh_impl.L2Tree method*), 49
get_nearest_indexd2_many() (*teqpflsh._teqpflsh_impl.L2Tree method*), 49
get_nonconstant_indices() (*teqpflsh._teqpflsh_impl.NRIterator method*), 50
get_nonsimple_xy() (*teqpflsh._teqpflsh_impl.QuadRegion2D method*), 54
get_NRIterator() (*teqpflsh._teqpflsh_impl.RegionedFlasher method*), 55
get_pair_from_chars() (*in module teqpflsh._teqpflsh_impl*), 58
get_pair_log_scaling() (*in module teqpflsh._teqpflsh_impl*), 58
get_polygon_xy() (*teqpflsh._teqpflsh_impl.PolyQuadTree method*), 52
get_PreparedGeometry() (*teqpflsh._teqpflsh_impl.Geometry method*), 47
get_PreparedGeometry() (*teqpflsh._teqpflsh_impl.Point method*), 51
get_property_chars() (*in module teqpflsh._teqpflsh_impl*), 58
get_quadtree_intersections() (*teqpflsh._teqpflsh_impl.RegionedFlasher method*), 55
get_quadtree_ro() (*teqpflsh._teqpflsh_impl.QuadRegion2D method*), 54

get_quadtree_rw() (*teqpflsh._teqpflsh_impl.QuadRegion2D method*), 54
get_regions_ro() (*teqpflsh._teqpflsh_impl.RegionedFlasher method*), 55
get_regions_rw() (*teqpflsh._teqpflsh_impl.RegionedFlasher method*), 56
get_rho() (*teqpflsh._teqpflsh_impl.NRIterator method*), 50
get_starting_Trho() (*teqpflsh._teqpflsh_impl.RegionedFlasher method*), 56
get_starting_Trho_many() (*teqpflsh._teqpflsh_impl.ThermodynamicRegion method*), 57
get_status() (*teqpflsh._teqpflsh_impl.PolyQuadTree method*), 52
get_step_count() (*teqpflsh._teqpflsh_impl.NRIterator method*), 50
get_T() (*teqpflsh._teqpflsh_impl.NRIterator method*), 50
get_T_from_p() (*teqpflsh._teqpflsh_impl.SuperAncillary method*), 56
get_transformed_region() (*teqpflsh._teqpflsh_impl.ThermodynamicRegion method*), 57
get_used_bytes() (*teqpflsh._teqpflsh_impl.L2Tree method*), 49
get_vals() (*teqpflsh._teqpflsh_impl.NRIterator method*), 50
get_vaporquality() (*teqpflsh._teqpflsh_impl.SuperAncillary method*), 56
get_x_for_y() (*teqpflsh._teqpflsh_impl.ChebyshevApproximation1D method*), 45
get_yval() (*teqpflsh._teqpflsh_impl.SuperAncillary method*), 56
get_yval_many() (*teqpflsh._teqpflsh_impl.SuperAncillary method*), 57
getCentroid() (*teqpflsh._teqpflsh_impl.Geometry method*), 47
getCentroid() (*teqpflsh._teqpflsh_impl.Point method*), 51
getCoordinates() (*teqpflsh._teqpflsh_impl.Geometry method*), 47
getCoordinates() (*teqpflsh._teqpflsh_impl.Point method*), 51
getGeometryN() (*teqpflsh._teqpflsh_impl.Geometry method*), 47
getGeometryN() (*teqpflsh._teqpflsh_impl.Point method*), 51
getNode() (*teqpflsh._teqpflsh_impl.PolyQuadNode method*), 52
getNumGeometries() (*teqpflsh._teqpflsh_impl.Geometry method*), 47
getNumGeometries() (*teqpflsh._teqpflsh_impl.Point method*), 51
getNumPoints() (*teqpflsh._teqpflsh_impl.Geometry method*), 47
getNumPoints() (*teqpflsh._teqpflsh_impl.Point method*), 51
getSize() (*teqpflsh._teqpflsh_impl.CoordinateSequence method*), 46
getX() (*teqpflsh._teqpflsh_impl.CoordinateSequence method*), 46
getX() (*teqpflsh._teqpflsh_impl.Point method*), 51
getXY() (*teqpflsh._teqpflsh_impl.Geometry method*), 47
getXY() (*teqpflsh._teqpflsh_impl.Point method*), 51
getY() (*teqpflsh._teqpflsh_impl.CoordinateSequence method*), 46
getY() (*teqpflsh._teqpflsh_impl.Point method*), 51

H

h (*teqpflsh._teqpflsh_impl.PropertySet property*), 53
has_pair() (*teqpflsh._teqpflsh_impl.ThermodynamicRegion method*), 57
HelmholtzInterface (*class in teqpflsh._teqpflsh_impl*), 48
HP (*teqpflsh._teqpflsh_impl.PropertyPairs attribute*), 53
HS (*teqpflsh._teqpflsh_impl.PropertyPairs attribute*), 53
HT (*teqpflsh._teqpflsh_impl.PropertyPairs attribute*), 53
HU (*teqpflsh._teqpflsh_impl.PropertyPairs attribute*), 53

I

idx (*teqpflsh._teqpflsh_impl.MonotonicExpansionMatch property*), 50
indexer() (*in module teqpflsh._teqpflsh_impl*), 58
indexer33() (*in module teqpflsh._teqpflsh_impl*), 58
inside (*teqpflsh._teqpflsh_impl.PQTStatus attribute*), 51
intersection (*teqpflsh._teqpflsh_impl.PQTStatus attribute*), 51
intersection() (*teqpflsh._teqpflsh_impl.Geometry method*), 47
intersection() (*teqpflsh._teqpflsh_impl.Point method*), 51
IntervalMatch (*class in teqpflsh._teqpflsh_impl*), 48
invlnp (*teqpflsh._teqpflsh_impl.SuperAncillary property*), 57
is_complete() (*teqpflsh._teqpflsh_impl.PolyQuadTree method*), 52
is_intersection() (*teqpflsh._teqpflsh_impl.PolyQuadTree method*), 52
isSimple (*teqpflsh._teqpflsh_impl.Geometry property*), 48
isSimple (*teqpflsh._teqpflsh_impl.Point property*), 51
isValid (*teqpflsh._teqpflsh_impl.Geometry property*), 48
isValid (*teqpflsh._teqpflsh_impl.Point property*), 51

K

keep_going (*teqpflsh._teqpflsh_impl.StoppingConditionReason attribute*), 56

L

L2Tree (*class in teqpflsh._teqpflsh_impl*), 48
L2TreeHolder (*class in teqpflsh._teqpflsh_impl*), 49
LeafContents (*class in teqpflsh._teqpflsh_impl*), 49

M

MainFlasher (*class in teqpflsh._teqpflsh_impl*), 49
make_valid() (*teqpflsh._teqpflsh_impl.Geometry method*), 48
make_valid() (*teqpflsh._teqpflsh_impl.Point method*), 51
makeclosedpolygon() (*teqpflsh._teqpflsh_impl.GeometryFactoryHolder method*), 48
MaxAbsErrorCondition (*class in teqpflsh._teqpflsh_impl*), 49
maxabsr (*teqpflsh._teqpflsh_impl.RegionedFlashReturn property*), 55
MinMaxLogScaler (*class in teqpflsh._teqpflsh_impl*), 49
MinMaxScaler (*class in teqpflsh._teqpflsh_impl*), 49
mole_fractions (*teqpflsh._teqpflsh_impl.FlashPhase property*), 47
monotonic_intervals (*teqpflsh._teqpflsh_impl.ChebyshevApproximation1D property*), 45
MonotonicExpansionMatch (*class in teqpflsh._teqpflsh_impl*), 50
msg (*teqpflsh._teqpflsh_impl.RegionedFlashReturn property*), 55

N

NanXDXErrorCondition (*class in teqpflsh._teqpflsh_impl*), 51
NE (*teqpflsh._teqpflsh_impl.PolyQuadNode property*), 52
nearestPoints() (*teqpflsh._teqpflsh_impl.PreparedGeometry method*), 53
newton_duration_us (*teqpflsh._teqpflsh_impl.RegionedFlashReturn property*), 55
Nphases (*teqpflsh._teqpflsh_impl.FlashSolution property*), 47
NRIterator (*class in teqpflsh._teqpflsh_impl*), 50
NW (*teqpflsh._teqpflsh_impl.PolyQuadNode property*), 52

O

outside (*teqpflsh._teqpflsh_impl.PQTStatus attribute*), 51

P

p (*teqpflsh._teqpflsh_impl.PropertySet property*), 54
path_integration() (*teqpflsh._teqpflsh_impl.NRIterator method*), 50
phases (*teqpflsh._teqpflsh_impl.FlashSolution property*), 47
Point (*class in teqpflsh._teqpflsh_impl*), 51
PolyQuadNode (*class in teqpflsh._teqpflsh_impl*), 52
PolyQuadTree (*class in teqpflsh._teqpflsh_impl*), 52
PQTStatus (*class in teqpflsh._teqpflsh_impl*), 51
PreparedGeometry (*class in teqpflsh._teqpflsh_impl*), 52
PropertyPairs (*class in teqpflsh._teqpflsh_impl*), 53
PropertySet (*class in teqpflsh._teqpflsh_impl*), 53
propset_bounding (*teqpflsh._teqpflsh_impl.ThermodynamicRegion property*), 58
propset_Trhogrid (*teqpflsh._teqpflsh_impl.ThermodynamicRegion property*), 57
PS (*teqpflsh._teqpflsh_impl.PropertyPairs attribute*), 53
PT (*teqpflsh._teqpflsh_impl.PropertyPairs attribute*), 53
PU (*teqpflsh._teqpflsh_impl.PropertyPairs attribute*), 53

Q

q (*teqpflsh._teqpflsh_impl.SuperAncillaryTwoPhaseSolution property*), 57
qmolar (*teqpflsh._teqpflsh_impl.FlashPhase property*), 47
QuadRegion2D (*class in teqpflsh._teqpflsh_impl*), 54

R

reason (*teqpflsh._teqpflsh_impl.RegionedFlashReturn property*), 55
regioned_flasher (*teqpflsh._teqpflsh_impl.MainFlasher property*), 49
RegionedFlasher (*class in teqpflsh._teqpflsh_impl*), 55
RegionedFlashReturn (*class in teqpflsh._teqpflsh_impl*), 54
remove_all_regions() (*teqpflsh._teqpflsh_impl.RegionedFlasher method*), 56
reset() (*teqpflsh._teqpflsh_impl.NRIterator method*), 50
rho (*teqpflsh._teqpflsh_impl.PropertySet property*), 54
rho (*teqpflsh._teqpflsh_impl.RegionedFlashReturn property*), 55
rho (*teqpflsh._teqpflsh_impl.TrhoLookup property*), 58
rho_molm3 (*teqpflsh._teqpflsh_impl.FlashPhase property*), 47
run_DouglasPeuckerSimplifier() (*teqpflsh._teqpflsh_impl.Geometry method*), 48
run_DouglasPeuckerSimplifier() (*teqpflsh._teqpflsh_impl.Point method*), 52
run_TopologyPreservingSimplifier() (*teqpflsh._teqpflsh_impl.Geometry method*), 48

run_TopologyPreservingSimplifier() (*teqpflsh._teqpflsh_impl.Point method*), 52

S

s (*teqpflsh._teqpflsh_impl.PropertySet property*), 54
sample_gridded() (*teqpflsh._teqpflsh_impl.QuadRegion2D method*), 54
sample_gridded_w_tree() (*teqpflsh._teqpflsh_impl.QuadRegion2D method*), 54
sample_random() (*teqpflsh._teqpflsh_impl.QuadRegion2D method*), 54
SE (*teqpflsh._teqpflsh_impl.PolyQuadNode property*), 52
solve_for_T() (*teqpflsh._teqpflsh_impl.SuperAncillary method*), 57
solve_for_Tq_DX() (*teqpflsh._teqpflsh_impl.SuperAncillary method*), 57
solve_for_Tq_DX_many() (*teqpflsh._teqpflsh_impl.SuperAncillary method*), 57
solve_for_x() (*teqpflsh._teqpflsh_impl.ChebyshevExpansion method*), 46
solve_for_x_count() (*teqpflsh._teqpflsh_impl.ChebyshevExpansion method*), 46
solve_for_x_many() (*teqpflsh._teqpflsh_impl.ChebyshevExpansion method*), 46
ST (*teqpflsh._teqpflsh_impl.PropertyPairs attribute*), 53
status (*teqpflsh._teqpflsh_impl.LeafContents property*), 49
step_count (*teqpflsh._teqpflsh_impl.RegionedFlashReturn property*), 55
StoppingCondition (*class in teqpflsh._teqpflsh_impl*), 56
StoppingConditionReason (*class in teqpflsh._teqpflsh_impl*), 56
SU (*teqpflsh._teqpflsh_impl.PropertyPairs attribute*), 53
success (*teqpflsh._teqpflsh_impl.StoppingConditionReason attribute*), 56
SuperAncillary (*class in teqpflsh._teqpflsh_impl*), 56
SuperAncillaryTwoPhaseSolution (*class in teqpflsh._teqpflsh_impl*), 57
SW (*teqpflsh._teqpflsh_impl.PolyQuadNode property*), 52

T

T (*teqpflsh._teqpflsh_impl.PropertySet property*), 53
T (*teqpflsh._teqpflsh_impl.RegionedFlashReturn property*), 54
T (*teqpflsh._teqpflsh_impl.SuperAncillaryTwoPhaseSolution property*), 57
T (*teqpflsh._teqpflsh_impl.TrhoLookup property*), 58
T_K (*teqpflsh._teqpflsh_impl.FlashSolution property*), 47
take_steps() (*teqpflsh._teqpflsh_impl.NRIterator method*), 50
teqpHelmholtzInterface (*class in teqpflsh._teqpflsh_impl*), 58
terminal (*teqpflsh._teqpflsh_impl.PolyQuadNode property*), 52
ThermodynamicRegion (*class in teqpflsh._teqpflsh_impl*), 57
toms748_solve() (*in module teqpflsh._teqpflsh_impl*), 58
total_duration_us (*teqpflsh._teqpflsh_impl.RegionedFlashReturn property*), 55
transformed_regions (*teqpflsh._teqpflsh_impl.ThermodynamicRegion property*), 58
tree (*teqpflsh._teqpflsh_impl.L2TreeHolder property*), 49
tree (*teqpflsh._teqpflsh_impl.PolyQuadTree property*), 52
TrhoLookup (*class in teqpflsh._teqpflsh_impl*), 58
TU (*teqpflsh._teqpflsh_impl.PropertyPairs attribute*), 53

U

u (*teqpflsh._teqpflsh_impl.PropertySet property*), 54

V

verbose (*teqpflsh._teqpflsh_impl.NRIterator property*), 50

X

x_at_extrema (*teqpflsh._teqpflsh_impl.ChebyshevApproximation1D property*), 45
x_max (*teqpflsh._teqpflsh_impl.Envelope property*), 46
x_min (*teqpflsh._teqpflsh_impl.Envelope property*), 46
xmax (*teqpflsh._teqpflsh_impl.ChebyshevApproximation1D property*), 45
xmax (*teqpflsh._teqpflsh_impl.ChebyshevExpansion property*), 46
xmax (*teqpflsh._teqpflsh_impl.IntervalMatch property*), 48
xmax (*teqpflsh._teqpflsh_impl.MonotonicExpansionMatch property*), 50
xmax() (*teqpflsh._teqpflsh_impl.PolyQuadNode method*), 52
xmin (*teqpflsh._teqpflsh_impl.ChebyshevApproximation1D property*), 45
xmin (*teqpflsh._teqpflsh_impl.ChebyshevExpansion property*), 46
xmin (*teqpflsh._teqpflsh_impl.IntervalMatch property*), 48
xmin (*teqpflsh._teqpflsh_impl.MonotonicExpansionMatch property*), 50
xmin() (*teqpflsh._teqpflsh_impl.PolyQuadNode method*), 52

Y

y_max (*teqpflsh._teqpflsh_impl.Envelope property*), 46

y_min (*teqpflsh._teqpflsh_impl.Envelope property*), 46
ymax (*teqpflsh._teqpflsh_impl.IntervalMatch property*), 48
ymax (*teqpflsh._teqpflsh_impl.MonotonicExpansionMatch property*), 50
ymax() (*teqpflsh._teqpflsh_impl.PolyQuadNode method*), 52
ymin (*teqpflsh._teqpflsh_impl.IntervalMatch property*), 48
ymin (*teqpflsh._teqpflsh_impl.MonotonicExpansionMatch property*), 50
ymin() (*teqpflsh._teqpflsh_impl.PolyQuadNode method*), 52