GHEtool

Release 2.2.1

Wouter Peere and Tobias Blanke

Jan 27, 2024

GHETOOL

Dear visitor

Welcome on this ReadTheDocs for the open-source borefield sizing tool GHEtool.

Here one can find (extensive) documentation of GHEtool codebase. Please find below the content of this ReadTheDocs.

CHAPTER

ONE

GRAPHICAL USER INTERFACE

There are two graphical user interfaces available which are built using GHEtool: GHEtool Pro and GHEtool Community

1.1 GHEtool Pro

GHEtool Pro is the official and supported version of GHEtool which supports drilling companies, engineering firms, architects, government organizations in their geothermal design process. With GHEtool Pro they can minimize the environmental and societal impact while maximizing the cost-effective utilization of geothermal projects. Visit our website at https://ghetool.eu to learn more about the synergy between this open-source package and GHEtool Pro.

The user manual for GHEtool Pro, together with instructions for installation, you can find here.

1.2 GHEtool Community

Besides GHEtool Pro, an open-source alternative for the graphical user interface is available in the form of *GHEtool Community*. This version is built and maintained by the community, and **has no official support like GHEtool Pro**. You can read all about this *GHEtool Community* on their GitHub Repo.

1.2.1 Cite GHEtool

General citation

Whenever you work with GHEtool (in general), please reference this tool using the JOSS paper.

> Peere, W., Blanke, T.(2022). GHEtool: An open-source tool for borefield sizing in Python. *Journal of Open Source Software, 7*(76), 4406, https://doi.org/10.21105/joss.04406

Whenever you use a specific functionality within GHEtool, one can also reference a specific paper/article related to this functionality.

Hybrid sizing methodology (L2) | borefield quadrants

Whenever you use the hybrid sizing methodology (L2 in GHEtool) or make use of borefield quadrants, please use the citations below.

> Peere, W., Picard, D., Cupeiro Figueroa, I., Boydens, W., and Helsen, L. (2021). Validated combined first and last year borefield sizing methodology. In *Proceedings of International Building Simulation Conference 2021*. Brugge (Belgium), 1-3 September 2021. https://doi.org/10.26868/25222708.2021.30180
>
> Peere, W. (2020). Methode voor economische optimalisatie van geothermische verwarmings- en koelsystemen. Master thesis, Department of Mechanical Engineering, KU Leuven, Belgium.

Active-passive cooling example

Whenever you use the example of active-passive cooling, please use the citations below.

> Coninx, M., De Nies, J., Hermans, L., Peere, W., Boydens, W., Helsen, L. (2024). Cost-efficient cooling of buildings by means of geothermal borefields with active and passive cooling. *Applied Energy*, 355, Art. No. 122261, https://doi.org/10.1016/j.apenergy.2023.122261.

1.2.2 GHEtool in literature

Below is a general list of all the articles which use or mention GHEtool in literature. Please let us know if we missed a contribution.

Note that for some of the publications, the code used in that publication, is available in this ReadTheDocs. In this way, we want to contribute to the transparency in the academic world by sharing computer code so it is easier to replicate and verify published results.

The articles for which this is the case, are:

Verleyen et al. (2022)

The code below is used in the article of Verleyen et al. (2022)[1].

```
"""
This document contains all the scripts for the figures of Verleyen et al. (2022).
One needs GHEtool version 2.0.6 to run this code.
"""
import math

import matplotlib.pyplot as plt
import numpy as np

# code for making figures black-and-white ready
from cycler import cycler

from GHEtool import Borefield, GroundData

color_c = cycler('color', ['k'])
style_c = cycler('linestyle', ['-', '--', ':', '-.'])
markr_c = cycler('marker', ['', '.', 'o'])
c_cms = color_c * markr_c * style_c
c_csm = color_c * style_c * markr_c
plt.rc('axes', prop_cycle=c_cms)
plt.rcParams['lines.markersize'] = 8


def figure_1():
    """
    This function generates the first figure (part a/b) of the article.
    """

    # initiate ground data
    ground_data = GroundData(2.4, 10, 0.12)

    # initiate borefield model
    borefield = Borefield()
    borefield.set_ground_parameters(ground_data)
    borefield.create_rectangular_borefield(10, 10, 6, 6, 100, 4)

    # initiate depth array
    depth = 150

    # dimensionless time
    ts = 150**2 / (9 * ground_data.k_s)

    # time array
    nb_of_timesteps = 50
    time_dimensionless = np.linspace(2, 14, nb_of_timesteps)

    # convert to seconds
```

(continues on next page)

[1] Verleyen, L., Peere, W., Michiels, E., Boydens, W., Helsen, L. (2022). The beauty of reason and insight: a story about 30 years old borefield equations. IEA HPT Magazine 40(3), 36-39, https://doi.org/10.23697/6q4n-3223

(continued from previous page)

```
    time_in_seconds = np.exp(time_dimensionless) * ts

    # calculate g-functions
    result = np.zeros(nb_of_timesteps)
    result = borefield.gfunction(time_in_seconds, depth)

    # create figure
    fig, axs = plt.subplots(1, 2, figsize=(10, 3), constrained_layout=True)

    # create figure g-function (lin)
    # plot g-functions
    axs[0].plot(time_in_seconds/8760/3600, result / (2 * math.pi * 2.4))

    # layout
    axs[0].set_title("Step response when applying a constant heat injection")
    axs[0].set_xlabel("Time (years)")
    axs[0].set_ylabel("Temperature difference (K)")
    axs[0].set_ylim(0, 5)
    axs[0].set_xlim(0, 40)

    # create figure g-function (semi-log)
    # plot g-functions
    axs[1].plot(time_dimensionless, result)

    # layout
    axs[1].set_title("Equivalent g-function for the step response")
    axs[1].set_xlabel("ln(t/ts)")
    axs[1].set_ylabel("g-function value")
    axs[1].set_ylim(-2, 60)

    # plt.legend()
    plt.show()


def figure_2():
    """
    This function generates the second figure of the article.
    """

    # initiate ground data
    ground_data = GroundData(2.4, 10, 0.12)

    # initiate borefield model
    borefield = Borefield()
    borefield.set_ground_parameters(ground_data)
    borefield.create_rectangular_borefield(10, 10, 6, 6, 100, 4)

    # initiate depth array
    depths = np.array([25, 50, 100, 150, 200])

    # dimensionless time
    ts = 150**2 / (9 * ground_data.k_s)
```

(continues on next page)

(continued from previous page)

```
    # time array
    nb_of_timesteps = 50
    time_dimensionless = np.linspace(2, 14, nb_of_timesteps)

    # convert to seconds
    time_in_seconds = np.exp(time_dimensionless) * ts

    # calculate g-functions
    results = np.zeros((5, nb_of_timesteps))
    for i in range(5):
        results[i] = borefield.gfunction(time_in_seconds, depths[i])

    # create figure
    plt.figure()

    # plot g-functions
    plt.plot(time_dimensionless, results[0], label="25m")
    plt.plot(time_dimensionless, results[1], label="50m")
    plt.plot(time_dimensionless, results[2], label="100m")
    plt.plot(time_dimensionless, results[3], label="150m")
    plt.plot(time_dimensionless, results[4], label="200m")

    # plot lines for Ra
    line1 = math.log(6*3600/ts)
    line2 = math.log((20 * 8760 + 730) * 3600 / ts)
    plt.vlines(line1, ymin=-5, ymax=5, colors="black", lw=0.75, ls="--")
    plt.vlines(line2, ymin=-5, ymax=50, colors="black", lw=0.75, ls="--")

    plt.annotate('', xy=(line1, -0.4), xytext=(line2, -0.4),
                 arrowprops=dict(arrowstyle='<->', color='black'))
    plt.text((line1+line2)/2, 0.2, "Ra", horizontalalignment='center')

    # layout
    # plt.title("G-function values for different borefield depths")
    plt.xlabel("ln(t/ts)")
    plt.ylabel("g-function value")
    plt.ylim(-2, 60)

    plt.legend()
    plt.show()


def figure_3():
    """
    This function creates the third figure of the article.
    """

    # initiate borefield model
    borefield = Borefield()

    # initiate depth for evaluations
```

(continues on next page)

(continued from previous page)

```
    nb_depths = 20
    depth_array = np.linspace(50, 350, nb_depths)

    # initiate list of borefield configurations
    configs = [(10, 10), (11, 11), (12, 12), (14, 14),
               (15, 15), (18, 18), (20, 20)]

    results = []

    for n1, n2 in configs:
        depths = []
        for H in depth_array:
            # set ground data
            ground_data = GroundData(2.4, 10, 0.12)

            # set ground data
            borefield.set_ground_parameters(ground_data)

            # set borefield
            borefield.create_rectangular_borefield(n1, n2, 7, 7, H, 4)

            # calculate gfunction
            gfunction = borefield.gfunction(borefield.time, H)

            # calculate Ra
            Ra = (gfunction[2] - gfunction[1]) / (2 * math.pi * ground_data.k_s)

            # add to depths
            depths.append(Ra)

        # add to results
        results.append(depths)

    # create figure
    plt.figure()
    for i, config in enumerate(configs):
        plt.plot(depth_array, results[i], label=str(config[0]) + "x" + str(config[1]))

    # plt.title("Ra for different borefield configurations")
    plt.xlabel("Depth (m)")
    plt.ylabel("Ra (mK/W)")

    plt.legend()
    plt.show()


def figure_4():
    """
    This code creates the fourth figure of the article
    """
    # initiate borefield
    borefield = Borefield()
```

(continues on next page)

(continued from previous page)

```
    # set the correct sizing method
    borefield.sizing_setup(L2_sizing=True)

    # initiate array with imbalances
    imbalance_array = np.linspace(200, 1600, 20)

    # initiate list of borefield configurations
    configs = [(10, 10), (11, 11), (12, 12), (14, 14),
               (15, 15), (18, 18), (20, 20)]

    # initiate loads
    monthly_load_heating_percentage = np.array([0.155, 0.148, 0.125, .099, .064, 0., 0.,␣
↪0., 0.061, 0.087, .117, 0.144])
    monthly_load_cooling_percentage = np.array([0.025, 0.05, 0.05, .05, .075, .1, .2, .2,
↪ .1, .075, .05, .025])
    monthly_load_heating = monthly_load_heating_percentage * 100 * 10 ** 3  # kWh
    monthly_load_cooling_init = monthly_load_cooling_percentage * 100 * 10 ** 3  # kWh
    peak_cooling_init = np.array([0., 0, 34., 69., 133., 187., 213., 240., 160., 37., 0.,
↪ 0.])  # Peak cooling in kW
    peak_heating = np.array([160., 142, 102., 55., 0., 0., 0., 0., 40.4, 85., 119., 136.
↪])

    # set heating loads
    borefield.set_peak_heating(peak_heating)
    borefield.set_baseload_heating(monthly_load_heating)

    results = []
    for i, config in enumerate(configs):
        depth_array = []
        for imbalance in imbalance_array:
            # initiate ground data
            ground_data = GroundData(2.4, 10, 0.12)

            # set ground data
            borefield.set_ground_parameters(ground_data)

            # set borefield
            borefield.create_rectangular_borefield(config[0], config[1], 7, 7, 100, 4)

            # calculate loads
            extra_load = imbalance / 12 * 10 ** 3  # kWh
            monthly_load_cooling = monthly_load_cooling_init + extra_load
            peak_cooling = peak_cooling_init + extra_load / 730

            # set cooling loads
            borefield.set_peak_cooling(peak_cooling)
            borefield.set_baseload_cooling(monthly_load_cooling)
            try:
                depth = borefield.size()
                depth_array.append(depth)
            except:
```

(continues on next page)

(continued from previous page)

```
                pass

        results.append(depth_array)

    # create figure
    plt.figure()
    for i, config in enumerate(configs):
        plt.plot(imbalance_array[:len(results[i])], results[i], label=str(config[0]) + "x
↪" + str(config[1]))

    # plt.title("Depth for different imbalances")
    plt.xlabel("Imbalance (MWh/y)")
    plt.ylabel("Depth (m)")

    plt.legend()
    plt.show()


def figure_5():
    """
    This function creates the fifth figure of the article.
    """

    # initiate borefield
    borefield = Borefield()

    # set the correct sizing method
    borefield.sizing_setup(L2_sizing=True)

    # initiate array with imbalances
    imbalance_array = np.linspace(200, 1600, 20)

    # initiate list of borefield configurations
    configs = [(10, 10), (11, 11), (12, 12), (14, 14),
               (15, 15), (18, 18), (20, 20)]

    # initiate loads
    monthly_load_heating_percentage = np.array([0.155, 0.148, 0.125, .099, .064, 0., 0.,␣
↪0., 0.061, 0.087, .117, 0.144])
    monthly_load_cooling_percentage = np.array([0.025, 0.05, 0.05, .05, .075, .1, .2, .2,
↪ .1, .075, .05, .025])
    monthly_load_heating = monthly_load_heating_percentage * 100 * 10 ** 3  # kWh
    monthly_load_cooling_init = monthly_load_cooling_percentage * 100 * 10 ** 3  # kWh
    peak_cooling_init = np.array([0., 0, 34., 69., 133., 187., 213., 240., 160., 37., 0.,
↪ 0.])  # Peak cooling in kW
    peak_heating = np.array([160., 142, 102., 55., 0., 0., 0., 0., 40.4, 85., 119., 136.
↪])

    # set heating loads
    borefield.set_peak_heating(peak_heating)
    borefield.set_baseload_heating(monthly_load_heating)

```

(continues on next page)

(continued from previous page)

```
    results = []
    for i, config in enumerate(configs):
        Ra_array = []
        for imbalance in imbalance_array:
            # initiate ground data
            ground_data = GroundData(2.4, 10, 0.12)

            # set ground data
            borefield.set_ground_parameters(ground_data)

            # set borefield
            borefield.create_rectangular_borefield(config[0], config[1], 7, 7, 100, 4)

            # calculate loads
            extra_load = imbalance / 12 * 10 ** 3  # kWh
            monthly_load_cooling = monthly_load_cooling_init + extra_load
            peak_cooling = peak_cooling_init + extra_load / 730

            # set cooling loads
            borefield.set_peak_cooling(peak_cooling)
            borefield.set_baseload_cooling(monthly_load_cooling)
            try:
                depth = borefield.size()

                # calculate gfunction
                gfunction = borefield.gfunction(borefield.time, depth)

                # calculate Ra
                Ra = (gfunction[2] - gfunction[1]) / (2 * math.pi * borefield.k_s)

                Ra_array.append(Ra)
            except:
                pass

        results.append(Ra_array)

    # create figure
    plt.figure()
    for i, config in enumerate(configs):
        plt.plot(imbalance_array[:len(results[i])], results[i], label=str(config[0]) + "x
↪" + str(config[1]))

    # plt.title("Ra for different borefield configurations")
    plt.xlabel("Imbalance (MWh/y)")
    plt.ylabel("Ra (mK/W)")

    plt.legend()
    plt.show()


def figure_7():
    """
```

(continues on next page)

(continued from previous page)

```
    This function creates the seventh figure in the article.
    """

    # initiate borefield
    borefield = Borefield()

    # set the correct sizing method
    borefield.sizing_setup(L2_sizing=True)

    # initiate array with imbalances
    imbalance_array = np.linspace(100, 500, 10)

    # initiate list of borefield configurations
    configs = [(7, 15), (15, 15)]

    # initiate loads
    monthly_load_heating_percentage = np.array([0.155, 0.148, 0.125, .099, .064, 0., 0.,␣
↪0., 0.061, 0.087, .117, 0.144])
    monthly_load_cooling_percentage = np.array([0.025, 0.05, 0.05, .05, .075, .1, .2, .2,
↪ .1, .075, .05, .025])
    monthly_load_heating = monthly_load_heating_percentage * 100 * 10 ** 3  # kWh
    monthly_load_cooling_init = monthly_load_cooling_percentage * 100 * 10 ** 3  # kWh
    peak_cooling_init = np.array([0., 0, 34., 69., 133., 187., 213., 240., 160., 37., 0.,
↪ 0.])  # Peak cooling in kW
    peak_heating = np.array([160., 142, 102., 55., 0., 0., 0., 0., 40.4, 85., 119., 136.
↪])

    # set heating loads
    borefield.set_peak_heating(peak_heating)
    borefield.set_baseload_heating(monthly_load_heating)

    results = []
    for i, config in enumerate(configs):
        Ra_array = []
        for imbalance in imbalance_array:
            # initiate ground data
            ground_data = GroundData(2.4, 10, 0.12)

            # set ground data
            borefield.set_ground_parameters(ground_data)

            # set borefield
            borefield.create_rectangular_borefield(config[0], config[1], 7, 7, 100, 4)

            # calculate loads
            extra_load = imbalance / 12 * 10 ** 3  # kWh
            monthly_load_cooling = monthly_load_cooling_init + extra_load
            peak_cooling = peak_cooling_init + extra_load / 730

            # set cooling loads
            borefield.set_peak_cooling(peak_cooling)
            borefield.set_baseload_cooling(monthly_load_cooling)
```

(continues on next page)

(continued from previous page)

```
            try:
                depth = borefield.size()

                # calculate gfunction
                gfunction = borefield.gfunction(borefield.time, depth)

                # calculate Ra
                Ra = (gfunction[2] - gfunction[1]) / (2 * math.pi * borefield.k_s)

                Ra_array.append(Ra)
            except:
                pass

        results.append(Ra_array)

    # create figure
    plt.figure()
    for i, config in enumerate(configs):
        plt.plot(imbalance_array[:len(results[i])], results[i], label=str(config[0]) + "x
↪" + str(config[1]))

    # plt.title("Ra for different borefield configurations")
    plt.xlabel("Imbalance (MWh/y)")
    plt.ylabel("Ra (mK/W)")

    plt.annotate('', xy=(300, 2.167), xytext=(350, 2.205),
                 arrowprops=dict(arrowstyle='<-', color='black'))
    plt.annotate('', xy=(280, 2.15), xytext=(240, 2.02),
                 arrowprops=dict(arrowstyle='<-', color='black'))

    plt.legend()
    plt.show()


def figure_8():
    """
    This function creates the eight figure of the article.
    """

    # initiate borefield
    borefield1 = Borefield()
    borefield2 = Borefield()

    # set the correct sizing method
    borefield1.sizing_setup(L2_sizing=True)
    borefield2.sizing_setup(L2_sizing=True)

    # initiate array with imbalances percentages
    imbalance_array = np.linspace(30, 70, 20)

    # initiate list of borefield configurations
    configs = [((20, 6), (20, 6)),
```

(continues on next page)

(continued from previous page)

```
                ((18, 8), (16, 6)),
                ((14, 12), (18, 4)),
                ((16, 12), (16, 5))]

    # initiate imbalance
    imbalance = 800

    # initiate loads
    monthly_load_heating_percentage = np.array([0.155, 0.148, 0.125, .099, .064, 0., 0.,␣
↪0., 0.061, 0.087, 0.117, 0.144])
    monthly_load_cooling_percentage = np.array([0.025, 0.05, 0.05, .05, .075, .1, .2, .2,
↪ .1, .075, .05, .025])
    monthly_load_heating = monthly_load_heating_percentage * (100 + imbalance) * 10 ** 3
    monthly_load_cooling = monthly_load_cooling_percentage * 100 * 10 ** 3  # kWh
    peak_cooling = np.array([0., 0., 22., 44., 83., 117., 134., 150., 100., 23., 0., 0.])
    peak_heating = np.array([300., 268., 191., 103., 75., 0., 0., 38., 76., 160., 224.,␣
↪255.])

    results = []
    for i, config_pair in enumerate(configs):
        config1, config2 = config_pair
        ratio_of_nb_of_boreholes = config1[0] * config1[1] / (config2[0] * config2[0] +␣
↪config1[0] * config1[1])

        # initiate ground data
        ground_data1 = GroundData(2.4, 10, 0.14)
        ground_data2 = GroundData(2.4, 10, 0.14)

        # set ground data
        borefield1.set_ground_parameters(ground_data1)
        borefield2.set_ground_parameters(ground_data2)

        # set borefields
        borefield1.create_rectangular_borefield(config1[0], config1[1], 7, 7, 100, 4)
        borefield2.create_rectangular_borefield(config2[0], config2[1], 7, 7, 100, 4)

        # set cooling peak according to the ratio of nb_of_boreholes
        borefield1.set_peak_cooling(peak_cooling * ratio_of_nb_of_boreholes)
        borefield2.set_peak_cooling(peak_cooling * (1-ratio_of_nb_of_boreholes))

        results_temp = []

        for imbalance_percentage in imbalance_array:

            # set the imbalance loads
            borefield1.set_baseload_heating(monthly_load_heating * imbalance_percentage/
↪100)
            borefield2.set_baseload_heating(monthly_load_heating * (100 - imbalance_
↪percentage)/100)

            # set peak load heating
            borefield1.set_peak_heating((peak_heating + imbalance_percentage/100 *␣
```

(continues on next page)

(continued from previous page)

```
↪imbalance / 12 / 730 * 10 ** 3))
            borefield2.set_peak_heating((peak_heating + (100 - imbalance_percentage) /␣
↪100 * imbalance / 12 / 730 * 10 ** 3))

            # set baseload cooling equally over the fields
            borefield1.set_baseload_cooling(monthly_load_cooling * imbalance_percentage/
↪100)
            borefield2.set_baseload_cooling(monthly_load_cooling * (100 - imbalance_
↪percentage)/100)

            try:
                depth1 = borefield1.size()
                depth2 = borefield2.size()

                results_temp.append(depth1 * config1[0] * config1[1] + depth2 *␣
↪config2[0] * config2[1])
            except:
                results_temp.append(0)

        results.append(results_temp)

    # create figure
    plt.figure()
    for i, config in enumerate(configs):
        plt.plot(imbalance_array[:len(results[i])], results[i], label=str(config[0]) + "x
↪" + str(config[1]))

    # plt.title("Effect of imbalance distribution on total borefield length")
    plt.xlabel("Percentage of imbalance on field with largest number of boreholes")
    plt.ylabel("Total borefield length (m)")

    plt.legend()
    plt.show()


if __name__ == "__main__":
    figure_1()
    figure_2()
    figure_3()
    figure_4()
    figure_5()
    figure_7()
    figure_8()
```

References

Coninx, M., De Nies, J., Hermans, L., Peere, W., Boydens, W., Helsen, L. (2024). Cost-efficient cooling of buildings by means of geothermal borefields with active and passive cooling. *Applied Energy*, 355, Art. No. 122261, https://doi.org/10.1016/j.apenergy.2023.122261.

Peere, W., Hermans, L., Boydens, W., and Helsen, L. (2023). Evaluation of the oversizing and computational speed of different open-source borefield sizing methods. In *Proceedings of International Building Simulation Conference 2023*. Shanghai (Belgium), 4-6 September 2023.

Weynjes, J. (2023). Methode voor het dimensioneren van een geothermisch systeem met regeneratie binnen verschillende ESCO-structuren. Master thesis, Department of Mechanical Engineering, KU Leuven, Belgium.

Hermans, L., Haesen, R., Uytterhoeven, A., Peere, W., Boydens, W., Helsen, L. (2023). Pre-design of collective residential solar districts with seasonal thermal energy storage: Importance of level of detail. *Applied thermal engineering* 226, Art.No. 120203, 10.1016/j.applthermaleng.2023.120203

Cimmino, M., Cook., J. C. (2022). pygfunction 2.2 : New Features and Improvements in Accuracy and Computational Efficiency. In *Proceedings of IGSHPA Research Track 2022*. Las Vegas (USA), 6-8 December 2022. https://doi.org/10.22488/okstate.22.000015

Verleyen, L., Peere, W., Michiels, E., Boydens, W., Helsen, L. (2022). The beauty of reason and insight: a story about 30 years old borefield equations. *IEA HPT Magazine 40*(3), 36-39, https://doi.org/10.23697/6q4n-3223

Peere, W., Boydens, W., Helsen, L. (2022). GHEtool: een open-sourcetool voor boorvelddimensionering. Presented at the 15e warmtepompsymposium: van uitdaging naar aanpak, Quadrivium, Heverlee, België.

Peere, W., Coninx, M., De Nies, J., Hermans, L., Boydens, W., Helsen, L. (2022). Cost-efficient Cooling of Buildings by means of Borefields with Active and Passive Cooling. Presented at the 15e warmtepompsymposium: van uitdaging naar aanpak, Quadrivium, Heverlee, België.

Peere, W. (2022). Technologieën voor de energietransitie. Presented at the Energietransitie in meergezinswoningen en kantoorgebouwen: uitdagingen!, VUB Brussel Bruxelles - U Residence.

Peere, W., Blanke, T.(2022). GHEtool: An open-source tool for borefield sizing in Python. *Journal of Open Source Software, 7*(76), 4406, https://doi.org/10.21105/joss.04406

Sharifi., M. (2022). Early-Stage Integrated Design Methods for Hybrid GEOTABS Buildings. PhD thesis, Department of Architecture and Urban Planning, Faculty of Engineering and Architecture, Ghent University.

Coninx M., De Nies J. (2022). Cost-efficient Cooling of Buildings by means of Borefields with Active and Passive Cooling. Master thesis, Department of Mechanical Engineering, KU Leuven, Belgium.

Michiels, E. (2022). Dimensionering van meerdere gekoppelde boorvelden op basis van het type vraagprofiel en de verbinding met de gebruikers. Master thesis, Department of Mechanical Engineering, KU Leuven, Belgium.

Vanpoucke, B. (2022). Optimale dimensionering van boorvelden door een variabel massadebiet. Master thesis, Department of Mechanical Engineering, KU Leuven, Belgium.

Haesen, R., Hermans, L. (2021). Design and Assessment of Low-carbon Residential District Concepts with (Collective) Seasonal Thermal Energy Storage. Master thesis, Departement of Mechanical Engineering, KU Leuven, Belgium.

Peere, W., Picard, D., Cupeiro Figueroa, I., Boydens, W., and Helsen, L. (2021). Validated combined first and last year borefield sizing methodology. In *Proceedings of International Building Simulation Conference 2021*. Brugge (Belgium), 1-3 September 2021. https://doi.org/10.26868/25222708.2021.30180

Peere, W. (2020). Methode voor economische optimalisatie van geothermische verwarmings- en koelsystemen. Master thesis, Department of Mechanical Engineering, KU Leuven, Belgium.

1.2.3 GHEtool's Changelog and future developments

All notable changes to this project will be documented in this file. For future developments, please visit our project board on GitHub.

The format is based on Keep a Changelog.

[2.2.1] - 2024-01-26

Added

- GHEtool is available on conda-forge (issue #107).
- Possibility to start in another month (issue #140).
- Equal functions for HourlyGeothermalLoad and MonthlyGeothermalLoadAbsolute (issue #189).
- Cylindrical borehole correction (issue #187).
- **add** functionality for the load classes (issue #202).

changed

- Negative reference temperatures for the fluid are now possible (issue #192).
- Move code related to the GUI to a separate repo (issue #210).
- Autorelease to PyPi and testPyPi (issue #212).

fixed

- Problem with multiyear hourly data and L3 sizing (issue #153).
- Problem with negative g-function values (issue #187).
- Bug in load-duration curve when not working with optimize load profile (issue #189).
- Bug in hourly data (issue #196).
- Bug in saving after a file has been moved (issue #198).
- Bug in DHW and peak heating power(issue #202).

2.2.0 - 2023-10-17

Added

- Extra warning message if one wants to load a GHEtool file that was created with a newer version.
- Borehole thermal resistance is now visible at the borehole thermal resistance page (issue #51).
- New class of GroundData: GroundTemperatureGradient added (issue #145).
- Load classes (issue #45).

- Pipe classes (single, double, coaxial, Multiple U Tube) (issue #40 and #45).
- Added another methodology for sizing with a variable ground temperature (issue #144).
- Custom error when the field cannot be sized due to a ground temperature gradient (issue #156).
- Interpolation option in calculate function in Gfunction class (issue #159).
- Absolute and relative tolerances for the sizing methods even as a maximum number of iterations is added, so there is more transparency and flexibility in the trade-off between accuracy and speed (issue #159).
- Added advanced options to GHEtool GUI (issue #165).
- Added a result class so all calculated temperatures are now in a separate Result class object within the borefield object (issue #167).
- Added domestic hot water (DHW) to GHEtool (issue #172).
- Glycol-water mixtures can now be selected from within the GUI (issue #174).
- Pygfunction media object can be imported into the FluidData object in GHEtool (issue #174).
- Temperature and flux database (Europe) implemented (issue #178).
- Yearly heating/cooling load in LoadClass (issue #180).

Changed

- GUI was moved to a separate project: ScenarioGUI.
- H_init was removed from the sizing functions since it was not used.
- Rb is now solely handled by the borehole object.
- load_hourly_profile is moved to the separate load classes (issue #45).
- Removed 'set_hourly_cooling_load', 'set_hourly_heating_load' from main_class and move it to separate load class (issue #45).
- Moved draw_borehole_internals to PipeClass (issue #45).
- Borehole equivalent resistances is now calculated in one step, centralised in the pipe class (issue #45).
- Go to 100% code coverage with 350 tests.
- Threshold interpolation for g-functions set to a relative threshold of 25% relative to the demanded depth (issue #144).
- Implemented a custom error for crossing the maximum number of iterations: 'MaximumNumberOfIterations' (issue #144).
- _size_based_on_temperature_profile now returns two arguments: the required depth and a boolean flag to check if the field is sized (issue #144).
- Speed up of L3/L4 sizing by halving calculation time due to intermediate checks if the field is sized (issue #144).
- Changed ValueError when the field cannot be sized due to a temperature gradient to the custom UnsolvableDueToTemperatureGradient Exception (issue #156).
- Rename SizingSetup class to CalculationSetup class (issue #159).
- Move H_init to CalculationSetup class (issue #159).
- Move use_precalcated_data to CalculationSetup class and rename to: 'use_precalculate_dataset' (issue #159).
- Changed 'set_max_ground_temperature' and 'set_min_ground_temperature' to correct names: 'set_max_avg_fluid_temperature' and 'set_min_avg_fluid_temperature'

- Changed ‘minimal average fluid temperature’ to ‘minimum average fluid temperature’ in GUI (issue #172).
- Max value of SEER is now 1000 (issue #178).

Fixed

- Fixed problem with L2 sizing, when the peak load was the same in all months (issue #146).
- Small bug in faster g-function calculation solved. When changing the borefield, the previously calculated g-functions where not removed.
- When using interpolation for the g-functions, the results could vary a little bit based on the previous sizings. By reinstating the H_init parameter, this is solved.
- Borehole internals can no longer overlap in the GUI.
- Optimise load profile crashes with small borefields (issue #180).

2.1.2 - 2023-04-28

Added

- Logger for GHEtool (issue #96).
- Examples are now also in RTD.
- Reynolds number is shown on the result page (issue #112).
- Example for the combination of active and passive cooling (issue #114).
- It is now possible to use building loads (with a SCOP/SEER) instead of ground loads(issue #115).

Changed

- In figure plotting, the interval[x[i], x[i+1]) now has the value y[i] (instead of y[i-1]).
- Scroll behaviour on the result page (issue #99).
- Changed icon of GHEtool.
- Imbalance changed to property so it can handle hourly loads as well (issue #106).
- Remove recalculation option (issue #109).
- When data is loaded in a two-column format, the button for ‘two columns’ is set (issue #133).
- GUI doesn’t crash anymore when wrong seperator and decimal points are selected when loading a .csv.
- One can now use monthly calculations which do not assume equal month length.

Fixed

- Sizing doesn't crash when either no heating or cooling load is present (issue #91).
- Wrong heating load in april in GUI (issue #94).
- Results are now cleared when new loads are loaded (issue #106).
- Options for g-function calculations are not working (issue #119).
- Wrong naming aim optimise load profile.
- GHEtool now can start after a crash without removing the backup file.
- Some translations were not correct.
- Solves issue with loading .csv file and optimise load profile (issue #130).
- Figure in optimize load profile keeps getting bigger and bigger (issue #131).
- Problem with sizing with temperature gradients (issue #136).
- Problem solved with calculate_multiple_scenarios.

2.1.1 - 2023-01-30

Added

- Added NavigationToolbar to figure (issue #55).
- Added different peak lengths for heating and cooling separately (issue #72).
- Readable saving format for gui (JSON).
- A super class that contains functions relevant for all GHEtool classes.
- Exe can be installed either locally for one user without admin permission or for all users using admin permission.
- Saved files (*.GHEtool) can be loaded from GHEtool by double click.

Changed

- Created a class for the custom g-functions (issue #57).
- Created a class for g-function calculation that stores the previously calculated g-values to speed up the iterative algorithms (issue #57).
- Created a class for sizing_setup to clean up the code. The speed improvement is over a factor 10 for heavy iterative procedures (like optimise load profile). A full speed improvement report can be found under: code version > speed improvements > v2.1.1.
- The sizing methods themselves are now faster due to the fact that only the first and last year are calculated (issue #44). For more info, one can check: code version > speed improvements > v2.1.1.
- Faster loading time of the GUI.
- Further documentation for optimise_load_profile functionality.
- Smaller exe-file size by setting up a virtual environment and using a pyinstall folder instead of a single file.

Fixed

- The hourly_heating_load_on_the_borefield and hourly_cooling_load_on_the_borefield are now correctly calculated.
- When an hourly temperature profile is plotted after an optimise_load_profile optimisation, the hourly load on the borefield (and not the entire hourly load) is shown.
- Correct conversion from hourly to monthly load (issue #62).
- Problem with np.float16 when using simulation periodes >80 years due to overflow errors.
- Implemented FIFO-class to prevent cycling in iterative sizing.
- A scenario name cannot occur twice in the scenario list.
- Sometimes some gui options were not shown.
- The drag-and-drop behaviour of the scenario list is fixed (issue #80).
- The renaming of a scenario was not possible (issue #86).
- Problems with borehole internals and pipe overlaps.

2.1.0 - 2022-11-30

Added

- Documentation with ReadTheDocs
- GUI Documentation
- Changelog
- New features in the GUI

Changed

- GUI workflow to be simpler
- precalculated data is removed
- general speed improvements

Removed

- size by length and width for it is not compatible with the just-in-time calculation of the g-functions.

2.0.6 - 2022-10-07

Added

- new functionalities for g-function calculation (inherited from pygfunction) are implemented

Changed

- just-in-time calculation of g-functions is included (and will be expanded later)
- custom borefields can be way faster calculated

Fixed

- Hyperlinks in PyPi should work now
- Sizing by length and width had problems with temperatures below the minimum temperature

2.0.5 - 2022-08-31

Added

- Hourly sizing method (L4) is implemented
- Hourly plotting method
- Volumetric heat capacity is included in the ground data

Changed

- Implemented numpy arrays everywhere
- Implemented convolution instead of matrix multiplication
- New implementation for L3 sizing

Fixed

- No more problems with iteration (L2) and sub 1m depth fields
- Fixed bug in main_functionalities example

Varia

- New validation document for the effective thermal borehole resistance, comparison with EED

2.0.4 - 2022-08-17

Fixed

- Final JOSS paper update

2.0.3 - 2022-08-12

Added

- Variable ground temperature
- Sizing with dynamic Rb*

Fixed

- General bug fixes

Changed

- Sizing setup with more streamlined sizing options

2.0.2 - 2022-06-12

Added

- Included a function (and example) on sizing a borefield by length and width

2.0.1 - 2022-06-12

Added

- Included a pytest document to check if package is correctly installed

2.0.0 - 2022-04-01

Added

- GUI
- Borehole thermal resistance (based on the pygfunction package)

Changed

- More documentation and examples

1.0.1 - 2021-12-11

Changed

- longer simulation period up to 100 years

Fixed

- fixed bug in interpolation

1.2.4 Legal Notice

1.2.5 Installation

Requirements

This code is tested with Python 3.8, 3.9, 3.10, 3.11 and 3.12 and requires the following libraries (the versions mentioned are the ones with which the code is tested)

- matplotlib >= 3.5.2
- numpy >= 1.23.1
- pandas >= 1.4.3
- pygfunction >= 2.2.1
- scipy >= 1.8.1
- scikit-optimize >= 0.9.0

For the tests

- Pytest >= 7.1.2

For the active/passive example

- scikit-optimize >= 0.9.0

Installation

One can install GHEtool by running Pip and running the command

```
pip install GHEtool
```

or one can install a newer development version using

```
pip install --extra-index-url https://test.pypi.org/simple/ GHEtool
```

GHEtool is also available as a conda package. Therefore, you can install GHEtool with the command:

```
conda install GHEtool
```

Developers can clone this repository.

It is a good practise to use virtual environments (venv) when working on a (new) Python project so different Python and package versions don't conflict with eachother. For GHEtool, Python 3.8 or higher is recommended. General information about Python virtual environments can be found here and in this article.

Check installation

To check whether everything is installed correctly, run the following command

```
pytest --pyargs GHEtool
```

This runs some predefined cases to see whether all the internal dependencies work correctly. 9 test should pass successfully.

1.2.6 Get started with GHEtool

Building blocks of GHEtool

GHEtool is a flexible package that can be extend with methods from pygfunction (and ScenarioGUI for the GUI part). To work efficiently with GHEtool, it is important to understand the main structure of the package.

Borefield

The Borefield object is the central object within GHEtool. It is within this object that all the calculations and optimizations take place. All attributes (ground properties, load data ...) are set inside the borefield object.

Ground properties

Within GHEtool, there are multiple ways of setting the ground data. Currently, your options are:

- *GroundConstantTemperature*: if you want to model your borefield with a constant, know ground temperature.
- *GroundFluxTemperature*: if you want to model your ground with a varying ground temperature due to a constant geothermal heat flux.
- *GroundTemperatureGradient*: if you want to model your ground with a varying ground temperature due to a geothermal gradient.

Please note that it is possible to add your own ground types by inheriting the attributes from the abstract _GroundData class.

Pipe data

Within GHEtool, you can use different structures for the borehole internals: U-tubes or coaxial pipes. Concretely, the classes you can use are:

- *Multiple U-tubes*
- *Single U-tubes (special case of multiple U-tubes)*
- *Double U-tubes (special case of multiple U-tubes)*
- *Coaxial pipe*

Please note that it is possible to add your own pipe types by inheriting the attributes from the abstract _PipeData class.

Fluid data

You can set the fluid data by using the FluidData class. In the future, more fluid data classes will be made available.

Load data

One last element which you will need in your calculations, is the load data. Currently, you can only set the primary (i.e. geothermal) load of the borefield. In a future version of GHEtool, also secundary building loads will be included. For now, you can use the following inputs:

- *MonthlyGeothermalLoadAbsolute*: You can set one the monthly baseload and peak load for heating and cooling for one standard year which will be used for all years within the simulation period.
- *HourlyGeothermalLoad*: You can set (or load) the hourly heating and cooling load of a standard year which will be used for all years within the simulation period.
- *HourlyGeothermalLoadMultiYear*: You can set (or load) the hourly heating and cooling load for multiple years (i.e. for the whole simulation period). This way, you can use secundary loads already with GHEtool as shown in this example.

All load classes also have the option to add a yearly domestic hot water usage.

Please note that it is possible to add your own load types by inheriting the attributes from the abstract _LoadData class.

Simple example

To show how all the pieces of GHEtool work together, below you can find a step-by-step example of how, traditionally, one would work with GHEtool. Start by importing all the relevant classes. In this case we are going to work with a ground model which assumes a constant ground temperature (e.g. from a TRT-test), and we will provide the load with a monthly resolution.

```
from GHEtool import Borefield, GroundDataConstantTemperature,␣
↪MonthlyGeothermalLoadAbsolute
```

After importing the necessary classes, the relevant ground data parameters are set.

```
data =
GroundDataConstantTemperature(3,   # ground thermal conductivity (W/mK)
                                                        10,  # initial/undisturbed␣
↪ground temperature (deg C)
                              2.4*10**6) # volumetric heat capacity of the ground (J/
↪m3K)
```

Furthermore, for our loads, we need to set the peak loads as well as the monthly base loads for heating and cooling.

```
peak_cooling = [0., 0, 34., 69., 133., 187., 213., 240., 160., 37., 0., 0.]   # Peak␣
↪cooling in kW
peak_heating = [160., 142, 102., 55., 0., 0., 0., 0., 40.4, 85., 119., 136.]  # Peak␣
↪heating in kW

monthly_load_heating = [46500.0, 44400.0, 37500.0, 29700.0, 19200.0, 0.0, 0.0, 0.0,␣
↪18300.0, 26100.0, 35100.0, 43200.0]        # in kWh
monthly_load_cooling = [4000.0, 8000.0, 8000.0, 8000.0, 12000.0, 16000.0, 32000.0, 32000.
↪0, 16000.0, 12000.0, 8000.0, 4000.0]  # in kWh

# set load object
load = MonthlyGeothermalLoadAbsolute(monthly_load_heating, monthly_load_cooling, peak_
↪heating, peak_cooling)
```

Next, we create the borefield object in GHEtool and set the temperature constraints and the ground data. Here, since we do not use a pipe and fluid model (see Examples if you need examples were no borehole thermal resistance is given), we set the borehole equivalent thermal resistance.

```
# create the borefield object
borefield = Borefield(load=load
peak_heating = peak_heating,
peak_cooling = peak_cooling,
baseload_heating = monthly_load_heating,
baseload_cooling = monthly_load_cooling)

# set ground parameters
borefield.set_ground_parameters(data)

# set the borehole equivalent resistance
borefield.Rb = 0.12

# set temperature boundaries
borefield.set_max_avg_fluid_temperature(16)  # maximum temperature
borefield.set_min_avg_fluid_temperature(0)  # minimum temperature
```

Next we create a rectangular borefield.

```
# set a rectangular borefield
borefield.create_rectangular_borefield(10, 12, 6, 6, 110, 4, 0.075)
```

Note that the borefield can also be set using the pygfunction package, if you want more complex designs.

```
import pygfunction as gt

```

(continues on next page)

(continued from previous page)

```
# set a rectangular borefield
borefield_gt = gt.boreholes.rectangle_field(10, 12, 6, 6, 110, 1, 0.075)
borefield.set_borefield(borefield_gt)
```

Once a Borefield object is created, one can make use of all the functionalities of GHEtool. One can for example size the borefield using:

```
depth = borefield.size()
print("The borehole depth is: ", depth, "m")
```

Or one can plot the temperature profile by using

```
borefield.print_temperature_profile(legend=True)
```

1.2.7 Functionalities of GHEtool

GHEtool has a couple of different functionalities, all centered around borefield sizing. In the examples listed below, one can find example code on how to use the different functionalities.

Sizing the borefield (i.e. calculating the required depth) for a given injection and extraction load for the borefield (three sizing methods are available).

Main functionalities of GHEtool

```
"""
This file contains all the main functionalities of GHEtool being:
    * sizing of the borefield
    * sizing of the borefield for a specific quadrant
    * plotting the temperature evolution
    * plotting the temperature evolution for a specific depth
    * printing the array of the temperature
"""

import numpy as np

# import all the relevant functions
from GHEtool import Borefield, FluidData, DoubleUTube, GroundConstantTemperature,␣
↪MonthlyGeothermalLoadAbsolute


def main_functionalities():
    # relevant borefield data for the calculations
    data = GroundConstantTemperature(3,             # conductivity of the soil (W/mK)
                                     10,            # Ground temperature at infinity␣
↪(degrees C)
                                     2.4 * 10**6)   # ground volumetric heat capacity (J/
↪m3K)

    # monthly loading values
    peak_cooling = np.array([0., 0, 34., 69., 133., 187., 213., 240., 160., 37., 0., 0.
```

(continues on next page)

(continued from previous page)

```
↪])  # Peak cooling in kW
    peak_heating = np.array([160., 142, 102., 55., 0., 0., 0., 0., 40.4, 85., 119., 136.
↪])  # Peak heating in kW

    # annual heating and cooling load
    annual_heating_load = 300 * 10 ** 3  # kWh
    annual_cooling_load = 160 * 10 ** 3  # kWh

    # percentage of annual load per month (15.5% for January ...)
    monthly_load_heating_percentage = np.array([0.155, 0.148, 0.125, .099, .064, 0., 0.,␣
↪0., 0.061, 0.087, 0.117, 0.144])
    monthly_load_cooling_percentage = np.array([0.025, 0.05, 0.05, .05, .075, .1, .2, .2,
↪ .1, .075, .05, .025])

    # resulting load per month
    monthly_load_heating = annual_heating_load * monthly_load_heating_percentage   # kWh
    monthly_load_cooling = annual_cooling_load * monthly_load_cooling_percentage   # kWh

    # set the load
    load = MonthlyGeothermalLoadAbsolute(monthly_load_heating, monthly_load_cooling,␣
↪peak_heating, peak_cooling)

    # create the borefield object
    borefield = Borefield(load=load)

    # one can activate or deactive the logger, by default it is deactivated
    # borefield.activate_logger()
    # borefield.deactivate_logger()

    borefield.set_ground_parameters(data)
    borefield.create_rectangular_borefield(10, 12, 6, 6, 100, 4, 0.075)

    borefield.Rb = 0.12  # equivalent borehole resistance (K/W)

    # set temperature boundaries
    borefield.set_max_avg_fluid_temperature(16)   # maximum temperature
    borefield.set_min_avg_fluid_temperature(0)    # minimum temperature

    # size borefield
    depth = borefield.size()
    print("The borehole depth is: ", depth, "m")

    # print imbalance
    print("The borefield imbalance is: ", borefield._borefield_load.imbalance, "kWh/y.␣
↪(A negative imbalance means the the field is heat extraction dominated so it cools␣
↪down year after year.)") # print imbalance

    # plot temperature profile for the calculated depth
    borefield.print_temperature_profile(legend=True)

    # plot temperature profile for a fixed depth
    borefield.print_temperature_profile_fixed_depth(depth=75, legend=False)
```

(continues on next page)

(continued from previous page)

```

    # print gives the array of monthly temperatures for peak cooling without showing the plot
    borefield.calculate_temperatures(depth=90)
    print("Result array for cooling peaks")
    print(borefield.results.peak_cooling)
    print("-----------------------------------------------")

    # size the borefield for quadrant 3
    # for more information about borefield quadrants, see (Peere et al., 2021)
    depth = borefield.size(quadrant_sizing=3)
    print("The borehole depth is: ", str(round(depth, 2)), "m for a sizing in quadrant 3")
    # plot temperature profile for the calculated depth
    borefield.print_temperature_profile(legend=True)

    # size with a dynamic Rb* value
    # note that the original Rb* value will be overwritten!

    # this requires pipe and fluid data
    fluid_data = FluidData(0.2, 0.568, 998, 4180, 1e-3)
    pipe_data = DoubleUTube(1, 0.015, 0.02, 0.4, 0.05)
    borefield.set_fluid_parameters(fluid_data)
    borefield.set_pipe_parameters(pipe_data)

    # disable the use of constant_Rb with the setup, in order to plot the profile correctly
    # when it is given as an argument to the size function, it will size correctly, but the plot will be with
    # constant Rb* since it has not been changed in the setup function
    borefield.calculation_setup(use_constant_Rb=False)
    depth = borefield.size()
    print("The borehole depth is: ", str(round(depth, 2)), "m for a sizing with dynamic Rb*.")
    borefield.print_temperature_profile(legend=True)


if __name__ == "__main__":  # pragma: no cover
    main_functionalities()
```

Using dynamically calculated borehole thermal resistance (this is directly based on the code of pygfunction).

Sizing with equivalent borehole resistance calculation

```
"""
This document compares the sizing with a constant Rb*-value with sizing where the Rb*-value is being recalculated.
For the test, the L2 sizing method is used.
The comparison is based on speed and relative accuracy in the result.
It is shown that the speed difference is significant, but so is the difference in the result. With a constant Rb* value, it is important that the initial depth is rather
```

(continues on next page)

(continued from previous page)

```
    accurate.
"""

import time

import numpy as np
import pygfunction as gt

from GHEtool import Borefield, FluidData, GroundConstantTemperature, DoubleUTube, MonthlyGeothermalLoadAbsolute


def sizing_with_Rb():
    number_of_iterations = 50
    max_value_cooling = 700
    max_value_heating = 800

    # initiate the arrays
    results_Rb_static = np.empty(number_of_iterations)
    results_Rb_dynamic = np.empty(number_of_iterations)
    difference_results = np.empty(number_of_iterations)

    monthly_load_cooling_array = np.empty((number_of_iterations, 12))
    monthly_load_heating_array = np.empty((number_of_iterations, 12))
    peak_load_cooling_array = np.empty((number_of_iterations, 12))
    peak_load_heating_array = np.empty((number_of_iterations, 12))

    # populate arrays with random values
    for i in range(number_of_iterations):
        for j in range(12):
            monthly_load_cooling_array[i, j] = np.random.randint(0, max_value_cooling)
            monthly_load_heating_array[i, j] = np.random.randint(0, max_value_heating)
            peak_load_cooling_array[i, j] = np.random.randint(monthly_load_cooling_array[i, j], max_value_cooling)
            peak_load_heating_array[i, j] = np.random.randint(monthly_load_heating_array[i, j], max_value_heating)

    # initiate borefield model
    data = GroundConstantTemperature(3, 10)  # ground data with an inaccurate guess of 100m for the depth of the borefield
    fluid_data = FluidData(0.2, 0.568, 998, 4180, 1e-3)
    pipe_data = DoubleUTube(1, 0.015, 0.02, 0.4, 0.05)

    borefield_gt = gt.boreholes.rectangle_field(10, 12, 6, 6, 100, 1, 0.075)

    # Monthly loading values
    peak_cooling = np.array([0., 0, 34., 69., 133., 187., 213., 240., 160., 37., 0., 0.])  # Peak cooling in kW
    peak_heating = np.array([160., 142, 102., 55., 0., 0., 0., 0., 40.4, 85., 119., 136.])  # Peak heating in kW

    # annual heating and cooling load
```

(continues on next page)

(continued from previous page)

```
    annual_heating_load = 300 * 10 ** 3  # kWh
    annual_cooling_load = 160 * 10 ** 3  # kWh

    # percentage of annual load per month (15.5% for January ...)
    monthly_load_heating_percentage = np.array([0.155, 0.148, 0.125, .099, .064, 0., 0.,␣
↪0., 0.061, 0.087, 0.117, 0.144])
    monthly_load_cooling_percentage = np.array([0.025, 0.05, 0.05, .05, .075, .1, .2, .2,
↪ .1, .075, .05, .025])

    # resulting load per month
    monthly_load_heating = annual_heating_load * monthly_load_heating_percentage   # kWh
    monthly_load_cooling = annual_cooling_load * monthly_load_cooling_percentage   # kWh

    # set the load
    load = MonthlyGeothermalLoadAbsolute(monthly_load_heating, monthly_load_cooling,␣
↪peak_heating, peak_cooling)

    # create the borefield object
    borefield = Borefield(load=load)

    borefield.set_ground_parameters(data)
    borefield.set_fluid_parameters(fluid_data)
    borefield.set_pipe_parameters(pipe_data)
    borefield.Rb = 0.2
    borefield.set_borefield(borefield_gt)

    # create custom gfunction to speed up the calculation
    borefield.create_custom_dataset()

    # set temperature boundaries
    borefield.set_max_avg_fluid_temperature(16)   # maximum temperature
    borefield.set_min_avg_fluid_temperature(0)    # minimum temperature

    # size with constant Rb* value
    borefield.calculation_setup(use_constant_Rb=True)

    # calculate the Rb* value
    borefield.Rb = borefield.borehole.calculate_Rb(100, 1, 0.075, data.k_s)

    start_Rb_constant = time.time()
    for i in range(number_of_iterations):
        # set the load
        load = MonthlyGeothermalLoadAbsolute(monthly_load_heating_array[i], monthly_load_
↪cooling_array[i],
                                             peak_load_heating_array[i], peak_load_
↪cooling_array[i])
        borefield.load = load
        results_Rb_static[i] = borefield.size()
    end_Rb_constant = time.time()

    # size with a dynamic Rb* value
    borefield.calculation_setup(use_constant_Rb=False)
```

(continues on next page)

(continued from previous page)

```

    start_Rb_dynamic = time.time()
    for i in range(number_of_iterations):
        # set the load
        load = MonthlyGeothermalLoadAbsolute(monthly_load_heating_array[i], monthly_load_
↪cooling_array[i],
                                             peak_load_heating_array[i], peak_load_
↪cooling_array[i])
        borefield.load = load
        results_Rb_dynamic[i] = borefield.size()
    end_Rb_dynamic = time.time()
    print(results_Rb_dynamic[1])

    print("These are the results when an inaccurate constant Rb* value is used.")
    print("Time for sizing with a constant Rb* value:", end_Rb_constant - start_Rb_
↪constant, "s")
    print("Time for sizing with a dynamic Rb* value:", end_Rb_dynamic - start_Rb_dynamic,
↪ "s")

    # calculate differences
    for i in range(number_of_iterations):
        difference_results[i] = results_Rb_dynamic[i] - results_Rb_static[i]

    print("The maximal difference between the sizing with a constant and a dynamic Rb*␣
↪value:", np.round(np.max(difference_results), 3), "m or", np.round(np.max(difference_
↪results) / results_Rb_static[np.argmax(difference_results)] * 100, 3), "% w.r.t. the␣
↪constant Rb* approach.")
    print("The mean difference between the sizing with a constant and a dynamic Rb*␣
↪value:", np.round(np.mean(difference_results), 3), "m or", np.round(np.mean(difference_
↪results) / np.mean(results_Rb_static) * 100, 3), "% w.r.t. the constant Rb* approach.")
    print("-------------------------------------------------------------------------------
↪")

    # Do the same thing but with another constant Rb* value based on a borehole depth of␣
↪185m.

    borefield_gt = gt.boreholes.rectangle_field(10, 12, 6, 6, 185, 1, 0.075) # borefield␣
↪with an accurate guess of 185m for the depth of the borefield
    borefield.set_borefield(borefield_gt)

    # size with a constant Rb* value
    borefield.calculation_setup(use_constant_Rb=True)

    # calculate the Rb* value
    borefield.Rb = borefield.borehole.calculate_Rb(100, 1, 0.075, data.k_s)

    start_Rb_constant = time.time()
    for i in range(number_of_iterations):
        # set the load
        load = MonthlyGeothermalLoadAbsolute(monthly_load_heating_array[i], monthly_load_
↪cooling_array[i],
                                             peak_load_heating_array[i], peak_load_
```

(continues on next page)

(continued from previous page)

```
↪cooling_array[i])
        borefield.load = load
        results_Rb_static[i] = borefield.size()
    end_Rb_constant = time.time()

    # size with a dynamic Rb* value
    borefield.calculation_setup(use_constant_Rb=False)

    start_Rb_dynamic = time.time()
    for i in range(number_of_iterations):
        # set the load
        load = MonthlyGeothermalLoadAbsolute(monthly_load_heating_array[i], monthly_load_
↪cooling_array[i],
                                             peak_load_heating_array[i], peak_load_
↪cooling_array[i])
        borefield.load = load
        results_Rb_dynamic[i] = borefield.size()
    end_Rb_dynamic = time.time()

    print("These are the results when an accurate constant Rb* value is used.")
    print("Time for sizing with a constant Rb* value:", end_Rb_constant - start_Rb_
↪constant, "s")
    print("Time for sizing with a dynamic Rb* value:", end_Rb_dynamic - start_Rb_dynamic,
↪ "s")

    # calculate differences
    for i in range(number_of_iterations):
        difference_results[i] = results_Rb_dynamic[i] - results_Rb_static[i]

    print("The maximal difference between the sizing with a constant and a dynamic Rb*␣
↪value:",
          np.round(np.max(difference_results), 3), "m or",
          np.round(np.max(difference_results) / results_Rb_static[np.argmax(difference_
↪results)] * 100, 3),
          "% w.r.t. the constant Rb* approach.")
    print("The mean difference between the sizing with a constant and a dynamic Rb*␣
↪value:",
          np.round(np.mean(difference_results), 3), "m or",
          np.round(np.mean(difference_results) / np.mean(results_Rb_static) * 100, 3),
          "% w.r.t. the constant Rb* approach.")


if __name__ == "__main__":  # pragma: no cover
    sizing_with_Rb()
```

Optimising the load profile for a given heating and cooling load.

Optimise load profile

```
"""
This document is an example of load optimisation.
First an hourly profile is imported and a fixed borefield size is set.
Then, based on a load-duration curve, the heating and cooling load is altered in order to fit as much load as possible on the field.
The results are returned.

"""
import numpy as np

# import all the relevant functions
from GHEtool import *


def optimise():

    # initiate ground data
    data = GroundConstantTemperature(3, 10)

    # initiate borefield
    borefield = Borefield()

    # set ground data in borefield
    borefield.set_ground_parameters(data)

    # set Rb
    borefield.Rb = 0.12

    # set borefield
    borefield.create_rectangular_borefield(10, 10, 6, 6, 110, 1, 0.075)

    # load the hourly profile
    load = HourlyGeothermalLoad()
    load.load_hourly_profile("hourly_profile.csv", header=True, separator=";")

    # optimise the load for a 10x10 field (see data above) and a fixed depth of 150m.
    borefield.optimise_load_profile(building_load=load, depth=150, print_results=True)

    # calculate temperatures
    borefield.calculate_temperatures(hourly=True)

    # print resulting external peak cooling profile
    print(borefield._external_load.max_peak_cooling)

    # print resulting monthly load for an external heating source
    print(np.sum(borefield._external_load.hourly_heating_load))


if __name__ == "__main__":  # pragma: no cover
    optimise()
```

Importing heating and cooling loads from .csv files.

Import data

```
"""
This document is an example on how to import hourly load profiles into GHEtool.
It uses the hourly_profile.csv data.
"""
import pygfunction as gt

# import all the relevant functions
from GHEtool import *

# initiate ground data
data = GroundConstantTemperature(3, 10, 2.4*10**6)
borefield_gt = gt.boreholes.rectangle_field(10, 12, 6, 6, 110, 1, 0.075)

# initiate borefield
borefield = Borefield()

# set ground data in borefield
borefield.set_ground_parameters(data)

# set Rb
borefield.Rb = 0.12

# set borefield
borefield.set_borefield(borefield_gt)

# load the hourly profile
load = HourlyGeothermalLoad()
load.load_hourly_profile("hourly_profile.csv", header=True, separator=";")
borefield.load = load

# size the borefield and plot the resulting temperature evolution
depth = borefield.size(100, L2_sizing=True)
print(depth)
borefield.print_temperature_profile()
```

Using your custom borefield configurations.

Custom borefield configuration

```
"""
This file gives an example on how to work with a custom borefield within GHEtool using␣
↪pygfunction.

When working on a custom borefield configuration, one needs to import this configuration␣
↪into the GHEtool.
Based on the pygfunction, one creates his custom borefield and gives it as an argument␣
↪to the class initiater Borefield of GHEtool.

You also need a custom g-function file for interpolation. This can also be given as an␣
↪argument to the class initiater as _custom_gfunction.
```

(continues on next page)

(continued from previous page)

```
This custom variable, must contain gfunctions for all time steps in Borefield.DEFAULT_TIME_ARRAY, and should be structured as follows:
{"Time":Borefield.DEFAULT_TIME_ARRAY,"Data":[[Depth1,[Gfunc1,Gfunc2 ...]],[Depth2,[Gfunc1, Gfunc2 ...]]]}.

However, one can use the function 'create_custom_dataset' when a custom borefield is given. This will make the required dataset for the optimisation.
 """

import numpy as np
import pygfunction as gt

# import all the relevant functions
from GHEtool import *


def custom_borefield_configuration():
    # set the relevant ground data for the calculations
    data = GroundConstantTemperature(3, 10)

    # Monthly loading values
    peak_cooling = np.array([0., 0, 3.4, 6.9, 13., 18., 21., 50., 16., 3.7, 0., 0.])  # Peak cooling in kW
    peak_heating = np.array([60., 42., 10., 5., 0., 0., 0., 0., 4.4, 8.5, 19., 36.])  # Peak heating in kW

    # annual heating and cooling load
    annual_heating_load = 30 * 10 ** 3  # kWh
    annual_cooling_load = 16 * 10 ** 3  # kWh

    # percentage of annual load per month (15.5% for January ...)
    monthly_load_heating_percentage = np.array([0.155, 0.148, 0.125, .099, .064, 0., 0., 0., 0.061, 0.087, 0.117, 0.144])
    monthly_load_cooling_percentage = np.array([0.025, 0.05, 0.05, .05, .075, .1, .2, .2, .1, .075, .05, .025])

    # resulting load per month
    monthly_load_heating = annual_heating_load * monthly_load_heating_percentage  # kWh
    monthly_load_cooling = annual_cooling_load * monthly_load_cooling_percentage  # kWh

    # set the load
    load = MonthlyGeothermalLoadAbsolute(monthly_load_heating, monthly_load_cooling, peak_heating, peak_cooling)

    # create the borefield object
    borefield = Borefield(load=load)

    borefield.set_ground_parameters(data)
    borefield.Rb = 0.2

    # set temperature boundaries
    borefield.set_max_avg_fluid_temperature(16)  # maximum temperature
```

(continues on next page)

(continued from previous page)

```
    borefield.set_min_avg_fluid_temperature(0)  # minimum temperature

    # create custom borefield based on pygfunction
    custom_field = gt.boreholes.L_shaped_field(N_1=4, N_2=5, B_1=5., B_2=5., H=100., D=4,
↪ r_b=0.05)

    # set the custom borefield (so the number of boreholes is correct)
    borefield.set_borefield(custom_field)
    borefield.create_custom_dataset()

    # size borefield
    depth = borefield.size()
    print("The borehole depth is: ", depth, "m")

    # print imbalance
    print("The borefield imbalance is: ", borefield.load.imbalance, "kWh/y. (A negative␣
↪imbalance means the the field is heat extraction dominated so it cools down year after␣
↪year.)") # print imbalance

    # plot temperature profile for the calculated depth
    borefield.print_temperature_profile(legend=True)

    # plot temperature profile for a fixed depth
    borefield.print_temperature_profile_fixed_depth(depth=75, legend=False)

    # print gives the array of monthly temperatures for peak cooling without showing the␣
↪plot
    borefield.calculate_temperatures(depth=90)
    print("Result array for cooling peaks")
    print(borefield.results.peak_cooling)


if __name__ == "__main__":  # pragma: no cover
    custom_borefield_configuration()
```

1.2.8 Modules

Main class

This file contains all the code for the borefield calculations.

class GHEtool.main_class.**Borefield**(*peak_heating: Optional[Union[ndarray, list]] = None*, *peak_cooling: Optional[Union[ndarray, list]] = None*, *baseload_heating: Optional[Union[ndarray, list]] = None*, *baseload_cooling: Optional[Union[ndarray, list]] = None*, *borefield=None*, *custom_gfunction: Optional[CustomGFunction] = None*, *load: Optional[_LoadData] = None*)

Bases: *BaseClass*

Main borefield class

Parameters

peak_heating
[list, numpy array] Monthly peak heating values [kW]

peak_cooling
[list, numpy array] Monthly peak cooling values [kW]

baseload_heating
[list, numpy array] Monthly baseload heating values [kWh]

baseload_cooling
[list, numpy array] Monthly baseload heating values [kWh]

borefield
[pygfunction borehole/borefield object] Set the borefield for which the calculations will be carried out

custom_gfunction
[CustomGFunction] Custom gfunction dataset

Examples

monthly peak values [kW]

```
>>> peak_cooling = np.array([0., 0, 34., 69., 133., 187., 213., 240., 160., 37., 0.,
↪ 0.])
>>> peak_heating = np.array([160., 142, 102., 55., 0., 0., 0., 0., 40.4, 85., 119.,␣
↪136.])
```

annual heating and cooling load [kWh]

```
>>> annual_heating_load = 300 * 10 ** 3
>>> annual_cooling_load = 160 * 10 ** 3
```

percentage of annual load per month (15.5% for January ...)

```
>>> monthly_load_heating_percentage = np.array([0.155, 0.148, 0.125, .099, .064, 0.,
↪ 0., 0., 0.061, 0.087, 0.117, 0.144])
>>> monthly_load_cooling_percentage = np.array([0.025, 0.05, 0.05, .05, .075, .1, .
↪2, .2, .1, .075, .05, .025])
```

resulting load per month [kWh]

```
>>> monthly_load_heating = annual_heating_load * monthly_load_heating_percentage
>>> monthly_load_cooling = annual_cooling_load * monthly_load_cooling_percentage
```

create the borefield object

```
>>> borefield = Borefield()
```

set the load

```
>>> load = MonthlyGeothermalLoadAbsolute(monthly_load_heating, monthly_load_cooling,
↪ peak_heating, peak_cooling)
>>> borefield.load = load
```

property Rb: float

This function returns the equivalent borehole thermal resistance.

Returns

Rb
[float] Equivalent borehole thermal resistance [mK/W]

property Re: float

Reynolds number.

Returns

float
Reynolds number

static activate_logger() → None

This function activates the logging.

Returns

None

property borefield

Returns the hidden _borefield variable.

Returns

Hidden _borefield object

calculate_next_depth_deep_sizing(*current_depth: float*) → float

This method is a slower but more robust way of calculating the next depth in the sizing iteration when the borefield is sized for the maximum fluid temperature when there is a non-constant ground temperature. The method is based (as can be seen in its corresponding validation document) on the assumption that the difference between the maximum temperature in peak cooling and the average undisturbed ground temperature is irreversibly proportional to the depth. In this way, given this difference in temperature and the current depth, a new depth can be calculated.

Parameters

current_depth
[float] The current depth of the borefield [m]

Returns

float
New depth of the borefield [m]

calculate_quadrant() → int

This function returns the borefield quadrant (as defined by Peere et al., 2021[1]) based on the calculated temperature profile. If there is no limiting quadrant, None is returned.

Quadrant 1 is limited in the first year by the maximum temperature

Quadrant 2 is limited in the last year by the maximum temperature

Quadrant 3 is limited in the first year by the minimum temperature

Quadrant 4 is limited in the last year by the maximum temperature

[1] Peere, W., Picard, D., Cupeiro Figueroa, I., Boydens, W., and Helsen, L. (2021) Validated combined first and last year borefield sizing methodology. In Proceedings of International Building Simulation Conference 2021. Brugge (Belgium), 1-3 September 2021. https://doi.org/10.26868/25222708.2021.30180

Returns

quadrant
[int] The quadrant which limits the borefield

References

calculate_temperatures(*depth: Optional[float] = None*, *hourly: bool = False*) → None

Calculate all the temperatures without plotting the figure. When depth is given, it calculates it for a given depth.

Parameters

depth
[float] Depth for which the temperature profile should be calculated for [m]

hourly
[bool] True when the temperatures should be calculated based on hourly data

Returns

None

calculation_setup(*calculation_setup: Optional[CalculationSetup] = None*, *use_constant_Rb: Optional[bool] = None*, ***kwargs*) → None

This function sets the options for the sizing function.

- The L2 sizing is the one explained in (Peere et al., 2021)[2] and is the quickest method (it uses 3 pulses)
- The L3 sizing is a more general approach which is slower but more accurate (it uses 24 pulses/year)
- The L4 sizing is the most exact one, since it uses hourly data (8760 pulses/year)

Parameters

calculation_setup
[CalculationSetup] An instance of the CalculationSetup class. When this argument differs from None, all the other parameters are set based on this calculation_setup

use_constant_Rb
[bool] True if a constant borehole equivalent resistance (Rb*) value should be used

kwargs
Dictionary with all the other options that can be set within GHEtool. For a complete list, see the documentation in the CalculationSetup class.

Returns

None

[2] Peere, W., Picard, D., Cupeiro Figueroa, I., Boydens, W., and Helsen, L. (2021) Validated combined first and last year borefield sizing methodology. In Proceedings of International Building Simulation Conference 2021. Brugge (Belgium), 1-3 September 2021. https://doi.org/10.26868/25222708.2021.30180

References

create_circular_borefield(*N: int*, *R: float*, *H: float*, *D: float = 1*, *r_b: float = 0.075*)

This function creates a circular borefield. It calls the pygfunction module in the background. The documentation of this function is based on pygfunction.

Parameters

N
[int] Number of boreholes in the borefield

R
[float] Distance of boreholes from the center of the field

H
[float] Borehole depth [m]

D
[float] Borehole buried depth [m]

r_b
[float] Borehole radius [m]

Returns

pygfunction borefield object

create_custom_dataset(*time_array: Optional[Union[ndarray, list]] = None*, *depth_array: Optional[Union[ndarray, list]] = None*, *options: dict = {}*) → None

This function makes a datafile for a given custom borefield and sets it for the borefield object. It automatically sets this datafile in the current borefield object so it can be used as a source for the interpolation of g-values.

Parameters

time_array
[list, np.array] Time values (in seconds) used for the calculation of the datafile

depth_array
[list, np.array] List or arrays of depths for which the datafile should be created

options
[dict] Options for the g-function calculation (check pygfunction.gfunction.gFunction() for more information)

Returns

None

Raises

ValueError
When no borefield or ground data is set

create_rectangular_borefield(*N_1: int*, *N_2: int*, *B_1: int*, *B_2: int*, *H: float*, *D: float = 1*, *r_b: float = 0.075*)

This function creates a rectangular borefield. It calls the pygfunction module in the background. The documentation of this function is based on pygfunction.

Parameters

N_1
[int] Number of boreholes in the x direction

N_2
[int] Number of boreholes in the y direction

B_1
[int] Distance between adjacent boreholes in the x direction [m]

B_2
[int] Distance between adjacent boreholes in the y direction [m]

H
[float] Borehole depth [m]

D
[float] Borehole buried depth [m]

r_b
[float] Borehole radius [m]

Returns

pygfunction borefield object

`static deactivate_logger()` → None

This function deactivates the logging.

Returns

None

`gfunction(time_value: list | float | numpy.ndarray, H: Optional[float] = None)` → ndarray

This function returns the gfunction value. It can do so by either calculating the gfunctions just-in-time or by interpolating from a loaded custom data file.

Parameters

time_value
[list, float, np.ndarray] Time value(s) in seconds at which the gfunctions should be calculated

H
[float] Depth [m] at which the gfunctions should be calculated. If no depth is given, the current depth is taken.

Returns

gvalue
[np.ndarray] 1D array with the g-values for all the requested time_value(s)

`property ground_data: _GroundData`

" This function returns the ground data.

Returns

ground data
[GroundData]

`property investment_cost: float`

This function calculates the investment cost based on a cost profile linear to the total borehole length.

Returns

float
Investment cost

property load: *GHEtool.VariableClasses.LoadData._LoadData._LoadData* | **GHEtool.VariableClasses.LoadData.GeothermalLoad.HourlyGeothermalLoad.HourlyGeothermalLoad** | **GHEtool.VariableClasses.LoadData.GeothermalLoad.MonthlyGeothermalLoadAbsolute.MonthlyGeothermalLoadAbsolute**

This returns the LoadData object.

Returns

Load data: LoadData

load_custom_gfunction(*location: str*) → None

This function loads the custom gfunction.

Parameters

location
[str] Path to the location of the custom gfunction file

Returns

None

optimise_load_profile(*building_load: HourlyGeothermalLoad*, *depth: Optional[float] = None*, *SCOP: float = 1000000*, *SEER: float = 1000000*, *print_results: bool = False*, *temperature_threshold: float = 0.05*) → None

This function optimises the load based on the given borefield and the given hourly load. (When the load is not geothermal, the SCOP and SEER are used to convert it to a geothermal load.) It does so based on a load-duration curve. The temperatures of the borefield are calculated on a monthly basis, even though we have hourly data, for an hourly calculation of the temperatures would take a very long time.

Parameters

building_load
[_LoadData] Load data used for the optimisation

depth
[float] Depth of the boreholes in the borefield [m]

SCOP
[float] SCOP of the geothermal system (needed to convert hourly building load to geothermal load)

SEER
[float] SEER of the geothermal system (needed to convert hourly building load to geothermal load)

print_results
[bool] True when the results of this optimisation are to be printed in the terminal

temperature_threshold
[float] The maximum allowed temperature difference between the maximum and minimum fluid temperatures and their respective limits. The lower this threshold, the longer the convergence will take.

Returns

None

Raises

ValueError
ValueError if no hourly load is given or the threshold is negative

`plot_load_duration`(*legend: bool = False*) → Tuple[Figure, Axes]

This function makes a load-duration curve from the hourly values.

Parameters

legend
[bool] True if the figure should have a legend

Returns

Tuple
plt.Figure, plt.Axes

`print_temperature_profile`(*legend: bool = True*, *plot_hourly: bool = False*) → None

This function plots the temperature profile for the calculated depth. It uses the available temperature profile data.

Parameters

legend
[bool] True if the legend should be printed

plot_hourly
[bool] True if the temperature profile printed should be based on the hourly load profile.

Returns

fig, ax
Figure object

`print_temperature_profile_fixed_depth`(*depth: float*, *legend: bool = True*, *plot_hourly: bool = False*)

This function plots the temperature profile for a fixed depth. It uses the already calculated temperature profile data, if available.

Parameters

depth
[float] Depth at which the temperature profile should be shown

legend
[bool] True if the legend should be printed

plot_hourly
[bool] True if the temperature profile printed should be based on the hourly load profile.

Returns

fig, ax
Figure object

`set_Rb`(*Rb: float*) → None

This function sets the constant equivalent borehole thermal resistance.

Parameters

Rb
[float] Equivalent borehole thermal resistance (mK/W)

Returns

None

set_borefield(*borefield: Optional[list[pygfunction.boreholes.Borehole]] = None*) → None

This function set the borefield object. When None is given, the borefield will be deleted.

Parameters

borefield
[List[pygfunction.boreholes.Borehole]] Borefield created with the pygfunction package

Returns

None

set_fluid_parameters(*data:* FluidData) → None

This function sets the fluid parameters.

Parameters

data
[FluidData] All the relevant fluid data

Returns

None

set_ground_parameters(*data:* _GroundData) → None

This function sets the relevant ground parameters.

Parameters

data
[GroundData] All the relevant ground data

Returns

None

set_investment_cost(*investment_cost: Optional[list] = None*) → None

This function sets the investment cost. This is linear with respect to the total field length. If None, the default is set.

Parameters

investment_cost
[list] 1D array of polynomial coefficients (including coefficients equal to zero) from highest degree to the constant term

Returns

None

set_length_peak(*length: float = 6*) → None

This function sets the length of the peak.

Parameters

length
[float] Length of the peak [hours]

Returns

None

set_load(*load: _LoadData*) → None

This function sets the _load attribute.

Parameters

load
[_LoadData] Load data object

Returns

None

set_max_avg_fluid_temperature(*temp: float*) → None

This functions sets the maximal average fluid temperature to temp.

Parameters

temp
[float] Maximal average fluid temperature [deg C]

Returns

None

Raises

ValueError
When the maximal average fluid temperature is lower than the minimal average fluid temperature

set_min_avg_fluid_temperature(*temp: float*) → None

This functions sets the minimal average fluid temperature to temp.

Parameters

temp
[float] Minimal average fluid temperature [deg C]

Returns

None

Raises

ValueError
When the maximal average temperature is lower than the minimal average temperature

set_options_gfunction_calculation(*options: dict*) → None

This function sets the options for the gfunction calculation of pygfunction. This dictionary is directly passed through to the gFunction class of pygfunction. For more information, please visit the documentation of pygfunction.

Parameters

options
[dict] Dictionary with options for the gFunction class of pygfunction

Returns

None

set_pipe_parameters(*data: _PipeData*) → None

This function sets the pipe parameters.

Parameters

data
[PipeData] All the relevant pipe parameters

Returns

None

property simulation_period: int

This returns the simulation period from the LoadData object.

Returns

Simulation period [years]
[int]

size(*H_init: Optional[float] = None*, *use_constant_Rb: Optional[bool] = None*, *L2_sizing: Optional[bool] = None*, *L3_sizing: Optional[bool] = None*, *L4_sizing: Optional[bool] = None*, *quadrant_sizing: Optional[int] = None*, ***kwargs*) → float

This function sets the options for the sizing function.

- The L2 sizing is the one explained in (Peere et al., 2021)[Page 41, 2] and is the quickest method (it uses 3 pulses)
- The L3 sizing is a more general approach which is slower but more accurate (it uses 24 pulses/year)
- The L4 sizing is the most exact one, since it uses hourly data (8760 pulses/year)

Please note that the changes sizing setup changes here are not saved! Use self.setupSizing for this.

Parameters

H_init
[float] Initial depth for the iteration. If None, the default H_init is chosen.

use_constant_Rb
[bool] True if a constant borehole equivalent resistance (Rb*) value should be used

L2_sizing
[bool] True if a sizing with the L2 method is needed

L3_sizing
[bool] True if a sizing with the L3 method is needed

L4_sizing
[bool] True if a sizing with the L4 method is needed

quadrant_sizing
[int] Differs from 0 when a sizing in a certain quadrant is desired. Quadrants are developed by (Peere et al., 2021)[Page 41, 2, 3]

kwargs
[dict] Dictionary with all the other options that can be set within GHEtool. For a complete list, see the documentation in the CalculationSetup class.

Returns

borehole depth
[float]

Raises

[3] Peere, W. (2020) Methode voor economische optimalisatie van geothermische verwarmings- en koelsystemen. Master thesis, Department of Mechanical Engineering, KU Leuven, Belgium.

ValueError
ValueError when no ground data is provided

`size_L2`(*H_init: Optional[float] = None*, *quadrant_sizing: int = 0*) → float

This function sizes the of the given configuration according to the methodology explained in (Peere et al., 2021)[Page 41, 2], which is a L2 method. When quadrant sizing is other than 0, it sizes the field based on the asked quadrant. It returns the borefield depth.

Parameters

H_init
[float] Initial depth from where to start the iteration [m]

quadrant_sizing
[int] If a quadrant is given the sizing is performed for this quadrant else for the relevant

Returns

H
[float] Required depth of the borefield [m]

Raises

ValueError
ValueError when no ground data is provided or quadrant is not in range.

`size_L3`(*H_init: Optional[float] = None*, *quadrant_sizing: int = 0*) → float

This function sizes the borefield based on a monthly (L3) method.

Parameters

H_init
[float] Initial depth from where to start the iteration [m]

quadrant_sizing
[int] If a quadrant is given the sizing is performed for this quadrant else for the relevant

Returns

H
[float] Required depth of the borefield [m]

Raises

ValueError
ValueError when no ground data is provided or quadrant is not in range.

UnsolvableDueToTemperatureGradient
Error when the field cannot be sized.

`size_L4`(*H_init: Optional[float] = None*, *quadrant_sizing: int = 0*) → float

This function sizes the borefield based on an hourly (L4) sizing methodology.

Parameters

H_init
[float] Initial depth from where to start the iteration [m]

quadrant_sizing
[int] If a quadrant is given the sizing is performed for this quadrant else for the relevant

Returns

H
[float] Required depth of the borefield [m]

Raises

ValueError
ValueError when no ground data is provided or quadrant is not in range.

UnsolvableDueToTemperatureGradient
When the field cannot be sized due to the temperature gradient.

BaseClass

This document contains the information for the BaseClass. This class is used as a super class for different variable classes.

class GHEtool.VariableClasses.BaseClass.**BaseClass**

Bases: object

This class contains basic functionality of different classes within GHEtool. It contains the code to generate a dictionary from the class (in order to be able to export to JSON), to load a class based on a dictionary and to check whether or not all attributes differ from None.

This class should only be altered whenever a highly general method should be implemented.

check_values() → bool

This functions checks if the class attributes differ from None.

Returns

bool
True if all values are correct. False otherwise

from_dict(*dictionary: dict*) → None

This function converts the dictionary values to the class attributes. Currently, it can handle np.ndarray, list, set, str, int, float, tuple, pygfunction.Borehole and classes within GHEtool.

Parameters

dictionary
Dictionary with all the attributes of the class

Returns

None

to_dict() → dict

This function converts the class variables to a dictionary so it can be saved in a JSON format. Currently, it can handle np.ndarray, list, set, str, int, float, tuple, pygfunction.Borehole and classes within GHEtool.

Returns

dict
Dictionary with all the attributes of the class

exception GHEtool.VariableClasses.BaseClass.**MaximumNumberOfIterations**(*iter: int*)

Bases: RuntimeError

This Error occurs when the maximum number of interation is reacted.

exception GHEtool.VariableClasses.BaseClass.**UnsolvableDueToTemperatureGradient**

Bases: Exception

This Exception occurs when there is an unsizeble borefield due to incompatibility between 1) peak cooling, which requires a deeper borefield and 2) a temperature gradient, which causes a higher ground temperature when the field is drilled deeper. This leads to unsizeble solutions.

Borehole

This document contains all the information of the borehole class.

class GHEtool.VariableClasses.Borehole.**Borehole**(*fluid_data: Optional[FluidData] = None*, *pipe_data: Optional[_PipeData] = None*)

Bases: *BaseClass*

The borehole class contains all the functionalities related to the calculation of the equivalent borehole thermal resistance and contains a fluid and pipe class object.

Parameters

fluid_data
[FluidData] Fluid data

pipe_data
[PipeData] Pipe data

property Rb: float

This returns the constant, equivalent borehole thermal resistance [mK/W].

Returns

Rb*
[float] Equivalent borehole thermal resistance [mK/W]

property Re: float

Reynolds number.

Returns

Reynolds number
[float]

calculate_Rb(*H: float*, *D: float*, *r_b: float*, *k_s: float*) → float

This function calculates the equivalent borehole thermal resistance.

Parameters

H
[float] Borehole depth [m]

D
[float] Borehole burial depth [m]

r_b
[float] Borehole radius [m]

k_s
[float] Ground thermal conductivity [mk/W]

Returns

Rb
[float] Equivalent borehole thermal resistance

Raises

ValueError
ValueError when the pipe and/or fluid data is not set correctly.

property fluid_data: *FluidData*

This function returns the fluid data object.

Returns

FluidData

get_Rb(*H: float*, *D: float*, *r_b: float*, *k_s: float*) → float

This function returns the equivalent borehole thermal resistance. If use_constant_Rb is True, self._Rb is returned, otherwise the resistance is calculated.

Parameters

H
[float] Borehole depth [m]

D
[float] Borehole burial depth [m]

r_b
[float] Borehole radius [m]

k_s
[float] Ground thermal conductivity [mk/W]

Returns

Rb*
[float] Equivalent borehole thermal resistance [mK/W]

property pipe_data: *_PipeData*

This function returns the pipe data object.

Returns

PipeData

Variable Classes

GHEtool uses a couple of variable classes for handling the load data, ground and pipe properties. Please find below the different classes and their modules.

Ground data

The ground data classes you can use have either:

1. A constant ground temperature
2. A constant ground geothermal heat flux
3. A constant ground temperature gradient

All of these three classes are children of the abstract class _GroundData.

class GHEtool.VariableClasses.GroundData._GroundData.**_GroundData**(*k_s: Optional[float] = None*, *volumetric_heat_capacity: float = 2400000.0*)

Bases: *BaseClass*, ABC

Contains information regarding the ground data of the borefield.

Parameters

k_s
[float] Ground thermal conductivity [W/mK]

volumetric_heat_capacity
[float] The volumetric heat capacity of the ground [J/m3K]

abstract calculate_Tg(*H: float*) → float

This function gives back the ground temperature

Parameters

H
[float] Depth of the borefield [m]

Returns

Tg
[float] Ground temperature [deg C]

abstract calculate_delta_H(*temperature_diff: float*) → float

This function calculates the difference in depth for a given difference in temperature.

Parameters

temperature_diff
[float] Difference in temperature [deg C]

Returns

Difference in depth [m]
[float]

max_depth(*max_temp: float*) → float

This function returns the maximum depth, based on the maximum temperature. The maximum is the depth where the ground temperature equals the maximum temperature limit.

Parameters

max_temp
[float] Maximum temperature [deg C]

Returns

Depth
[float] Maximum depth [m]

class GHEtool.VariableClasses.GroundData.GroundConstantTemperature.**GroundConstantTemperature**(*k_s: Optional[float] = None*, *T_g: Optional[float] = None*, *volumetric_heat_capacity: float = 2400000.0*)

Bases: *_GroundData*

Parameters

k_s
[float] Ground thermal conductivity [W/mK]

T_g
[float] Ground temperature at infinity [deg C]

volumetric_heat_capacity
[float] The volumetric heat capacity of the ground [J/m3K]

calculate_Tg(*H: Optional[float] = None*) → float

This function gives back the ground temperature.

Parameters

H
[float] Depth of the borefield [m] (not used)

Returns

Tg
[float] Ground temperature [deg C]

calculate_delta_H(*temperature_diff: float*) → float

This function calculates the difference in depth for a given difference in temperature.

Parameters

temperature_diff
[float] Difference in temperature [deg C]

Returns

float
Difference in depth [m]

class GHEtool.VariableClasses.GroundData.GroundFluxTemperature.**GroundFluxTemperature**(*k_s: Optional[float] = None*, *T_g: Optional[float] = None*, *volumetric_heat_capacity: float = 2400000.0*, *flux: float = 0.06*)

Bases: *_GroundData*

Parameters

k_s
[float] Ground thermal conductivity [W/mK]

T_g
[float] Surface ground temperature [deg C]

volumetric_heat_capacity
[float] The volumetric heat capacity of the ground [J/m3K]

flux
[float] The geothermal heat flux [W/m2]

calculate_Tg(*H: float*) → float

This function gives back the ground temperature at a depth H.

Parameters

H
[float] Depth at which the temperature should be calculated [m]

Returns

Tg
[float] Ground temperature [deg C]

calculate_delta_H(*temperature_diff: float*) → float

This function calculates the difference in depth for a given difference in temperature.

Parameters

temperature_diff
[float] Difference in temperature [deg C]

Returns

float
Difference in depth [m]

class GHEtool.VariableClasses.GroundData.GroundTemperatureGradient.**GroundTemperatureGradient**(*k_s: Optional[float] = None*, *T_g: Optional[float] = None*, *volumetric_heat_capacity: float = 2400000.0*, *gradient: float = 3.0*)

Bases: *_GroundData*

Parameters

k_s
[float] Ground thermal conductivity [W/mK]

T_g
[float] Surface ground temperature [deg C]

volumetric_heat_capacity
[float] The volumetric heat capacity of the ground [J/m3K]

gradient
[float] The geothermal temperature gradient [K/100m]

calculate_Tg(*H: float*) → float

This function gives back the ground temperature at a depth H.

Parameters

H
[float] Depth at which the temperature should be calculated [m]

Returns

Tg
[float] Ground temperature [deg C]

calculate_delta_H(*temperature_diff: float*) → float

This function calculates the difference in depth for a given difference in temperature.

Parameters

temperature_diff
[float] Difference in temperature [deg C]

Returns

float
Difference in depth [m]

Fluid data

This document contains the variable classes for the fluid data.

class GHEtool.VariableClasses.FluidData.**FluidData**(*mfr: Optional[float] = None*, *k_f: Optional[float] = None*, *rho: Optional[float] = None*, *Cp: Optional[float] = None*, *mu: Optional[float] = None*)

Bases: *BaseClass*

Contains information regarding the fluid data of the borefield.

Parameters

mfr
[float] Mass flow rate per borehole [kg/s]

k_f
[float] Thermal Conductivity of the fluid [W/mK]

rho
[float] Density of the fluid [kg/m3]

Cp
[float] Thermal capacity of the fluid [J/kgK]

mu
[float] Dynamic viscosity of the fluid [Pa/s]

import_fluid_from_pygfunction(*fluid_object: Fluid*) → None

This function loads a fluid object from pygfunction and imports it into GHEtool. Note that this object does not contain the mass flow rate!

Parameters

fluid_object
[Fluid object from pygfunction]

Returns

None

set_mass_flow_rate(*mfr: float*) → None

This function sets the mass flow rate per borehole.

Parameters

mfr
[fluid] Mass flow rate per borehole [kg/s]

Returns

None

Load data

GHEtool supports different types of load data (and more are coming, check our project for more information: https://github.com/users/wouterpeere/projects/2) Currently you can use:

1. Geothermal loads with a monthly resolution for one year (so it repeats itself every year)
2. Geothermal loads with an hourly resolution for one year (so it repeats itself every year)
3. Geothermal loads with an hourly resolution but multiple years (it does not repeat itself)

All of the load classes are based children of the abstract _LoadData class.

class GHEtool.VariableClasses.LoadData._LoadData.**_LoadData**(*hourly_resolution: bool*, *simulation_period: int = 20*)

Bases: *BaseClass*, ABC

This class contains information w.r.t. load data for the borefield sizing.

Parameters

hourly_resolution
[bool] True if the load class uses an hourly resolution

simulation_period
[int] Length of the simulation period in years

property UPM: ndarray

Depending on whether all months are assumed to have equal length, the UPM are either constant or vary during the year.

Returns

Hours per month
[np.ndarray]

_calculate_first_year_params(*HC: bool*) → tuple

This function calculates the parameters for the sizing based on the first year of operation. This is needed for the L2 sizing.

Parameters

HC
[bool] True if the borefield is limited by extraction load

Returns

th, tpm, tcm, qh, qpm, qcm
[float] Peak duration [s], cumulative time passed at the start of the month [s], cumulative time passed at the end of the month [s], peak load [W], average cumulative load of the past months [W avg], average load of the current month [W avg]

_calculate_last_year_params(*HC: bool*) → tuple

This function calculates the parameters for the sizing based on the last year of operation. This is needed for the L2 sizing.

Parameters

HC
[bool] True if the borefield is limited by extraction load

Returns

th, qh, qm, qa
[float] Peak length [s], peak load [W], corresponding monthly load [W], yearly imbalance [W]

abstract _check_input(*input: Union[ndarray, list, tuple]*) → bool

This function checks whether the input is valid or not.

Parameters

input
[np.ndarray, list, tuple] Thermal load input

Returns

bool
True if the input is correct for the load class

add_dhw(*dhw: float*) → None

This function adds the domestic hot water (dhw). An error is raies is the dhw is not positive.

Parameters

dhw
[float] Yearly consumption of domestic hot water [kWh/year]

Returns

None

property all_months_equal: bool

Returns the attribute all months are equal

Returns

bool
True if the months are assumed to be of equal length (i.e. 730 hours/month). False if the correct number of hours is used.

abstract baseload_cooling() → ndarray

This function returns the baseload cooling in kWh/month.

Returns

baseload cooling
[np.ndarray]

property baseload_cooling_power: ndarray

This function returns the baseload cooling in kW avg/month.

Returns

baseload cooling
[np.ndarray]

property baseload_cooling_power_simulation_period: ndarray

This function returns the average cooling power in kW avg/month for a whole simulation period.

Returns

average cooling power
[np.ndarray] average cooling for the whole simulation period

property baseload_cooling_simulation_period: ndarray

This function returns the baseload cooling in kWh/month for a whole simulation period.

Returns

baseload cooling
[np.ndarray] baseload cooling for the whole simulation period

abstract baseload_heating() → ndarray

This function returns the baseload heating in kWh/month.

Returns

baseload heating
[np.ndarray]

property baseload_heating_power: ndarray

This function returns the baseload heating in kW avg/month.

Returns

baseload heating
[np.ndarray]

property baseload_heating_power_simulation_period: ndarray

This function returns the avergae heating power in kW avg/month for a whole simulation period.

Returns

average heating power
[np.ndarray] average heating power for the whole simulation period

property baseload_heating_simulation_period: ndarray

This function returns the baseload heating in kWh/month for a whole simulation period.

Returns

baseload heating
[np.ndarray] baseload heating for the whole simulation period

abstract correct_for_start_month(*array: ndarray*) → ndarray

This function corrects the load for the correct start month. If the simulation starts in september, the start month is 9 and hence the array should start at index 9.

Parameters

array
[np.ndarray] Load array

Returns

load
[np.ndarray]

property dhw: float

This function returns the yearly domestic hot water consumption.

Returns

dhw
[float] Yearly domestic hot water consumption [kWh/year]

property dhw_power: float

This function returns the power related to the dhw production.

Returns

dhw power
[float]

static get_month_index(*peak_load*, *avg_load*) → int

This function calculates and returns the month index (i.e. the index of the month in which the field should be sized). It does so by taking 1) the month with the highest peak load. 2) if all the peak loads are the same, it takes the month with the highest average load 3) if all average loads are the same, it takes the last month

Parameters

peak_load
[np.ndarray] array with the peak loads [kW]

avg_load
[np.ndarray] array with the monthly average loads [kW]

Returns

month_index
[int] 0 = jan, 1 = feb ...

property imbalance: float

This function calculates the ground imbalance. A positive imbalance means that the field is injection dominated, i.e. it heats up every year.

Returns

imbalance
[float]

property max_peak_cooling: float

This returns the max peak cooling in kW.

Returns

max peak cooling
[float]

property max_peak_heating: float

This returns the max peak heating in kW.

Returns

max peak heating
[float]

property monthly_average_load: ndarray

This function calculates the average monthly load in kW.

Returns

monthly average load
[np.ndarray]

property monthly_average_load_simulation_period: ndarray

This function calculates the average monthly load in kW for the whole simulation period.

Returns

monthly average load
[np.ndarray]

abstract peak_cooling() → ndarray

This function returns the peak cooling load in kW/month.

Returns

peak cooling
[np.ndarray]

property peak_cooling_duration: float

Duration of the peak in cooling.

Returns

Duration of the peak in cooling [s]

property peak_cooling_simulation_period: ndarray

This function returns the peak cooling in kW/month for a whole simulation period.

Returns

peak cooling
[np.ndarray] peak cooling for the whole simulation period

property peak_duration: None

Dummy object to set the length peak for both heating and cooling.

Returns

None

abstract peak_heating() → ndarray

This function returns the peak heating load in kW/month.

Returns

peak heating
[np.ndarray]

property peak_heating_duration: float

Length of the peak in heating.

Returns

Length peak in heating [s]

property peak_heating_simulation_period: ndarray

This function returns the peak heating in kW/month for a whole simulation period.

Returns

peak heating
[np.ndarray] peak heating for the whole simulation period

property start_month: int

This function returns the start month.

Returns

float
Start month

property time_L3: ndarray

Time for L3 sizing, i.e. an array with monthly the cumulative seconds that have passed. [744, 1416 ...] * 3600

Returns

Times for the L3 sizing
[np.ndarray]

property time_L4: ndarray

Times for the L4 sizing, i.e. an array with hourly the cumulative seconds that have passed. [1, 2, 3, 4 ...] * 3600

Returns

Times for the L4 sizing
[np.ndarray]

property ty: float

Simulation period in seconds.

Returns

Simulation period in seconds

property yearly_cooling_load: float

This function returns the yearly cooling load in kWh/year.

Returns

float
Yearly cooling load kWh/year

property yearly_heating_load: float

This function returns the yearly heating load in kWh/year.

Returns

float
Yearly heating load kWh/year

This class contains all the information for geothermal load data with a monthly resolution and absolute input. This means that the inputs are both in kWh/month and kW/month. This class contains all the information for geothermal load data with a monthly resolution and absolute input. This means that the inputs are both in kWh/month and kW/month. This class contains all the information for geothermal load data with a monthly resolution and absolute input. This means that the inputs are both in kWh/month and kW/month.

Pipe data

GHEtool supports both U-type and coaxial type heat exchangers. You can use:

1. Multiple U-tubes
2. Single U-tubes (special case of multiple U-tubes)
3. Double U-tubes (special case of multiple U-tubes)
4. Coaxial pipe

All of the classes above are children from the abstract _PipeData class.

class GHEtool.VariableClasses.PipeData._PipeData.**_PipeData**(*k_g: Optional[float] = None*, *k_p: Optional[float] = None*, *epsilon: float = 1e-06*)

Bases: BaseClass, ABC

Contains information regarding the pipe data of the borefield.

Parameters

k_g
[float] Grout thermal conductivity [W/mK]

k_p
[float] Pipe thermal conductivity [W/mK]

epsilon
[float] Pipe roughness [m]

abstract Re(*fluid_data:* FluidData) → float

Reynolds number.

Parameters

fluid_data: FluidData
Fluid data

Returns

Reynolds number
[float]

abstract calculate_resistances(*fluid_data:* FluidData) → None

This function calculates the conductive and convective resistances, which are constant.

Parameters

fluid_data
[FluidData] Fluid data

Returns

None

abstract draw_borehole_internal(*r_b: float*) → None

This function draws the internal structure of a borehole. This means, it draws the pipes inside the borehole.

Parameters

r_b
[float] Borehole radius [m]

Returns

None

abstract pipe_model(*fluid_data:* FluidData, *k_s: float*, *borehole: Borehole*) → _BasePipe

This function returns the BasePipe model.

Parameters

fluid_data
[FluidData] Fluid data

k_s
[float] Ground thermal conductivity

borehole
[Borehole] Borehole object

Returns

BasePipe

class GHEtool.VariableClasses.PipeData.MultipleUTube.**MultipleUTube**(*k_g: Optional[float] = None*, *r_in: Optional[float] = None*, *r_out: Optional[float] = None*, *k_p: Optional[float] = None*, *D_s: Optional[float] = None*, *number_of_pipes: int = 1*, *epsilon: float = 1e-06*)

Bases: *_PipeData*

Contains information regarding the Multiple U-Tube class.

Parameters

k_g
[float] Grout thermal conductivity [W/mK]

r_in
[float] Inner pipe radius [m]

r_out
[float] Outer pipe radius [m]

k_p
[float] Pipe thermal conductivity [W/mK]

D_s
[float] Distance of the pipe until center [m]

number_of_pipes
[int] Number of pipes [#] (single U-tube: 1, double U-tube:2)

epsilon
[float] Pipe roughness [m]

Re(*fluid_data:* FluidData) → float

Reynolds number.

Parameters

fluid_data: FluidData
fluid data

Returns

Reynolds number
[float]

calculate_resistances(*fluid_data:* FluidData) → None

This function calculates the conductive and convective resistances, which are constant.

Parameters

fluid_data
[FluidData] Fluid data

Returns

None

`draw_borehole_internal`(*r_b: float*) → None

This function draws the internal structure of a borehole. This means, it draws the pipes inside the borehole.

Parameters

r_b
[float] Borehole radius [m]

Returns

None

`pipe_model`(*fluid_data:* FluidData, *k_s: float*, *borehole: Borehole*) → _BasePipe

This function returns the BasePipe model.

Parameters

fluid_data
[FluidData] Fluid data

k_s
[float] Ground thermal conductivity

borehole
[Borehole] Borehole object

Returns

BasePipe

`class GHEtool.VariableClasses.PipeData.SingleUTube.SingleUTube`(*k_g: Optional[float] = None*, *r_in: Optional[float] = None*, *r_out: Optional[float] = None*, *k_p: Optional[float] = None*, *D_s: Optional[float] = None*, *epsilon: float = 1e-06*)

Bases: *MultipleUTube*

Class for the single U-Tube borehole.

Parameters

k_g
[float] Grout thermal conductivity [W/mK]

r_in
[float] Inner pipe radius [m]

r_out
[float] Outer pipe radius [m]

k_p
[float] Pipe thermal conductivity [W/mK]

D_s
[float] Distance of the pipe until center [m]

epsilon
[float] Pipe roughness [m]

class GHEtool.VariableClasses.PipeData.DoubleUTube.**DoubleUTube**(*k_g: Optional[float] = None*, *r_in: Optional[float] = None*, *r_out: Optional[float] = None*, *k_p: Optional[float] = None*, *D_s: Optional[float] = None*, *epsilon: float = 1e-06*)

Bases: *MultipleUTube*

Class for the double U-Tube borehole.

Parameters

k_g
[float] Grout thermal conductivity [W/mK]

r_in
[float] Inner pipe radius [m]

r_out
[float] Outer pipe radius [m]

k_p
[float] Pipe thermal conductivity [W/mK]

D_s
[float] Distance of the pipe until center [m]

epsilon
[float] Pipe roughness [m]

class GHEtool.VariableClasses.PipeData.CoaxialPipe.**CoaxialPipe**(*r_in_in: Optional[float] = None*, *r_in_out: Optional[float] = None*, *r_out_in: Optional[float] = None*, *r_out_out: Optional[float] = None*, *k_p: Optional[float] = None*, *k_g: Optional[float] = None*, *epsilon: float = 1e-06*, *is_inner_inlet: bool = True*)

Bases: *_PipeData*

Contains information regarding the Multiple U-Tube class.

Parameters

r_in_in
[float] Inner radius of the inner annulus [m]

r_in_out
[float] Outer radius of the inner annulus [m]

r_out_in
[float] Inner radius of the outer annulus [m]

r_out_out
[float] Outer radius of the outer annulus [m]

k_p
[float] Pipe thermal conductivity [W/mK]

k_g
[float] Thermal conductivity of the grout [W/mK]

epsilon
[float] Pipe roughness of the tube [m]

is_inner_inlet
[bool] True if the inlet of the fluid is through the inner annulus

`Re`(*fluid_data:* FluidData) → float

Reynolds number. Note: This code is based on pygfunction, 'convective_heat_transfer_coefficient_concentric_annulus' in the Pipes class.

Parameters

fluid_data: FluidData
fluid data

Returns

Reynolds number
[float]

`calculate_resistances`(*fluid_data:* FluidData) → None

This function calculates the conductive and convective resistances, which are constant.

Parameters

fluid_data
[FluidData] Fluid data

Returns

None

`draw_borehole_internal`(*r_b: float*) → None

This function draws the internal structure of a borehole. This means, it draws the pipes inside the borehole.

Parameters

r_b
[float] Borehole radius [m]

Returns

None

`pipe_model`(*fluid_data:* FluidData, *k_s: float*, *borehole: Borehole*) → _BasePipe

This function returns the BasePipe model.

Parameters

fluid_data
[FluidData] Fluid data

k_s
[float] Ground thermal conductivity

borehole
[Borehole] Borehole object

Returns

BasePipe

CustomGFunction

This file contains both the CustomGFunction class and all the relevant information w.r.t. custom gfunctions.

class GHEtool.VariableClasses.CustomGFunction.**CustomGFunction**(*time_array: Optional[ndarray] = None*, *depth_array: Optional[ndarray] = None*, *options: Optional[dict] = None*)

Bases: `object`

This class contains all the functionalities related to custom gfunctions.

Parameters

time_array
[np.ndarray] Time value(s) in seconds at which the gfunctions should be calculated

depth_array
[np.ndarray] Depths [m] for which the gfunctions should be calculated

options
[dict] Dictionary with options for the gFunction class of pygfunction

calculate_gfunction(*time_value: Union[list, float, ndarray]*, *H: float*, *check: bool = False*) → ndarray

This function returns the gfunction value, based on interpolation between precalculated values.

Parameters

time_value
[list, float, np.ndarray] Time value(s) in seconds at which the gfunctions should be calculated

H
[float] Depth [m] at which the gfunctions should be calculated. If no depth is given, the current depth is taken.

check
[bool] True if it should be check whether or not the requested gvalues can be interpolated based on the precalculated values

Returns

gvalues
[np.ndarray] 1D array with all the requested gvalues. False is returned if the check is True and the requested values are out of range for interpolation

create_custom_dataset(*borefield: List[Borehole]*, *alpha: float*) → None

This function creates the custom dataset.

Parameters

borefield
[list[pygfunction.boreholes.Borehole]] Borefield object for which the custom dataset should be created

alpha
[float] Ground thermal diffusivity [m2/s]

Returns

None

delete_custom_gfunction() → None

This function deletes the custom gfunction.

Returns

None

dump_custom_dataset(*path: str*, *name: str*) → None

This function dumps the current custom dataset.

Parameters

path
[str] Location where the dataset should be saved

name
[str] Name under which the dataset should be saved

Returns

None

set_options_gfunction_calculation(*options: dict*) → None

This function sets the options for the gfunction calculation of pygfunction. This dictionary is directly passed through to the gFunction class of pygfunction. For more information, please visit the documentation of pygfunction.

Parameters

options
[dict] Dictionary with options for the gFunction class of pygfunction

Returns

None

within_range(*time_value: Union[list, float, ndarray]*, *H: float*) → bool

This function checks whether or not the requested data can be calculated using the custom dataset.

Parameters

time_value
[list, float, np.ndarray] Time value(s) in seconds at which the gfunctions should be calculated

H
[float] Depth [m] at which the gfunctions should be calculated. If no depth is given, the current depth is taken.

Returns

bool
True if the requested values are within the range of the precalculated data, False otherwise

GHEtool.VariableClasses.CustomGFunction.**load_custom_gfunction**(*path: str*) → *CustomGFunction*

This function loads a custom gfunction dataset.

Parameters

path
[str] Location of the dataset

Returns

CustomGFunction
Dataset with the custom gfunction data

GFunction

class GHEtool.VariableClasses.GFunction.**FIFO**(*length: int = 2*)

Bases: object

This class is a container with n elements. If the n+1th element is added, the first is removed

Parameters

length
[int] Length of the fifo-array

add(*value: float*) → None

This function adds the value to the fifo array. If the array is full, the first element is removed.

Parameters

value
[float] Value to be added to the array

Returns

None

clear() → None

This function clears the fifo_array.

Returns

None

in_fifo_list(*value: float*) → bool

This function checks whether the value is in the fifo list, but not the last element!

Parameters

value
[float] Value potentially in the fifo list

Returns

bool
True if the value is in the fifo list, false otherwise

class GHEtool.VariableClasses.GFunction.**GFunction**

Bases: object

Class that contains the functionality to calculate gfunctions and to store previously calculated values that can potentially be used for interpolation to save time. This is done by storing previously calculated gvalues.

calculate(*time_value: Union[list, float, ndarray]*, *borefield: List[Borehole]*, *alpha: float*, *interpolate: Optional[bool] = None*)

This function returns the gvalues either by interpolation or by calculating them. It does so by calling the function gvalues which does this calculation. This calculation function also stores the previous calculated data and makes interpolations whenever the requested list of time_value are longer then DEFAULT_NUMBER_OF_TIMESTEPS.

Parameters

time_value
[list, float, np.ndarray] Array with all the time values [s] for which gvalues should be calculated

borefield
[list[pygfunction.boreholes.Borehole]] Borefield model for which the gvalues should be calculated

alpha
[float] Thermal diffusivity of the ground [m2/s]

interpolate
[bool] True if results should be interpolated when possible, False otherwise. If None, the default is chosen.

Returns

gvalues
[np.ndarray] 1D array with all the requested gvalues

`interpolate_gfunctions`(*time_value: Union[list, float, ndarray]*, *depth: float*, *alpha: float*, *borefield: List[Borehole]*) → ndarray

This function returns the gvalues by interpolation them. If interpolation is not possible, an emtpy array is returned.

Parameters

time_value
[list, float, np.ndarray] Time value(s) [s] for which gvalues should be calculated

depth
[float] Depth of the borefield [m]

alpha
[float] Thermal diffusivity of the ground [m2/s]

borefield
[list[pygfunction.boreholes.Borehole]] Borefield model for which the gvalues should be calculated

Returns

gvalues
[np.ndarray] 1D array with all the requested gvalues

`remove_previous_data`() → None

This function removes the previous calculated data by setting the depth_array, time_array and previous_gfunctions back to empty arrays.

Returns

None

`set_new_calculated_data`(*time_values: ndarray*, *depth: float*, *gvalues: ndarray*, *borefield*, *alpha*) → bool

This function stores the newly calculated gvalues if this is needed.

Parameters

time_values
[np.ndarray] Array with all the time values [s] for which gvalues should be calculated

depth
[float] Depth of the borefield [m]

gvalues
[np.ndarray] Array with all the calculated gvalues for the corresponding borefield, alpha and time_values

borefield
[list[pygfunction.borehole]] Borefield model for which the gvalues should be calculated

alpha
[float] Thermal diffusivity of the ground [m2/s]

Returns

bool
True if the data is saved, False otherwise

`set_options_gfunction_calculation`(*options: dict*, *add: bool = True*) → None

This function sets the options for the gfunction calculation of pygfunction. This dictionary is directly passed through to the gFunction class of pygfunction. For more information, please visit the documentation of pygfunction.

Parameters

options
[dict] Dictionary with options for the gFunction class of pygfunction

add
[bool] True if the options should be added, False is the options should be replaced.

Returns

None

SizingSetup

ghe_logger

Script to create the GHEtool logger for the console and the text file

`class GHEtool.logger.ghe_logger.CustomFormatter`(*fmt: str*)

Bases: `Formatter`

Class to create a special console coloring of the messages

Parameters

fmt
[str] Format of the log message

`format`(*record: LogRecord*) → str

Formats the record.

Parameters

record: logging.LogRecord
record to be formatted

Returns

str
Formatted log message

GHEtool.logger.ghe_logger.**addLoggingLevel**(*levelName*, *levelNum*, *methodName=None*)

Comprehensively adds a new logging level to the *logging* module and the currently configured logging class.

levelName becomes an attribute of the *logging* module with the value *levelNum*. *methodName* becomes a convenience method for both *logging* itself and the class returned by *logging.getLoggerClass()* (usually just *logging.Logger*). If *methodName* is not specified, *levelName.lower()* is used.

To avoid accidental clobberings of existing attributes, this method will raise an *AttributeError* if the level name is already an attribute of the *logging* module or if the method name is already present

1.2.9 Examples

Combination of active and passive cooling

```
"""
This file contains an example on how GHEtool can be used to size a borefield
using a combination of active and passive cooling.
This example is based on the work of Coninx and De Nies, 2021.
Coninx, M., De Nies, J. (2022). Cost-efficient Cooling of Buildings by means of Borefields
with Active and Passive Cooling. Master thesis, Department of Mechanical Engineering, KU Leuven, Belgium.
It is also published as: Coninx, M., De Nies, J., Hermans, L., Peere, W., Boydens, W., Helsen, L. (2024).
Cost-efficient cooling of buildings by means of geothermal borefields with active and passive cooling.
Applied Energy, 355, Art. No. 122261, https://doi.org/10.1016/j.apenergy.2023.122261.
"""

from GHEtool import Borefield, GroundConstantTemperature, HourlyGeothermalLoadMultiYear

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from skopt import gp_minimize


def active_passive_cooling(location='Active_passive_example.csv'):

    # load data
    columnNames = ['HeatingSpace', 'HeatingAHU', 'CoolingSpace', 'CoolingAHU']
    df = pd.read_csv(location, names=columnNames, header=0)
    heating_data = df.HeatingSpace + df.HeatingAHU
    cooling_data = df.CoolingSpace + df.CoolingAHU

    # variable COP and EER data
    COP = [0.122, 4.365]  # ax+b
    EER = [-3.916, 17,901]  # ax+b
    threshold_active_cooling = 16

    # set simulation period
    SIMULATION_PERIOD: int = 50
    heating_building: np.ndarray = np.tile(np.array(heating_data), SIMULATION_PERIOD)
```

(continues on next page)

(continued from previous page)

```
    cooling_building: np.ndarray = np.tile(np.array(cooling_data), SIMULATION_PERIOD)

    def update_load_COP(temp_profile: np.ndarray,
                        COP:np.ndarray,
                        load_profile: np.ndarray) -> np.ndarray:
        """
        This function updates the geothermal load for heating based on a variable COP

        Parameters
        ----------
        temp_profile : np.ndarray
            Temperature profile of the fluid
        COP : np.ndarray
            Variable COP i.f.o. temperature
        load_profile : np.ndarray
            Heating load of the building

        Returns
        -------
        Geothermal heating load : np.ndarray
        """
        COP_array = temp_profile * COP[0] + COP[1]
        return load_profile * (1 - 1/COP_array)


    def update_load_EER(temp_profile: np.ndarray,
                        EER: np.ndarray,
                        threshold_active_cooling: float,
                        load_profile: np.ndarray) -> np.ndarray:
        """
        This function updates the geothermal load for cooling based on a threshold for␣
↪active/passive cooling,
        and a variable EER.

        Parameters
        ----------
        temp_profile : np.ndarray
            Temperature profile of the fluid
        EER : np.ndarray
            Variable EER i.f.o. temperature
        threshold_active_cooling : float
            Threshold of the temperature above which active cooling is needed
        load_profile : np.ndarray
            Cooling load of the building

        Returns
        -------
        Geothermal cooling load : np.ndarray
        """
        EER_array = temp_profile * EER[0] + EER[1]
        passive: np.ndarray = temp_profile < threshold_active_cooling
        active = np.invert(passive)
```

(continues on next page)

(continued from previous page)

```
        return active * load_profile * (1 + 1/EER_array) + passive * load_profile


    costs = {"C_elec": 0.2159,      #  electricity cost (EUR/kWh)
             "C_borefield": 35,     #  inv cost per m borefield (EUR/m)
             "DR": 0.0011,          #  discount rate(-)
             "sim_period": SIMULATION_PERIOD}


    def calculate_costs(borefield: Borefield, heating_building: np.ndarray, heating_geothermal: np.ndarray,
                        cooling_building: np.ndarray, cooling_geothermal: np.ndarray, costs: dict) -> tuple:
        """
        This function calculates the relevant costs for the borefield.

        Parameters
        ----------
        borefield : Borefield
            Borefield object
        heating_building : np.ndarray
            Heating demand for the building
        heating_geothermal : np.ndarray
            Heating demand coming from the ground
        cooling_building : np.ndarray
            Cooling demand for the building
        cooling_geothermal : np.ndarray
            Cooling demand coming from the ground
        costs : dict
            Dictionary with investment cost for borefield/m, electricity cost, annual discount rate

        Returns
        -------
        investment cost borefield, operational cost heating, operational cost cooling, total operational cost:
            float, float, float, float
        """
        # calculate investment cost
        investment_borefield = costs["C_borefield"] * borefield.H * borefield.number_of_boreholes

        # calculate working costs
        elec_heating = heating_building - heating_geothermal
        elec_cooling = cooling_geothermal - cooling_building

        discounted_cooling_cost = []
        discounted_heating_cost = []
        for i in range(SIMULATION_PERIOD):
            tempc = costs["C_elec"] * (elec_cooling[730 * 12 * i:730 * 12 * (i + 1)])
            tempc = tempc * (1 / (1 + costs["DR"])) ** (i + 1)

```

(continues on next page)

(continued from previous page)

```
            temph = costs["C_elec"] * (elec_heating[730 * 12 * i:730 * 12 * (i + 1)])
            temph = temph * (1 / (1 + costs["DR"])) ** (i + 1)
            discounted_cooling_cost.append(tempc)
            discounted_heating_cost.append(temph)
        cost_cooling = np.sum(discounted_cooling_cost)
        cost_heating = np.sum(discounted_heating_cost)

        return investment_borefield, cost_heating, cost_cooling, cost_heating+cost_
↪cooling

    borefield = Borefield()
    borefield.simulation_period = SIMULATION_PERIOD
    borefield.set_max_avg_fluid_temperature(17)

    borefield.create_rectangular_borefield(12, 12, 6, 6, 100)
    borefield.set_ground_parameters(GroundConstantTemperature(2.1, 11))
    borefield.Rb = 0.12

    ### PASSIVE COOLING
    depths = [0.9, 0]

    # set initial loads
    cooling_ground = cooling_building.copy()
    heating_ground = heating_building.copy()

    while abs(depths[0] - depths[1]) > 0.1:
        # set loads
        load = HourlyGeothermalLoadMultiYear()
        load.hourly_heating_load = heating_ground
        load.hourly_cooling_load = cooling_ground
        borefield.load = load

        # size borefield
        depth_passive = borefield.size_L4()
        depths.insert(0, depth_passive)

        # get temperature profile
        temp_profile = borefield.results.peak_heating

        # recalculate heating load
        heating_ground = update_load_COP(temp_profile, COP, heating_building)


    ### ACTIVE COOLING
    depths = [0.9, 0]

    # set initial loads
    cooling_ground = cooling_building.copy()
    heating_ground = heating_building.copy()

    borefield.set_max_avg_fluid_temperature(25)
    while abs(depths[0] - depths[1]) > 0.1:
```

(continues on next page)

(continued from previous page)

```
        # set loads
        load = HourlyGeothermalLoadMultiYear()
        load.hourly_heating_load = heating_ground
        load.hourly_cooling_load = cooling_ground
        borefield.load = load

        # size borefield
        depth_active = borefield.size_L4()
        depths.insert(0, depth_active)

        # get temperature profile
        temp_profile = borefield.results.peak_heating

        # recalculate heating load
        heating_ground = update_load_COP(temp_profile, COP, heating_building)
        cooling_ground = update_load_EER(temp_profile, EER, 16, cooling_building)


    ### RUN OPTIMISATION

    # initialise parameters
    operational_costs = []
    operational_costs_cooling = []
    operational_costs_heating = []
    investment_costs = []
    total_costs = []
    depths = []


    def f(depth: list) -> float:
        """
        Optimisation function.

        Parameters
        ----------
        depth : list
            List with one element: the depth of the borefield in mm

        Returns
        -------
        total_cost : float
        """
        # convert to meters
        depth = depth[0] / 1000
        borefield._update_borefield_depth(depth)
        borefield.H = depth
        depths.append(depth)

        # initialise
        heating_ground = heating_building.copy()
        cooling_ground = cooling_building.copy()

```

(continues on next page)

(continued from previous page)

```
        heating_ground_prev = np.zeros(len(heating_ground))
        cooling_ground_prev = np.zeros(len(cooling_ground))

        # iterate until convergence in the load
        while np.sum(cooling_ground + heating_ground - heating_ground_prev - cooling_
↪ground_prev) > 100:
            # set loads
            load = HourlyGeothermalLoadMultiYear()
            load.hourly_heating_load = heating_ground
            load.hourly_cooling_load = cooling_ground
            borefield.load = load

            # get temperature profile
            borefield.calculate_temperatures(depth, hourly=True)
            temp_profile = borefield.results.peak_heating

            # set previous loads
            heating_ground_prev = heating_ground.copy()
            cooling_ground_prev = cooling_ground.copy()

            # recalculate heating load
            heating_ground = update_load_COP(temp_profile, COP, heating_building)
            cooling_ground = update_load_EER(temp_profile, EER, 16, cooling_building)

        # calculate costs
        investment, cost_heating, cost_cooling, operational_cost = calculate_
↪costs(borefield,
                                                                    heating_
↪building, heating_ground,
                                                                    cooling_
↪building, cooling_ground,
                                                                    costs)
        total_costs.append(investment + operational_cost)
        operational_costs.append(operational_cost)
        operational_costs_cooling.append(cost_cooling)
        operational_costs_heating.append(cost_heating)
        investment_costs.append(investment)
        return investment + operational_cost

    # add boundaries to figure
    # multiply with 1000 for numerical stability
    f([depth_active * 10 ** 3])
    f([depth_passive * 10 ** 3])

    res = gp_minimize(f,  # the function to minimize
                      [(depth_active * 10 ** 3, depth_passive * 10 ** 3)],  # the bounds␣
↪on each dimension of x
                      acq_func="EI",  # the acquisition function
                      n_calls=30,  # the number of evaluations of f
                      initial_point_generator="lhs",
                      n_random_starts=15,  # the number of random initialization points
                      # noise=0,       # the noise level (optional)
```

(continues on next page)

(continued from previous page)

```
                      random_state=1234)  # the random seed

    # plot figures
    fig = plt.figure()
    ax1 = fig.add_subplot(111)
    ax1.plot(depths, [x/1000 for x in total_costs], marker = 'o', label = "TC")
    ax1.plot(depths, [x/1000 for x in investment_costs], marker = 'o', label="IC")
    ax1.plot(depths, [x/1000 for x in operational_costs], marker = 'o', label="OC")
    ax1.plot(depths, [x/1000 for x in operational_costs_cooling], marker='o', label="OCc
↪")
    ax1.plot(depths, [x/1000 for x in operational_costs_heating], marker='o', label="OCh
↪")
    ax1.set_xlabel(r'Depth (m)', fontsize=14)
    ax1.set_ylabel(r'Costs ($k€$)', fontsize=14)
    ax1.legend(loc='lower left', ncol=3)
    ax1.tick_params(labelsize=14)
    plt.show()


if __name__ == "__main__":  # pragma: no cover
    active_passive_cooling()
```

Different borehole configurations

```
"""
This document is an example of how the borefield configuration can influence the total␣
↪borehole length and hence the cost of the borefield.
"""

# import all the relevant functions
from GHEtool import GroundConstantTemperature, Borefield, MonthlyGeothermalLoadAbsolute
import numpy as np
import pygfunction as gt


def effect_borefield_configuration():
    # GroundData for an initial field of 11 x 11
    data = GroundConstantTemperature(3, 10)
    borefield_gt = gt.boreholes.rectangle_field(11, 11, 6, 6, 110, 1, 0.075)

    # Monthly loading values
    peak_cooling = np.array([0., 0, 34., 69., 133., 187., 213., 240., 160., 37., 0., 0.
↪])  # Peak cooling in kW
    peak_heating = np.array([160., 142, 102., 55., 0., 0., 0., 0., 40.4, 85., 119., 136.
↪])  # Peak heating in kW

    # annual heating and cooling load
    annual_heating_load = 150 * 10 ** 3  # kWh
    annual_cooling_load = 400 * 10 ** 3  # kWh

```

(continues on next page)

(continued from previous page)

```
    # percentage of annual load per month (15.5% for January ...)
    monthly_load_heating_percentage = np.array([0.155, 0.148, 0.125, .099, .064, 0., 0.,␣
↪0., 0.061, 0.087, 0.117, 0.144])
    monthly_load_cooling_percentage = np.array([0.025, 0.05, 0.05, .05, .075, .1, .2, .2,
↪ .1, .075, .05, .025])

    # resulting load per month
    monthly_load_heating = annual_heating_load * monthly_load_heating_percentage    # kWh
    monthly_load_cooling = annual_cooling_load * monthly_load_cooling_percentage    # kWh

    # set the load
    load = MonthlyGeothermalLoadAbsolute(monthly_load_heating, monthly_load_cooling,␣
↪peak_heating, peak_cooling)

    # create the borefield object
    borefield = Borefield(load=load)

    borefield.set_ground_parameters(data)
    borefield.set_borefield(borefield_gt)
    borefield.Rb = 0.2

    # set temperature boundaries
    borefield.set_max_avg_fluid_temperature(16)    # maximum temperature
    borefield.set_min_avg_fluid_temperature(0)    # minimum temperature

    # size borefield
    depth = borefield.size()
    print("The borehole depth is:", depth, "m for a 11x11 field")
    print("The total length is:", int(depth * 11 * 11), "m")
    print("-------------------------")


    # borefield of 6x20
    data = GroundConstantTemperature(3, 10)
    borefield_gt = gt.boreholes.rectangle_field(6, 20, 6, 6, 110, 1, 0.075)

    # set ground parameters to borefield
    borefield.set_borefield(borefield_gt)
    borefield.set_ground_parameters(data)

    # set Rb
    borefield.Rb = 0.2

    # size borefield
    depth6_20 = borefield.size()
    print("The borehole depth is:", depth6_20, "m for a 6x20 field")
    print("The total length is:", int(depth6_20 * 6 * 20), "m")
    print("The second field is hence", -int(depth6_20 * 6 * 20) + int(depth * 11 * 11),
↪"m shorter")

    borefield.print_temperature_profile()

```

(continues on next page)

(continued from previous page)

```

if __name__ == "__main__":   # pragma: no cover
    effect_borefield_configuration()
```

Start simulation in a different month

```
"""
This example illustrates the importance of when a borefield is 'started' (i.e. when the first month of operation is).
"""

from GHEtool import *
from GHEtool.Validation.cases import load_case

import matplotlib.pyplot as plt


def start_in_different_month():
    # set data
    ground_data = GroundTemperatureGradient(2.5, 10)
    load = MonthlyGeothermalLoadAbsolute(*load_case(1))

    # create borefield object
    borefield = Borefield(load=load)
    borefield.ground_data = ground_data
    borefield.create_rectangular_borefield(10, 8, 6, 6, 100)

    borefield.set_max_avg_fluid_temperature(17)
    borefield.set_min_avg_fluid_temperature(3)
    borefield.calculation_setup(max_nb_of_iterations=100)

    depth_list = []

    # iterate over all the start months
    for month in range(1, 13, 1):
        borefield.load.start_month = month
        depth_list.append(borefield.size_L3())

    plt.figure()
    plt.bar(range(1, 13, 1), depth_list)
    plt.ylabel('Required depth [m]')
    plt.xlabel('First month of operation')
    plt.xlim(0)
    plt.ylim(0)
    plt.title('Required depth as a function of the first month of operation')
    plt.show()


if __name__ == "__main__":  # pragma: no cover
    start_in_different_month()
```

1.2.10 Validation

GHEtool is validated in a couple of ways. The goal is to increase the number of validation documents in the future.

First of all, it is internally checked for coherence, meaning that different methodology give (more or less) the same result. Examples of this can be found in the validation files below.

Sizing method comparison (L2/L3)

```
"""
This document compares both the L2 sizing method of (Peere et al., 2021) with a more general L3 sizing.
The comparison is based on speed and relative accuracy in the result.
"""

import time

import numpy as np
import pygfunction as gt

from GHEtool import Borefield, GroundConstantTemperature, MonthlyGeothermalLoadAbsolute


def sizing_method_comparison():
    number_of_iterations = 50
    max_value_cooling = 700
    max_value_heating = 800

    # initiate the arrays
    results_L2 = np.zeros(number_of_iterations)
    results_L3 = np.zeros(number_of_iterations)
    difference_results = np.zeros(number_of_iterations)

    monthly_load_cooling_array = np.empty((number_of_iterations, 12))
    monthly_load_heating_array = np.empty((number_of_iterations, 12))
    peak_load_cooling_array = np.empty((number_of_iterations, 12))
    peak_load_heating_array = np.empty((number_of_iterations, 12))

    # populate arrays with random values
    for i in range(number_of_iterations):
        for j in range(12):
            monthly_load_cooling_array[i, j] = np.random.randint(0, max_value_cooling)
            monthly_load_heating_array[i, j] = np.random.randint(0, max_value_heating)
            peak_load_cooling_array[i, j] = np.random.randint(monthly_load_cooling_array[i, j], max_value_cooling)
            peak_load_heating_array[i, j] = np.random.randint(monthly_load_heating_array[i, j], max_value_heating)

    # initiate borefield model
    data = GroundConstantTemperature(3, 10)
    borefield_gt = gt.boreholes.rectangle_field(10, 12, 6, 6, 110, 1, 0.075)

    # Monthly loading values
```

(continues on next page)

(continued from previous page)

```
    peak_cooling = np.array([0., 0, 34., 69., 133., 187., 213., 240., 160., 37., 0., 0.])  # Peak cooling in kW
    peak_heating = np.array([160., 142, 102., 55., 0., 0., 0., 0., 40.4, 85., 119., 136.])  # Peak heating in kW

    # annual heating and cooling load
    annual_heating_load = 300 * 10 ** 3  # kWh
    annual_cooling_load = 160 * 10 ** 3  # kWh

    # percentage of annual load per month (15.5% for January ...)
    monthly_load_heating_percentage = np.array([0.155, 0.148, 0.125, .099, .064, 0., 0., 0.061, 0.087, 0.117, 0.144])
    monthly_load_cooling_percentage = np.array([0.025, 0.05, 0.05, .05, .075, .1, .2, .2, .1, .075, .05, .025])

    # resulting load per month
    monthly_load_heating = annual_heating_load * monthly_load_heating_percentage    # kWh
    monthly_load_cooling = annual_cooling_load * monthly_load_cooling_percentage    # kWh

    # set the load
    load = MonthlyGeothermalLoadAbsolute(monthly_load_heating, monthly_load_cooling, peak_heating, peak_cooling)

    # create the borefield object
    borefield = Borefield(load=load)
    borefield.set_ground_parameters(data)
    borefield.set_borefield(borefield_gt)
    borefield.Rb = 0.2

    # set temperature boundaries
    borefield.set_max_avg_fluid_temperature(16)    # maximum temperature
    borefield.set_min_avg_fluid_temperature(0)    # minimum temperature

    # size according to L2 method
    start_L2 = time.time()
    for i in range(number_of_iterations):
        # set the load
        load = MonthlyGeothermalLoadAbsolute(monthly_load_heating_array[i], monthly_load_cooling_array[i],
                                             peak_load_heating_array[i], peak_load_cooling_array[i])
        borefield.load = load
        results_L2[i] = borefield.size(L2_sizing=True)
    end_L2 = time.time()

    # size according to L3 method
    start_L3 = time.time()
    for i in range(number_of_iterations):
        # set the load
        load = MonthlyGeothermalLoadAbsolute(monthly_load_heating_array[i], monthly_load_cooling_array[i],
                                             peak_load_heating_array[i], peak_load_
```

(continues on next page)

(continued from previous page)

```
↪cooling_array[i])
        borefield.load = load
        results_L3[i] = borefield.size(L3_sizing=True)
    end_L3 = time.time()

    print("Time for sizing according to L2:", end_L2 - start_L2, "s (or ", round((end_L2␣
↪- start_L2) / number_of_iterations * 1000, 3), "ms/sizing)")
    print("Time for sizing according to L3:", end_L3 - start_L3, "s (or ", round((end_L3␣
↪- start_L3) / number_of_iterations * 1000, 3), "ms/sizing)")

    # calculate differences
    for i in range(number_of_iterations):
        difference_results[i] = results_L3[i] - results_L2[i]

    print("The maximal difference between the sizing of L2 and L3 was:", np.round(np.
↪max(difference_results), 3), "m or", np.round(np.max(difference_results) / results_
↪L2[np.argmax(difference_results)] * 100, 3), "% w.r.t. the L2 sizing.")
    print("The mean difference between the sizing of L2 and L3 was:", np.round(np.
↪mean(difference_results), 3), "m or", np.round(np.mean(difference_results) / np.
↪mean(results_L2) * 100, 3), "% w.r.t. the L2 sizing.")


if __name__ == "__main__":   # pragma: no cover
    sizing_method_comparison()
```

Sizing method comparison (L2/L3/L4)

```
"""
This document is an example of the different sizing methods in GHEtool.
The example load profile is for a profile limited in the first year of operation.
"""
import time

import numpy as np
import pygfunction as gt

# import all the relevant functions
from GHEtool import *


def compare():
    # initiate ground data
    data = GroundConstantTemperature(3, 10)

    # initiate borefield
    borefield = Borefield()

    # set ground data in borefield
    borefield.set_ground_parameters(data)

```

(continues on next page)

(continued from previous page)

```
    # set Rb
    borefield.Rb = 0.12

    # set the borefield
    borefield.create_rectangular_borefield(10, 10, 6, 6, 110, 1, 0.075)

    # load the hourly profile
    load = HourlyGeothermalLoad()
    print(FOLDER)
    load.load_hourly_profile(FOLDER.joinpath("Examples/hourly_profile.csv"), header=True,
↪ separator=";")
    borefield.load = load
    borefield.simulation_period = 100

    ### size the borefield
    # according to L2
    L2_start = time.time()
    depth_L2 = borefield.size(100, L2_sizing=True)

    L2_stop = time.time()

    # according to L3
    L3_start = time.time()
    depth_L3 = borefield.size(100, L3_sizing=True)
    L3_stop = time.time()

    # according to L4
    L4_start = time.time()
    depth_L4 = borefield.size(100, L4_sizing=True)
    L4_stop = time.time()

    ### print results
    print("The sizing according to L2 took", round((L2_stop-L2_start) * 1000, 4), "ms␣
↪and was", depth_L2, "m.")
    print("The sizing according to L3 took", round((L3_stop-L3_start) * 1000, 4), "ms␣
↪and was", depth_L3, "m.")
    print("The sizing according to L4 took", round((L4_stop-L4_start) * 1000, 4), "ms␣
↪and was", depth_L4, "m.")

    borefield.plot_load_duration()
    borefield.print_temperature_profile(plot_hourly=True)


if __name__ == '__main__':   # pragma: no cover
    compare()
```

Speed comparison

```
"""
This document compares the speed of the L2 sizing method of (Peere et al., 2021) with and without the precalculated gfunction data.
This is done for two fields with different sizes. It shows that, specifically for the larger fields, the precalculated data is way faster.
"""

import time

import pygfunction as gt

from GHEtool import Borefield, GroundConstantTemperature, MonthlyGeothermalLoadAbsolute


def test_64_boreholes():
    data = GroundConstantTemperature(3, 10)
    borefield_64 = gt.boreholes.rectangle_field(8, 8, 6, 6, 110, 1, 0.075)

    # monthly loading values
    peak_cooling = [0., 0, 34., 69., 133., 187., 213., 240., 160., 37., 0., 0.]  # Peak cooling in kW
    peak_heating = [160., 142, 102., 55., 0., 0., 0., 0., 40.4, 85., 119., 136.]  # Peak heating in kW

    # annual heating and cooling load
    annual_heating_load = 300 * 10 ** 3  # kWh
    annual_cooling_load = 160 * 10 ** 3  # kWh

    # percentage of annual load per month (15.5% for January ...)
    monthly_load_heating_percentage = [0.155, 0.148, 0.125, .099, .064, 0., 0., 0., 0.061, 0.087, 0.117, 0.144]
    monthly_load_cooling_percentage = [0.025, 0.05, 0.05, .05, .075, .1, .2, .2, .1, .075, .05, .025]

    # resulting load per month
    monthly_load_heating = list(map(lambda x: x * annual_heating_load, monthly_load_heating_percentage))  # kWh
    monthly_load_cooling = list(map(lambda x: x * annual_cooling_load, monthly_load_cooling_percentage))  # kWh

    # set the load
    load = MonthlyGeothermalLoadAbsolute(monthly_load_heating, monthly_load_cooling, peak_heating, peak_cooling)

    # create the borefield object
    borefield = Borefield(load=load)

    borefield.set_ground_parameters(data)
    borefield.set_borefield(borefield_64)
    borefield.Rb = 0.2

```

(continues on next page)

(continued from previous page)

```
    # set temperature boundaries
    borefield.set_max_avg_fluid_temperature(16)  # maximum temperature
    borefield.set_min_avg_fluid_temperature(0)  # minimum temperature

    # precalculate
    borefield.create_custom_dataset()

    # size borefield
    t1 = time.time()
    depth_precalculated = borefield.size()
    t1_end = time.time()

    # delete precalculated data
    borefield.custom_gfunction.delete_custom_gfunction()

    ### size without the precalculation
    t2 = time.time()
    depth_calculated = borefield.size()
    t2_end = time.time()

    print("With precalculated data, the sizing took", round(t1_end - t1, 3), "s for 64 boreholes.")
    print("Without the precalculated data, the sizing took", round(t2_end - t2, 3), "s for 64 boreholes.")
    print("The difference in accuracy between the two results is",
          round((depth_calculated - depth_precalculated) / depth_calculated * 100, 3), "%.")


def test_10_boreholes():
    data = GroundConstantTemperature(3, 10)
    borefield_10 = gt.boreholes.rectangle_field(2, 5, 6, 6, 110, 1, 0.075)

    # monthly loading values
    peak_cooling = [0., 0, 3., 9., 13., 20., 43., 30., 16., 7., 0., 0.]  # Peak cooling in kW
    peak_heating = [16., 14, 10., 5., 0., 0., 0., 0., 4, 8., 19., 13.]  # Peak heating in kW

    # annual heating and cooling load
    annual_heating_load = 16 * 10 ** 3  # kWh
    annual_cooling_load = 24 * 10 ** 3  # kWh

    # percentage of annual load per month (15.5% for January ...)
    monthly_heating_load_percentage = [0.155, 0.148, 0.125, .099, .064, 0., 0., 0., 0.061, 0.087, 0.117, 0.144]
    monthly_load_cooling_percentage = [0.025, 0.05, 0.05, .05, .075, .1, .2, .2, .1, .075, .05, .025]

    # resulting load per month
    monthly_load_heating = list(map(lambda x: x * annual_heating_load, monthly_heating_load_percentage))  # kWh
```

(continues on next page)

(continued from previous page)

```
    monthly_load_cooling = list(map(lambda x: x * annual_cooling_load, monthly_load_cooling_percentage))  # kWh

    # set the load
    load = MonthlyGeothermalLoadAbsolute(monthly_load_heating, monthly_load_cooling, peak_heating, peak_cooling)

    # create the borefield object
    borefield = Borefield(load=load)

    borefield.set_ground_parameters(data)
    borefield.set_borefield(borefield_10)
    borefield.Rb = 0.2

    # set temperature boundaries
    borefield.set_max_avg_fluid_temperature(16)  # maximum temperature
    borefield.set_min_avg_fluid_temperature(0)  # minimum temperature

    # precalculate
    borefield.create_custom_dataset()

    # size borefield
    t1 = time.time()
    depth_precalculated = borefield.size()
    t1_end = time.time()

    # delete precalculated data
    borefield.custom_gfunction.delete_custom_gfunction()

    ### size without the precalculation
    t2 = time.time()
    depth_calculated = borefield.size()
    t2_end = time.time()

    print("With precalculated data, the sizing took", round(t1_end - t1, 3), "s for 10 boreholes.")
    print("Without the precalculated data, the sizing took", round(t2_end - t2, 3), "s for 10 boreholes.")
    print("The difference in accuracy between the two results is",
          round((depth_calculated-depth_precalculated) / depth_calculated * 100, 3), "%.\n")


if __name__ == "__main__":  # pragma: no cover

    test_10_boreholes()
    test_64_boreholes()
```

Peere et al. (2021) validates the hybrid sizing method of GHEtool. The validation code can be found below.

Four different sizing cases

```
"""
This document contains checks to see whether or not adaptations to the code still comply with some specific cases.
It also shows the difference between the original L2 sizing methode (Peere et al., 2021) and a more general L3 one.

This document contains 4 different cases referring to the paper: Peere, W., Picard, D., Cupeiro Figueroa, I., Boydens, W., and Helsen, L. Validated combined first and last year borefield sizing methodology. In Proceedings of International Building Simulation Conference 2021 (2021). Brugge (Belgium), 1-3 September 2021.

"""

import numpy as np
import pygfunction as gt

from GHEtool import Borefield, GroundConstantTemperature, MonthlyGeothermalLoadAbsolute

# relevant borefield data for the calculations
data = GroundConstantTemperature(3.5,  # conductivity of the soil (W/mK)
                                 10)   # Ground temperature at infinity (degrees C)

borefield_gt = gt.boreholes.rectangle_field(10, 12, 6.5, 6.5, 100, 4, 0.075)


def load_case(number):
    """This function returns the values for one of the four cases."""

    if number == 1:
        # case 1
        # limited in the first year by cooling
        monthly_load_heating_percentage = np.array([0.155, 0.148, 0.125, .099, .064, 0., 0., 0., 0.061, 0.087, 0.117, 0.144])
        monthly_load_cooling_percentage = np.array([0.025, 0.05, 0.05, .05, .075, .1, .2, .2, .1, .075, .05, .025])
        monthly_load_heating = monthly_load_heating_percentage * 300 * 10 ** 3  # kWh
        monthly_load_cooling = monthly_load_cooling_percentage * 150 * 10 ** 3  # kWh
        peak_cooling = np.array([0., 0., 22., 44., 83., 117., 134., 150., 100., 23., 0., 0.])
        peak_heating = np.zeros(12)

    elif number == 2:
        # case 2
        # limited in the last year by cooling
        monthly_load_heating_percentage = np.array([0.155, 0.148, 0.125, .099, .064, 0., 0., 0., 0.061, 0.087, .117, 0.144])
        monthly_load_cooling_percentage = np.array([0.025, 0.05, 0.05, .05, .075, .1, .2, .2, .1, .075, .05, .025])
        monthly_load_heating = monthly_load_heating_percentage * 160 * 10 ** 3  # kWh
        monthly_load_cooling = monthly_load_cooling_percentage * 240 * 10 ** 3  # kWh
        peak_cooling = np.array([0., 0, 34., 69., 133., 187., 213., 240., 160., 37., 0.,
```

(continues on next page)

(continued from previous page)

```
↪0.])  # Peak cooling in kW
        peak_heating = np.array([160., 142, 102., 55., 0., 0., 0., 0., 40.4, 85., 119.,␣
↪136.])

    elif number == 3:
        # case 3
        # limited in the first year by heating
        monthly_load_heating_percentage = np.array([0.155, 0.148, 0.125, .099, .064, 0.,␣
↪0., 0., 0.061, 0.087, .117, 0.144])
        monthly_load_cooling_percentage = np.array([0.025, 0.05, 0.05, .05, .075, .1, .2,
↪ .2, .1, .075, .05, .025])
        monthly_load_heating = monthly_load_heating_percentage * 160 * 10 ** 3  # kWh
        monthly_load_cooling = monthly_load_cooling_percentage * 240 * 10 ** 3  # kWh
        peak_cooling = np.zeros(12)
        peak_heating = np.array([300.0, 266.25, 191.25, 103.125, 0.0, 0.0, 0.0, 0.0, 75.
↪75, 159.375, 223.125, 255.0])

    else:
        # case 4
        # limited in the last year by heating
        monthly_load_heating_percentage = np.array([0.155, 0.148, 0.125, .099, .064, 0.,␣
↪0., 0., 0.061, 0.087, 0.117, 0.144])
        monthly_load_cooling_percentage = np.array([0.025, 0.05, 0.05, .05, .075, .1, .2,
↪ .2, .1, .075, .05, .025])
        monthly_load_heating = monthly_load_heating_percentage * 300 * 10 ** 3  # kWh
        monthly_load_cooling = monthly_load_cooling_percentage * 150 * 10 ** 3  # kWh
        peak_cooling = np.array([0., 0., 22., 44., 83., 117., 134., 150., 100., 23., 0.,␣
↪0.])
        peak_heating = np.array([300., 268., 191., 103., 75., 0., 0., 38., 76., 160.,␣
↪224., 255.])

    return monthly_load_heating, monthly_load_cooling, peak_heating, peak_cooling


def check_cases():

    """
    This function checks whether the borefield sizing gives the correct (i.e. validated)␣
↪results for the 4 cases.
    If not, an assertion error is raised.
    NOTE: these values differ slightly from the values in the mentioned paper. This is␣
↪due to the fact that GHEtool uses slightly different precalculated data.
    """

    correct_answers_L2 = (56.75, 117.23, 66.94, 91.32)
    correct_answers_L3 = (56.77, 118.74, 66.47, 91.24)

    for i in (1, 2, 3, 4):
        borefield = Borefield(load=MonthlyGeothermalLoadAbsolute(*load_case(i)))

        borefield.set_ground_parameters(data)
        borefield.set_borefield(borefield_gt)
```

(continues on next page)

(continued from previous page)

```
        borefield.Rb = 0.2

        # set temperature boundaries
        borefield.set_max_avg_fluid_temperature(16)  # maximum temperature
        borefield.set_min_avg_fluid_temperature(0)  # minimum temperature

        borefield.size(100, L2_sizing=True)
        print(f'correct answer L2: {correct_answers_L2[i-1]}; calculated answer L2:
↪{round(borefield.H,2)}; error: '
              f'{round(abs(1 - borefield.H / correct_answers_L2[i - 1]) * 100, 4)} %')
        assert np.isclose(borefield.H, correct_answers_L2[i-1], rtol=0.001)

        borefield.size(100, L3_sizing=True)
        print(f'correct answer L3: {correct_answers_L3[i - 1]}; calculated answer L3:
↪{round(borefield.H, 2)}; error: '
              f'{round(abs(1 - borefield.H / correct_answers_L3[i - 1]) * 100, 4)} %')
        assert np.isclose(borefield.H, correct_answers_L3[i-1], rtol=0.001)


def check_custom_datafile():
    """
    This function checks whether the borefield sizing gives the correct (i.e. validated)␣
↪results for the 4 cases given the custom datafile.
    If not, an assertion error is raised.
    """

    # create custom datafile

    correct_answers = (56.75, 117.23, 66.94, 91.32)

    custom_field = gt.boreholes.rectangle_field(N_1=12, N_2=10, B_1=6.5, B_2=6.5, H=110.,
↪ D=4, r_b=0.075)

    for i in (1, 2, 3, 4):
        borefield = Borefield(load=MonthlyGeothermalLoadAbsolute(*load_case(i)))

        borefield.set_ground_parameters(data)
        borefield.set_borefield(custom_field)
        borefield.Rb = 0.2

        # set temperature boundaries
        borefield.set_max_avg_fluid_temperature(16)  # maximum temperature
        borefield.set_min_avg_fluid_temperature(0)  # minimum temperature

        borefield.size(100, L3_sizing=True)
        print(f'correct answer: {correct_answers[i-1]}; calculated '
              f'answer: {round(borefield.H,2)}; error: '
              f'{round(abs(1-borefield.H/correct_answers[i - 1])*100,4)} %')


if __name__ == "__main__":   # pragma: no cover
    check_cases()  # check different cases
```

(continues on next page)

(continued from previous page)

```
    check_custom_datafile()  # check if the custom datafile is correct
```

The equivalent borehole thermal resistance is validated with the commercial software of Earth Energy Designer (EED) and can be found here.

Validation equivalent borehole thermal resistance

```
"""
This document contains the code to compare the equivalent borehole thermal resistance calculated with GHEtool
(based on pygfunction) with the results from Earth Energy Designer. The differences can be explained by using other
correlations and another assumption for the Nusselt number in the laminar regime.
"""

import math

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pygfunction as gt

from GHEtool import Borefield, FOLDER
from GHEtool.VariableClasses import FluidData, GroundConstantTemperature, DoubleUTube


def validate():
    # initiate parameters
    ground_data = GroundConstantTemperature(3, 10)  # ground data with an inaccurate guess of 100m for the depth of the borefield
    borefield_gt = gt.boreholes.rectangle_field(10, 12, 6, 6, 100, 1, 0.075)
    pipe_data = DoubleUTube(1, 0.015, 0.02, 0.4, 0.05, epsilon=1e-6)

    # initiate borefield model
    borefield = Borefield()
    borefield.set_ground_parameters(ground_data)
    borefield.set_pipe_parameters(pipe_data)
    borefield.set_borefield(borefield_gt)
    borefield.Rb = 0.12

    # initialise variables
    R_fp = []
    R_p = []
    Rb = []

    # load data EED
    data_EED = pd.read_csv(FOLDER.joinpath("Validation/resistances_EED.csv"), sep=";")

    mfr_range = np.arange(0.05, 0.55, 0.05)

    # calculate effective borehole thermal resistance (Rb*)
```

(continues on next page)

(continued from previous page)

```
    for mfr in mfr_range:
        fluid_data = FluidData(mfr, 0.568, 998, 4180, 1e-3)
        borefield.set_fluid_parameters(fluid_data)
        Rb.append(borefield.Rb)
        R_p.append(borefield.borehole.pipe_data.R_p)
        R_fp.append(borefield.borehole.pipe_data.R_f)


    # make figure
    plt.figure()
    plt.plot(R_fp, 'r+', label="GHEtool")
    plt.plot(data_EED["R_fp"], 'bo', label="EED")
    plt.xlabel("Mass flow rate per borehole l/s")
    plt.ylabel("Fluid-pipe resistance resistance mK/W")
    plt.title("Comparison R_fp from GHEtool with EED")
    plt.legend()

    plt.figure()
    plt.plot(mfr_range, (R_fp - data_EED["R_fp"])/data_EED["R_fp"]*100, 'bo')
    plt.xlabel("Mass flow rate per borehole l/s")
    plt.ylabel("Difference in fluid-pipe resistance %")
    plt.title("Comparison R_fp from GHEtool with EED (relative)")

    plt.figure()
    plt.plot(Rb, 'r+', label="GHEtool")
    plt.plot(data_EED["Rb*"], 'bo', label="EED")
    plt.xlabel("Mass flow rate per borehole l/s")
    plt.ylabel("Effective borehole thermal resistance mK/W")
    plt.title("Comparison Rb* from GHEtool with EED")
    plt.legend()

    plt.figure()
    plt.plot(mfr_range, (Rb - data_EED["Rb*"])/data_EED["Rb*"]*100, 'bo')
    plt.xlabel("Mass flow rate per borehole l/s")
    plt.ylabel("Difference in effective borehole thermal resistance %")
    plt.title("Comparison Rb* from GHEtool with EED (relative)")

    plt.show()


if __name__ == '__main__':   # pragma: no cover
    validate()
```

The deep sizing methodology, using a 1/depth assumption for the convergence of the borefield sizing when there is a temperature gradient can be found here.

Deep sizing

```
"""
This file contains the reasoning behind the sizing method when the field is limited by injection (i.e. cooling)
and there is a non-constant ground temperature. This is based on the assumption that the difference between the
maximum peak temperature in injection and the average, undistrubed ground temperature scales like 1/depth.

This can be understood since the main contributor to the peak temperature is the peak load, which, in the sizing,
is expressed as a load per meter (W/m), so it scales like 1/depth.
"""

from GHEtool import *
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import curve_fit


def validate():
    ground_data = GroundFluxTemperature(3, 10)
    fluid_data = FluidData(0.2, 0.568, 998, 4180, 1e-3)
    pipe_data = DoubleUTube(1, 0.015, 0.02, 0.4, 0.05)
    borefield = Borefield()
    borefield.create_rectangular_borefield(5, 4, 6, 6, 110, 4, 0.075)
    borefield.set_ground_parameters(ground_data)
    borefield.set_fluid_parameters(fluid_data)
    borefield.set_pipe_parameters(pipe_data)
    borefield.calculation_setup(use_constant_Rb=False)
    borefield.set_max_avg_fluid_temperature(17)
    borefield.set_min_avg_fluid_temperature(3)
    hourly_load = HourlyGeothermalLoad()
    hourly_load.load_hourly_profile(FOLDER.joinpath("test\methods\hourly_data\\auditorium.csv"), header=True, separator=";",
                                    col_cooling=0, col_heating=1)
    borefield.load = hourly_load

    # initiate lists
    Tg_list = []
    max_Tf_list = []
    depth_list = range(20, 450, 20)

    for depth in depth_list:
        print(f'The current depth is {depth} m.')
        borefield.calculate_temperatures(depth)
        Tg_list.append(borefield.ground_data.calculate_Tg(depth))
        max_Tf_list.append(np.max(borefield.results.peak_cooling))

    def f(x, a, b):
        return a/x + b

```

(continues on next page)

(continued from previous page)

```
    # determine temperature difference between peak cooling temperature and ground␣
↪temperature
    diff = np.array(max_Tf_list) - np.array(Tg_list)

    # fit to curve
    popt, pcov = curve_fit(f, depth_list, diff)
    print(popt, pcov)

    plt.figure()
    plt.plot(depth_list, Tg_list, label='Ground')
    plt.plot(depth_list, max_Tf_list, label='Fluid')
    plt.hlines(borefield.Tf_max, 0, depth_list[-1], label='Maximum temperature limit')
    plt.xlabel('Depth [m]')
    plt.ylabel('Temperature [deg C]')
    plt.legend()
    # plt.show()

    plt.figure()
    plt.plot(depth_list, diff, label='Actual calculated difference')
    plt.plot(depth_list, f(np.array(depth_list), *popt), label='Fitted difference')
    plt.xlabel('Depth [m]')
    plt.ylabel('Temperature difference [deg C]')
    plt.title('Temperature difference between maximum peak cooling fluid\ntemperature␣
↪and undistrubed ground temperature')
    plt.legend()
    plt.show()


if __name__ == "__main__":   # pragma: no cover
    validate()
```

1.2.11 Speed improvements

This folder contains information w.r.t. speed improvements in new versions of GHEtool.

v2.1.1

In GHEtool v2.1.1 there are two major code changes that reduce the computational time significantly. One has to do with the way the sizing methodology (L3/L4) is implemented and another with the new Gfunction class. Both improvements are explained below.

Improvement in sizing

Previously, in v2.1.0, for the sizing methodology L3 and L4 (i.e. monthly and hourly), the temperatures where calculated every time step. This however (especially for long simulation periods) requires a lot of time due to the convolution step, especially for an hourly sizing. In v2.1.1 this is changed so that only the first and last year are calculated, since only these years are relevant for the sizing[1]. This means that however long the simulation period may be, only two years are calculated. In the table below, the time required for one sizing iteration (i.e. one g-value convolution) for an hourly sizing is shown in µs. The code to come up with these numbers is added below.

Table 1: Speed improvement for g-value convolution in hourly sizing

Simulation period [years]	Required time old method [µs]	Required time new method [µs]
5 years	15625 µs	0 µs
15 years	15625 µs	0 µs
25 years	31250 µs	0 µs
35 years	46875 µs	0 µs
45 years	62500 µs	0 µs
55 years	78125 µs	0 µs
65 years	78125 µs	0 µs
75 years	78125 µs	0 µs
85 years	109375 µs	0 µs
95 years	125000 µs	0 µs

Gfunction class

Due to the implementation of the GFunction class in GHEtool, a substantial speed improvement is made w.r.t. GHEtool v2.1.0 for computationally expensive tasks. In the tables below, one can find this speed improvement for the different sizing methods and for several of the examples documents. The results can be recreated by running the code below. The computational times as shown in the table below, are an average of 5 runs.

Table 2: Speed benchmark sizing methods

Sizing method	Time v2.1.0 [ms]	Time v2.1.1 [ms]	Improvement [%]
L2 (three pulse) sizing	1.84 ms	1.43 ms	28%
L3 (monthly) sizing	12.34 ms	5.8 ms	113%
L4 (hourly) sizing	4.21 ms	3.63 ms	16%

Table 3: Speed benchmark examples

Example	Time v2.1.0 [ms]	Time v2.1.1 [ms]	Improvement [%]
Main functionalities	3.38 ms	2.57 ms	32%
Optimise load profile	15.07 ms	0.63 ms	2305%
Sizing with Rb calculation	9.97 ms	9.99 ms	0%
Effect borefield configuration	1.57 ms	1.5 ms	4%

```
import numpy as np
import pygfunction as gt
from GHEtool import GroundConstantTemperature, Borefield, HourlyGeothermalLoad, FOLDER
```

(continues on next page)

[1] Peere, W., Picard, D., Cupeiro Figueroa, I., Boydens, W., and Helsen, L. (2021) Validated combined first and last year borefield sizing methodology. In Proceedings of International Building Simulation Conference 2021. Brugge (Belgium), 1-3 September 2021. https://doi.org/10.26868/25222708.2021.30180

(continued from previous page)

```
from scipy.signal import convolve
from math import pi
from time import process_time_ns


def test_new_calc_method(simulation_period: int):
    """
    Test the new calculation method which is just considering the first and last year.

    Parameters
    ----------
    simulation_period : int
        simulation period [years]

    Returns
    -------
        None

    Raises
    -------
        AssertionError
    """

    h = 110

    # initiate ground data
    data = GroundConstantTemperature(3, 10)

    # initiate pygfunction borefield model
    borefield_gt = gt.boreholes.rectangle_field(10, 10, 6, 6, 110, 1, 0.075)

    # initiate borefield
    borefield = Borefield(100)

    # set borehole thermal equivalent resistance
    borefield.Rb = 0.12

    # set ground data in borefield
    borefield.set_ground_parameters(data)

    # set pygfunction borefield model
    borefield.set_borefield(borefield_gt)


    # load the hourly profile
    load = HourlyGeothermalLoad(simulation_period=simulation_period)
    load.load_hourly_profile(f'hourly_profile.csv', header=True, separator=";")

    borefield.load = load

    # borefield.g-function is a function that uses the precalculated data to interpolate␣
↪the correct values of the
```

(continues on next page)

(continued from previous page)

```
    # g-function. This dataset is checked over and over again and is correct
    g_values = borefield.gfunction(borefield.time_L4, borefield.H)

    # get process time at start of new method
    dt1 = process_time_ns()
    # determine load
    loads_short = borefield.hourly_cooling_load - borefield.hourly_heating_load
    # reverse the load
    loads_short_rev = loads_short[::-1]
    # init results vector
    results = np.zeros(loads_short.size * 2)
    # calculation of needed differences of the g-function values. These are the weight␣
↪factors in the calculation
    # of Tb.
    g_value_differences = np.diff(g_values, prepend=0)

    # convolution to get the results for the first year
    results[:8760] = convolve(loads_short * 1000, g_value_differences[:8760])[:8760]
    # sum up g_values until the pre last year
    g_sum_n1 = g_value_differences[:8760 * (borefield.simulation_period - 1)].
↪reshape(borefield.simulation_period - 1, 8760).sum(axis=0)
    # add up last year
    g_sum = g_sum_n1 + g_value_differences[8760 * (borefield.simulation_period - 1):]
    # add zero at start and reverse the order
    g_sum_n2 = np.concatenate((np.array([0]), g_sum_n1[::-1]))[:-1]
    # determine results for the last year by the influence of the year (first term) and␣
↪the previous years (last term)
    results[8760:] = convolve(loads_short * 1000, g_sum)[:8760] + convolve(loads_short_
↪rev * 1000, g_sum_n2)[:8760][::-1]
    # calculation the borehole wall temperature for every month i
    t_b = results / (2 * pi * borefield.ground_data.k_s) / (borefield.H * borefield.
↪number_of_boreholes) + borefield._Tg(borefield.H)

    # get process time
    dt2 = process_time_ns()
    # determine hourly load
    hourly_load = np.tile(borefield.hourly_cooling_load - borefield.hourly_heating_load,␣
↪borefield.simulation_period)
    # calculation of needed differences of the g-function values. These are the weight␣
↪factors in the calculation
    # of Tb.
    g_value_differences = np.diff(g_values, prepend=0)

    # convolution to get the monthly results
    results = convolve(hourly_load * 1000, g_value_differences)[:hourly_load.size]

    # calculation the borehole wall temperature for every month i
    t_b_new = results / (2 * pi * borefield.ground_data.k_s) / (h * borefield.number_of_
↪boreholes) + borefield._Tg(h)

    # print time for the different methods
    print(f'simulation period: {simulation_period}; old: {(process_time_ns() - dt2)/
```

(continues on next page)

(continued from previous page)

```
↪1000:.0f} µs;'
          f'new: {(dt2 - dt1)/1000:.0f} µs')
    # compare results to ensure they are the same
    assert np.allclose(t_b[:8760], t_b_new[:8760])
    assert np.allclose(t_b[8760:], t_b_new[8760*(borefield.simulation_period - 1):])


if __name__ == "__main__":
    for sim_year in np.arange(5, 101, 10):
        test_new_calc_method(sim_year)
```

```
from GHEtool import *
import numpy as np
import pygfunction as gt
import time
import os, contextlib

from GHEtool.Examples.main_functionalities import main_functionalities
from GHEtool.Examples.sizing_with_Rb_calculation import sizing_with_Rb
from GHEtool.Examples.effect_of_borehole_configuration import effect_borefield_
↪configuration
from GHEtool import HourlyGeothermalLoad

# disable the plot function by monkey patching over it
Borefield._plot_temperature_profile = lambda *args, **kwargs: None


# disable the print outputs
def supress_stdout(func):
    def wrapper(*a, **ka):
        with open(os.devnull, 'w') as devnull:
            with contextlib.redirect_stdout(devnull):
                return func(*a, **ka)
    return wrapper


@supress_stdout
def run_without_messages(callable) -> None:
    """
    This function runs the callable without messages.

    Parameters
    ----------
    callable : Callable
        Function to be called

    Returns
    -------
    None
    """
    callable()

```

(continues on next page)

(continued from previous page)

```

def optimise_load_profile() -> None:
    """
    This is a benchmark for the optimise load profile method.

    Returns
    -------
    None
    """
    # initiate ground data
    data = GroundData(3, 10, 0.2)

    # initiate pygfunction borefield model
    borefield_gt = gt.boreholes.rectangle_field(10, 10, 6, 6, 110, 1, 0.075)

    # initiate borefield
    borefield = Borefield()

    # set ground data in borefield
    borefield.set_ground_parameters(data)

    # set pygfunction borefield
    borefield.set_borefield(borefield_gt)

    # load the hourly profile
    load = HourlyGeothermalLoad()
    load.load_hourly_profile("hourly_profile.csv", header=True, separator=";")
    borefield.load = load

    # optimise the load for a 10x10 field (see data above) and a fixed depth of 150m.
    borefield.optimise_load_profile(depth=150, print_results=False)


def size_L2() -> None:
    """
    This is a benchmark for the L2 sizing method.

    Returns
    -------
    None
    """
    number_of_iterations = 5
    max_value_cooling = 700
    max_value_heating = 800

    monthly_load_cooling_array = np.empty((number_of_iterations, 12))
    monthly_load_heating_array = np.empty((number_of_iterations, 12))
    peak_load_cooling_array = np.empty((number_of_iterations, 12))
    peak_load_heating_array = np.empty((number_of_iterations, 12))

    # populate arrays with random values
    for i in range(number_of_iterations):
```

(continues on next page)

(continued from previous page)

```
        for j in range(12):
            monthly_load_cooling_array[i, j] = np.random.randint(0, max_value_cooling)
            monthly_load_heating_array[i, j] = np.random.randint(0, max_value_heating)
            peak_load_cooling_array[i, j] = np.random.randint(monthly_load_cooling_
↪array[i, j], max_value_cooling)
            peak_load_heating_array[i, j] = np.random.randint(monthly_load_heating_
↪array[i, j], max_value_heating)

    # initiate borefield model
    data = GroundData(3, 10, 0.2)
    borefield_gt = gt.boreholes.rectangle_field(10, 12, 6, 6, 110, 1, 0.075)

    # Monthly loading values
    peak_cooling = np.array([0., 0, 34., 69., 133., 187., 213., 240., 160., 37., 0., 0.
↪])  # Peak cooling in kW
    peak_heating = np.array([160., 142, 102., 55., 0., 0., 0., 0., 40.4, 85., 119., 136.
↪])  # Peak heating in kW

    # annual heating and cooling load
    annual_heating_load = 300 * 10 ** 3  # kWh
    annual_cooling_load = 160 * 10 ** 3  # kWh

    # percentage of annual load per month (15.5% for January ...)
    monthly_load_heating_percentage = np.array(
        [0.155, 0.148, 0.125, .099, .064, 0., 0., 0., 0.061, 0.087, 0.117, 0.144])
    monthly_load_cooling_percentage = np.array([0.025, 0.05, 0.05, .05, .075, .1, .2, .2,
↪ .1, .075, .05, .025])

    # resulting load per month
    monthly_load_heating = annual_heating_load * monthly_load_heating_percentage  # kWh
    monthly_load_cooling = annual_cooling_load * monthly_load_cooling_percentage  # kWh

    # create the borefield object
    borefield = Borefield(simulation_period=20,
                          peak_heating=peak_heating,
                          peak_cooling=peak_cooling,
                          baseload_heating=monthly_load_heating,
                          baseload_cooling=monthly_load_cooling)
    borefield.set_ground_parameters(data)
    borefield.set_borefield(borefield_gt)

    # set temperature boundaries
    borefield.set_max_avg_fluid_temperature(16)  # maximum temperature
    borefield.set_min_avg_fluid_temperature(0)  # minimum temperature

    # size according to L2 method
    for i in range(number_of_iterations):
        borefield.set_baseload_cooling(monthly_load_cooling_array[i])
        borefield.set_baseload_heating(monthly_load_heating_array[i])
        borefield.set_peak_cooling(peak_load_cooling_array[i])
        borefield.set_peak_heating(peak_load_heating_array[i])
        borefield.size(100, L2_sizing=True)
```

(continues on next page)

(continued from previous page)

```


def size_L3() -> None:
    """
    This is a benchmark for the L3 sizing method.

    Returns
    -------
    None
    """
    number_of_iterations = 5
    max_value_cooling = 700
    max_value_heating = 800

    monthly_load_cooling_array = np.empty((number_of_iterations, 12))
    monthly_load_heating_array = np.empty((number_of_iterations, 12))
    peak_load_cooling_array = np.empty((number_of_iterations, 12))
    peak_load_heating_array = np.empty((number_of_iterations, 12))

    # populate arrays with random values
    for i in range(number_of_iterations):
        for j in range(12):
            monthly_load_cooling_array[i, j] = np.random.randint(0, max_value_cooling)
            monthly_load_heating_array[i, j] = np.random.randint(0, max_value_heating)
            peak_load_cooling_array[i, j] = np.random.randint(monthly_load_cooling_
↪array[i, j], max_value_cooling)
            peak_load_heating_array[i, j] = np.random.randint(monthly_load_heating_
↪array[i, j], max_value_heating)

    # initiate borefield model
    data = GroundData(3, 10, 0.2)
    borefield_gt = gt.boreholes.rectangle_field(10, 12, 6, 6, 110, 1, 0.075)

    # Monthly loading values
    peak_cooling = np.array([0., 0, 34., 69., 133., 187., 213., 240., 160., 37., 0., 0.
↪])  # Peak cooling in kW
    peak_heating = np.array([160., 142, 102., 55., 0., 0., 0., 0., 40.4, 85., 119., 136.
↪])  # Peak heating in kW

    # annual heating and cooling load
    annual_heating_load = 300 * 10 ** 3  # kWh
    annual_cooling_load = 160 * 10 ** 3  # kWh

    # percentage of annual load per month (15.5% for January ...)
    monthly_load_heating_percentage = np.array(
        [0.155, 0.148, 0.125, .099, .064, 0., 0., 0., 0.061, 0.087, 0.117, 0.144])
    monthly_load_cooling_percentage = np.array([0.025, 0.05, 0.05, .05, .075, .1, .2, .2,
↪ .1, .075, .05, .025])

    # resulting load per month
    monthly_load_heating = annual_heating_load * monthly_load_heating_percentage  # kWh
    monthly_load_cooling = annual_cooling_load * monthly_load_cooling_percentage  # kWh
```

(continues on next page)

(continued from previous page)

```
    # create the borefield object
    borefield = Borefield(simulation_period=20,
                          peak_heating=peak_heating,
                          peak_cooling=peak_cooling,
                          baseload_heating=monthly_load_heating,
                          baseload_cooling=monthly_load_cooling)
    borefield.set_ground_parameters(data)
    borefield.set_borefield(borefield_gt)

    # set temperature boundaries
    borefield.set_max_avg_fluid_temperature(16)  # maximum temperature
    borefield.set_min_avg_fluid_temperature(0)  # minimum temperature

    # size according to L3 method
    for i in range(number_of_iterations):
        borefield.set_baseload_cooling(monthly_load_cooling_array[i])
        borefield.set_baseload_heating(monthly_load_heating_array[i])
        borefield.set_peak_cooling(peak_load_cooling_array[i])
        borefield.set_peak_heating(peak_load_heating_array[i])
        borefield.size(100, L3_sizing=True)


def size_L4() -> None:
    """
    This is a benchmark for the L4 sizing method.

    Returns
    -------
    None
    """
    # initiate ground data
    data = GroundData(3, 10, 0.2)

    # initiate pygfunction borefield model
    borefield_gt = gt.boreholes.rectangle_field(10, 10, 6, 6, 110, 1, 0.075)

    # initiate borefield
    borefield = Borefield()

    # set ground data in borefield
    borefield.set_ground_parameters(data)

    # set pygfunction borefield
    borefield.set_borefield(borefield_gt)

    # load the hourly profile
    load = HourlyGeothermalLoad()
    load.load_hourly_profile("hourly_profile.csv", header=True, separator=";")
    borefield.load = load

    borefield.size(L4_sizing=True)
```

(continues on next page)

(continued from previous page)

```


def benchmark(callable, name: str) -> None:
    """
    This function calls the callable five times and outputs the time needed to run the␣
↪callable.

    Parameters
    ----------
    callable : Callable
        Function to be called
    name : str
        Name of the function

    Returns
    -------
    None
    """
    diff, diff_without = 0., 0.

    for i in range(5):
        GFunction.DEFAULT_STORE_PREVIOUS_VALUES = True

        start_time = time.time()
        run_without_messages(callable)
        end_time = time.time()
        diff = diff * i/(i+1) + (end_time - start_time)/(i+1)

        GFunction.DEFAULT_STORE_PREVIOUS_VALUES = False

        start_time_without = time.time()
        run_without_messages(callable)
        end_time_without = time.time()
        diff_without = diff_without * i/(i+1) + (end_time_without - start_time_without)/
↪(i+1)

    print(f'{name} took  {round(diff_without, 2)} ms in v2.1.0 and '
          f'{round(diff, 2)} ms in v2.1.1. This is an improvement of {round((diff_
↪without-diff)/diff*100)}%.')


# run examples
benchmark(main_functionalities, "Main functionalities")
benchmark(optimise_load_profile, "Optimise load profile")
benchmark(sizing_with_Rb, "Sizing with Rb calculation")
benchmark(effect_borefield_configuration, "Effect borehole configuration")

# run benchmark sizing methods
benchmark(size_L2, "Sizing with L2 method")
benchmark(size_L3, "Sizing with L3 method")
benchmark(size_L4, "Sizing with L4 (hourly) method")
```

References

PYTHON MODULE INDEX

g

INDEX

Symbols

A

B

C

D

F

G

I

L

M

T

U

W

Y