TMM4270 Knowledge-based Engineering, Introduction (H2023)

Assignment:

A3: Web-based KBE system

Group 2:

Einar Myhrvold & Martin Bergstø

Problem

For assignment 3, we were challenged to create a web-based KBE system. This included a web browser as user interface, knowledge base with relevant information and coordination with NX for 3D modeling.

We started with deciding on using the solution from Assignment 2. We started early and it has been a long and hard process over several weeks, with a lot to learn regarding Fuseki, Sparql, genetic algorithms and web-integration.

After some research, we decided to use Python Flask to develop our web application, since it seemed simple to use. We followed the practice sessions in our development. Installing and learning to use Fuseki, Protege and Sparql was the first focus. After using some time in getting to know the Sparql query syntax and functionality, we started to integrate some simple queries in our app, for inserting new pumps. To develop our app further we had to create a genetic algorithm for pump design. How the algorithm is developed and used is further explained in its own section.

With the foundational queries established, we started to use Flask's template rendering capabilities. The HTML files in the "templates" folder are dynamically rendered by the Flask application, allowing us to pass variables to the HTML for user displays in the web browser. In the templates folder we have different HTML files for the different routes in our application. We also have templates for displaying a table with data of all pumps in the knowledge base, and a template for displaying a table with the data of a single pump.
To generate such tables with info we have used comprehensive queries to gather the relevant info, and dictionaries to store it in python. The dictionaries are then sent to the html files for displaying it to the user.

Briefly summarized: Behind the scenes, each user interaction triggers a series of parameters that flow through the system. These parameters guide the retrieval, insertion, and processing of data within the Fuseki knowledge base, ensuring that eachapp user action yields the correct response from the server.

The app.py file is the core of our Flask application, consisting of various Python functions each designed to fulfill specific roles within the system. We have tried to document the code with comprehensive comments and chosen descriptive names for functions and variables.

While the file is very long, we've taken measures to organize it logically. HTML templates are separated into the "templates" directory, preventing us from having logic and HTML in the same file. Furthermore, SPARQL queries are encapsulated within their own distinct functions. At the beginning of the file, you'll find the route declarations that Flask uses to map URLs to functions. Directly following each route decorator is the corresponding function, which is executed whenever the associated route is accessed. This direct mapping makes it easier to understand which part of the codebase is responsible for a given functionality.

Through this structured file organization and clear commenting, we’ve tried to make everything as intuitively as possible.

User guide

Getting Started with the Application:
To effectively use our application, a few initial setup steps are required:

Running the Fuseki-Server:

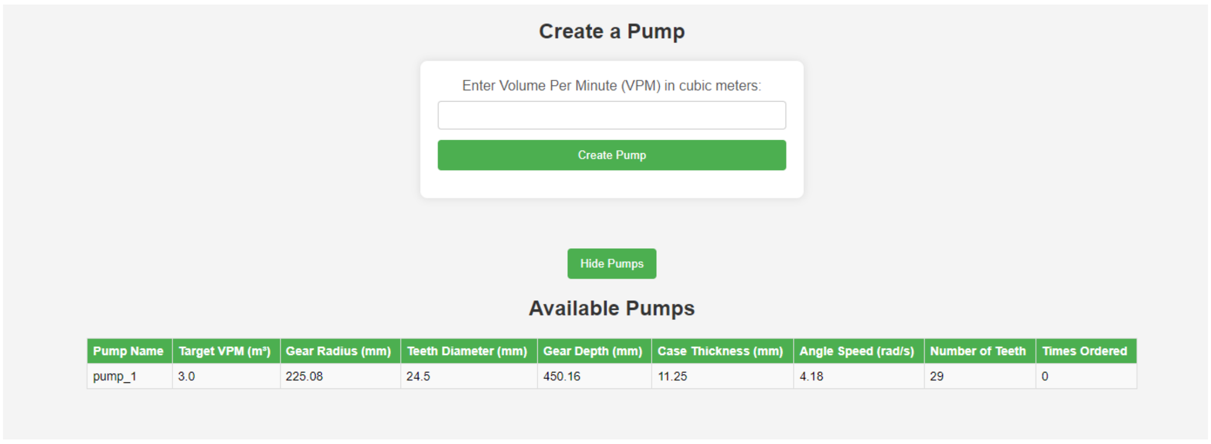
- Ensure that the Fuseki-Server is operational with our ontology, A3.rdf, properly uploaded. This step is crucial for the application to access and manipulate the pump data.

Setting Up the Python App (app.py):

- Before running app.py, install the necessary Python packages using pip install:
    - requests
    - flask

Navigating the Front Page:
The front page of our application features a form that allows users to design a pump based on a specified target flow rate (in cubic meters per minute), referred to as targetVPM.

- Understanding TargetVPM:
    - The targetVPM is an essential input for our genetic algorithm, which aims to design a pump that closely matches the specified flow rate.
- Exploring Existing Pumps:
    - If you are unsure about what targetVPM to input, you can explore existing pumps. Click on “Show available pumps” to display a detailed table of all pumps currently stored in our knowledge base. This information can help you identify a suitable pump. To hide this table, simply click “Hide pumps.” Remember to refresh the page for new pumps to be added to the table.


Create a Pump

Enter Volume Per Minute (VPM) in cubic meters:

Available Pumps

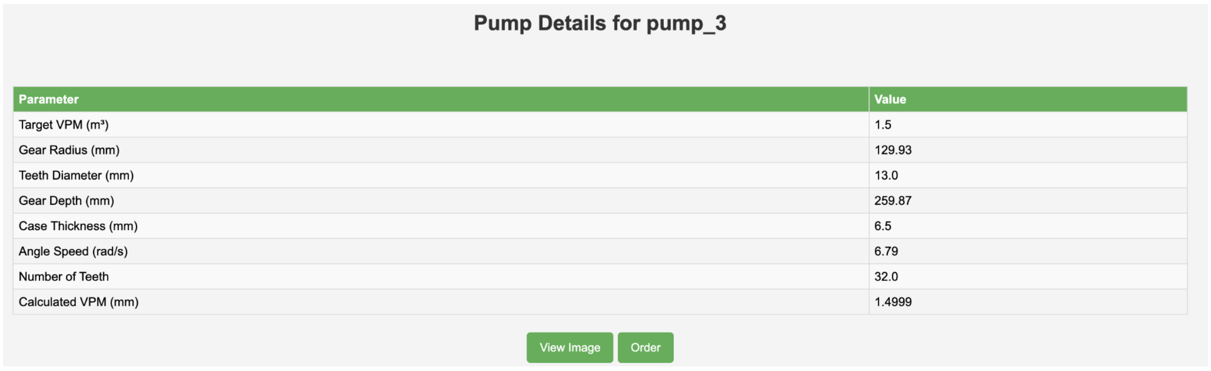
Pump Name	Target VPM (m³)	Gear Radius (mm)	Teeth Diameter (mm)	Gear Depth (mm)	Case Thickness (mm)	Angle Speed (rad/s)	Number of Teeth	Times Ordered
pump_1	3.0	225.08	24.5	450.16	11.25	4.18	29	0

Figure 1: Homepage showing available pumps. Parameters in mm.

Creating and Viewing Pump Details:

Upon clicking "Create pump," the application will display the details of a pump corresponding to the entered targetVPM.

- Existing Pumps:
    - If a pump with the specified targetVPM already exists in our knowledge base, you will receive information about this existing pump.
- New Pump Designs:
    - If the pump does not exist, our genetic algorithm kicks in to generate new pump parameters. The result, including the calculated VPM by the algorithm, will be displayed, usually close to your inputted targetVPM. Every parameter is calculated by the GA except the casing thickness, which is given a suitable value based on the size of the pump.


Pump Details for pump_3

Parameter	Value
Target VPM (m³)	1.5
Gear Radius (mm)	129.93
Teeth Diameter (mm)	13.0
Gear Depth (mm)	259.87
Case Thickness (mm)	6.5
Angle Speed (rad/s)	6.79
Number of Teeth	32.0
Calculated VPM (mm)	1.4999

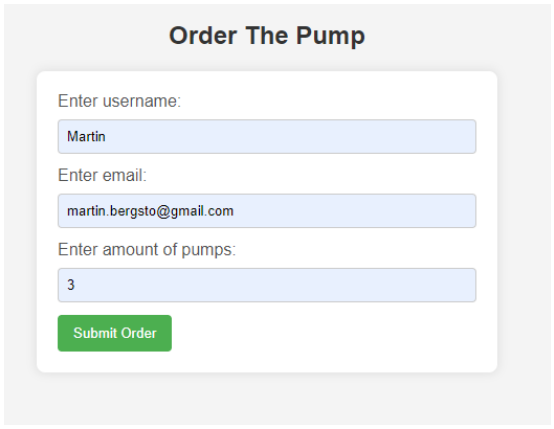
Figure 2: Pump details for a new pump.

Viewing Pump Images:

- Accessing Pump Images:
    - Select “View image” to attempt retrieving an image of the pump with the specified targetVPM.
- Image Generation Process:
    - For an image to be available, an external party, referred to as an engineer, must run pump.py in Siemens NX. This script, with the use of ImageGenerator.py, automatically captures a screenshot of the pump and stores it in the “Images” folder.
    - If this process has not been executed, the image of the pump will not be available.
    - In our submission we have already added some images for pumps we have generated during the work progress

Initiating an Order:
Once satisfied with the provided pump details, you can proceed to place an order by clicking the “Order” button. This action will navigate you to an order form, where you’ll need to provide some essential details.

- Completing the Order Form:
    - The order form requires your username, email, and the desired quantity of pumps.
    - In case your username does not exist in our knowledge base, we'll automatically create a new customer profile for you using the username provided.



Order The Pump

Enter username: Martin

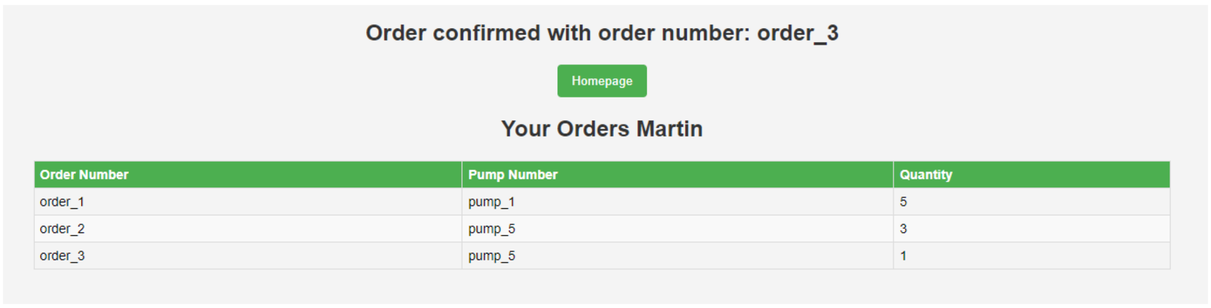
Enter email: martin.bergsto@gmail.com

Enter amount of pumps: 3

Figure 3: Order form

Confirming Your Order:
After submitting the order, you will be taken to the order confirmation page, which also provides an overview of your past orders.

- Order and History Overview:
    - Each order is assigned a unique order number for easy tracking. This is similar to how we uniquely identify each pump.
    - This page not only confirms your current order but also displays your complete order history, allowing you to review past transactions.
- Navigating Back to the Homepage:
    - If you wish to return to the homepage at any point, simply click the designated button to do so.



Order confirmed with order number: order_3

Your Orders Martin

Order Number	Pump Number	Quantity
order_1	pump_1	5
order_2	pump_5	3
order_3	pump_5	1

Figure 4: Order confirmation and order history for user named “Martin”

Structure

Main structure.
Our application has a complex structure, primarily due to the diverse components it integrates. While we run the application from the python code there is still communication with our database and NX to create the needed elements.

Flask-Powered Routing:
At the core of our architecture is the primary file, app.py, which uses the Flask library for efficient navigation between application pages. All routing is encapsulated within app.py, with dedicated functions created specifically for each page. Routing definitions are positioned at the top of the app.py file for clarity and ease of maintenance.

Dynamic Page Rendering:
As the application prepares to render a page, the app.py methodes gather or store information in the database, depending on what site is routed to. This is done by Sparql requests with relevant information. These calls are placed at the end of app.py, and solve unique problems and gather the specific data needed for the page to be rendered. When the sparql has been handled, then the different pages load in an HTML file from the folder “templates”. These often include information from the last page, sent through “POST” methods or simply with the flask method “render_template”.

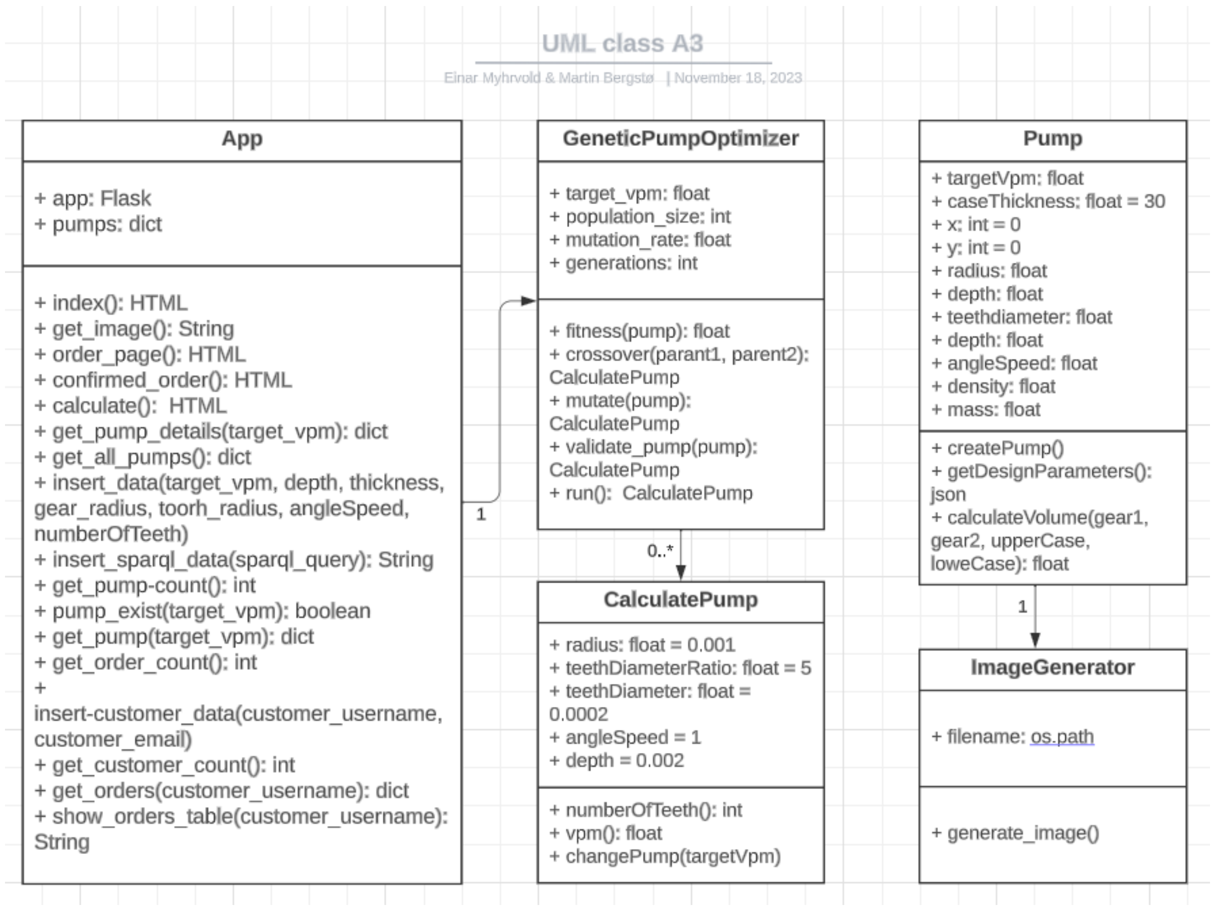
Pump Request Processing:
Upon a user's request for a pump with a targeted vpm, the application follows one of two paths. It either presents information on an existing pump from the Knowledge Base (KB) or employs the GeneticPumpOptimizer to assess and compare different pumps, returning an optimized pump tailored to the specified vpm. This optimized pump is not only stored in the KB for future use but also in the "pump_parameters" with its values. In cases where a new pump is created, an engineer needs to execute the pump.py file in NX. This file extracts values from the "pump_parameters" file, and subsequently, the ImageGenerator takes a screenshot, saving it in a folder accessible to app.py.

User Interaction and Ordering:
After extracting or creating a matching pump, users can choose to view an image of the pump or proceed to order it. The order page prompts users to provide a username, email, and the quantity of pumps before confirming. The database then extracts relevant information associated with the username, saving the new order under the respective customer and showcasing all previous orders.

Circular Navigation in the Web Application:

The navigation in our web application follows a circular path, guiding users through a sequence: creating a pump based on desired vpm, transitioning to an information page, proceeding to the order page, confirming the order, and circling back to the creation of a new pump. The text fields within the application that the user encounters are set to exclusive entry of numeric values where numerical input is necessary such as amount and vpm.

UML diagrams

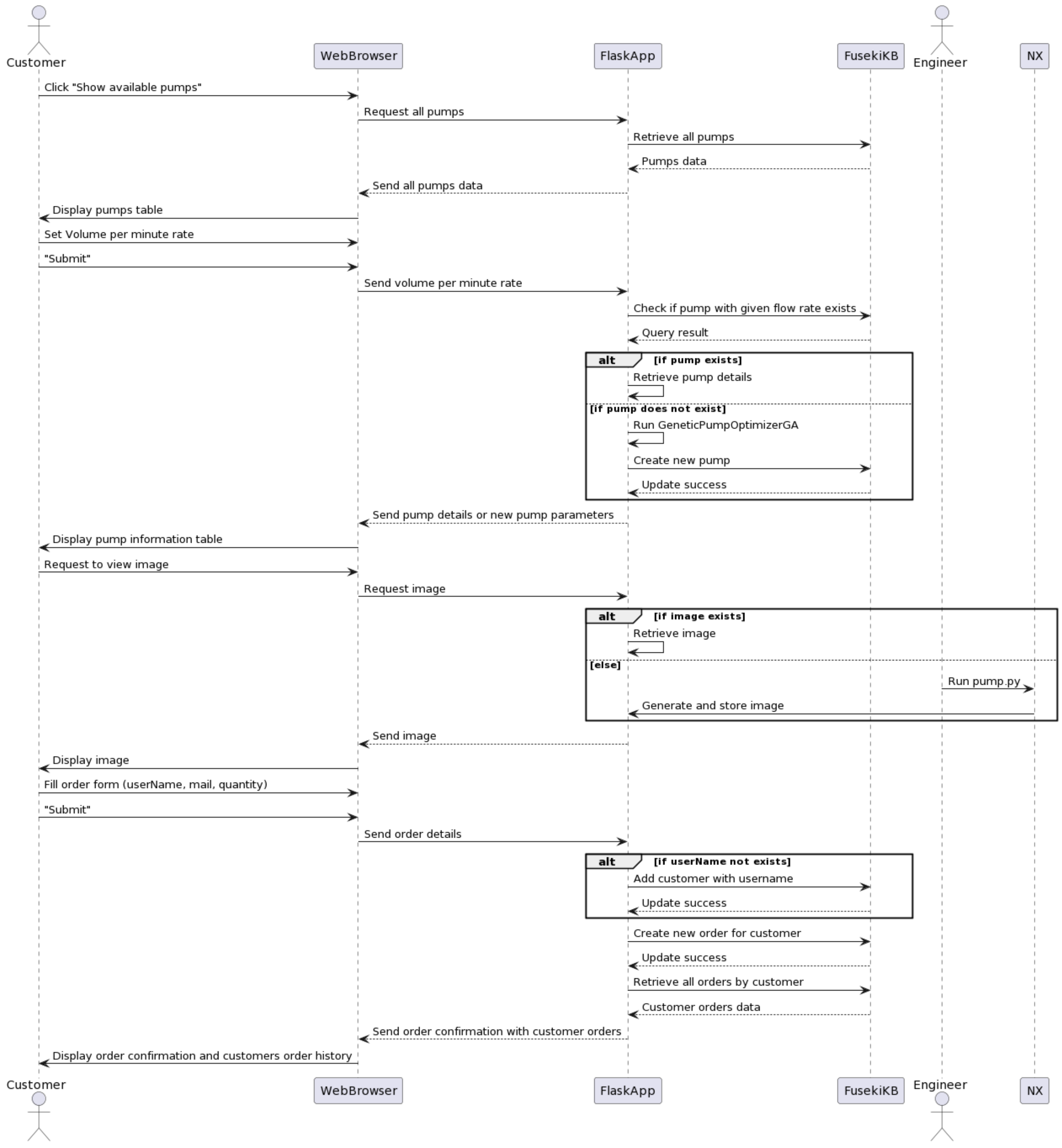

UML class A3
Einar Myhrvold & Martin Bergstø | November 18, 2023
App
+ app: Flask
+ pumps: dict
+ index(): HTML
+ get_image(): String
+ order_page(): HTML
+ confirmed_order(): HTML
+ calculate(): HTML
+ get_pump_details(target_vpm): dict
+ get_all_pumps(): dict
+ insert_data(target_vpm, depth, thickness, gear_radius, toorh_radius, angleSpeed, numberOfTeeth)
+ insert_sparql_data(sparql_query): String
+ get_pump-count(): int
+ pump_exist(target_vpm): boolean
+ get_pump(target_vpm): dict
+ get_order_count(): int
+ insert-customer_data(customer_username, customer_email)
+ get_customer_count(): int
+ get_orders(customer_username): dict
+ show_orders_table(customer_username): String
1
GeneticPumpOptimizer
+ target_vpm: float
+ population_size: int
+ mutation_rate: float
+ generations: int
+ fitness(pump): float
+ crossover(parant1, parent2): CalculatePump
+ mutate(pump): CalculatePump
+ validate_pump(pump): CalculatePump
+ run(): CalculatePump
0..*
CalculatePump
+ radius: float = 0.001
+ teethDiameterRatio: float = 5
+ teethDiameter: float = 0.0002
+ angleSpeed = 1
+ depth = 0.002
+ numberOfTeeth(): int
+ vpm(): float
+ changePump(targetVpm)
Pump
+ targetVpm: float
+ caseThickness: float = 30
+ x: int = 0
+ y: int = 0
+ radius: float
+ depth: float
+ teethdiameter: float
+ depth: float
+ angleSpeed: float
+ density: float
+ mass: float
+ createPump()
+ getDesignParameters(): json
+ calculateVolume(gear1, gear2, upperCase, loweCase): float
1
ImageGenerator
+ filename: os.path
+ generate_image()

Class

Here you can see the classes connected to our application.
App which is the main file has one instance of GeneteicPumpOptimizer (GPO). GPO will have many instances of CalculatePump since it uses an algorithm to compare and determine the most precise and effective pump of many. The values under CalculatePump are default values from Assignment 2. They are overruled when used in GPO to create pumps.

The Pump class with one instance of ImageGenerator is separate classes with no direct ties with the rest of the classes. This is because Pump will read from a Json file regardless of what happens in the App class. But App is where the Json file is updated with the new values, so it is still relevant for our solution.

Sequence



Structure reflection and challenges

Flask is well known to fit websites with few pages and low complexity. Since our application has 5 pages, flask is a perfect fit for routing in app.py. Other python html routes would also be possible, but more complex to set-up.

One of the more advanced parts of our web-application was sending the necessary information to the relevant page. It was important for the UI and KB to have the right data sent. Our decision to use POST and not GET is because we only need to read the value that we send between the pages. This makes it more secure and easier as we don't need to use any GET parameters for the call. An issue concerning sparql and extracting data from the KB was to gather enough and the right information to do an accurate query request. Luckily with flask and POST we were able to get the information needed by sending it between pages.

Because of our choice to build upon our solution from A2, we already had the functionality to create the 3D module in NX from python. Still, we had to adapt by making the user be able to choose the values in the input file. This worked out fine as we could update a json file from app.py, but we still need to manually create the assembly in NX before we can show an image to the user (unless the pump is in the KB already).

Our choice to have the customer information at the end when the order is complete lets us have circular navigation on our site. The benefit of this is more specific calls and fewer places for the user to get lost. Not to mention fewer error handlings since it's predetermined what is the next page.

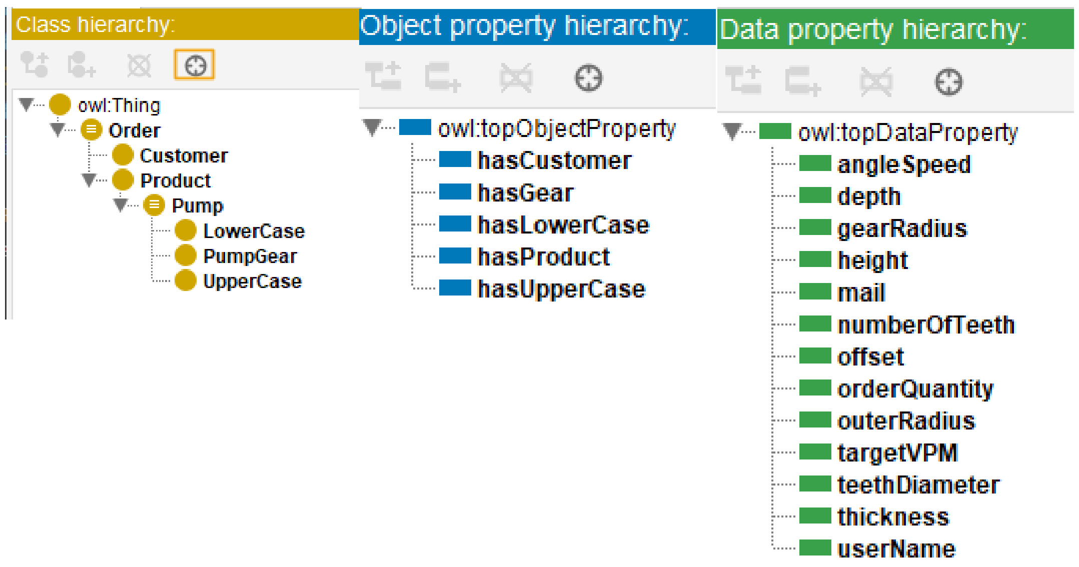
When loading the program we could gather all the information every time, and sort out what of it was needed for the issue at hand. But with larger information sent we found it much harder to work within the code. So we created multiple sparql queries for the unique situations. These queries let us have a more detailed code with a clear structure, something we saved a lot of time on later in the design process. For queries that gather a lot of data, we often decided to store the data in python dictionaries for easier use in other functions that handle the data. To begin with it was difficult to handle the data that queries returned. We were not familiar with the formatting. But after having dealt with it in one function we could reuse the logic in other functions with queries for gathering data.

Most of the functionality in our app needs access to one or more pumps data. In our development process we noticed that we had many calls to query-functions all over our code. It was messy and seemed like an unecessary amount of calls to the knowledge base. One solution that helped with the problem was a global dictionary variable with all necessary pump data. This variable is updated when new pumps are added and accessed where needed.

We also had difficulties with displaying images in our web-browser. We had to make a recorded python file in NX where we extracted an image of the pump. Then we made adjustments to the recorder file so it became a python class with input parameter filename. With this adjustment we could reuse the “ImageGenerator.py” and save images with new names each time. The issue was now to integrate this in our application as smoothly as possible. We decided on having to run “pump.py” in NX to automatically generate an image which the user then can access. The pump.py file can be adjusted as wanted, for example removing the casing to get a clearer picture of the gears.

OWL

Our ontology is structured in the following way, with Classes, Object Properties and Data Properties:



```
Class hierarchy:
owl:Thing
    Order
        Customer
        Product
            Pump
                LowerCase
                PumpGear
                UpperCase

Object property hierarchy:
owl:topObjectProperty
    hasCustomer
    hasGear
    hasLowerCase
    hasProduct
    hasUpperCase

Data property hierarchy:
owl:topDataProperty
    angleSpeed
    depth
    gearRadius
    height
    mail
    numberOfTeeth
    offset
    orderQuantity
    outerRadius
    targetVPM
    teethDiameter
    thickness
    userName
```

- Class Hierarchy:
    - The ontology includes several classes that form a hierarchy under the root owl:Thing class, which is standard in OWL ontologies.
    - The primary classes are Order, Customer, Product, and Pump, with Pump being further specialized into LowerCase, PumpGear, and UpperCase.
    - We decided not to represent Shapes classes, like block and cylinder, in our ontology. It is not necessary for representing our pumps in the knowledge base, since the shapes are mainly used as parts in unite or subtract methods in the design of a pump.
- Object Property Hierarchy:
    - The object properties define the relationships between instances of the classes. For example, hasCustomer, hasGear, hasLowerCase, hasProduct, and hasUpperCase describe how various entities like orders, pumps, and their parts are related. For instance, an Order has a Customer and a Product.
- Data Property Hierarchy:
    - The data properties specify attributes of the classes that take literal values, such as angleSpeed, depth, gearRadius, height, mail, numberofTeeth, orderQuantity, outerRadius, targetVPM, teethDiameter, thickness, and userName. These properties store specific data about the items in the ontology, like the measurements of pump components or customer details.
- Relationships and Cardinality:

    - The (=) inside the Order and Pump classes representsa cardinality constraint, indicating that an Order is associated with exactly one Customer and one Product. Similarly, the Pump class has a direct relationship with its subclasses, which indicate that a pump consists of these specific parts.
- Extensibility:
    - The ontology is designed with extensibility in mind. While the current focus is on pump design and storage, the Product class provides a pathway for adding new products in the future, thus making the ontology adaptable for a broader range of products beyond pumps.
- Usage:
    - The ontology serves as the backbone of a knowledge base for our application that manages pump orders, customer information, and pump design parameters. It enables sophisticated data management and retrieval, such as finding all orders associated with a customer, determining the specifications of pump components, and storing new customer orders with unique identifiers.

Genetic algorithm (GA)

Our genetic algorithm, found in GeneticPumpOptimizer.py, uses an evolutionary approach to optimize pump designs. It has been configured with a population size of 100, a mutation rate of 0.05, and a target of generating 1,000 iterations (generations) to converge on an optimal solution. For clarity and ease of understanding, the code within the Python file is extensively annotated.

The algorithm initiates with a diverse population of potential pump solutions, each characterized by a set of randomly assigned parameters and designed through our pump design module, as established in Assignment 2. Selection is conducted by choosing two parent solutions from the top 50 performers of the current population. These are then bred through crossover mechanisms, with the possibility of mutations, to produce a new offspring pump design. This process is iteratively applied until a complete new generation is populated.

After the creation of a thousand generations, the algorithm selects the 'best' pump. Here, 'best' is defined according to the pump's score from the fitness function.

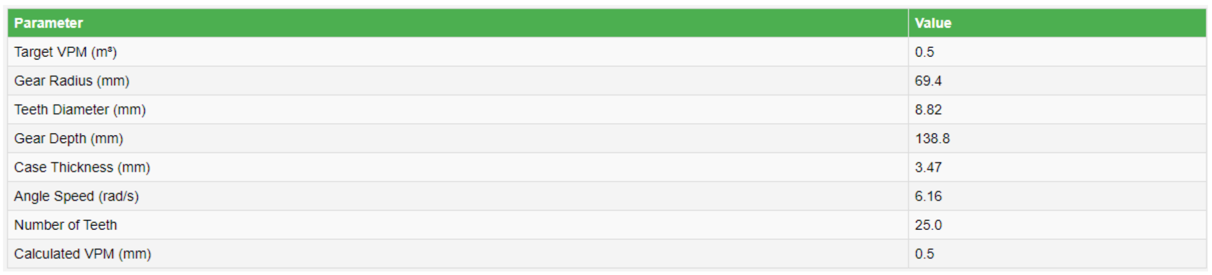
The fitness function is important to the GA's performance, evaluating each pump's potential by comparing its volumetric flow rate (VPM) against the target VPM. The fitness is quantified as the inverse of the square of the deviation from the target VPM. This formula ensures that pumps with flow rates closest to the target VPM are favored during the selection process.

The GA includes validation functions to maintain the feasibility of pump designs, ensuring parameters like the ratio between gear- and teeth size to stay within practical bounds. Limits are also placed on angle speed and gear size, but these can of course be adjusted if necessary. Such limitations prevent unrealistic pump designs, though they may limit the GA's ability to design pumps for very high targetVPMs (> 100 VPM) due to fixed speed and teeth amount boundaries. Nonetheless, for typical VPM values (0.1 to 100 VPM), the GA should deliver realistic and well-performing pump configurations.

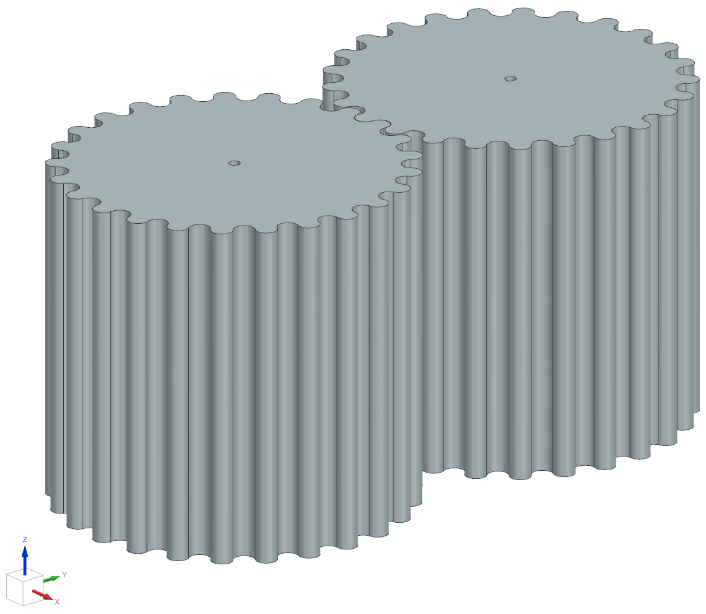
Examples

Below you can see 3 examples of pumps with different parameters. We have not shown the casing because it looks similar for every pump. The pumps may not look very different in the pictures because they are zoomed in to fit the size. If you look at the parameters you can see that the different solutions differ a lot in both teeth and gear size, as well as angle speed and number of teeth.

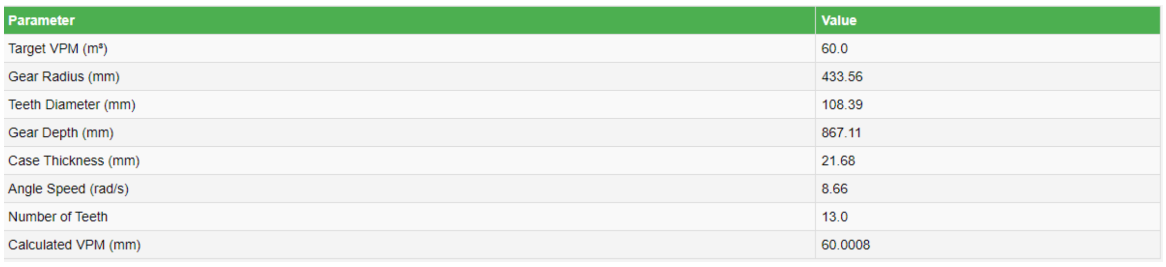
Example 1



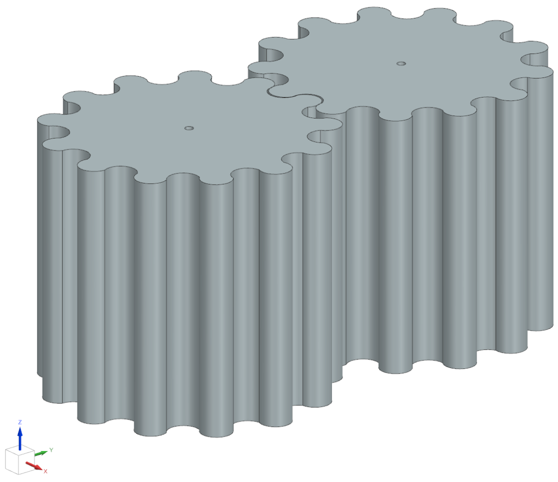
Parameter	Value
Target VPM (m^3)	0.5
Gear Radius (mm)	69.4
Teeth Diameter (mm)	8.82
Gear Depth (mm)	138.8
Case Thickness (mm)	3.47
Angle Speed (rad/s)	6.16
Number of Teeth	25.0
Calculated VPM (mm)	0.5



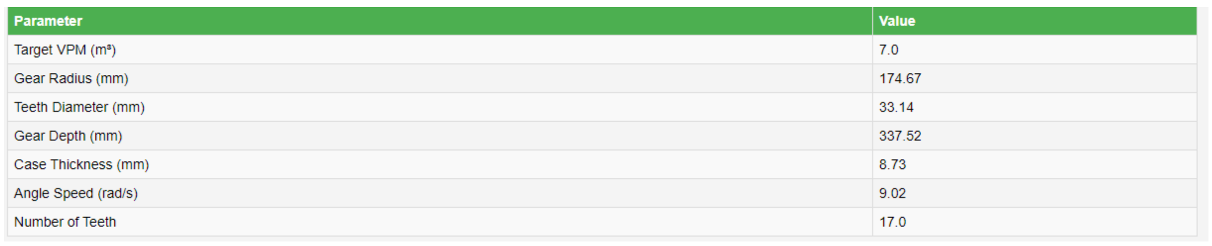
Example 2



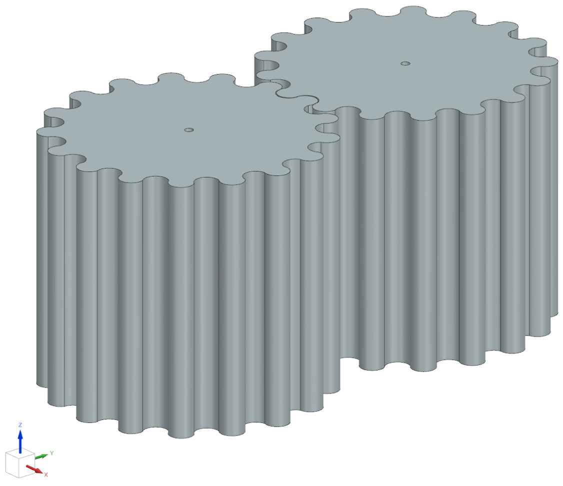
Parameter	Value
Target VPM (m^3)	60.0
Gear Radius (mm)	433.56
Teeth Diameter (mm)	108.39
Gear Depth (mm)	867.11
Case Thickness (mm)	21.68
Angle Speed (rad/s)	8.66
Number of Teeth	13.0
Calculated VPM (mm)	60.0008



Example 3



Parameter	Value
Target VPM (m^3)	7.0
Gear Radius (mm)	174.67
Teeth Diameter (mm)	33.14
Gear Depth (mm)	337.52
Case Thickness (mm)	8.73
Angle Speed (rad/s)	9.02
Number of Teeth	17.0



Group member roles

We have worked together in practice sessions and when we had time on Big Ben or Zevs, but also separate at home by using a github repo and different branches. Martin did the setup of Fuseki and the Flask app. After that Martin did most of the functionality connected to queries and knowledge base. Einar did most of the routing, interface and HTML tasks. In the report we did 50/50.