Article

Controlling Nanoparticle Formulation: A Low-Budget Prototype for the Automation of a Microfluidic Platform

Dominik M. Loy [1,*], Rafał Krzysztoń [2,3], Ulrich Lächelt [1], Joachim O. Rädler [2,3] and Ernst Wagner [1]

[1] Department of Pharmacy, Ludwig-Maximilians-Universität München, Butenandtstr. 5-13, 81377 Munich, Bavaria, Germany; ulrich.laechelt@cup.uni-muenchen.de (U.L.); ernst.wagner@cup.uni-muenchen.de (E.W.);
[2] Faculty of Physics, Ludwig-Maximilians-Universität München, Geschwister-Scholl-Platz 1, 80539 Munich, Bavaria, Germany; rafal.krzyszton@stonybrook.edu (R.K.); raedler@lmu.de (J.O.R.)
[3] Graduate School of Quantitative Biosciences (QBM), Ludwig-Maximilians-Universität München, Geschwister-Scholl-Platz 1, 80539 Munich, Bavaria, Germany
* Correspondence: dominik.loy@cup.uni-muenchen.de

Supplemental Information

1. Formulation module

1.1. Materials

Table 1. Materials formulation module.

Material	Source
Biopsy puncher (d_{inner} = 0.96 mm; d_{outer} = 1.26 mm)	World precision instruments[1]
Fluidmedic polyethylene tube (d_{inner} = 0.38 mm; d_{outer} = 1.09 mm, $thickness_{wall}$ = 0.35 mm)	ProLiquid[2]
Object slide 76x26x1.0 mm	Plano[3]
Object slide 76x50x1.0 mm	Plano[3]
Sylgard 184 polydimethylsiloxane (PDMS) silicon elastomer base & crosslinker	Dow Corning[4]

[1] World Precision Instruments, 175 Sarasota Center Blvd. Sarasota, FL 34240, USA. [2] ProLiquid GmbH, Heiligenbreite 19, 88662 Überlingen. [3] Plano GmbH, Ernst-Befort-Straße 12, 35578 Wetzlar, Germany. [4] Dow Corning GmbH, Rheingaustrasse 34, 65201 Wiesbaden, Germany

1.2. Schematics

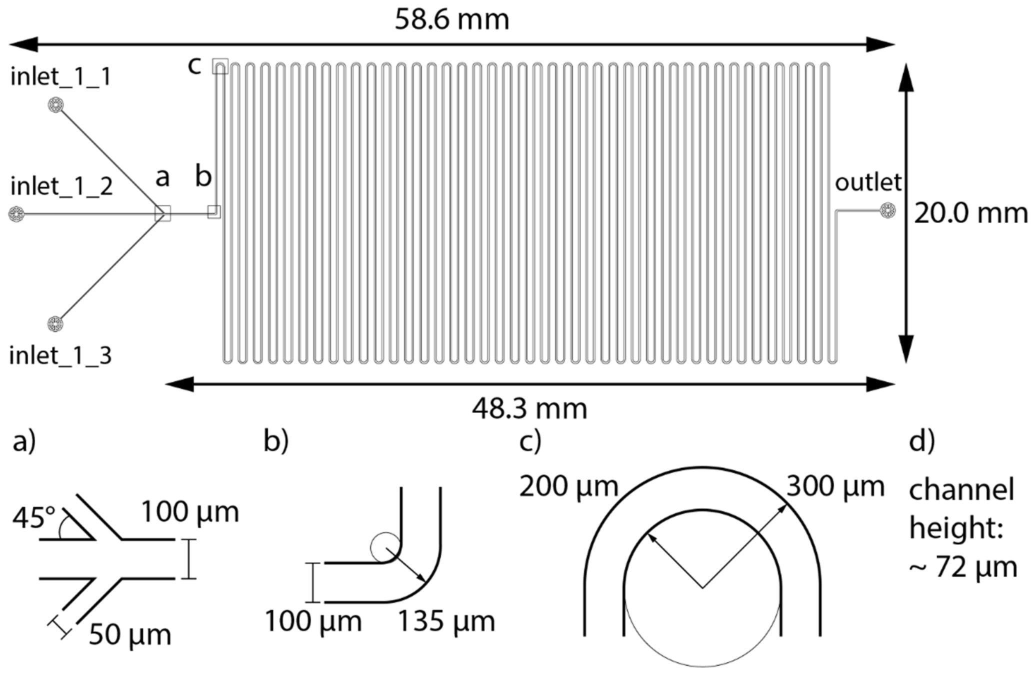

Figure S1: Channel design of the single meander channel.
Circles on the left represent inlets, a circle on the right an outlet. Liquids are pumped from left to right. The inserts a, b, and c present the details of the regions marked with squares in the channel sketch. Modified from Loy et al., 2019, https://doi.org/10.7717/peerj-matsci.1/supp-11

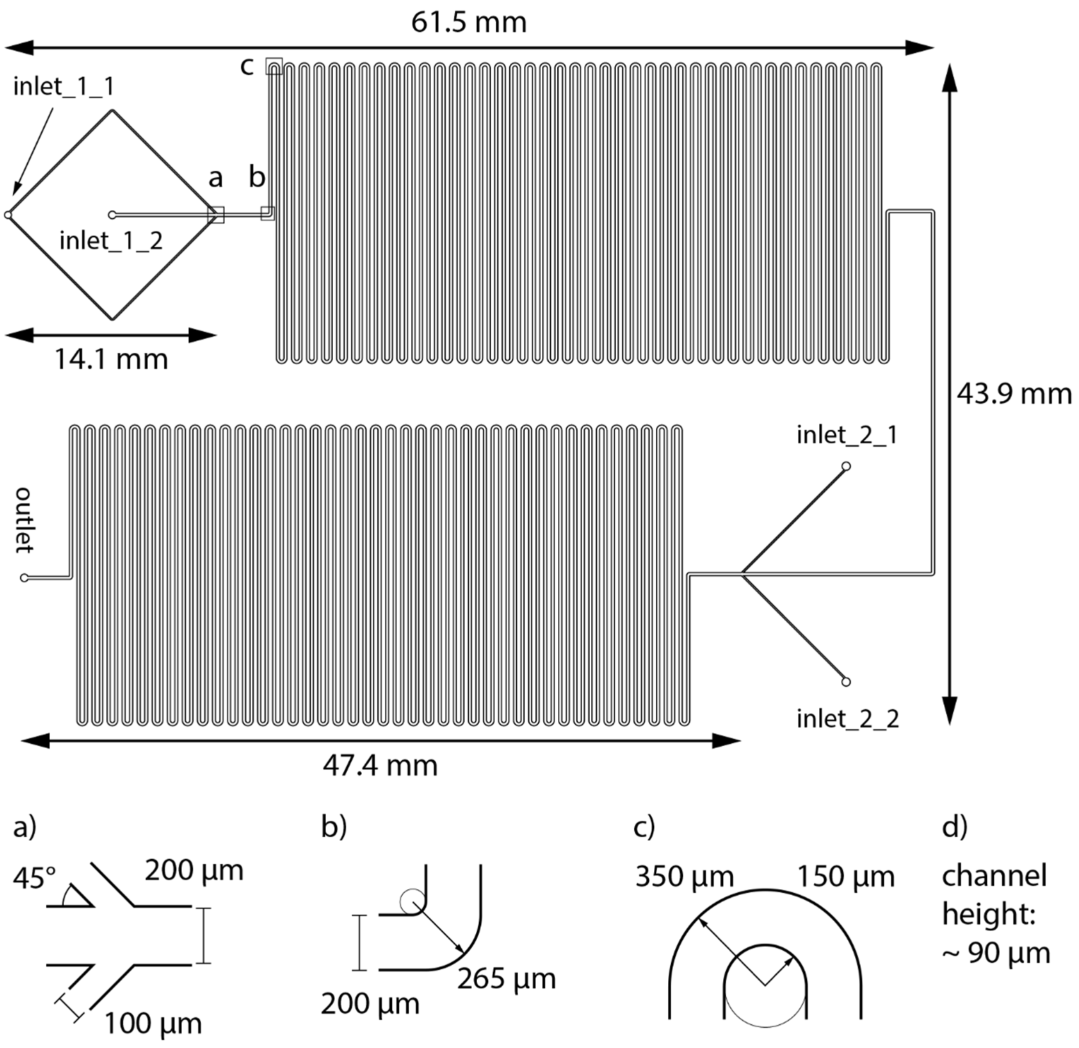

Figure S2: Channel design of the double meander channel.
Circles represent inlets, except the circle on the bottom of the left side, which is an outlet. Liquids are pumped from top left to bottom left. The inserts a, b, and c present the details of the regions marked with squares in the channel sketch. Modified from Loy et al., 2019, https://doi.org/10.7717/peerj-matsci.1/supp-12

2. Control module

2.1. Materials

Table S2. Materials control module

Material	Source
Jumper wires, JKMF40, JKFF40 Makerfactory	Conrad[1]
Raspberry Pi model 3B	Almost Anything Ltd[2]
TECHly USB serial wire (USB 2.0 - RS232)	Conrad[1]
Transcend TS16GUSDHC10E Class 10 microSDHC 16GB	Transcend Information, Inc.[4]
Universal Power Supply RPI-012	Pimoroni Ltd.[3]

[1] Conrad Electronic SE, Klaus-Conrad-Str. 1, 92240 Hirschau, Germany. [2] Thornaby Cecil Avenue, Salisbury, Wiltshire, Great Britain. [3] 2 Manton Street, Sheffield, S2 4BA, United Kingdom.
[4] Flughafenstraße 52b (Airport-Center), 22335 Hamburg, Germany

2.2. Software

Table S3. Software control module

Software	Version
PuTTY	0.71
Python	3.7.3 (Van Rossum & Drake Jr, 2009)
Python package: pySerial	3.4 (Liechti, 2017)
Python package: RPi.GPIO	0.7.0 (Croston, 2019)
Raspbian	Raspbian GNU/Linux 9 (stretch)
WinSCP	5.15.5 (Build 9925)

3. Feeding Module

3.1. Materials

Table S4. Materials feeding module

Material	Source
Needles: NDL ga27, 90 mm, pst4	Hamilton[1]
Syringe 1 ml 1001 TLL, d_{inner} = 4.61 mm,	Hamilton[1]
Syringe 100 µl 1710 TLL-XL, d_{inner} = 1.46 mm	Hamilton[1]
Syringe 500 µl 1750 TLL-XL, d_{inner} = 3.26 mm	Hamilton[1]
Syringe pump *LA*-120	Landgraf[2]
Syringe pump *LA*-122	Landgraf[2]
Syringe pump *LA*-160	Landgraf[2]

[1] Hamilton Bonaduz AG, Bonaduz, Switzerland. [2] Landgraf Laborsysteme HLL GmbH, Langenhagen, Germany.

3.2. Description of the python modules

The software for controlling the syringe pumps consists of seven modules: ‘channels.py’, ‘syringes.py’, ‘Module_pumps.py’, ‘setup.py’, ‘ramping_class.py’, ‘mixing_class.py’, and ‘main.py’. The UML class diagram showing the structure of the software and indicating the relations between all classes is shown in **Figure S3**.

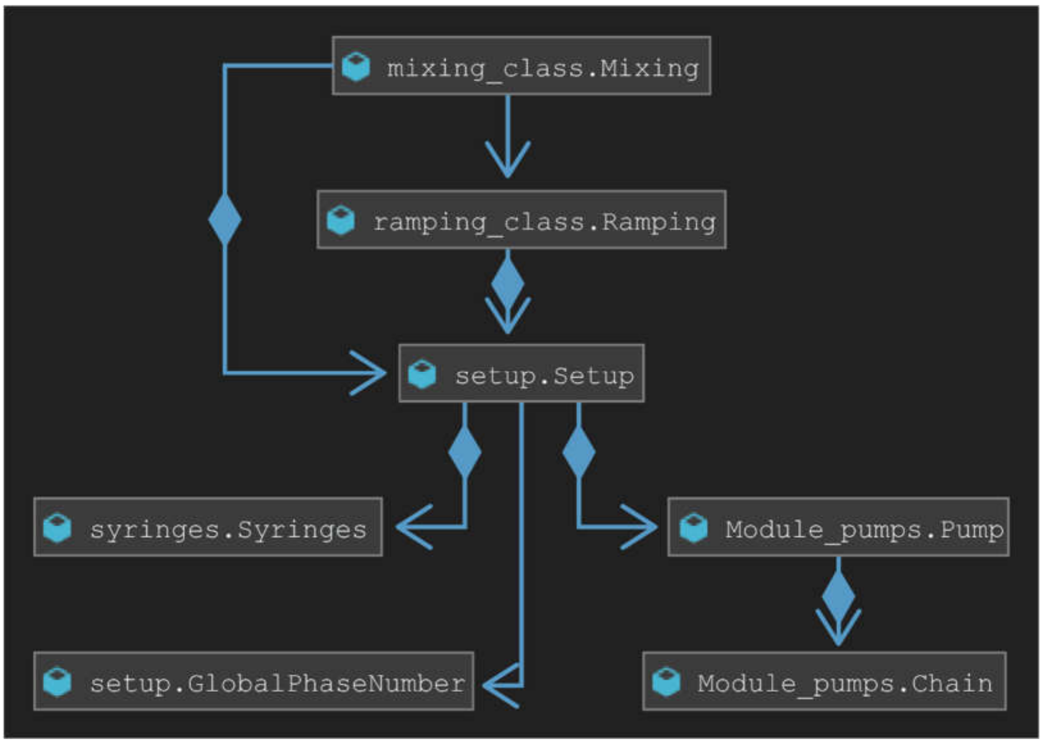

Figure S3: UML class diagram of the control software of the syringe pumps.
Each box represents a class. The name of the class is compounded from the module's name, and the class's name separated by a dot. Each box lists all functions from the respective class. An arrow indicates that one class can access the other. A diamond indicates dependence: for example, class 'Pump' cannot exist without class 'Chain'.

The workflow of this program is straightforward (**Figure S4**). First, either the module 'main.py' for manual parameter input or the module 'main_[…]_automated.py' for automated parameter input needs to be customized to the specific experiment. Second, the user executes the program (**Figure S4**, first orange box), and the software will either ask for parameter input during runtime (manual approach, orange boxes in **Figure S4**), or execute the program automatically (automated approach).
The following paragraphs describe the functions of each module.

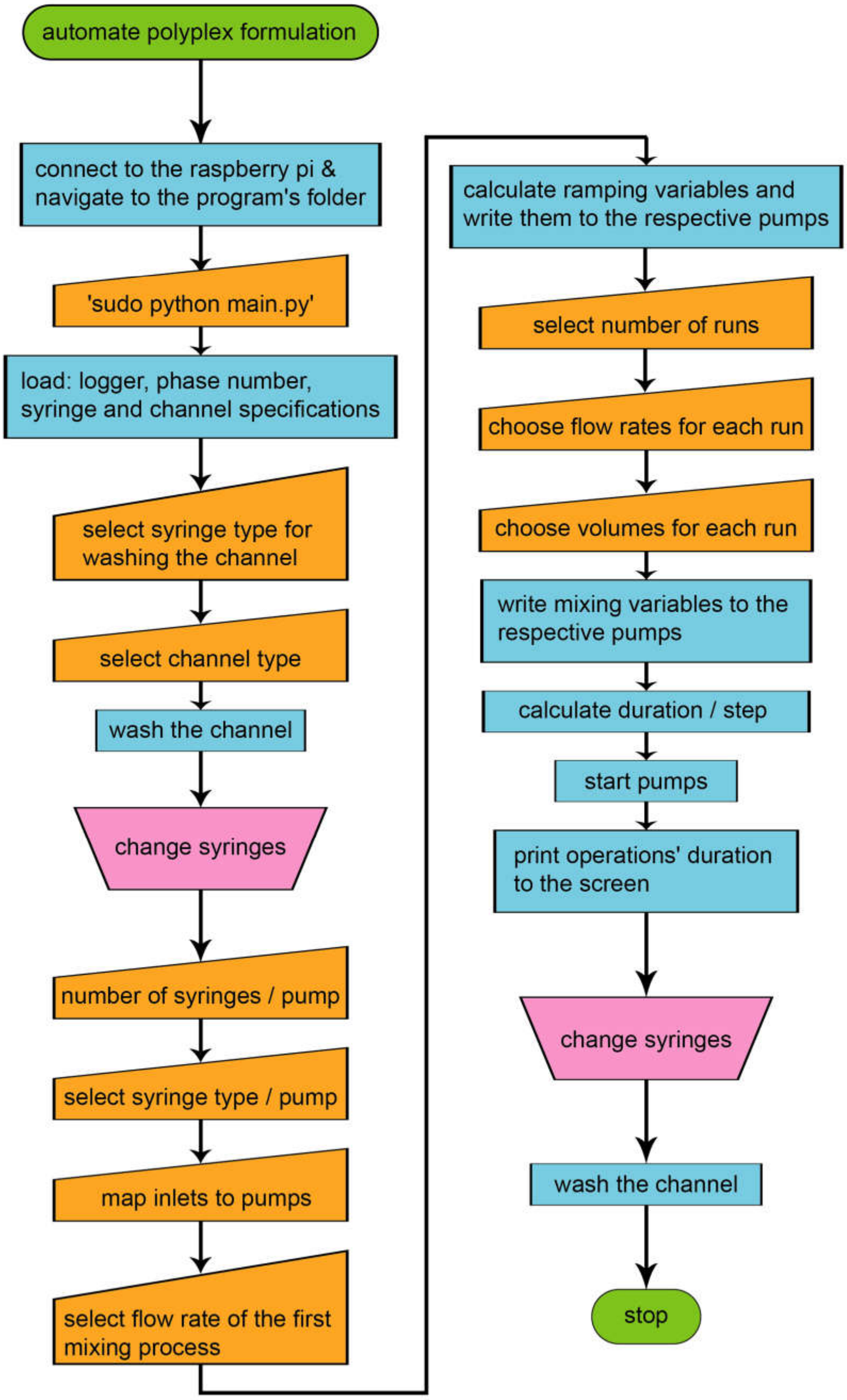
automate polyplex formulation
connect to the raspberry pi & navigate to the program's folder
'sudo python main.py'
load: logger, phase number, syringe and channel specifications
select syringe type for washing the channel
select channel type
wash the channel
change syringes
number of syringes / pump
select syringe type / pump
map inlets to pumps
select flow rate of the first mixing process
calculate ramping variables and write them to the respective pumps
select number of runs
choose flow rates for each run
choose volumes for each run
write mixing variables to the respective pumps
calculate duration / step
start pumps
print operations' duration to the screen
change syringes
wash the channel
stop

Figure S4: Flowchart describing the automation process of polyplex formulations. Green ellipses signify the beginning and end of the process. Blue rectangles denote a process executed by the program or by the user. Orange non-symmetrical parallelograms denote user input. Pink symmetrical parallelograms indicate manual operations.

3.2.1.Module: channels.py

The module 'channels.py' holds the class 'Channel()' that reads the specifications of any channel from a *.txt file and stores them in variables. This approach guarantees the accuracy of the specifications' data and makes adding or adjusting values a matter of changing a simple text file. The file needs to be stored in the same folder as the module. A set of regular expressions is used to extract the relevant information from the file enabling the simple implementation of additional channel designs to the program. The text files for the single or double meander channel can serve as templates for custom specification files. It is important to transfer the names of the variables from the template file to the new specifications file when new channels are added since the regular expressions recognizes the variable names and write their value to the respective variables of the program. An excerpt from the '_set_from_spec_file()' function with an example of a regular expression is shown in **Code 1**.
Additionally, the class defines the functions to calculate the volume of any section of the target channel from the variables.

```
def _set_from_spec_file(self, filename):
    """ This function opens the file specified in filename. If/elif
    statements are used to detect keywords. If a keyword is detected, regex
    is used to extract the desired information. The information is stored
    in variables defined in __init__.
    """

    with open(filename, "r") as file:  # file is closed automatically when
                                         scope is exited.
        for line in file:
            if "inlets_number" in line:
                digit = [float(s.replace(",", ".")) for s in line.split()
                        if re.findall(r'\d+\.*\d*', s)]
                self.inlets_number = digit[0]
[…]
```

Code 1: Excerpt from the '_set_from_spec_file()' function.
Lines 108 – 119 from the module 'channels.py'. The function opens the file specified in filename and searches for keywords in every line (here: 'inlets_number'). When a keyword is detected, the desired information is extracted using regular expressions and it is stored in the target variable (here: the number of inlets of the channel is extracted and stored in 'self.inlets_number').

3.2.2. Module: syringes.py

The module 'syringes.py' holds the class 'Syringes()' and maps each syringe to its inner diameter inside a dictionary. In order to enable the implementation of additional syringes, it defines the function 'import_syringes()' to add specifications of new syringes to the program. Additionally, new syringes can be added permanently to the program by customizing the dictionary in the 'syringes.py' module which maps the name of the syringes to their diameter in mm.

3.2.3. Module: Module_pumps.py

The module 'Module_pumps.py' holds two classes and the logging function 'start_logging()'. **Code 2** shows the last lines of the logging function. In this excerpt, two separate loggers for the pumps and the collector are instantiated. The function of the loggers is assured by writing "started" to the screen and the log file. The logger writes all events with their respective timestamp to a *.txt file and saves the file in the folder 'logs'. When used for the first time, this folder is created in the same directory as the module 'Module_pumps.py'. Additionally, the module contains the class 'Chain(serial.Serial)' which enables the initialization of the serial connection to the syringe pumps, and the class

'Pump()' which defines all functions to control the basic parameters of each pump, for example, functions for setting the pumping rate or the volume to be dispensed. If a pump with a different command structure is to be used, all commands in this module which are sent to the pump must be adjusted. First, the dictionary holding the units for rates and volumes (self.units_dict) must be updated. Second, all occurrences of 'self.serialcon.write()' must be reviewed to find deviations from the required syntax.

```
[...]
# Define loggers which represent areas in the application:
logger_pump = logging.getLogger('pump')
logger_collector = logging.getLogger('collector')
# Confirm the function of the loggers by printing "started" to the console
# and to the log file.
logger_pump.info("started")
logger_collector.info("started")
return logger_pump, logger_collector
```

Code 2: Excerpt from the initialization of the logger function.
Lines 47 – 54 from the module 'Module_pumps.py'. The appearance of 'logger_pump' and 'logger_collector' at any point in the code will write the string between the brackets to the log file (always) and to the screen (except the argument '.debug' is given).

3.2.4. Module: setup.py

The module 'setup.py' holds two classes and the 'countdown()' function. The countdown function prints the remaining time of the current operation to the screen. The code of the function is shown in **Code 3**. The function takes two inputs: the time of the operation in seconds and the name of the operation. The output is the remaining time in minutes together with the name of the operation, e.g. "operation: 10:22".

```
def countdown(t, name):
    """ This function takes two inputs: t in seconds (float or int) and any
    string as name. The time is converted to minutes and seconds and every
    second the name and the time (dd:dd) is printed to the screen
    effectively counting down to zero.
    """
    t = round(t)
    while t >= 0:
        mins, secs = divmod(t, 60)
        timeformat = '{:02d}:{:02d}'.format(mins, secs)
        print("{}: {}".format(name, timeformat), end= '\r')
        time.sleep(1)
        t -= 1
    print("\n")
```

Code 3: countdown() function.
Lines 7 – 20 from the module 'setup.py'. Inputs are t in seconds and any string for a name. When the function is executed, the name is printed next to the remaining time in minutes + seconds (name: 10:22). The function terminates when the time reaches 00:00.

The first class in the 'setup.py' module is called 'GlobalPhaseNumber()' which defines the functions that control the global phase numbers. This number is used to serialize events on the syringe pumps. It contains two classmethods that are used independently of the current scope of any function to assign numbers to individual steps. This class ensures sequence consistency over all steps. The class is shown in **Code 4**.

```
class GlobalPhaseNumber(object):
    """ This class holds two classmethods to control the phase number.
    'next' increases the phase number by +1, while 'reset' resets it to 0.
    This class is used to assign a phase number to every event, creating a
    defined sequence of steps.
    """
    curr_phn = 1

    @classmethod
    def next(cls):
        cls.curr_phn += 1
        return cls.curr_phn - 1

    @classmethod
    def reset(cls):
        cls.curr_phn = 0
```

Code 4: ‚GlobalPhaseNumber()' class.
Lines 22 – 36 from the module 'setup.py'. This class holds two classmethods to control the phase number. 'next' increases the phase number by +1, while 'reset' resets it to 0. This class is used to assign a phase number to every event, creating a defined sequence of steps.

The second class in the 'setup.py' module is called 'Setup()'. It combines the functions and variables from 'syringes.py', 'channels.py', and 'Module_pumps.py' to enable the connection to the pumps and to select the utilized channel and syringes. Connection to each pump is established with functions from the 'Module_pumps.py' module, and the status of each pump is printed to the screen during instantiation of this class (**Code 5**). The washing function from this module is used to prepare the channel for the intended experiment. This function is based on two assumptions: all active pumps are used for washing and that the same type of syringes is connected to all inlets of the channel. Subsequently, the channel is washed with twice its volume and with the combined flowrates equaling the maximal flow rate defined in the variable 'self.max_flowrate' from the class 'Setup()'. The countdown function (**Code 3**) is used to print the washing sequence's remaining time to the screen.

```
class Setup(object):
    """ This class holds all functions and variables related to the setup
     of the micro mixer. Upon instantiation, it creates an instance of the
     Chain class from 'Module_pumps.py' and from the Syringes Class from
     the module 'syringes.py'. Afterwards, each pump is contacted and
     their status (active / inactive) is stored in a variable.
     The functions in this class are used to select the utilized channel
     and syringes and to wash the setup.
     """
    def __init__(self, pumps):
[...]
        self.max_flowrate = 1500  # ul/h
[...]
        # get the information which pumps are active
        try:
            self.LA120 = p.Pump(self.chain, str(sorted(pumps)[0]),
                                str(pumps[sorted(pumps)[0]]))
            self.pumps_active["LA120"] = True
            self.dict_pump_instances["LA120"] = self.LA120

        except p.PumpError:
            p.logger_pump.info("{} is not responding at address
                               {}.".format(sorted(pumps)[0],
                               pumps[sorted(pumps)[0]]))
[...]
```

Code 5: Excerpt from the 'Setup()' class.
Lines 39 – 47, 54, and 57 – 64 from the module 'setup.py'. Upon instantiation of the class, the '__init__()' function is called automatically. Here, the variable 'self.max_flowrate' is set to 1500, and the connection to the pump LA120 is established. If the pump LA120 were offline, the logger would print 'LA120 is not responding at address 01' to the screen and to the log file.

The exact configuration of the setup, for example, the mapping of inlets to pumps, is provided to the program by the user. To avoid unnecessary errors, the program checks each input for plausibility before it is committed to a variable. Simple checks verify the format of the input, for example, flow rates and volumes must be numbers, while more elaborate checks safeguard the integrity of the device by ensuring that the maximum total flow rate or the maximum volume of a syringe are not exceeded. **Code 6**, for example, shows the routine for checking if the number of inlets connected to each syringe pump does not exceed the number of syringes each pump can hold.

```
def check_connections(self):
    """Checks if the number of syringes matches the number of inlets."""
    check_LA120 = sum(1 for x in self.dict_inlets_pumps.values() if
                      x == "LA120")
[...]
    if check_LA120 > self.pump_max_syr["LA120"]:
        print("""Pump LA120 cannot hold {} syringes.
              Please repeat the selection process.""".format(check_LA120))
        self.tubing_connections()
[...]
```

Code 6: Excerpt from the 'check_connections()' function.
Lines 207 – 223 from the module 'ramping_class.py'. The function counts the occurrences of the name of the pump (e.g. "LA120") in the dictionary 'dict_inlets_pumps'. This dictionary maps the inlets of the channel to the pumps as specified by the user. If a pump is mapped to more inlets than it has channels, a message is printed to the screen, and the mapping process is repeated.

Another way to avoid unnecessary errors is the confirmation of selections. The logging function described in **Code 2** confirms every selection by simultaneously printing it to the screen and writing it to the log file. A typical line of code that confirms the selection of a flow rate is shown in **Code 7**. If the automated approach is chosen, only the result of the function is printed. If the manual approach is chosen, then the call to the 'rate()' function from the 'mixing_class.py' module prints a question to the screen depending on the number of runs selected previously (e.g., 'What is the flow rate for run 2 for pump LA120?') and the program waits for the input of the user. The input is converted to a floating-point number (float), if possible, and the selection is confirmed by printing it to the screen (e.g., 'Run 2: LA120's rate is 500').

```
def rate(self, pumps_active, **kwargs):
    """ This function asks the user to provide rates for each run. The
    rates' unit is selected once for all subsequent runs on this pump.
    Alternatively, the rates and the respective unit can be passed directly
    to the function via the kwargs. The names of the arguments should be
    <name_of_pump>_rate for rates and <name_of_pump>_unit for units. Rates
    must be stored in a list. All active pumps must be used.
    Example: LA120_rates = [120,140,160], LA160_rates = [1200, 1400, 1600],
    LA120_unit = 'ul/h', LA160_unit = 'ul/h'.
    """
[...]
     print("What is the flow rate for run {} for pump
           {}?".format(i+1, sorted(pumps_active)[0]))
           rate = input("> ").replace(",", ".")
           try:
               self.rates_LA120.append(float(rate))
               p.logger_pump.info("Run {}: {}'s rate is {}.".format(i+1,
                                   sorted(pumps_active)[0],
                                   self.rates_LA120[-1]))
           except ValueError:
               print("Please choose a number.")
               return self.rate(pumps_active)
[...]
```

Code 7: Excerpt from the 'rate()' function.
Lines 152 – 359 from the module 'mixing_class.py'. Every time the user assigns a flow rate to a pump the function 'rate()' is called. If the automated approach is chosen, the function gets its parameters via the **kwargs. If the manual approach is chosen, the function asks the user for the respective flow rates and units for each pump. In both cases, the input is validated and stored in a separate list for each pump, e.g., 'rates_LA120' and the result is printed to the screen and the log.

After the setup of the device and the washing of the channel have been completed, the ramping and the mixing process must be defined.

3.2.5. Module: ramping_class.py

The module 'ramping_class.py' holds two classes. The class 'Ramping()' defines all functions related to bringing the reactants to the mixing zone at the intended point in time. First, it reads the parameters from the 'Setup()' class. The program assumes that the channel type is not changed after it has been washed, but the syringes can be replaced. Second, functions from the class are used to calculate volumes and rates for the respective pumps and write the ramping sequence (default: ten steps) to the pumps. The rate at which the reactants reach the mixing zone is also the first rate at which the 'mixing_class.py' module mixes reactants. When all information about the upcoming mixing process has been gathered, the ramping protocol is executed to bring all reactants to the mixing zone at the same time. This approach prevents waste of reactants and the formation of unwanted side products due to unintended mixing ratios. Moreover, this approach minimizes the displacement of reactants pumped with relatively small flow rates from the mixing zone by reactants pumped with relatively higher flow rates.

To this end, the flow rate of the first mixing operation is taken as the target flow rate for each pump, and a ten-step descending, or ascending sequence of flow rates and respective volumes is calculated. The direction of the ramping sequence is dependent on the magnitude of the target flow rate relative to the mean flow rate of all flow rates. If the target flow rate of a pump is lower than the mean flow rate, the ramping sequence will be descending and vice versa for target flow rates above the mean. This approach minimizes the displacement of reactants due to pressure gradients. The implementation of the calculation in the program is shown in **Code 8**.

```
def ramping_calc(self):
    """This function calculates the flow rate and volume of each step
    and stores them in a list"""

    for key in self.dict_rates_pumps.keys():
        if "LA120" in key:  # surrogate test: is pump LA120 being used?
            # Decides if ramping to the final flow rate (FR) is done from a
            # higher or lower FR

            if self.dict_rates_pumps[key] > self.total_flowrate /
            sum(self.pump_configuration_n.values()):
                self.rates_LA120.append(self.total_flowrate * 0.25)

                while len(self.rates_LA120) < self.steps:
                    self.rates_LA120.append(round(self.rates_LA120[-1]
                    + (self.dict_rates_pumps[key]
                    - self.rates_LA120[0])/9, 3))
            else:
                if self.dict_rates_pumps[key] <= self.total_flowrate /
                sum(self.pump_configuration_n.values()):
                    self.rates_LA120.append(self.mean_flowrate * 2
                    - self.dict_rates_pumps[key])

                    while len(self.rates_LA120) < self.steps:
                        self.rates_LA120.append(round(self.rates_LA120[-1]
                        + (self.dict_rates_pumps[key]
                        - self.rates_LA120[0])/9, 3))

            p.logger_pump.debug("Ramping rates LA120: {}".format(
                                ",".join(str(x) for x in self.rates_LA120))
                                )
[...]
```

Code 8: Excerpt from the 'ramping_calc()' function
Lines 418 – 435 from the module 'ramping_class.py'. In this excerpt, the calculation of the flow rates for the pump LA120's ramping process is shown. 'dict_rates_pumps' holds the first flow rate of the mixing process mapped to the respective pump. This dictionary is used to decide if a ramping sequence must be calculated for this pump (i.e., if the pump is active). The next if-clause decides if the target flow rate of the pump LA120 is lower or higher than the mean flow rate of all active pumps. The while-clause nested inside the if-clause calculates the respective sequence of flowrates and stores them in the list 'rates_LA120'. At the end of this function, the sequence of rates is written to the log file without printing it to the screen by the logger function.

The class 'EmptyClass()' is used to illustrate that the named arguments 'LA120'. 'LA122', and 'LA160' in the function 'writing()' from the 'Ramping()' class expect an instance of the respective pump from the class ‘Pump()’ from the module ‘Module_pumps.py’. If the default ‘EmptyClass()’ is passed to the named argument, an error message detailing the problem is printed to the screen when the function is called.

The mixing operations will start seamlessly when the ramping process has finished. The countdown printed to the screen will follow the individual steps and inform the user when to start gathering the product from the outlet and when to discard the product (e.g., because of the overlap volume between mixing operations).

3.2.6. Module: mixing_class.py

The module ‘mixing_class.py’ holds the class ‘Mixing()’ which defines the functions to write the mixing protocol to the pumps. It also has the capability to purge the product from the mixing zone after the last mixing step was executed in order to reduce the waste of educts. To be precise, the program expects the number of separate mixing operations and their parameters, how much total overlap volume between fraction will be given, and the number of pumps which will purge the channel.

The overlap volume is chosen once by the user. The program calculates the relative overlap volume for each pump, depending on the volume the pump is pumping relative to the total volume. The calculated volumes are then inserted between all mixing operations to avoid cross-contamination of consecutive products. An excerpt of this function can be inspected in **Code 9**.

The ‘end_process()’ function defines the pumps designated to purging the channel. When the mixing protocol has completed, a fraction of the last formulation remains in the channel section from the mixing zone to the outlet. In order to gather this product as well, the channel needs to be purged. Therefore, we advise to purge the channel after the last run. The purging function calculates the required flow rate, volume, and time for each pump, and appends them to the respective lists holding the values for each pump.

In theory, the program can store an almost unlimited number of steps. However, the internal memory of our pumps is limited to 41 steps or phases (e.g., combinations from flow rates and volumes). Although, usually, the volume of the syringes is the limiting factor for the length of the program.

```
def overlap_calc(self, overlap=None):
    """ This function asks for the overlap between runs and stores them
    in the variable 'self.overlap'. A sensible value is 8 µl. Afterwards,
    it adds volumes and rates in between runs in self.rates_LAxxx und
    self.vol_LAxxx.
    Alternatively, the overlap volume can be passed to the function via
    the kwargs. The name of the argument must be 'overlap'. For example:
    overlap = 8
    """
[...]
    # calculate relative overlap for each pump
    def _relative_overlap_calc(rates_list):
        """
        This function takes the list of the rates from one pump as
        parameter and calculates the relative overlap volume for each pump.
        """
        for j in range(0, len(rates_list)):
            # calculate total flow rate
            flowrate = 0
            if self.rates_LA120:
                flowrate += self.rates_LA120[j] *
                            self.pump_configuration_n["LA120"]
[...]
    # checks, if self.rates_LA120 exists, calculates the necessary
    # overlaps and inserts them into the pump volume's list.
    # Additionally, the name of the overlap is inserted into the
    # variable self.name which is used to inform the user with the
    # countdown function.
    if self.rates_LA120:
        _relative_overlap_calc(self.rates_LA120)
        for i in range(len(self.overlap_LA120)):
            self.name.insert(i*2, "overlap {}".format(i))
        del self.name[0]
[...]
     # insert relative overlap into each self.volume
     for i in range(0, len(self.volumes_LA120)):
         self.volumes_LA120.insert(i*2, self.overlap_LA120[i])
     if self.volumes_LA120:
         # removes first item in the list. Overlap is only necessary
         # between runs.
         del self.volumes_LA120[0]

[...]
    # insert overlap flow rate (rate of the next run) into each self.rate
    for i in range(0, len(self.rates_LA120)):
        self.rates_LA120.insert(i*2, self.rates_LA120[i*2])
        # *2 because with each iteration of the loop the
        # length of rates.LA120 grows
```

Code 9: Excerpt from the 'overlap_calc()' function.
Lines 462 – 579 from the module 'mixing_class.py'. For brevity reasons, only the code for pump LA120 is shown. This function consists of four parts. First, the relative overlap for each pump is calculated. Second, the string 'overlap' and its number is inserted into the list holding the names of all operations. The names in the list are used for the 'countdown()' function. Third, and fourth, the respective overlap volume and flowrate is inserted in the respective lists. The contents of self.rates_LA120 and self.volumes_LA120 are subsequently written to the respective pump by another function and the list 'self.name' is used to inform the user about the name of the current pumping operation.

3.2.7. Modules: main[…].py

The module ‘main.py’ holds the sequence of functions from all the modules above to execute the mixing protocol. It can be customized in order to fit any mixing regime by altering the sequence of functions and their respective arguments.

Arguments to the respective functions can be provided in two ways: either manually during execution of the ‘main.py’ module or automatically by passing the variables directly to the respective functions via the ‘**kwargs’ arguments inside the ‘main.py’ module.

The advantage of the manual approach is the increased flexibility during the experiment since variables can be changed on the fly and the ease of use since no python code must be customized. However, the disadvantage of this approach is the increased time consumption during the experiment due to the many user inputs required. A ‘main.py’ module for the manual approach can be inspected in **Code 10**.

The advantage of the automated approach is the ability to automated complete experiments without requiring a single user input during execution. However, some python knowledge is essential to leverage the full potential of this approach. Two modules – one for the single meander channel and one for the double meander channel – showcasing the potential of the automated approach are available on GitHub as well: ‘main_single_meander_automated.py’ and ‘main_double_meander_automated.py’ (Loy, 2020).

```python
import ramping_class as r_c
import mixing_class as m_c
import setup
# -- Program: --
# Usage: E.g., Formulation of core nanoparticles from two or more
# components.
# All relevant parameters are asked from the user during program execution.

# ------------------------------------------------------------------------
# -- Initialize the pumps and prepare the channel --

# Define Name and address of all pumps:
pumps = {"LA120": "01", "LA122": "02", "LA160": "03"}

# instantiate global phase number
phase_number = setup.GlobalPhaseNumber()

# test which pumps are active, select the channel and the syringes,
# wash the channel
pumps_setup = setup.Setup(pumps)
pumps_setup.select_syringe_washing()
pumps_setup.select_channel()
pumps_setup.washing()

# -- ramp your educts to the mixing zone --
ramping = r_c.Ramping(pumps_setup.channel_used)
ramping.syringes_number(pumps_setup.pumps_active)
ramping.syringes_type(pumps_setup.dict_pump_instances,
pumps_setup.pumps_active)
ramping.tubing_connections()
ramping.first_rate()
ramping.calc_mean_flowrate(pumps_setup.channel)
ramping.ramping_calc()
ramping.writing(phase_number,
                LA120=pumps_setup.dict_pump_instances["LA120"],
                LA122=pumps_setup.dict_pump_instances["LA122"],
                LA160=pumps_setup.dict_pump_instances["LA160"])

# -- mixing: formulate your products --
mixing = m_c.Mixing(ramping_instance=ramping)
mixing.number_of_runs()
mixing.rate(pumps_setup.pumps_active)
mixing.volume(pumps_setup.pumps_active)
mixing.overlap_calc()
mixing.end_process(pumps_setup.channel,
                   pumps_setup.pumps_active)
mixing.writing(pumps_setup.dict_pump_instances,
               pumps_setup.pumps_active,
               phase_number)
mixing.mixing(pumps_setup.channel_used,
              setup.countdown,
              pumps_setup.dict_pump_instances,
              pumps_setup.channel,
              pumps_setup.pumps_active,
              pumps,
              ramping_time=ramping.ramping_time,
              dict_rate_pumps=ramping.dict_rates_pumps)

# -- washing --
pumps_setup.select_syringe_washing()
pumps_setup.select_channel()
pumps_setup.washing()
```

Code 10: 'main.py' module
Lines 1 – 55 from the 'main.py' module. The first three lines import all the other modules to enable access to their functions. Subsequently, the code for setting up the machine ('pumps_setup.*'), ramping the educts to the mixing zone ('ramping.*'), and producing the formulation ('mixing.*') is executed. In the end, another washing step is appended. A flow chart of this module is shown in **Figure S4.**

4. Collection Module

4.1. Materials

Table S5. Materials collection module

Material	Standard / Source
Brass hexagonal bar 50 X 12	
Clamping plate for toothed belt T5	
Dowel pin Ø4 X 25	
L298N H Dual-Bridge DC stepper motor driver controller	Boboshop[2]
Linear ball bearing Ø10 X Ø17	
M2,5x10 screw DIN 963	
M3x10 screw DIN 84	
M3x10 screw DIN 963	
M3x16 screw DIN 912	
M3x8 grub screw	
M4 screw, knurled head, plastic	
M4 x10 screw, plastic	
M4x16 screw DIN 912	
M4x40 screw DIN 912	
M6 washer	
M6x20 screw DIN 912	
NEMA 14 bipolar stepper 1.8 °, 40 Ncm, 1.5 A, 4.2V 35x35x52 mm	Stepper online[3]
NEMA 14 bipolar stepper, 1.8 °, 13.7 Ncm, 1 A, 12 V, 35x35x40 mm	Phidgets Inc.[4]
PChero mechanical end switch	P&Cstore [5]
Revolt universal switching power supply, 1000 mA, 3-12 V	PEARL[6]
Rod bar, stainless steel, Ø10	
Toothed belt disk 21 T5 14/2	Sahlberg[1]
Toothed belt Type AT5, PU, 10, T5 mm, 480 mm, Optibelt alpha torque	Sahlberg[1]
Toothed belt Type AT5, PU, 10, T5 mm, 545 mm, Optibelt alpha torque	Sahlberg[1]

Source indicated unless Standard Part. [1] Sahlberg GmbH, Friedrich-Schüle-Str. 20, 85622 Feldkirchen, Germany. [2] Boboshop, Zhejiang Quxiu Ecommerce Co., Limited, Quzhou Zhejiang 324000, China. [3] Stepper online, OMC corp. Ltd., #7 Zhongke Road, Jiangning District Nanjing City, 211100 China. [4] Phidgets Inc. nit 1 - 6115 4 St SE Calgary AB T2H 2H9 Canada. [5] P&Cstore Brunhuberstr.116, Wasserburg, Germany. [6] PEARL GmbH Pearl-Straße 1-3 79426 Buggingen

4.2. Description of the python modules

The control program for the fraction collector consists of three modules, 'initialize.py', 'move.py' and 'main.py'. 'initialize.py' and 'move.py' contain the code to initialize the fraction collector and to move the dispenser head around. 'main.py' imports both aforementioned modules and executes a customizable sequence of their functions according to the needs of the user. **Figure S5** shows an UML class diagram to illustrate the dependencies between the classes of the modules. A complete list of all classes and functions of the two modules can be found on GitHub (Loy, 2020).

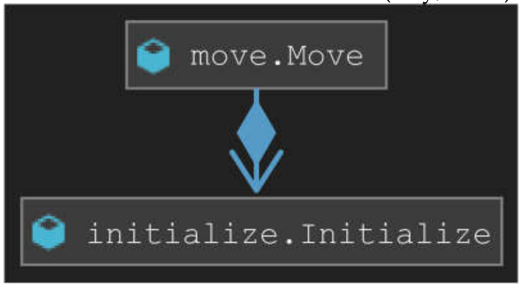

Figure S5: UML class diagram of the fraction collector's control software.
Each box represents a class. The name of the class is compounded from the name of the module, and the name of the class separated by a dot. An arrow indicates that one class has access to the functions of the other class. The diamond indicates dependence: class 'Move' cannot exist without class 'Initialize'.

The workflow of this program is straightforward (**Figure S6**). First, the module 'main.py' needs to be customized to drive the dispenser head to the desired wells at the intended time. Second, the user executes the program (**Figure S6**, orange box), and the software will follow the instructions (**Figure S6**, blue boxes 2 - 5).

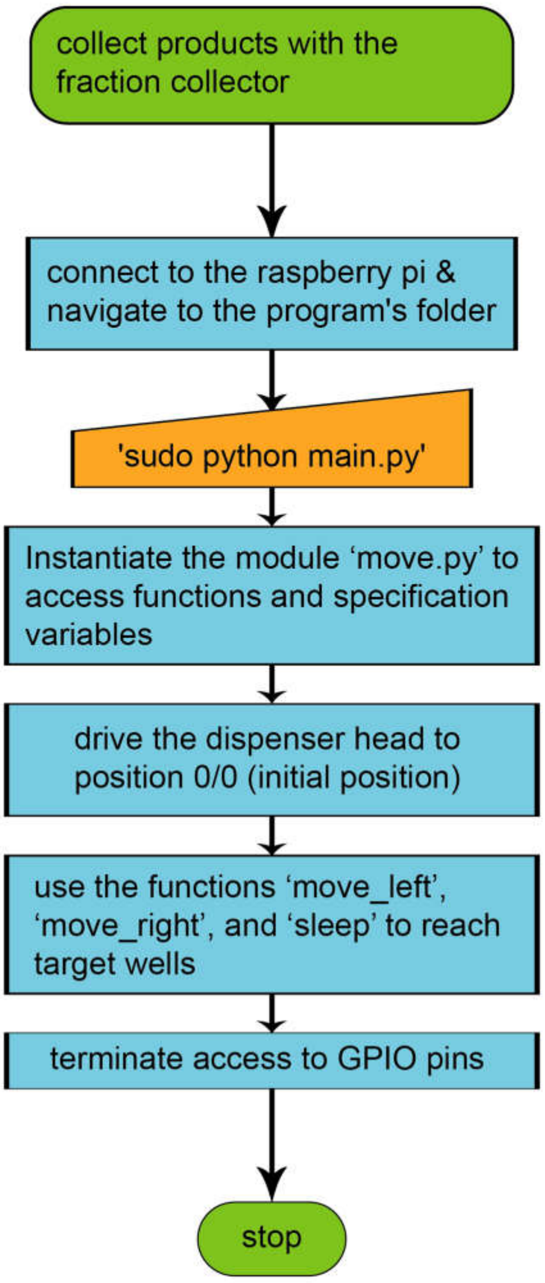
collect products with the fraction collector
connect to the raspberry pi & navigate to the program's folder
'sudo python main.py'
Instantiate the module 'move.py' to access functions and specification variables
drive the dispenser head to position 0/0 (initial position)
use the functions 'move_left', 'move_right', and 'sleep' to reach target wells
terminate access to GPIO pins
stop

Figure S1: Flowchart describing the workflow of the fraction collector. Green ellipses signify the start and beginning of the process. Blue rectangles denote a process executed by the program or by the user. Orange non-symmetrical parallelograms denote user input. User input is possible either during run time or automated before execution.

4.2.1. Module: initialize.py

The class 'Initialize()' from the module 'initialize.py' has three main functions. It maps the GPIO pins to the stepper motors and end switches, it defines the patterns to turn the stepper in the desired direction, and it holds the functions to enable the end switches. Whenever the program is executed, this class must be instantiated first because it lays the groundwork for the subsequent modules. The assignment of the GPIO pins and the definition of the step pattern to turn the motor left or right is shown in **Code 11**.

```
class Initialize(object):
    """ This class initializes the fraction collector. It maps the raspi's
    GPIO pins to the steppers' coils and to the end switches. It defines
    the patterns to turn the steppers in a specific direction and it holds
    the functions to enable the end switches.
    """
    def __init__(self):
        # callable variables used in this Method
[…]
        self.mask_dl = []  # pattern to turn the stepper left
        self.mask_dr = []  # pattern to turn the stepper right
        # map pins to a stepper and its end switch
        self.pins_stepper1 = {"A": 18, "B": 23, "C": 24, "D": 25,
                              "stop_1": 27}
        self.pins_stepper2 = {"A": 5, "B": 6, "C": 13, "D": 26,
                              "stop_2": 17}
[…]
        # define pattern for each step of the steppers
        patterns = [[1, 0, 1, 0], [1, 0, 0, 1], [0, 1, 0, 1], [0, 1, 1, 0]]
        # turn stepper in direction "left"
        self.mask_dl = patterns[0:4]
        # turn stepper in direction "right"
        self.mask_dr = list(reversed(patterns[0:4]))
```

Code 11: Excerpt from the 'Initialize()' class.
Lines 7 – 20 and 63 – 68 from the 'initialize.py' module. Upon instantiation of this class, the pins of stepper motor 1 and stepper motor 2 are stored in 'self.pins_stepper1' and 'self.pins_stepper2' and the pattern that drives the stepper motors left or right is generated and stored in 'self.mask_dl' and 'self.mask_dr'.

4.2.2. Module: move.py

The class 'Move()' from the module 'move.py' holds all parameters related to moving the dispenser head around: defined speeds, the maximum number of steps in x and y direction, and the number of steps between wells of the default 96 well plate. Functions for reaching the starting position and moving the dispenser head left or right are defined in this class as well. As depicted in the UML class diagram (**Figure S5**), it must create an instance of the class 'Initialize()' to access its functions. Important variables for the physical integrity of the collector are shown in **Code 12**.
The variable 'speeds' is a dictionary mapping strings – i.e., names given to certain speeds – to numbers defining the turning rate of the stepper motors. Functions moving a stepper motor in a particular direction use these numbers to determine the time in seconds to wait until the next step is taken (i.e., electrical current is directed to the next coil).
The variable 'maximum_steps_stepper1' or 'maximum_steps_stepper2' defines the boundaries for the target stepper. The program counts the steps a stepper takes in the global variable 'step_counter_stepper_1' and 'step_counter_stepper_2', and it will stop the execution of the program if the value stored in this variable exceeds the maximum number of steps or if it falls below zero. The variables 'steps_stepper_1' and 'steps_stepper_2' hold lists of empirically found distances between wells of – in this case – 96 well plates. Since the distance between wells cannot be covered exactly with full steps, variations of the number of steps between wells were introduced to account for this inaccuracy. If a different container (e.g., a 24 well plate) were to be used with this collector, these two variables would have to be adjusted accordingly.

```
class Move(object):
    """
    Holds all the commands and attributes to move both steppers.
    Move_initial and move_initial2 move both steppers to position 0/0 on
    the x/y grids coordinate system. move_left means that the stepper
    is turning left. The carriage, however, is moving right due to the
    positioning of the steppers. Same is true for move_right.
    """
    def __init__(self):
        # initialize components and wiring
        # step counter to know the exact position of the dispenser head
        global step_counter_stepper_1
        global step_counter_stepper_2
        # speeds: 0.3 sec is very slow -> no stepping errors
        # 0.002 sec is possible without errors
        self.speeds = {"s100": 0.005, "s75": 0.01, "s50": 0.025,
                       "s25":0.05, "s0": 0.1}
        # Instantiate the class 'Initialize' from 'initialize.py' to enable
        # access to its functions and to set up the mapping of the GPIO
        # pins to the steppers and end switches.
        self.system = ini.Initialize()
        step_counter_stepper_1 = 0
        self.maximum_steps_stepper_1 = 260
        step_counter_stepper_2 = 0
        self.maximum_steps_stepper_2 = 340
        self.total_sub_steps = len(self.system.mask_dl)
        # steps for a 96 well plate.
        self.steps_stepper_1 = [43, 28, 27, 27, 28, 27, 28, 28]
        # steps from positions zero to wells A - H.
        self.steps_stepper_2 = [35, 28, 27, 27, 28, 27, 28, 27,
                                27, 28, 27, 28]  # steps from pos. 0 to
                                                 # wells 12 - 1.
[...]
```

Code 12: Excerpt from the 'Move()' class.
Lines 8 – 34 from the 'move.py' module. This class holds all commands and attributes related to moving the dispenser head around. As soon as the class is instantiated, an instance of the class 'Initialize()' from the module 'initialize.py' is created in order to access GPIO pin mappings and functions to control end switches. The 'step_counter_stepper_1 or 2' is used in combination with 'maximum_steps_stepper_1 or 2' to ensure that the dispenser head is only moved inside the boundaries of the collector.

4.2.3. Module: main.py

The module 'main.py' utilizes the classes and functions from the other two modules. It must always start with an instance of 'Move()' from 'move.py.' to enable access to the functions moving the dispenser head around and defining the utilized GPIO pins. Subsequently, the 'move_initial()' function must be called to move the dispenser head to its initial position. Afterwards, any sequence of 'move_left()' and 'move_right()' commands can be programmed. The program counts the number of steps taken and will abort if its predefined maximum number of steps is reached. The function 'sleep()' from the 'time' module can be used to pause the dispenser head at any position. The program should be terminated with a call to 'GPIO.cleanup()' to reset the GPIO assignment. The structure of the 'main.py' module is shown in the **Code 13**. This code drives the dispenser head to its initial position and subsequently dispenses products in the first 24 wells of a 96 well plate following the pattern A1 -> A12 -> B12 -> B1. The dispenser head pauses at each well for three seconds. A video documenting the execution of this code can be found on GitHub (Loy, 2020).

```
import RPi.GPIO as GPIO
import move
from time import *

# Instantiate the class 'Move' from the module 'move.py' to enable access
to functions and variables.
commands = move.Move()

# Move the dispenser head to its initial position x/y = 0/0.
commands.move_initial(commands.speeds["s25"], commands.speeds["s0"])
commands.move_initial2(commands.speeds["s25"], commands.speeds["s0"])

# program: serve wells from a 96 well plate: A1-> A12 -> B12 -> B1
# go to well A1
commands.move_left(2, sum(commands.steps_stepper_2) - 16,
commands.speeds["s50"])  # -16: drives to far.
commands.move_right(1, commands.steps_stepper_1[0], commands.speeds["s50"])

# wait 3s (i.e. the time to collect your sample)
sleep(3)

# for-clause. Iterates over the elements of the list 'steps_stepper_2'
# and drives the target steps to the right (A1 -> A12)
for i in list(reversed(commands.steps_stepper_2))[:-1]:
    commands.move_right(2, i, commands.speeds["s50"])
    sleep(3)

# drives stepper 1 target steps to the right (A12 -> B12)
commands.move_right(1, commands.steps_stepper_1[1], commands.speeds["s50"])
sleep(3)

# drives stepper 2 target steps to the left (B12 -> B1)
for i in commands.steps_stepper_2[1:]:
    commands.move_left(2, i, commands.speeds["s50"])
    sleep(3)

# removes access to and any voltage from the GPIO pins.
GPIO.cleanup()
```

Code 13: The 'main.py' module of the fraction collector.
Lines 1 – 35 from the 'main.py' module. The first three lines import all necessary modules to enable access to their functions. Subsequently, the code for setting up the machine ('move.Move()') and moving the dispenser head to its initial position ('move_initial()') is executed. The following lines drive the dispenser head from its initial position to the following 24 wells: A1 -> A12 -> B12 -> B1 with a pause of 3 s at each well. In the end, access to GPIO pins is terminated ('GPIO.cleanup()').

5. Oligomers

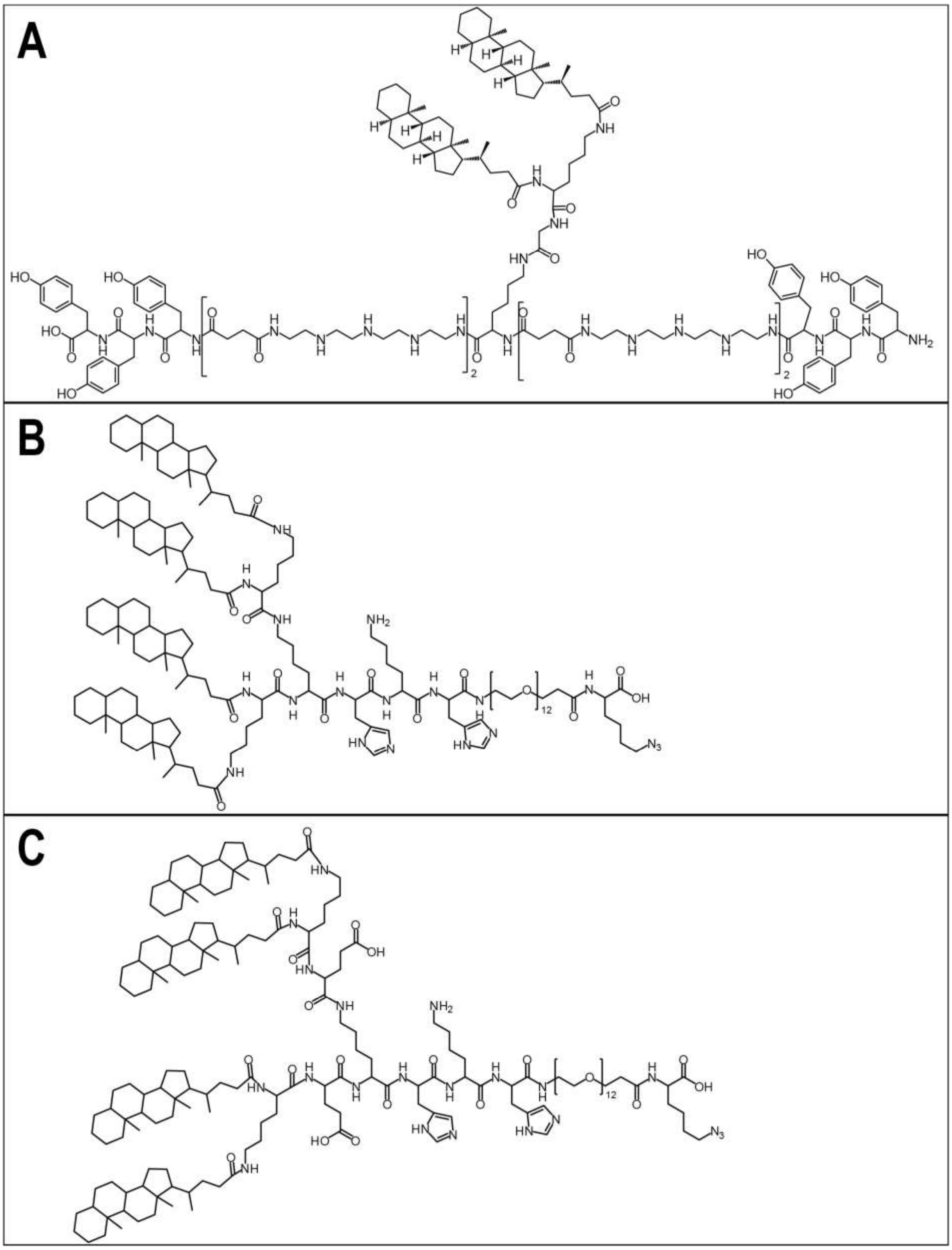

Figure S7: Chemical structures of CO, LPO, LPOE.
(A) CO. Calculated molecular weight: 3081.07 Da (B) LPO. Calculated molecular weight: 2929.16 Da. (C) LPOE. Calculated molecular weight: 3187.39 Da.

6. DLS measurements

Table S6. Solvents used for DLS measurements

Solvent	Dispersant RI	Viscosity [cP]
HBG	1.337	1.037
HBG (4.2 % [V/V] acetone)	1.340	1.119
HBG (8.3 % [V/V] acetone)	1.342	1.188

Note: Refractive indices (RI) and viscosities in centi poise (cP).

7. References

Croston B. 2019.raspberry-gpio-python. *Available at https://sourceforge.net/projects/raspberry-gpio-python/* (accessed on January 18, 2020).

Liechti C. 2017. pySerial. *Available at https://pyserial.readthedocs.io/en/latest/index.html* (accessed on January 7, 2021).

Loy DM. 2020.Dominik Loy on GitHub. *Available at https://github.com/Dominikmloy* (accessed on November 7, 2020).

Loy DM, Klein PM, Krzysztoń R, Lächelt U, Rädler JO, Wagner E. 2019. A microfluidic approach for sequential assembly of siRNA polyplexes with a defined structure-activity relationship. *PeerJ Materials Science* 1:e1. DOI: 10.7717/peerj-matsci.1.

Van Rossum G, Drake Jr FL. 2009. *Python 3 Reference Manual*. CA: CreateSpace. Scotts Valley, CA. 2009 ISBN 1441412697