THE UNIVERSITY *of* EDINBURGH
School of Biological Sciences

MetCloOT2: an automated DNA assembly framework for the Edinburgh Genome Foundry

Student Exam Number:

In partial fulfilment of the requirement for the Degree of Master of Science in ***Synthetic Biology and Biotechnology*** at the University of Edinburgh

2021 / 2022

Name of Dissertation Supervisor: Peter Vegh

Word Count: 7,590

Contents

Abstract

The essential use of novel genetic constructs provides unique behaviours and functionalities to biological systems, aid biological research, and advances the field of biotechnology and synthetic biology. MetClo is a hierarchical DNA assembly method derived from Golden Gate assembly. It uses a vector-set framework for DNA assembly. It uses a single type of type IIS restriction enzyme sites regulated by methylation, and a predetermined overhang set. Multiple standardised DNA assembly methods, including MetClo, have been developed to have consistent design patterns, standardized steps, and predictable results. These traits permit the translation of laboratory protocols into computer aided design (CAD) software and automation script. Biofoundries, such as the Edinburgh Genome Foundry (EGF), are established on engineering principles and consequently provides design and automated build support to normal molecular laboratories that cannot obtain such infrastructures. Such automation platforms include the Opentrons OT2 liquid handling robot, an economic and versatile device capable of conducting molecular biology protocols. To aid in the expansion of the foundry's capabilities and the democratisation of high throughput assembly technology, the MetCloOT2 pipeline was created to modularly fit into the EGF's established DNA assembly system. All novel protocols, hardware or software entering the foundry's system needs to be scrutinised to assure that only vetted operations are used. The MetClo kit was simulated, and its overhang collection was validated using various EGF's CAD software. A MetClo kit specific CAD capable of calculating all reagents and part volumes as well as allocating them positions within the OT2 was developed. It can work independently or in conjugation with a MetClo protocol generator. The generator was developed to streamline and automate the MetClo assembly on the OT2. The software, called MetCloOT2 can be found within its GitHub repository

1. Introduction

1.1. Definition of terms

For clarity throughout the report, a set of terms must be defined. A part is defined as a standardized genetic element or fragment that (1) comply with the standards of a DNA assembly method and (2) will be comprised within a genetic construct (Figure 1). These may include basic genetic components (examples: promoters, genes, terminators) or previously assembled parts that will participate in another iteration of assembly (example: a transcription unit). A construct or assembly refers to the complete annealing of parts held within an assembly vector backbone or not. A final assembly is the last construct within a hierarchy. Donor plasmids will accept, carry, and donate parts while assembly vectors will accept multiple parts to form the assembly.

Type IIS restriction enzymes digest double stranded DNA to produce overhangs or sticky ends (3-4bp). The location within the DNA that will become an overhang will be referred to as a restriction site (RS). An overhang set or adapter set are two sequence specific restriction sites flanking a genetic element (Figure 4).

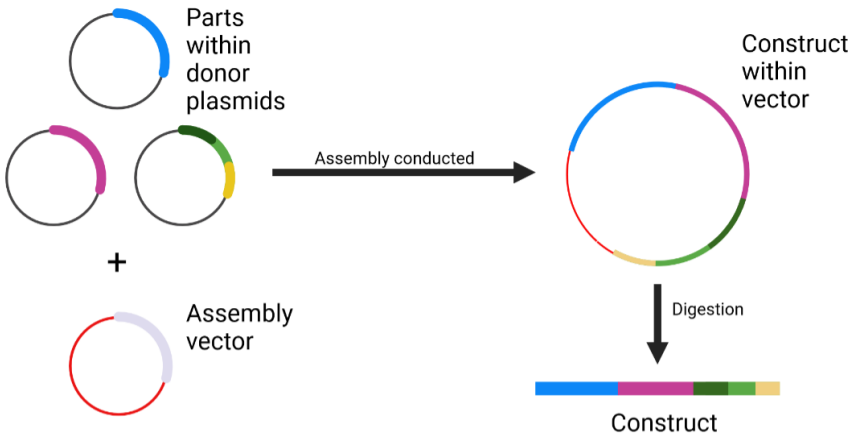


Figure 1 – Simplified restriction-ligation assembly schematic. *Multiple donor plasmids contain a part (blue and pink fragment) or a previously constructed assembly (green, light green & yellow fragment). The assembly vector has a negative selection marker (Gray fragment). The plasmids, vector and an enzyme mix are combined in a one-pot assembly reaction producing the construct held within the assembly vector backbone. The construct can be released by digesting out of the backbone.*

1.2. DNA assembly, a foundational practice within Synthetic Biology

Synthetic Biology's prevalent use of the iterative design-build-test-learn (DBTL) cycle aids researchers to develop and ultimately create novel biological functionalities and systems (Bryant et al., 2022). DNA assembly is a fundamental process within synthetic biology and related molecular biological fields; it would benefit from faster and error-free iterations of the DBTL cycle to comprehensively explore the genetic construct design space (Walsh et al., 2019; Marillonnet & Grützner, 2020; Bryant et al., 2022). The creation of Biofoundries, such as the Edinburgh Genome Foundry (EGF), have assumed the role of high-throughput builders with the implementation of automated procedures, which offer a quick, time-effective, economical, and idempotent mean of assembling constructs for academic and non-academic researchers. Automation can work as well or better than manually performed procedures, enabling hasty progression of research and bio-based engineering.

Large strives have been made in DNA assembly technology, these are continuously developing. Many of these methods employ a strategic engineering approach emphasizing the use and need of standardization, characterization and modularity within their assembly vector framework and genetic parts (Exley et al., 2019; Kang et al., 2022). Despite the advances in the methods, the effort required for plasmid design and their tedious application are regarded as weakly scalable, partially optimized and labour intensive, making these steps rate-limiting within the DBTL cycle (Enghiad et al., 2022). However, many DNA assembly method's standardized architecture and combinatorial assembly approach enables their integration within computer-aided-design (CAD) software and automation which can accelerate the process.

1.3. Biofoundries: The Edinburgh Genome Foundry

Biofoundries are high-throughput resource epicentres for the scientific community. They are a significant investments to synthetic biology that house automation platforms, analytical equipment, software, standardized procedures and expertise generally not available or untenable by common molecular biology laboratories (Ortiz et al., 2017; Holowko et al., 2021). They are governed by curated workflows. A workflow is a

process, that executes repeatable tasks and/or manipulates information, with the intend to increase the speed of production and/or quality of an outcome. These repetitive tasks and manipulations can be performed by computational or automated means, reducing the need of human input.

The EGF specializes on the manufacturing of complex genetic constructs using a highly automated platform to address all four phases of the DBTL cycle. Design begins with the use of their various CADs; CUBA, their open-access suite of web-application tools, and the EGF's GitHub library of open-source software; both of which facilitate and aid the in-silico design and evaluation of assembly projects, intended final constructs, and combinatorial part-library exploration. Once optimized, the project's specifications can be submitted to the EGF's integrated computational and liquid handling infrastructure for the assembly of the DNA constructs; these are then validated with onsite equipment. Otherwise, the assembly generated with the aid of the CADs can be conducted independently by the researcher.

The design of experiments (DoE) is an investigation framework that explores the effect of multiple variables and their interactions; it has the objective of optimization and minimization of experiments over a vast array of factors while maximizing the knowledge outcome or desired biosynthetic system (Casas et al., 2022; Exley et al., 2019). The part repositories create a large combinatorial landscape for DNA assembly, the landscape only increases with larger assemblies, making it more difficult to query numerous combinatorial instances. Manual DoE, especially of modular vector-based assembly methods, is laborious, error-prone and time consuming, delaying the design and automated-build steps. CAD can support the implementation of DoE as it helps users create their biological designs or builds based on set algorithmic rules and logic gates to reduce human involvement (Exley et al., 2019). Multifactorial designs are developed faster, at greater scales, with reduced error and higher consistency. CADs can additionally optimise the methodology used for the build stage of the project from design-stage information (Exley et al., 2019). Methodology optimisation would reduce experimental cost, time, error propensity, and labour.

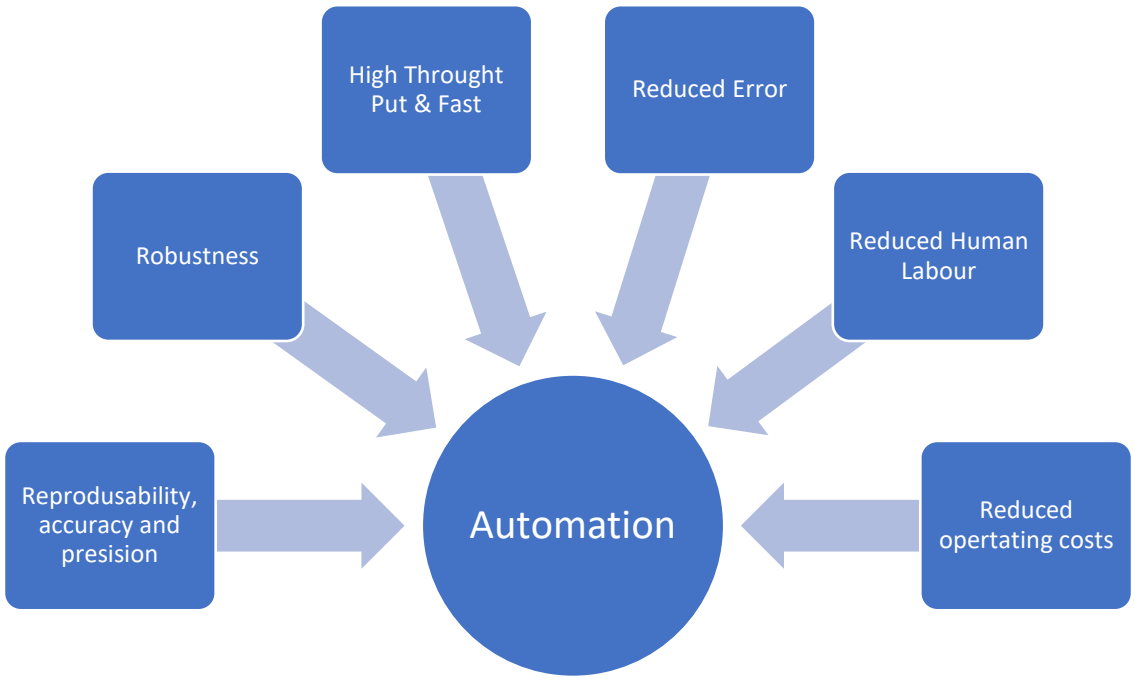


Figure 2 - Benefits of Laboratory Automation
(James et al., 2022; Vazquez-Vilar, Orzaez & Patron, 2018)

Unsurprisingly, the use of automation within biofoundries have become an attractive platform to execute the DNA assembly, as papers consistently report that they outperform in both accuracy, precision and speed compared to human-executed protocols (Figure 2) (James et al., 2022; Vazquez-Vilar, Orzaez & Patron, 2018; Zhang et al., 2017; Walsh et al., 2019; Vasilev et al., 2011; Storch, Haines & Baldwin, 2019; Ortiz et al., 2017; Kanigowska et al., 2016; Chory et al., 2021; Bryant et al., 2022). Ortiz et al. (2017) reports that manual and Opentrons OT-One liquid handling robot (OT1) automatic assembly are equally as efficient (95% assembly success), however the OT1 conducted 96 assemblies in 2 hours compared to the 4 hours with manual-work. Although the OT1 cut the time in half and had equal success rates, it required double the expense. Storch et al. (2019) reports that 88 automatic BASIC assemblies took 1 hour and 30 minutes on the improved Opentrons OT-Two liquid handling robot (OT2), while the same project took 5 hours of manual work. James et al., (2022)'s report that acoustic droplet injection automation of the EMMA assembly method produced 83.3% correct complex assemblies (≥20 parts), while manual execution had a 77.8% success rate. In the same study, assemblies with equal number of parts, but varying final

assembly sizes only saw better automated assembly success rates in sizes >9600 bp (Success rate: automation = 77.7%, manual = 72.2%). Studies reporting on the price of automated assemblies after 2017, were not found, however, it is expected that the price must have dropped since Ortiz et al. 2017's report in accordance with Moore's law and the trend of other technologies used in synthetic biology.

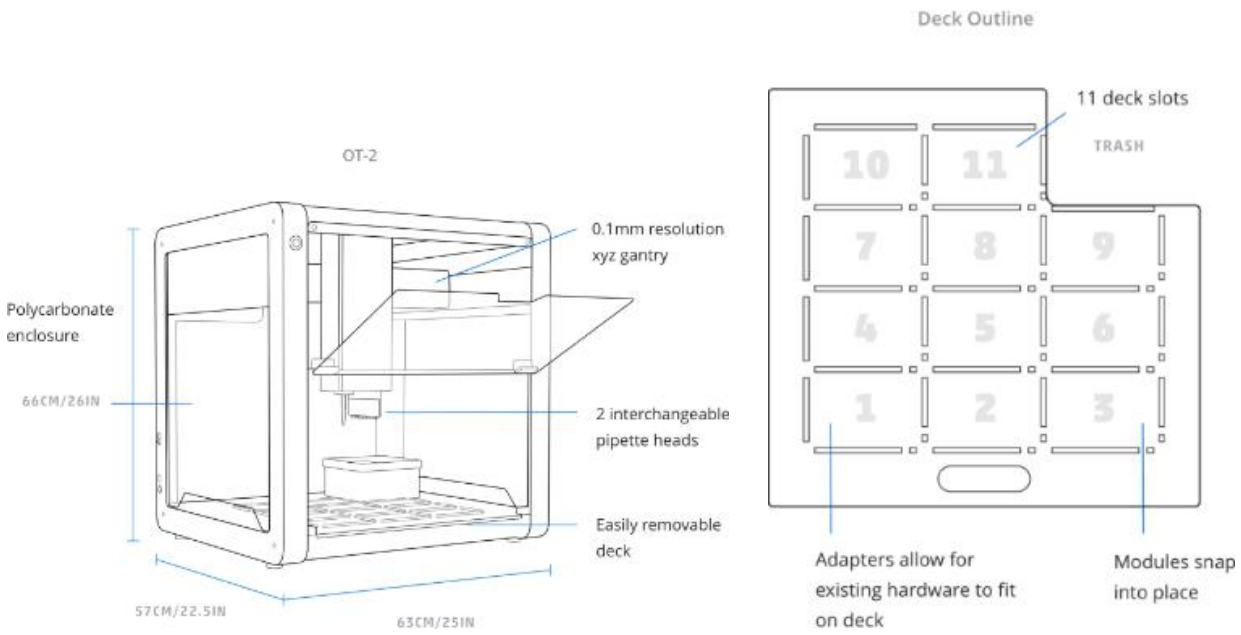
Within the EGF, the automation platforms are generally interoperable with the inhouse software and other equipment, permitting the easy flow of information and experimental processes. However, workflows can range from fully automated, semi-automated, or completely manual, depending on the needs of the project and the capabilities of the foundry. The eventual goal is to allocate full experimental responsibility to the automated and computational platforms, an automation engineers, with extensive molecular biology knowledge, would only be needed to maintain and supervise their functioning (Holowko et al., 2021).

The cost of owning large automation platforms, operating, and maintaining them are beyond the means of many small laboratories, limiting their use. Democratizing the access of larger platforms via the Biofoundries, is one solution, however cost of leasing of the larger expensive platforms may still be out of means. The use of cheaper automation platforms, like the Opentrons OT2 (Figure 3), within the EGF would offer research a more cost-effective option, while benefiting from all the other resources the foundry offers. The OT2 is an affordable open-source liquid handling robot (Opentrons, 2022). Vetted workflows and protocols for a variety of different applications can be found in the Opentrons APIv2 Protocol Library. Personalized workflows can be created by translating human-language protocols to Python code (programming language) (Figure 3B) with their user-friendly Protocol Designer application, on request, or on Opentrons' Python API programming framework for users with coding and wet-lab skills.

Apart from the mentioned ways of obtaining protocols, OT2 users can access public git repositories, such as GitHub, these are platforms where version-controlled files are shared among developers. The EGF is exploring the possibility of adding accessible

liquid handling protocols into their existing GitHub repository already populated with CAD, so these tools can be shared with the community.

A)



B)

```python
from opentrons import protocol_api

metadata = {'apiLevel': '2.13'}

def run(protocol: protocol_api.ProtocolContext):
    plate = protocol.load_labware('corning_96_wellplate_360ul_flat', 1)
    tiprack_1 = protocol.load_labware('opentrons_96_tiprack_300ul', 2)
    p300 = protocol.load_instrument('p300_single', 'right', tip_racks=[tiprack_1])

    p300.transfer(100, plate['A1'], plate['B1'])
```

```
Picking up tip from A1 from Opentrons 96 tip rack 300ul on 2
Aspirating 100ul from A1 of corning 96 flat well plate 360 ul on slot 1 at 500.0ul/sec
Dispensing 100ul into B1 of corning 96 flat well plate 360 ul on slot 1 at 1000.0ul/sec
Dropping tip into A1 of Opentrons fixed trash on 12
```

***Figure 3 - (A) Layout of the OT2's hardware framework**. The OT2 has a 2-pipette mounts for any combination of a single channel and 8-channel pipette. The deck is divided into 11 slots allocated to various modules and equipment necessary to the protocol. A trash bin is incorporated into the deck. The OT2 require to be connected to a computer with the Opentrons App to function (Opentrons, 2022). **(B) Exemplary script on the Python API and picklist generated by Opentrons App.** The protocol transfers 100ul from a 96-well-plate's location A1 to B1 with a 300ul single channel pipette. The Protocol API package was imported to allow communication between the user, the script and the OT2 (Opentrons, 2022). The picklist is a highly specific list of steps generated from the protocol script that instructs the OT2 what actions to perform.*

1.4. Golden Gate Assembly

Type IIS restriction-ligation assembly, otherwise known as Golden Gate, is a single-pot directional hierarchical multiple DNA fragment assembly method (Engler, Kandzia & Marillonnet, 2008). Type IIS restriction enzymes digest outside of their recognition site, producing arbitrary overhangs of 3-4bp. The removal of these sites, along with the negative selection marker, assures the permanent conjunction between the parts. The part's overhangs are typically designed by the user with a set of empirical guidelines to (1) define the order of these parts in the assembly and (2) safeguard mismatched ligation. Part-containing-plasmids and assembly vectors are digested with the restriction enzyme and ligated with the highly efficient T4 DNA ligase. The Golden Gate assembly is popular as it can seamlessly assemble multiple parts within a single-pot reaction, produces small scars, is easily scalable and permits hierarchal construction (Young et al., 2021).

The Golden Gate's assembly mechanism has been used as the bases of many new assembly methods with the purpose of extending and improving its capabilities. These methods include: MoClo (Weber et al., 2011), PODAC (Van Hove et al., 2017), MetClo (Lin & O'Callaghan, 2018), Golden Braid (Sarrion-Perdigones et al., 2011), GreenGate (Lampropoulos et al., 2013), among others. Increased standardization, scale, reliability, idempotency, orthogonality and efficiency are constant goals when developing these new methods.

The need of part domestication, the lack of overhang standardization, and the presence of scars are shortcomings of Golden Gate and, at various degrees, in its improved

iterations (Young et al., 2021). Domestication is the removal of unwanted restriction sites within the DNA fragment by either overlapping-PCR, direct mutagenesis or direct DNA synthesis (Sarrion-Perdigones et al., 2011). Mutation of the fragments may impact the functionality and performance of the desired product. The overhang set design for all of these Golden Gate-based methods are not based on studies but empirical experience, which unreliably guides experimental design, hinders ligation efficiency and averts the discovery of additional overhang sets (Potapov et al., 2018a; Pryor et al., 2020; Sarrion-Perdigones et al., 2011). The lack of method and part standardisation work in tandem, preventing the creation of extensive libraries, limiting the combinatorial design space, and decreasing part exchange within the community (Weber et al., 2011; Young et al., 2021).

The holistic application of Golden Gate-based methods within the field of synthetic biology should be equally considered in their creation, such as their ease of integrating within established automation infrastructure, software tools and databases.

Metclo: Methylase-assisted hierarchical DNA assembly using a single type IIS restriction enzyme

Metclo (Lin & O'Callaghan, 2018) is a one-pot hierarchical DNA assembly method based on Golden Gate. The method parallels MoClo (Weber et al., 2011) in that both their DNA assembly mechanism relies on a vector set framework (Appendix Figure 1), and multiple adapter sets (Figure 5). MetClo recruits the use of in vivo methylation-switching to mediate enzymatic digestion (Figure 6), permitting the use of a single restriction enzyme throughout all levels of assembly, where MoClo utilizes 2-3 enzymes depending on the project's hierarchical demand. DNA methylation is the chemical modification of DNA catalysed by a DNA methyltransferase (Moore, Le & Fan, 2013). The methylation-switch enables the deactivation and activation of the enzyme's recognition site to prevent or allow overhang exposure. The switch-like mediation of these sites adjusts the need for multiple restriction enzymes to one, surmounting other assembly methods like MoClo.

The implementation of MetClo begins with the design of the final construct and the assembly plan. All parts used must comply with MetClo's parts requirements. Parts must have (1) two BsaI recognition sites (BRS) and (2) 'p' and 'q' overhangs flanking the genetic element of interest (G.O.I.); and (3) no BRSs within the genetic element (Figure 4). The arrangement and number of parts within the final assembly guides the vector selection from the Metclo vector kit (Appendix Figure 1) and the number of hierarchical reactions.

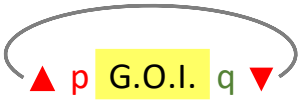


Figure 4 – Metclo part architecture. *The G.O.I. is placed in-between p & q RS and inverse BRSs. Ideally, all parts within the same level of hierarchical assembly, carry the same antibiotic selection marker which differs from that of the donor plasmid or assembly vector.*

The MetClo vector set contains a set of 30 low and 18 high copy vectors (Appendix Figure 1). The vectors can be of three different antibiotic selection markers and have 1 of the 10 adapter sets (Figure 5). These adapters determine the position of the parts. These 48 vectors can be used as both donor and assembly vectors.

Donor plasmids and assembly vectors are prepared by being transformed into a methylase expressing Escherichia coli (E.coli) strain (MES) (Figure 6). The outer-most BRS partially overlaps with a methylase recognition site (MRS), causing BRS methylation and its subsequent inactivation. The internal BRS remains active. The methylated donor plasmid is populated with its respective parts by restriction-ligation as per Lin & O'Callaghan (2018) (Figure 7). These are then transformed into a normal E. coli strain (NS) to reactivate the outer-most BRS. The donor plasmids, an assembly vector, and reaction reagents are combined to form the final assembly by restriction ligation. The assembly vector containing the final construct can participate in subsequent assembly reactions or be used in further applications.

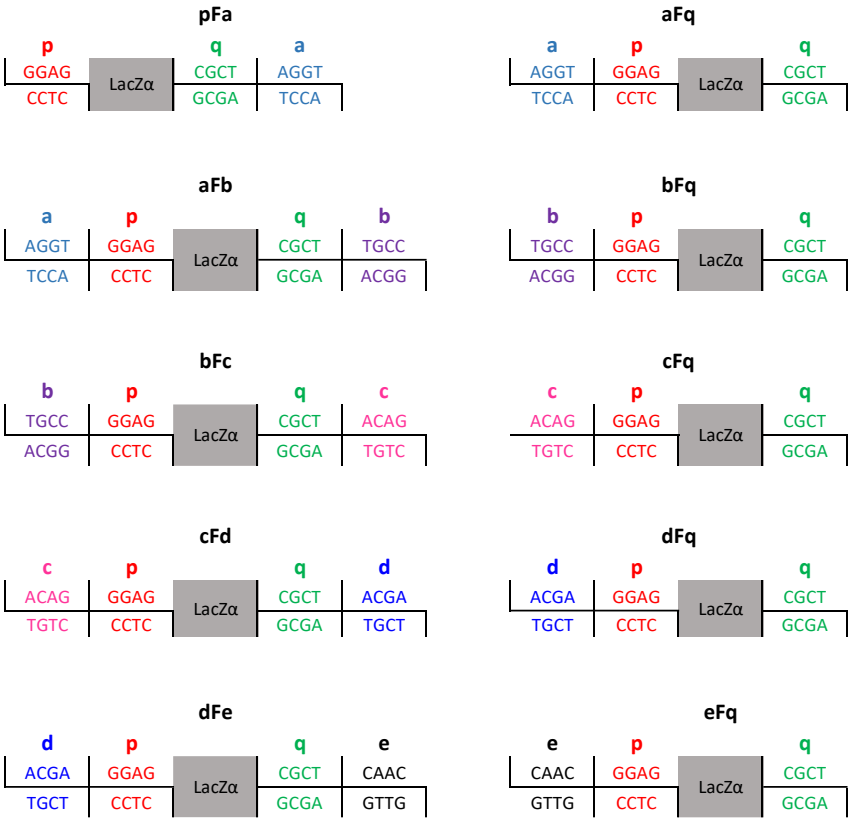


Figure 5 – MetClo's 10 adapter sets composed of permutations of 7 overhang sequences. *There are 7 different overhangs labelled a-e, p and q. The internal overhangs must be p and q, these adaptors readily accept standardised parts while negatively selecting LacZα. The outermost overhang determines the position of the fragments and thus the number of fragments within an assembly. Overhangs p and q can partake as either internal or external overhangs. Overhang p always precedes the starting fragment of a DNA construct (pFa), overhang q follows the last fragment of a DNA construct (xFq). These adapter sets populate the MetClo 48 vector set. The "F" within the name of the adapter set represents the fragment flanked by two overhangs, the overhangs within the name are the outermost overhangs as these determine the position of the part. The black line indicates the cut produced by the restriction enzyme.*

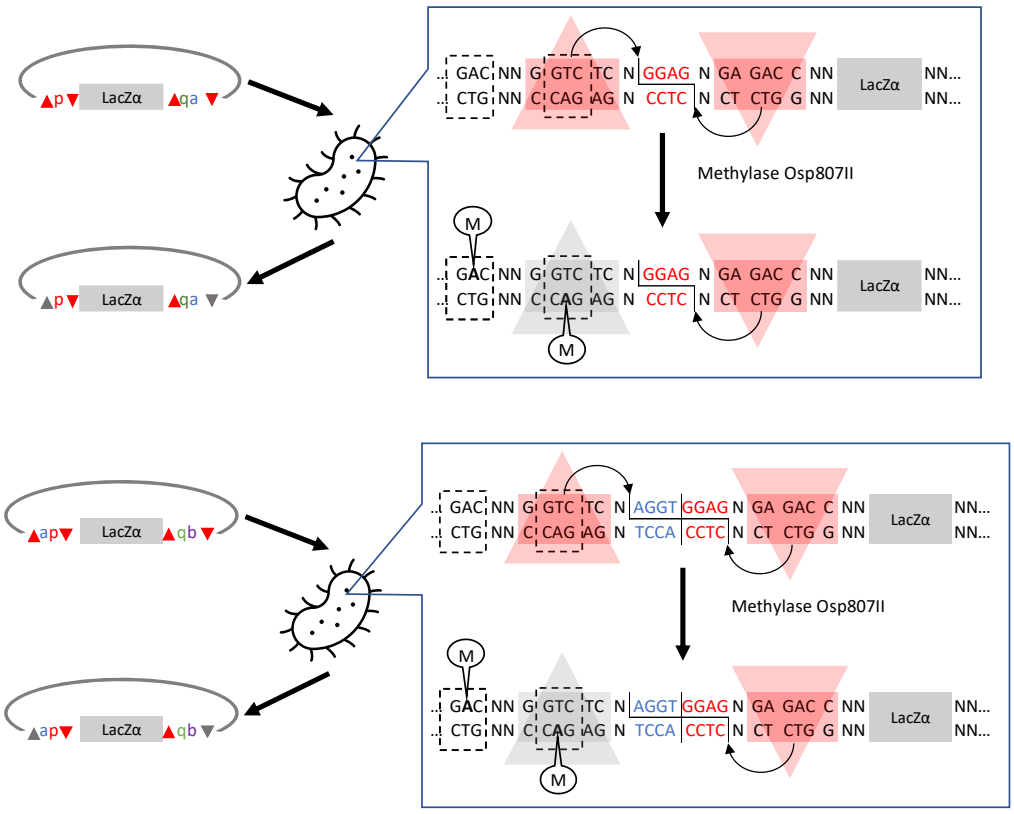
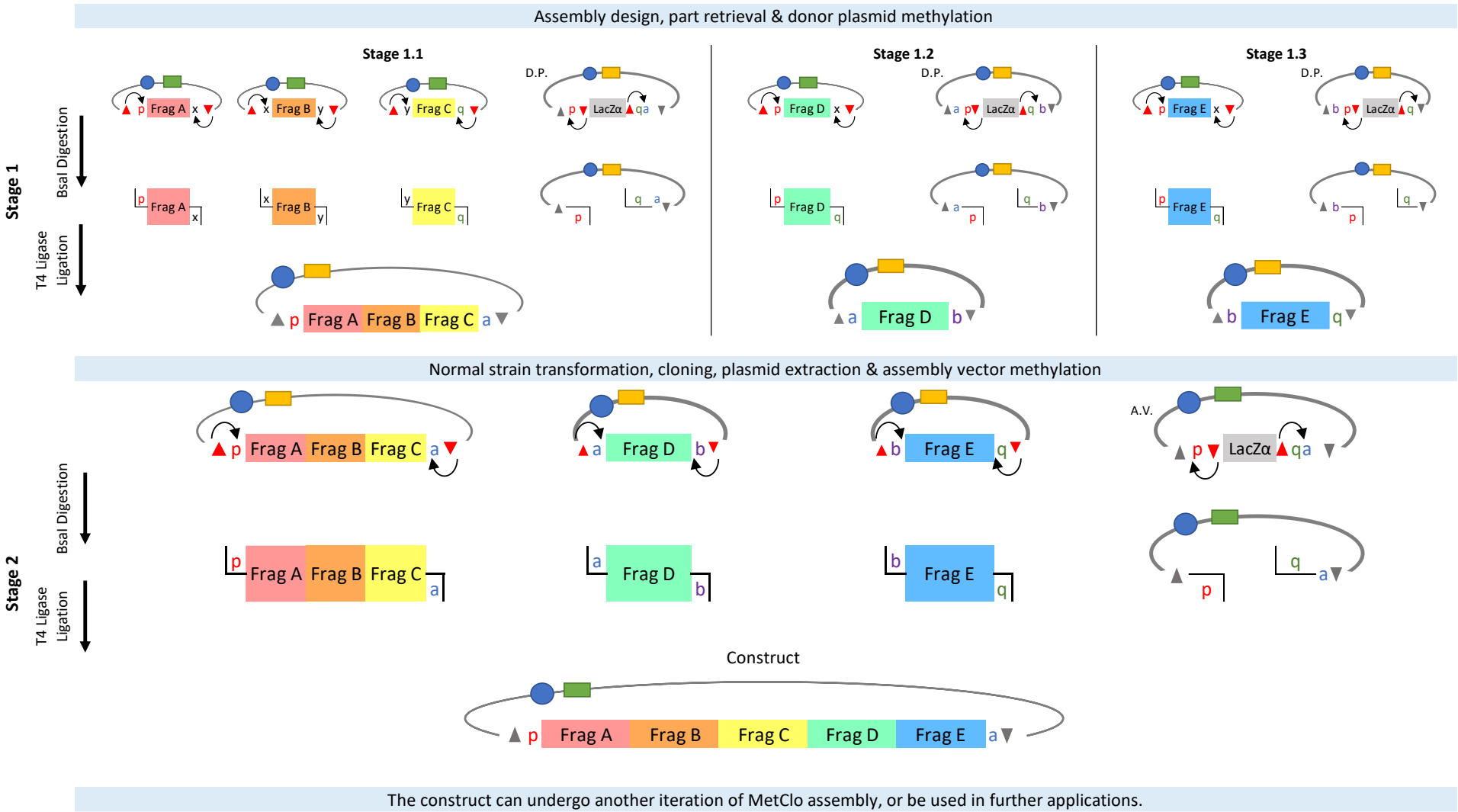


Figure 6 – The methylation-switchable restriction enzyme recognition sites' (MSRERS) architecture. *Methylase Osp807II (M.Osp807II) expression cassettes were inserted into DH10B cells. Donor plasmids and assembly vectors are individually transformed into the methylase expressing strain. Two methylase-switchable restriction enzyme recognition sites flank the LacZα negative selection marker. The left-MSRERS (magnified) and right-MSRERS are inverse mirrors of each other, where all elements are the same except for the RS (a-blue, b-purple, p-red, and q-green overhangs). A single MSRERS is composed of head-to-head BRS (triangled, shaded grey or red) which enclose the restriction site/s, and a methylation recognition sites (dashed lined boxes) that partially overlaps the outermost BRS. The RS can be comprised of a single type of overhang (top), or two (bottom). The single overhang can be acted upon (black curving arrow) by either-enclosing BRS. Each of the binary overhangs is acted upon by the closest BRS. BRSs need to be active/unmethylated (red triangle) to digest a RS. MRS calls M.Osp807II for the methylation of the adenine (bold and labelled with M) within the BRS and the MRS, causing the deactivation (grey triangle) of the outermost BRS.*

Figure 7- Exemplar MetClo hierarchical DNA assembly (view following page). *The figure depicts a 5-part assembly (Frag A, B, C, D & E) into the final construct (ABCDE). The assembly is separated into stage 1 and stage 2. Stage 1 is divided up into 1.1 (to produce construct ABC), 1.2, and 1.3. In stage 1, the parts of interest are held within insert plasmids flanked by a single BRS, and MetClo's methylated donor plasmids (D.P.) carries a negative selection marker (LacZα) flanked by MSRERS; all plasmids generate compatible overhangs (p, x, y, or q). The insert plasmids are prepared within NS which are unable to methylate the BRS. The D.P. and assembly vector (A.V.) are prepared within MES to inactivate the outer-most BRS, as these sites lay within the methylation-switch. The inner pair of BRS are not within the methylation-switch. Parts and donor plasmids are digested (BsaI) and ligated within independent wells. Stage 1.1-3 products are transformed into NS, to produce unmethylated BRS clones, and retrieved to enter Stage 2. In Stage 2 the methylated A.V. and Stage 1 products are restricted ligated. Stage 2's final construct can be used within subsequent rounds of hierarchical assembly as it possesses pFa overhangs much like Frag. A or Construct ABC, or be used for other applications.*


Assembly design, part retrieval & donor plasmid methylation
Stage 1
BsaI Digestion
T4 Ligase Ligation
Stage 1.1
Stage 1.2
Stage 1.3
D.P.
Frag A
Frag B
Frag C
Frag D
Frag E
LacZα
Normal strain transmformation, cloning, plasmid extraction & assembly vector methylation
Stage 2
A.V.
Construct
The construct can undergo another iteration of MetClo assembly, or be used in further applications.

Inactive methylated BsaI recognition site

Active BsaI recognition site

Direction of restriction site digestion

p

Sticky end

qa

Adjacent restriction sites

q

Single restriction site

Antibiotic selection markers

Replication origin

LacZα

Negative selection marker

Frag B

DNA fragment to assemble

1.5. Aim

This work reports on the creation of (1) a computer aided design software dedicated to help the user create a detailed MetClo assembly plan and (2) an multi-assembly automated OT2 protocol for MetClo. Collectively, the two scripts form a semi-automatic pipeline for DNA assembly, which can be fitted within the EGF's larger system. The goal of the project is to expand on the capabilities of the foundry and democratise the use of lab-automation. Additionally, the MetClo assembly and overhang ligation efficiency was examined and validated using EGF's CAD.

2. Results

2.1. Metclo Kit Validation

An assembly plan describes the parts, assembly method and project specification necessary for the completion of a DNA assembly. Assembly plan design can readily accrue many errors, especially in project of increasing complexity, from the improper standardization of parts, mistaken overhang set allocation, poorly designed vectors, incorrect primer design and more (Zulkower, 2021; Pereira et al., 2015). Improper plans would likely fail to produce the intended DNA construct, delaying the project and increasing its overhead. It is necessary to validate the Metclo assembly framework as the EGF has to assure that its implementation within their operations is reliable and functional.

Assembly Simulation

To verify the assembly ability of the MetClo Kit, all 48 vectors participated in simulated assemblies with EGF's DNA Cauldron and SnapGene applications. Six different parts (Frag. A-F) were allocated to the different MetClo's vectors within its 48-vector set (Appendix Figure 1). The vector appointment depended on the planned order of the parts within the final assembly. Frag. A is intended to be the first part in all assemblies; therefore, it was implanted into all vectors with pFa overhangs (where F constitutes the position of the fragment within the overhangs), regardless of the antibiotic resistance marker and replication origin. Frag B is intended to be the second part; therefore, it was

implanted into all vectors with aFb or aFq adapters. In such manner the remaining vectors were populated with the remaining fragments (Figure 8A). All assemblies must begin with vectors containing overhangs pFa and end with overhangs xFq (where x is any letter from a-e). The Metclo vector set can append a minimum of 2, to a maximum of 6 or 4 parts depending on the replication origin of the donor plasmids selected (Figure 8C). Figure 8B shows the possible assemblies conducted with each type of vector within the vector set.

24 total assemblies were simulated within SnapGene and also in DNA Cauldron, EGF's inhouse genetic cloning simulation framework. Each of the 3 antibiotic marker groups with the p15a replication origin, were successfully simulated. Every antibiotic market group produced the expected five assemblies (Figure 8B) within the two simulation programs. Similarly, each 3 antibiotic marker groups with the F replication origins, experienced successful simulation of its 3 assemblies within both programs (Figure 8B).

A)

Part	Frag. A	Frag. B	Frag. C	Frag. D	Frag. E	Frag. D
Vector adapter sequence	pFa	aFb	bFc	cFd	dFe	eFq
		aFq	bFq	cFq	dFq	

B)

2 Part Assembly

pFa	Frag. A
aFq	Frag. B

3 Part Assembly

pFa	Frag. A
aFb	Frag. B
bFq	Frag. C

4 Part Assembly

pFa	Frag. A
aFb	Frag. B
bFc	Frag. C
cFq	Frag. D

5 Part Assembly*

pFa	Frag. A
aFb	Frag. B
bFc	Frag. C
cFd	Frag. D
dFq	Frag. E

6 Part Assembly*

pFa	Frag. A
aFb	Frag. B
bFc	Frag. C
cFd	Frag. D
dFe	Frag. E
eFq	Frag. D

C)

Selection Marker	Replication Origin	Number of Vectors	Adapter sets	Possible number of parts to assemble	Assemblies conducted
Chloramphenicol	p15a	10	pFa, aFb, bFc, cFd, dFe, aFq, bFq, cFq, dFq, eFq	2-6	5
Kanamycin	p15a	10	pFa, aFb, bFc, cFd, dFe, aFq, bFq, cFq, dFq, eFq	2-6	5
Ampicillin	p15a	10	pFa, aFb, bFc, cFd, dFe, aFq, bFq, cFq, dFq, eFq	2-6	5
Chloramphenicol	F	6	pFa, aFb, bFc, aFq, bFq, cFq	2-4	3
Kanamycin	F	6	pFa, aFb, bFc, aFq, bFq, cFq	2-4	3
Ampicillin	F	6	pFa, aFb, bFc, aFq, bFq, cFq	2-4	3

Figure 8 - Experimental design of the DNA simulated assembly with the MetClo vector set. *(A) Parts (Frag. A-D) were allocated to the different overhang sets. Some fragments were implanted within two different sets, as these positions can be either a construct's internal or final position. (B) There are 5 different assembly lengths that can be constructed within the set. These 5 assemblies depict the possible permutation of the 10 adapter sets. All replication origins within the set can assemble 2-4 parts, only vectors with the p15a replication origin can additionally assembly 5 & 6 parts (*). (C) There are 6 different groups of vectors, each with a unique antibiotic resistance marker and replication origin. The replication origin limits the number of parts able to be assembled, influencing the number of adapter sets and vectors constituting each vector group.*

Overhang Validation

The order of the DNA assembly is determined by the vector's adapter set, and its success is dependent on the ligation efficiency of the overhangs within the adapter sets (Potapov et al., 2018a). Potapov et al. (2018a; 2018b) and Pryor et al. (2020) report on the profiling of high-fidelity junction sets from experimental data, the predicting of arbitrary overhang effectiveness within an assembly, and the creation of computer-assisted data optimized assembly. Based on the experimental data generated by these papers, the EGF created web-applications to evaluate the suitability of the overhang set for golden gate DNA assembly. The MetClo's overhangs were participants of 24 successful simulated assembly demonstrating functionality; however, their efficiency can be further explored. MetClo's 7 overhangs (Figure 5) were introduced to KappaGate (Vegh, 2020) and the Overhangs (Vegh, 2022) EGF's python packages to evaluate their propensity of cross talk and their effectiveness.

The Overhang Set report (Appendix Figure 2) indicated that overhang **p** (CTCC/GGAG) (Figure 9, feature 1) and **e** (CAAC/GTTG) (Figure 9, feature 5) are weakly annealing, meaning that the ligation of their Watson and Crick pairs is not strong, possibly reducing the efficiency of parts flanked by either of these adapters. Overhang **b** (ACGG/CCGT) (Figure 9, feature 3) and **d** (AGCA/TGCT) (Figure 9, feature 4) are self-missannealing, implying that the overhangs can ligate with other instances of itself. Overhang **b** and **q** (GCGA/TCGC) seems to missanneal with each other instead of their complement (Figure 9, feature 2). Overhangs **a** (AGGT/ACCT) and **c** (TGTC/GACA) are robust within this adapter group (a-e, p and q), they do not show any significant predisposition to self-missannealing, missannealing among other overhangs, nor weakly annealing.

KappaGate is a CAD that predicts the percentage of good clones (transformations carrying the correct assembly) by modelling part ligation within an assembly based on overhang sequence analysis. A KappaGate evaluation of 13 overhangs predicted that 98.7% of simulated clones contained a valid assembly; the experimental assembly (Potapov et al., 2018b) utilizing the same adapter group resulted in 99.2%± 0.6% (1 std) of valid assembly carrying clones. These results suggest that KappaGate could aid research design and assembly pre-validation, but the program is still claimed

experimental. The KappaGate analysis does not detect unwanted overhang interaction (Figure 10A) and it predicts that 98.7% - 99.5% of clones will carry a valid assembly, implying that there is less than a 5% chance a single colony picked will carry the incorrect assembly (Figure 10B). Figure 10A additionally shows the overhang interaction strength between the overhangs and their pairs, where **q, b,** and **e** have similar and the largest ligation efficiency strength, while **d** has the weakest. These result conflict with the results seen in the Overhangs report.

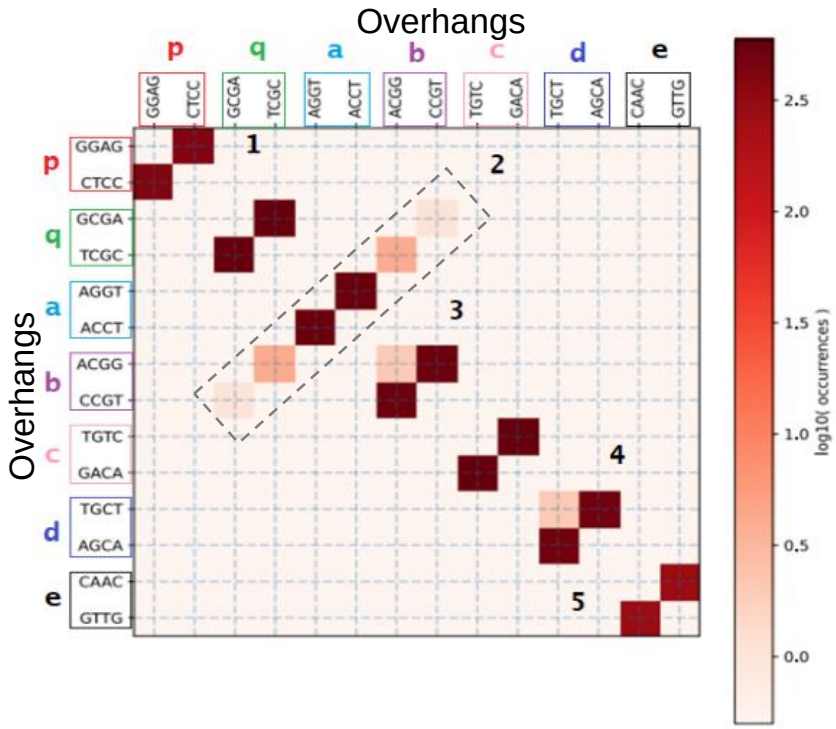
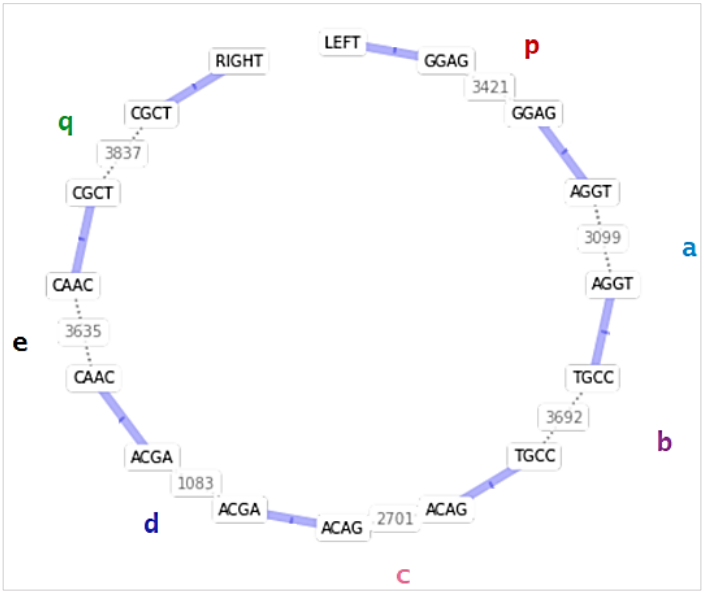


Figure 9 - MetClo's partial overhang set report generated from a simulated assembly with the use of BsaI restriction enzyme at 37°C for 1 hour. *The figure shows the propensity (log_{10} occurrence) of the overhangs to ligate with each other and themselves instead of their complementary sequence. The strong ligation-efficiency of the correct overhang junction is represented by two dark red points along the diagonal. Lighter tones indicate that the ligation of those overhangs is less frequent. Ideally the adapter set should have two strong diagonal lines, any deviation of this represents possible missannealing (2), self-missannealing (3 & 4) or weak-annealing (1 & 5). The 10 overhang sets are found on the right and top axis.*

A)



Overhang	Interaction Strength
q	3837
b	3692
e	3635
p	3421
a	3099
c	2701
d	1083

B)

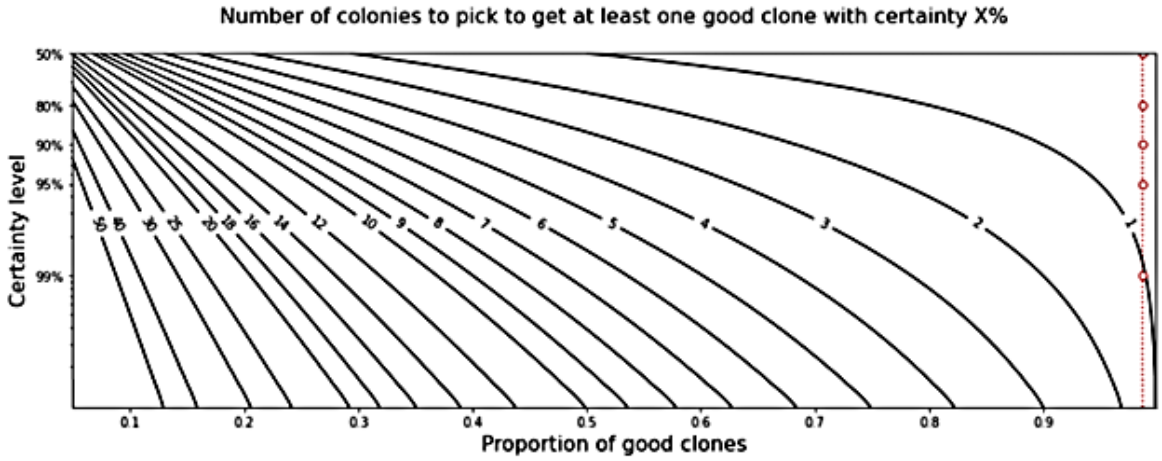


Figure 10 - KappaGate analysis of Medco's Overhang Set. *(A) Circular plot depicts the interaction strength among overhangs. Unwanted overhang interactions would be represented as a red line dissecting the circle, connecting the two overhangs. The table summarises the strength of the interaction in order of their strengths. B) The recommended number of colonies needed to be picked is reliant on the certainty and the proportion of good colonies out of 1. The black lines within the graph recommends the number of colonies to be picked. The red line is correlated with the success rate of the correct overhang ligation, and the red dots depict the estimated minimum, maximum and average number of trials that should be conducted.*

2.2. MetCloOT2 Pipeline Generation and Integration within the EGF

The EGF's DNA assembly system follows the DBTL cycle; it's composed of three consecutive portions: design, assembly (build) and validation (test & learn) (Figure 11). Each section is accompanied with a variety of software and hardware tools. The appropriate tool, or set of tools, used is dependent on the assembly project and its specifications. Having a diverse set of interoperable tools for each stage of the DBTL cycle within the EGF's system architecture is advantageous as it expands the foundry's capabilities, permits modular exchange of tools, and streamlines projects. Aside from a large suite of design tools, the EGF offers two primary modes of assembly, manual and automatic. Assembly can be performed on an array of liquid handling machines within the foundry with varying degrees of automation and human involvement. The MetCloOT2 pipeline, created within this report, contains a design-phase CAD tool and a semi-automation build-phase tool, these two programs easily transfer information between each other to facilitate the use of the MetClo assembly method, and were designed to swiftly fit within the EGF's system.

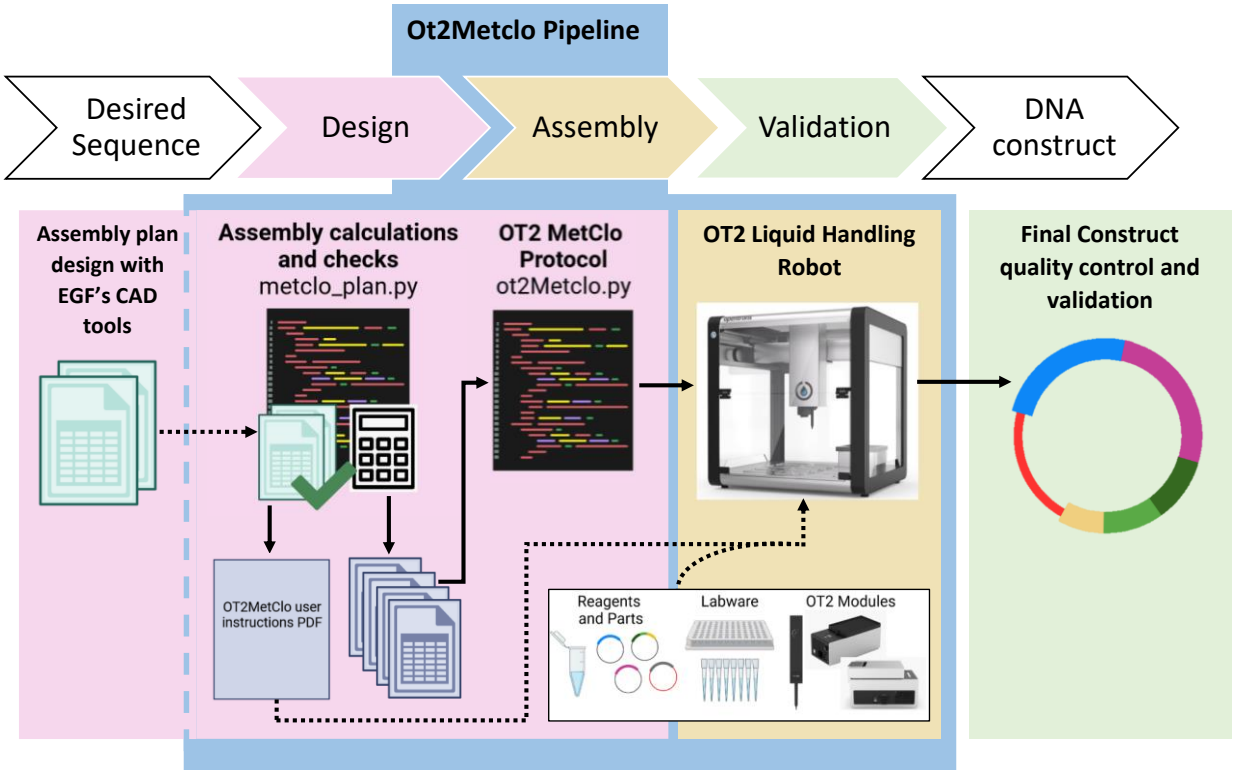


Figure 11 - MetCloOT2 pipeline within the EGF system's architecture. *The above cheveron process illustrates the stages within the system's architecture, from the request of a specific sequence, to the*

delivery of a physical verified construct. The process is based on the DBTL cycle. The MetCloOT2 pipeline (within the blue area) contains a tool for both the design (pink area) and build/assembly phase (yellow area). EGF' CAD tools are linked (blue dotted line) by means of files (green spreadsheets) to the MetCloOT2 pipeline, these files are needed as the only input. Dashed black arrows depict the need of human intervention, where solid black arrows are fully automatic.The first script (metclo_plan.py) calculates nessesary values for assembly from the input data to generate four .csv files (purple spreadsheets) which are not intended for human-use, and a human-readable PDF. These .csv files are directly detected by the second script (MetCloOT2.py), and used to create the assembly protocol specific to the project. The PDF instructs the user of the reagents, parts, labware, and OT2 robot apparatus nesesary to be installed in the robot. Once the protocol, the assembly files, and other materials are arranged, the assembly can start. The final construct can then go into the testing phase (green area) to be validated and undergo quality control (QC). The EGF's assembly process is complete once the assembly has completed all rounds of hierarchical assembly and passes validation, now the construct can be used for its intended purpose.

MetClo Computer Aided Design Tool

To conduct an assembly specific actionable protocol, the volume of the reagents and parts needs to be calculated. A single MetClo reaction requires 15-30 fmol of each insert, donor plasmid or assembly vector, 2 µl of 10x T4 ligase buffer, 0.5 µl of T4 ligase, 0.5µl of BsaI-HFv2 (or BsaI) for small assemblies (<30kbp) or 1.0 µl for large assemblies (>30kbp), and the top-up water volume to reach the final reaction volume of 20 µl. Additional effort is the need to designate the location of parts, reagent and reactions within wells. These assembly formulation and organisational steps can be a time consuming and an error prone process within the design stage, especially for numerous large assembly projects. Calculations and organisation become extra tedious when the reduction of laboratory waste and the optimised use of labware is a prioritised in the experimental design. To remove the user from this stage of planning, the metclo_plan.py script (Appendix Script 1 & Appendix Figure 3) was developed to automate the calculation and well allocation for parts, reagents, and assembly reactions for multiple assemblies.

The user-created assembly plan spreadsheet (.csv) input for the DNA Cauldron assembly simulator can be used, with a slight modification, as the input for metclo_plan.py. The use of this file reduces redundancy and the chance of miss-typing information; and streamlines information transmission between tools at the EGF (Figure 11). Part information (name, concentration (ng/µl), size (bp)) is an additional needed .csv input (Figure 12).

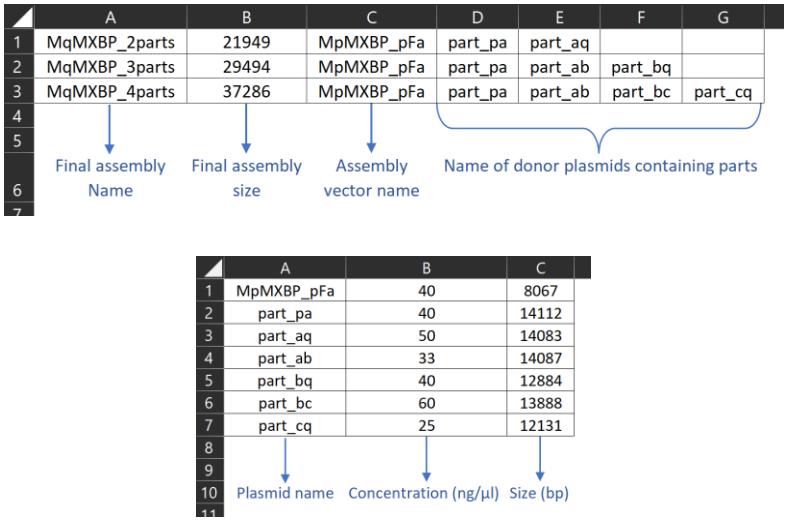


Figure 12- metlco_plan.py exemplary input assembly.csv (top) and part.csv (bottom) files*. The assembly.csv is analogous to the assembly plan spreadsheet for DNA Cauldron, except for the additional 'final assembly size' column. Each row contains the name of the final assembly, its size, followed by the assembly vector and the donor plasmids containing the parts. The part.csv contains rows with each plasmid/vector's concentration and size.*

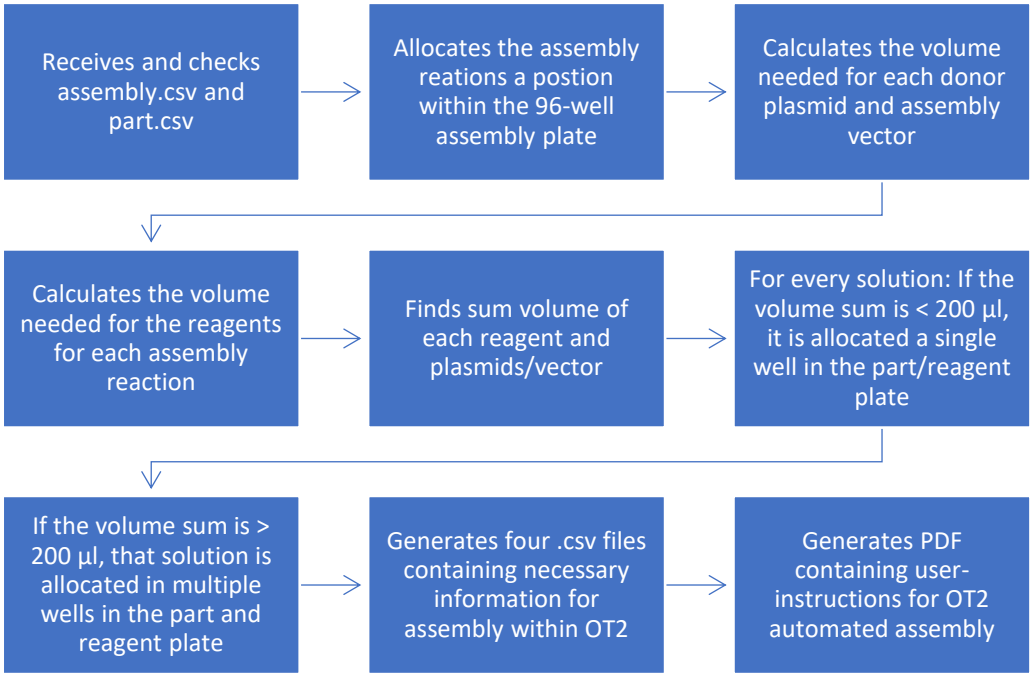


Figure 13 - Summary of metclo_plan.py workflow. *The flowchart shows the essential steps within the script to generate the protocol information necessary for assembly. This is a summary of Appendix Figure 3, which is an extensive algorithm detailing the Appendix Script 1 operations.*

The initial version of the program could calculate the part and reagent volumes necessary for a single assembly reaction. The script experienced multiple

improvements to increase its utility. The current metclo_plan.py can calculate the reagent and part volumes of multiple assemblies in a single program run (Figure 13). The volumes for each reagent and part are aggregated, if the volume is under 200µl (the part/reagent 96-well plate well's limit) then a single well is allocated to that specified reagent or part. The program will allocate additional wells for aggregated volumes over 200 µl. The purpose of this complex form of well designation is to optimise the use of the part/reagent plate and simplify the reagent and part pipetting for the OT2 set-up.

The script has inbuilt checks, that verify the execution of methods and results within the program. If a fault is found, the program is interrupted and a descriptive error message will arise to permit user-mediated troubleshooting. Assembly information necessary for the protocol execution is stored within the user's device in four new .csv files (Figure 14). The .csv files streamline information to the next phase of the DBTL cycle, building (Figure 11). Lastly, an easily human-readable PDF (Appendix Figure 6) is generated containing (1) a guide indicating how the user must set up the OT2 for their assembly project, (2) part, reagent and assembly reaction location within two 96-well plates, (3) summarised assembly specific details, (4) parts and reagents' calculated volumes, (5) recommendation of cloning technology depending on final assembly vector size. The PDF facilitates adaptability to diverse laboratory settings, as information enclosed can serve manual assembly and automated assembly.

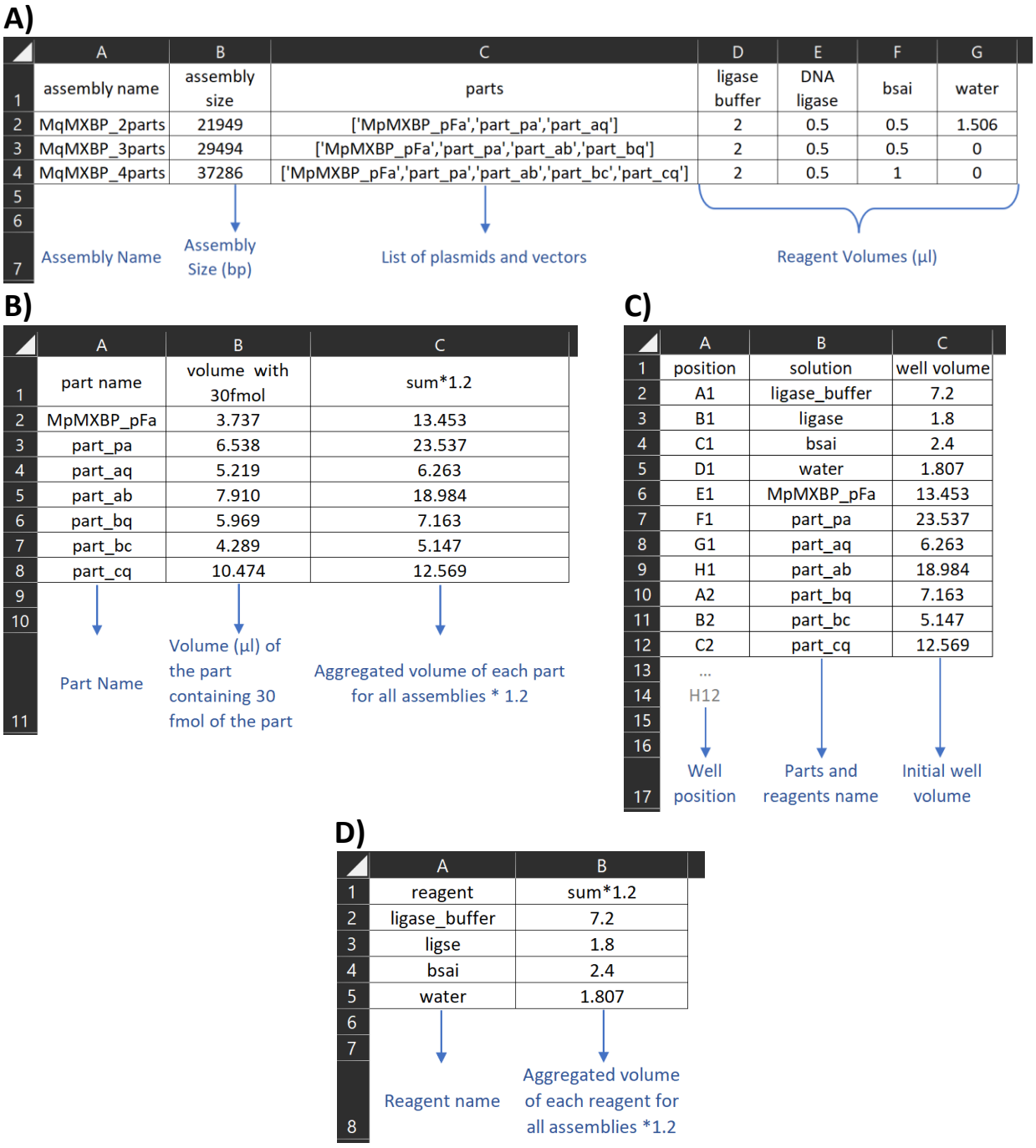


Figure 14 - metclo_plan.py exemplary output .csv files. *(A) the assembly_data.csv contains the assembly specific information, such as its name, size (bp), array of parts, and volume needed of each reagent. (B) The part_data.csv contains part specific information, like the volume of each part containing 30 fmol of plasmids. The total volume needed for every part for all assemblies is added and multiplied by 1.2. The 1.2 multiplication assures a small surplus is provided to account for any pipetting errors. For example, part MpMXBP_pFa (3.737 µl containing 30 fmol) is needed within 3 assemblies, therefore it is multiplied by 3 and 1.2 volume to find the total required volume of 13.453 µl. (C) The position_data.csv*

contains the location of the parts and reagents within the part and reagent 96-well plate on the OT2. (D) The reagent_data.csv contains the sum volume of all the reagents for all the assemblies.

MetClo Protocol Translation for Automated Assembly

The OT-2 Python Protocol API is a python framework used to write high-level human-readable python source code lab protocols. The Opentrons App receives the protocol and translates it into executable machine code for the OT2 robot. The MetCloOT2.py script (Appendix Script 2) was created within the API, it is a translation from Lin & O'Callaghan (2020)'s human-readable MetClo assembly protocol to an open-source python script that commands the execution of multiple MetClo based assemblies within the liquid handling robot.

The code was initially hardcoded to run a single assembly reaction and required reprogramming to conduct other assemblies. The tool had very low through-put and require programming skills, making manual assembly a more attractive option. To increase MetCloOT2.py's through-put, it was made to automatically detect and import the detailed assembly design .csv files generated by metclo_plan.py, giving it the flexibility to form as many as 1 to 96 assembly reactions in a single run and prevent code alteration by the user. To reduce the number of tips necessary and the time for the pipette to travel between the deck slots, the program instructed reagent batch dispensing, meaning that a single pipette tip was used to absorb a large quantity of a reagent and dispense the assembly specific amount into the appropriate assembly PCR well. Parts were not batch dispensed as contamination between assemblies would occur. To optimise the use of labware, the metclo_plan.py aggregates the reagent and parts into single wells, or multiple wells if the sum volume was over 200µl. The MetCloOT2.py was programmed to keep track of each well's volume and aspirate from wells containing sufficient solution. The script has inbuilt checks, that verifies the execution of methods and results within the program. If a fault is found, the program is interrupted, and a descriptive error message will arise to permit user-mediated troubleshooting.

Figure 15 is a simplified workflow of the assembly process, from preparing the OT2, script execution, to assembly completion. A complete flowchart detailing the algorithm within MetCloOT2.py is found in Appendix Figure 4. The deck is mounted with the necessary apparatus, and pipettes; reagents and parts are manually loaded to their designated position as described in the metclo_plan.pdf. The app generated a picklist (a list of high granular instructions) (Figure 3B) from the MetCloOT2.py script for the OT2 with all the nuances described above. The App commands the OT2 to begin executing the commands within the picklist, until they are complete. The user then can retrieve the assemblies from the assembly plate which has been stored at 4°C.

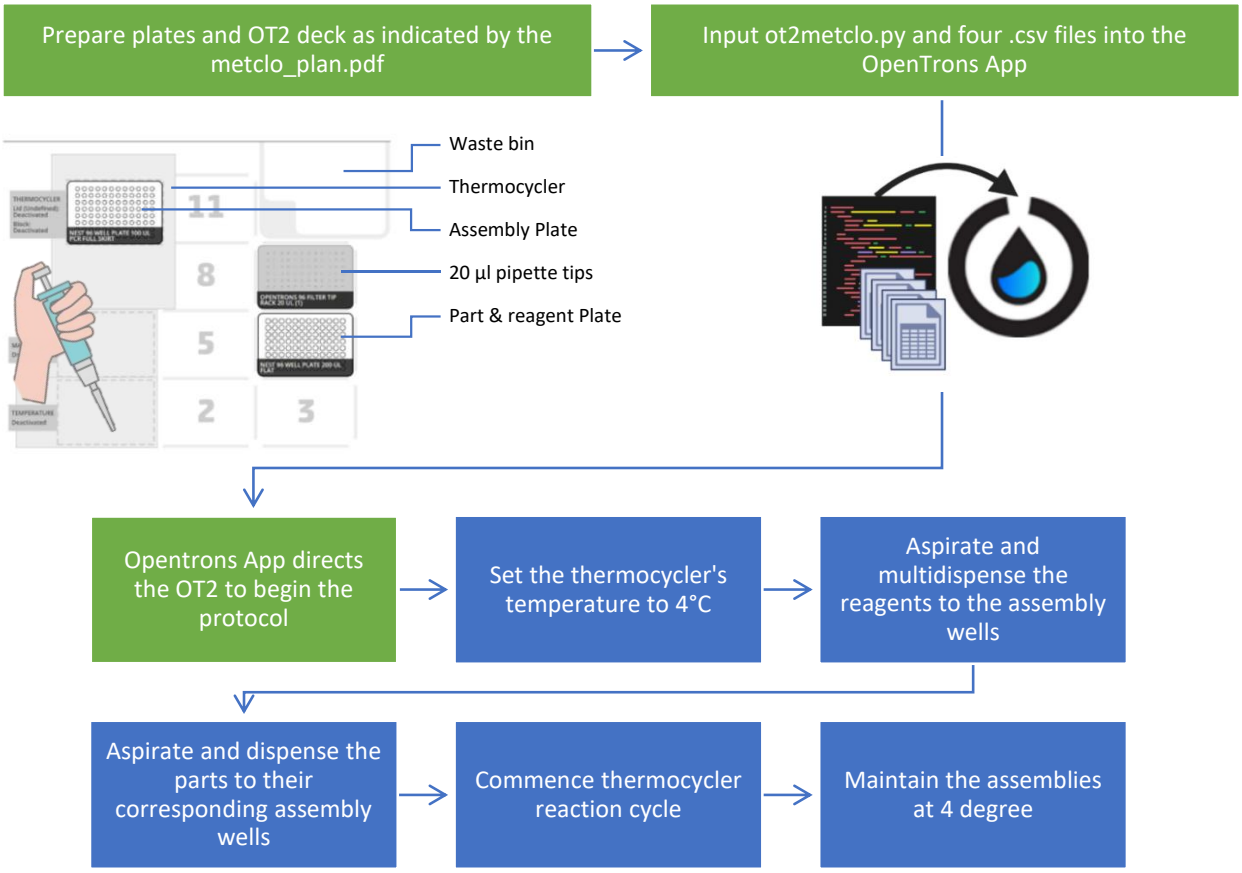


Figure 15 - Summary of MetCloOT2.py workflow. *The flowchart shows the essential steps prior the scrip execution (green shapes) and the summarised key functions within the script (blue shapes). The OT2 deck is mounted with the necessary apparatus as shown. The parts and reagents are manually pipetted to their designated location by the user. The user then runs the MetCloOT2.py on the Opentrons App that creates a picklist and instructs the OT2 to initiate executing the picklist's instructions.*

MetClo Pipeline Validation

To validate the pipeline, the metclo_plan.py and ot2metclo.py were used to conduct a dry-run assembly within the EGF's facility. A dry-run means that no assemblies were produced; the pipette moved around the OT2's deck, aspirating and dispensing air as it would reagents and parts. The pipette's route was closely monitored, and the picklist carefully checked to assure the correct mixing of solutions for the assembly reaction. The thermocycler was also monitored to assure that correct temperature was held for the appropriate amount of time. The OT2 completed the three-assembly project as described in Figure 12. The intermediate files with the assembly data can be found in Figure 14 and Appendix Figure 6.

3. Method and Materials

3.2. Biofoundry and User Implementation of MetCloOT2

The MetCloOT2 open-source codes can be found and downloaded (cloned) from the GitHub repository (Appendix Figure 7), these will be saved as a file called "MetCloOT2". The source code and library packages are available for systems supporting python. Once the OT2 MetClo python files are downloaded, as instructed by the 'Read Me' on GitHub, the user must assure they have csv, math, string, sys, os, FPDF, profile, and Opentrons packages within their device. The metclo_plan.py scrip is run, and the user will be asked to input .csv files containing the assembly plan and part information into the command line. The program will save a single .pdf file (metclo_plan.pdf) and four .csv files (assembly_data.csv, part_data.csv, reagent_data.csv, position_data.csv) in a directory called 'MetCloOT2/metclo_plan_files' on the user's device.

The user must prepare the OT2 robot as indicated by the .pdf file. The user must install the OT2 Thermocycler Module Gen 2, a 20 µl OT2 pipette, a 20 µl tip box, a NEST 100 µl PCR Full Skirt 96-Well Plate (assembly reaction plate), and a NEST 200 µl Flat 96-Well Plate (reagent/part plate). Additionally, the reagents and parts must be manually pipetted into the assigned location within the reagent/part plate.

To verify the robot will complete the correct actions, it is recommended to simulate the automatic assembly, stored within MetCloOT2.py, as described in the Opentrons API website (https://docs.Opentrons.com/v2/). The picklist generated from the assembly will inform the user that the correct actions were taken. The .pdf file contains specific instructions about running the OT2 robot and the MetCloOT2.py protocol. The entire MetCloOT2 file needs to be on a computer connected to an OT2 and running the Opentrons App, additional information about the OT2 and the App can be found online (https://Opentrons.com/ot-2/). Open the Opentrons App, and follow the instructions shown on screen and provide the necessary files when requested.

The OpenTron App will generate the picklist from the MetCloOT2.py protocol and provide it to the OT2. The OT2 pipettes aspirate, dispense, and mix the necessary volumes for each reagent, plasmids and vectors, as indicated by the picklist, from the reagent/part plate to the assembly plate to achieve a total reaction volume of 20 µl. The assembly plate is incubated in the thermocycler for 15 minutes at 35°C, followed by 45 cycles (2 minutes at 37°C, 5 minutes at 16°C, 20 minutes at 37°C, and 5 minutes at 80°C). The assemblies are then maintained at 4°C.

3.3. Simulation

The DNA Cauldron web-application was used, as per Zulkower (2021). The program does not detect the methylation and deems BRS as active. Any attempt to run a simulated assembly of the MetClo vectors produces an error saying that the assembly cannot be performed as there are too many restriction sites. To mimic the inhibition of the BRS by methylation, a point mutation was done on the outer flanking BRS, in a DNA sequence editor prior submitting the assembly plan into the software, preventing the software from recognising the BRS and thus permitting assembly. The simulations were duplicated in SnapGene (www.snapgene.com). Similarly, a single nucleotide was changed in the outer flanking BRS, to mimic digestion inactivation by methylation, to allow proper assembly. The altered BRS, were returned to normal to continue the next phases of simulated assembly.

3.4. Overhang analysis

The 7 MetClo overhangs served as inputs into the EGF's Overhangs and KappaGate Python packages. The packages were obtained from the EGF's GitHub repository and used as per each program's repository's indications.

4. Discussion

The Metclo vector kit successfully produced different assembly combinations in both SnapGene and DNA Cauldron programs. The simulations were not straightforward, the sequences required minimal manipulation (single base substitution) as these programs do not recognise the inhibition of restriction enzyme recognition sites by methylation. The positive outcome within the programs partially confirms the author's claims of a functioning assembly kit. Definitive proof of MetClo's ability to work cannot be completely tested within these simulation programs as they do not accommodate the inhibition of RS inhibition methylation, consider other subtleties that come with real cloning, and are frequently cited to lack the incorporation of the latest research data and continuous computational maintenance (Peng et al., 2015). For these reasons and more, these programs have efficiency and stability issues causing low utilisation rates within the community. Simulation programs could be improved to produce more accurate results and cater to the subtleties of assembly methods.

A deep inspection of the 48 MetClo vectors verified the presence of the necessary components for an assembly plasmid: an origin of replication, a selectable antibiotic resistance gene, and an insertion site flanked by RE for an insert fragment. As a kit capable of generating large constructs, the two origins of replication p15a and OriF offered, gives the kit greater control of plasmid copies. The selection of three antibiotic resistance vectors permitted choice flexibility to combine assembly vectors and donor plasmids into a single-pot reaction with distinct selection markers. As expected, the insert site contained a negative selection fragment flanked by RS and BRS, making it replaceable by the G.O.I.

Three factors permitted the 24 successful simulations, which may yield to successful lab-based assemblies: (1) the correct assembly vector structure for all 48 vectors, (2) an assembly framework structured to facilitate large DNA construction and (3) Lin & O'Callaghan (2020)'s clear and comprehensive MetClo kit instructions. The MetClo method could potentially be implemented in to the EGF, however further simulations or lab-bench assemblies are needed to verify the method's effectiveness prior investing in and adopting the method.

As the MetClo overhangs were empirically designed and published prior the overhang ligation-efficiency studies by Potapov et al. (2018a; 2018b) and Pryor et al. (2020), they may have flaws or opportunities of enhancement. Their efficiency was examined by two EGF CAD software (based on the experimental data from the forementioned papers). The chance of clones containing the incorrect assembly using the MetClo overhangs is improbable (<5%) as reported by KappaGate. Based on this result, the researcher may plan to pick a single colony. The report does not detect any significant unwanted ligations between the adapters; however, the strength of correct ligation varies greatly between the overhangs.

There is room for improvement within Kappa Gate's documentation. The precise way the program determines these outcomes is unspecified, there are no declared limitations to the program and there are no comparisons of these theoretical results to actual bench assemblies. The results have little explanation support, for example, there is no mention of how the ligation strength values in Figure 10A were calculated or information of their unit of measurement. A single statement on the KappaGate GitHub page reports that it has closely predicted Potapov et al. (2018b) experimental results; no published reports evaluating Kappa Gate's effectiveness and accuracy have been found. The user must be prudent to use this data to aid experimental design.

Contrary to Kappa Gate's results, the Overhang's report identified problematic MetClo overhangs. Some overhangs ligated weakly, missanneal with itself or other overhangs. Two overhangs were shown to be efficient. However, the effectiveness of a collection of overhangs is as good as its most unreliable overhang, effectively discounting the efficiency of the better adapters. The resulting undesirable efficiencies may be due to

the empirical-based method of overhang selection. Much like KappaGate, no published papers claim the use of the Overhang's program, therefore, there is no experimental confirmation of its predicted results, aside from statements on its GitHub page. It should be noted that these two programs do derive from experimental data (Potapov et al., 2018b; Pryor et al., 2020; Potapov et al., 2018a) that did show accurate predictions of overhang ligation-efficiency and the rate of assembly success, therefore the results of these two programs should not be disregarded but used cautiously.

It is uncertain if the assembly simulation programs like DNA Cauldron or SnapGene consider the ligation efficiency of the simulation's overhangs. DNA Cauldron's documentation (Zulkower, 2020) states that it "detects incorrect overhang design", however it does not describe the type of errors it detects. The DNA Cauldron's documentation does not specify the use of experimental data from Potapov et al. nor Pryor et al. within the code, unlike Overhangs or KappaGate, and might not use this information to predict assembly success. DNA Cauldron could evaluate the overhang based on their sequence alone, and not on the comprehensive profiling of the interactions of all possible permutations of 4 bp overhangs. It should not be assumed that overhangs will always ligate with their intended counterpart, the high efficiency of T4 ligase and the DNA's promiscuity of forming H-bonds with other overhangs should be included in such simulations. There is no proof that SnapGene also utilises this information to determine whether the assembly is feasible or not. Therefore, to use these results to state that the assembly is feasible is a premature assumption.

The recurring hardware sum cost (not including reagents nor parts) of a single MetCloOT2 run (96 assemblies) is approximately 330 UDS. The cost of the smallest pipette unfiltered tip bundle (9,216 tips) is 412.50 USD, or 4.30 USD per rack. The thermocycler (9,750 USD) and the OT2 with a 20µl single-channel pipette (7,750 USD) are one-time purchases. The cost of the first MetCloOT2 assembly would be approximately 18,300 USD (not including reagents nor parts) if the laboratory does not already have the necessary hardware infrastructure. If the OT2, pipette, tips and necessary modules are present, then the expected cost would be the forementioned

330 USD. The MetClo kit containing the 48 vectors and the necessary E.coli strains, as sold by AddGene, is retailed at 375 USD.

The MetClo Pipeline successfully executed the protocol for three dry-assemblies. The metclo_plan.py correctly calculated all necessary values, created the output files, and arose errors where necessary. The ot2metclo.py accepted and checked all files, and interacted appropriately with the Opentrons App. The generated picklist and the OT2 movements were as expected. The pipette moved to the intended locations, picked up and dispensed of tips when necessary. The thermocycler followed the coded temperature conditions. The pipeline was successfully used within the EGF's system without any detectable problems. The ability to acquire the open-source scripts, actively interact with them and conduct a dry-run, demonstrates that the MetcCloOT2 pipeline has potential to conduct semi-automated MetClo based assemblies within the OT2 robot. As to whether the pipeline can yield correct assemblies is dependent on the MetClo method and correct programming of the scripts, this can only be confirmed by wet-protocol runs which were not conducted in this project. The ability of the pipeline to outcompete manual-work in output, labware and reagent use efficiency, time, and total cost is unknown.

DNA assembly automation papers mainly evaluate their platform's functioning on four criteria: cost, output, time of execution and human involvement. They are then compared to other automation platforms or manual assembly. A DNA assembly metric (Q-metric) has even been developed and used to quantitate the benefit of automation over manual-work based on cost, output and time quantities derived from automation operations (Walsh et al., 2019). No quantitative metrics were obtained during the validation phase. This project solely reports on the development of the of an automated DNA assembly platform and its ability to follow protocol procedures, future work should aim at characterizing it.

Walsh et al. (2019) describes a threshold that determines whether automated or manual assembly is favourable. If the number of assembly projects is high enough, the use of automation is more advantageous over manual. Any future dry or non-dry validation of MetCloOT2's ability to outperform manual assembly, must perform 96 assemblies and

be compared to the manual execution in output, time and costs. Performing too few assemblies, as done in this report, may not permit the discovery of the threshold, and thus the minimum number of assemblies that makes the use of the pipeline worthwhile.

Many assembly methods validate their implementation by assembling the lycopene pathway for the synthesis of lycopene, as it has become a model metabolic pathway in synthetic biology (Exley et al., 2019). The assembly simulation was conducted with arbitrary fragments (Figure 8). Future *in silico,* automated or manual validation assemblies of the MetClo kit should use this pathway, to standardise validation among assembly methods, automation systems and relevant CADs.

A protocol script is as good as the programmer's understanding of the lab-protocol and their ability to translate it from human-readable to robot-readable. Any error in interpretation or programming can result in inappropriate execution. Any unoptimized portion of code, can result in an inefficient use of the robot. Similarly, the system/pipeline is as useful as its creator's capability of designing it. Many code and pipeline improvements can be made to MetCloOT2.

The pipeline could be described as semi-automatic, as it requires human intervention past the first step, metclo_plan.py's file input (Figure 12). The MetCloOT2 requires the manual setup of the OT2 and the command line intervention to run both scripts. To make the pipeline fully automated, an additional automated system needs to be created that retrieves parts and labware from a repository to populate the OT2's deck. Once populated, the second script can be ran on command to commence the assembly. The increase of automation past the point described in this paper would prevent the accessible use of the pipeline, as it would demand more infrastructure than that of an OT2. This pipeline seems to be as automated as possible.

The user requires some coding ability, and knowledge of the use of an OT2 robot. The user must know how to use basic Git package functions, download python packages, and run python programs within a command line. This may limit the uptake of the pipeline and its ability to serve the community. An improvement would be the creation of a user-friendly web-application that only requires the upload of assembly information to

generate a downloadable OT2 protocol which can easily be uploaded into the OT2 App, effectively bypassing the need for the user to have programming skills.

Many other areas of optimisation could be described. The 96-well assembly plate could be replaced with a 384-well plate, consequently increasing the throughput capacity. Reagents and parts could be held within separate plates. A transformation protocol can be appended to ot2_metclo.py to free the researcher from conducting the downstream procedure. The pipette route can be optimised to minimise travel distance and reduce the time of operation. Higher order optimisations may include the use of machine learning to learn from past assembly data to produce better and more efficient future versions of the MetCloOT2 machine-protocols.

One significant limitation of all software, including MetCloOT2, is the constant need to be updated and maintained. MetCloOT2 is reliant on other hardware and code, such as the OT2, the Opentrons App and the packages. Any change to these devices or programs could affect the functioning of MetCloOT2. The ability for MetCloOT2 to serve the scientific community is dependent on its dedicated maintenance.

5. Concussion

The MetCloOT2 was developed to automate the MetClo DNA assembly method within the Opentrons OT2 liquid handling robot. The project gives an assembly method the benefits of automation, extend the capabilities within the EGF and make MetClo automated assembly accessible to researchers in possession of the OT2. The MetCloOT2 pipeline can (1) receive 1-96 assembly projects' specifications, (2) compute necessary MetClo reaction experimental data from said specifications (3) create datasets that serve as the input for the protocol directing the OT2 robot. The framework was designed to assimilate within the EGF's modular DNA assembly system, to streamline its addition into the DNA assembly process in the foundry.

The EGF's DNA Cauldron CAD was used to simulate multiple MetClo assemblies to (1) assure build feasibility and (2) use the validated CAD output files as inputs into the MetCloOT2 pipeline. The pipeline was designed to accept the DNA Cauldron output

files to streamline the movement from the design to build phase, effectively easing the transmission of information and encouraging the use of CADs prior assembly.

The pipeline is available to all users via its GitHub repository. Minimal programming knowledge is needed for its basic functioning. It is recommended that the pipeline is used in conjugation with the EGF's simulation tools and other CADS to aid experimental design. The OT2 itself is a liquid handling robot that is designed to enable its easy use, has an extensive how-to guide, and provides support for its users. The easy use of the pipeline (from its installation, application and equipment use) was a priority in its design to prevent a high dropout rate common with such platforms.

Overall, the automation of MetClo intends to reduce human involvement, free researchers' time, increase output accuracy, improve DNA assembly throughput, and standardise lab procedures. The opensource quality of the platform permits the modification of its code by the research with coding skills. This permits improvements to the code and/or project specific modification. It is provided free of charge to democratize the technology in the hopes to expedite the progression of experiments and research. The MetClo method seems to be a viable approach to create very large constructs (218kbp) while circumventing problems associated with other Golden Gate based assembly methods. It, however, lacks application as no publication declares its use. This method was chosen because (1) it has not yet been automated, and (2) to encourage its use by making it simpler to implement.

The validation of MetClo is imperative to its implementation in the EGF. The MetClo creators (Lin & O'Callaghan, 2018) report positive results for the creation of large assemblies, and state that MetClo has comparable performance to MoClo, a popular cloning method. This report evaluates the MetClo vector set and confirms that it comprises of all necessary DNA assembly vector components and a framework that permits the appending of 4-6 fragments per hierarchical step. Computational simulation suggests that it can successfully assemble DNA. These simulations do not consider many intricacies associated with the project's specifications, physical cloning, and MetClo assemblies, thus conceivably producing misleading results. The method's forementioned lack of use and, mixed and dubious overhang analysis results decrease

the confidence that it is a reliable DNA assembly approach. The elusive results from the overhang analysis and simulations may be due to the evaluation software itself, instead of the actual overhangs and assembly framework.

Future work could evaluate the accuracy and precision of CAD programs results. Additionally, they can be improved upon to achieve results comparable to experimental assemblies. Further validation is needed to implement MetClo into the EGF; both wet and dry lab testing would assure the method's functioning. The MetCloOt2 pipeline should also be tested and characterised to assure proper performance. The assembly of the lycopene pathway, a pathway assembled by many other methods, would make it easier to compare the MetClo's and MetCloOT2's functionality.

6. Acknowledgement

I thank my supervisor Peter Vegh for his support and guidance with the content and the creation of this report. As well as Elise Cachat, David Radford, Shona Nixon, my family, friends, and my peers from the MSc in Synthetic Biology & Biotechnology and the MSc Biotechnology 2021 graduating class for the support.

7. Abbreviations

A.V.	Assembly vector
BRS	BsaI Recognition Site
CAD	Computer Aided Design
D.P.	Donor plasmid
DBTL	Design-Build-Test-Learn cycle
DoE	Design of experiments
E.coli	Escherichia coli
EGF	Edinburgh Genome Foundry
G.O.I	Genetic Element of Interest
M.Osp807II	Methylase Osp807II

MES	Methylase expressing strain
MRS	Methylase Recognition Site
MSRERS	Methylase-switchable restriction enzyme recognition site
NS	Normal E. coli strain
OT1	Opentrons OT-One liquid handling robot
OT2	Opentrons OT-Two liquid handling robot
QC	Quality control
RS	Restriction site

8. Code availability

The open-source MetCloOT2 software is available at: https://github.com/Edinburgh-Genome-Foundry/OT2Metclo

A copy of the commented source code in found in Appendix Script 1 & 2

9. References

Bryant, J.A., Kellinger, M., Longmire, C., Miller, R. & Wright, R.C. (2022) *AssemblyTron: Flexible automation of DNA assembly with Opentrons OT-2 lab robots*. doi:10.1101/2022.09.29.510219.

Casas, A., Bultelle, M., Motraghi, C. & Kitney, R. (2022) PASIV: A Pooled Approach-Based Workflow to Overcome Toxicity-Induced Design of Experiments Failures and Inefficiencies. *ACS Synthetic Biology*. 11 (3), 1272–1291. doi:10.1021/acssynbio.1c00562.

Chory, E.J., Gretton, D.W., DeBenedictis, E.A. & Esvelt, K.M. (2021) Enabling high-throughput biology with flexible open-source automation. *Molecular Systems Biology*. 17 (3), e9942. doi:10.15252/msb.20209942.

ConceptDraw (n.d.) *Flow Chart Symbols*. https://www.conceptdraw.com. https://www.conceptdraw.com/How-To-Guide/flow-chart-symbols [Accessed: 21 December 2022].

Enghiad, B., Xue, P., Singh, N., Boob, A.G., Shi, C., Petrov, V.A., Liu, R., Peri, S.S., Lane, S.T., Gaither, E.D. & Zhao, H. (2022) PlasmidMaker is a versatile, automated, and high throughput end-to-end platform for plasmid construction. *Nature Communications*. 13 (1), 2697. doi:10.1038/s41467-022-30355-y.

Engler, C., Kandzia, R. & Marillonnet, S. (2008) A One Pot, One Step, Precision Cloning Method with High Throughput Capability. *PLOS ONE*. 3 (11), e3647. doi:10.1371/journal.pone.0003647.

Exley, K., Reynolds, C.R., Suckling, L., Chee, S.M., Tsipa, A., Freemont, P.S., McClymont, D. & Kitney, R.I. (2019) Utilising datasheets for the informed automated design and build of a synthetic metabolic pathway. *Journal of Biological Engineering*. 13 (1), 8. doi:10.1186/s13036-019-0141-z.

Holowko, M.B., Frow, E.K., Reid, J.C., Rourke, M. & Vickers, C.E. (2021) Building a biofoundry. *Synthetic Biology*. 6 (1), ysaa026. doi:10.1093/synbio/ysaa026.

James, J.S., Jones, S., Martella, A., Luo, Y., Fisher, D.I. & Cai, Y. (2022) Automation and Expansion of EMMA Assembly for Fast-Tracking Mammalian System Engineering. *ACS Synthetic Biology*. 11 (2), 587–595. doi:10.1021/acssynbio.1c00330.

Kang, D.H., Ko, S.C., Heo, Y.B., Lee, H.J. & Woo, H.M. (2022) RoboMoClo: A Robotics-Assisted Modular Cloning Framework for Multiple Gene Assembly in Biofoundry. *ACS Synthetic Biology*. 11 (3), 1336–1348. doi:10.1021/acssynbio.1c00628.

Kanigowska, P., Shen, Y., Zheng, Y., Rosser, S. & Cai, Y. (2016) Smart DNA Fabrication Using Sound Waves. *Jala (Charlottesville, Va.)*. 21 (1), 49–56. doi:10.1177/2211068215593754.

Lampropoulos, A., Sutikovic, Z., Wenzl, C., Maegele, I., Lohmann, J.U. & Forner, J. (2013) GreenGate - A Novel, Versatile, and Efficient Cloning System for Plant Transgenesis. *PLoS ONE*. 8 (12), e83043. doi:10.1371/journal.pone.0083043.

Lin, D. & O'Callaghan, C.A. (2020) Hierarchical Modular DNA Assembly Using MetClo. In: S. Chandran & K.W. George (eds.). *DNA Cloning and Assembly*. Methods in Molecular Biology. New York, NY, Springer US. pp. 143–159. doi:10.1007/978-1-0716-0908-8_9.

Lin, D. & O'Callaghan, C.A. (2018) MetClo: methylase-assisted hierarchical DNA assembly using a single type IIS restriction enzyme. *Nucleic Acids Research*. 46 (19), e113. doi:10.1093/nar/gky596.

Marillonnet, S. & Grützner, R. (2020) Synthetic DNA Assembly Using Golden Gate Cloning and the Hierarchical Modular Cloning Pipeline. *Current Protocols in Molecular Biology*. 130 (1), e115. doi:10.1002/cpmb.115.

Moore, L.D., Le, T. & Fan, G. (2013) DNA Methylation and Its Basic Function. *Neuropsychopharmacology*. 38 (1), 23–38. doi:10.1038/npp.2012.112.

OpenTrons (2022) *OT-2 Liquid Handler | Opentrons Lab Automation from $5,000*. 2022. https://opentrons.com/ot-2/ [Accessed: 8 December 2022].

Ortiz, L., Pavan, M., McCarthy, L., Timmons, J. & Densmore, D.M. (2017) Automated Robotic Liquid Handling Assembly of Modular DNA Devices. *JoVE (Journal of Visualized Experiments)*. (130), e54703. doi:10.3791/54703.

Peng, B., Chen, H.-S., Mechanic, L.E., Racine, B., Clarke, J., Gillanders, E. & Feuer, E.J. (2015) Genetic data simulators and their applications: an overview. *Genetic epidemiology*. 39 (1), 2–10. doi:10.1002/gepi.21876.

Pereira, F., Azevedo, F., Carvalho, Â., Ribeiro, G.F., Budde, M.W. & Johansson, B. (2015) Pydna: a simulation and documentation tool for DNA assembly strategies using python. *BMC Bioinformatics*. 16 (1), 142. doi:10.1186/s12859-015-0544-x.

Potapov, Ong, J.L., Kucera, R.B., Langhorst, B.W., Bilotti, K., Pryor, J.M., Cantor, E.J., Canton, B., Knight, T.F., Evans, T.C. & Lohman, G.J.S. (2018a) *Optimization of Golden Gate assembly through application of ligation sequence-dependent fidelity and bias profiling*.p.322297. doi:10.1101/322297.

Potapov, V., Ong, J.L., Kucera, R.B., Langhorst, B.W., Bilotti, K., Pryor, J.M., Cantor, E.J., Canton, B., Knight, T.F., Evans, T.C. & Lohman, G.J.S. (2018b) Comprehensive Profiling of Four Base Overhang Ligation Fidelity by T4 DNA Ligase and Application to DNA Assembly. *ACS Synthetic Biology*. 7 (11), 2665–2674. doi:10.1021/acssynbio.8b00333.

Pryor, J.M., Potapov, V., Kucera, R.B., Bilotti, K., Cantor, E.J. & Lohman, G.J.S. (2020) Enabling one-pot Golden Gate assemblies of unprecedented complexity using data-optimized assembly design. *PLOS ONE*. 15 (9), e0238592. doi:10.1371/journal.pone.0238592.

Sarrion-Perdigones, A., Falconi, E.E., Zandalinas, S.I., Juárez, P., Fernández-del-Carmen, A., Granell, A. & Orzaez, D. (2011) GoldenBraid: An Iterative Cloning System for Standardized Assembly of Reusable Genetic Modules. *PLOS ONE*. 6 (7), e21622. doi:10.1371/journal.pone.0021622.

Storch, M., Haines, M.C. & Baldwin, G.S. (2019) *DNA-BOT: A low-cost, automated DNA assembly platform for synthetic biology*. doi:10.1101/832139.

Van Hove, B., Guidi, C., De Wannemaeker, L., Maertens, J. & De Mey, M. (2017) Recursive DNA Assembly Using Protected Oligonucleotide Duplex Assisted Cloning (PODAC). *ACS Synthetic Biology*. 6 (6), 943–949. doi:10.1021/acssynbio.7b00017.

Vasilev, V., Liu, C., Haddock, T., Bhatia, S., Adler, A., Yaman, F., Beal, J., Babb, J., Weiss, R. & Densmore, D. (2011) *A Software Stack for Specification and Robotic Execution of Protocols for Synthetic Biological Engineering*.

Vazquez-Vilar, M., Orzaez, D. & Patron, N. (2018) DNA assembly standards: Setting the low-level programming code for plant biotechnology. *Plant Science*. 273, 33–41. doi:10.1016/j.plantsci.2018.02.024.

Vegh, P. (2022) *EGF's Compendium of overhangs*. https://github.com/Edinburgh-Genome-Foundry/Overhang.

Vegh, P. (2020) *GitHub - Edinburgh-Genome-Foundry/kappagate: Predict DNA assembly clone validity rates - powered by Kappa*. 2020. https://github.com/Edinburgh-Genome-Foundry/kappagate [Accessed: 14 December 2022].

Walsh, D.I., Pavan, M., Ortiz, L., Wick, S., Bobrow, J., Guido, N.J., Leinicke, S., Fu, D., Pandit, S., Qin, L., Carr, P.A. & Densmore, D. (2019) Standardizing Automated DNA Assembly: Best Practices, Metrics, and Protocols Using Robots. *SLAS Technology*. 24 (3), 282–290. doi:10.1177/2472630318825335.

Weber, E., Engler, C., Gruetzner, R., Werner, S. & Marillonnet, S. (2011) A Modular Cloning System for Standardized Assembly of Multigene Constructs. *PLOS ONE*. 6 (2), e16765. doi:10.1371/journal.pone.0016765.

Young, R., Haines, M., Storch, M. & Freemont, P.S. (2021) Combinatorial metabolic pathway assembly approaches and toolkits for modular assembly. *Metabolic Engineering*. 63, 81–101. doi:10.1016/j.ymben.2020.12.001.

Zhang, M., McLaughlin, J.A., Wipat, A. & Myers, C.J. (2017) SBOLDesigner 2: An Intuitive Tool for Structural Genetic Design. *ACS Synthetic Biology*. 6 (7), 1150–1160. doi:10.1021/acssynbio.6b00275.

Zulkower, V. (2021) Computer-Aided Planning for the Verification of Large Batches of DNA Constructs. In: F. Menolascina (ed.). *Synthetic Gene Circuits*. Methods in Molecular Biology. New York, NY, Springer US. pp. 167–174. doi:10.1007/978-1-0716-1032-9_7.

Zulkower, V. (2020) *DNA Cauldron Documentation*. https://edinburgh-genome-foundry.github.io/DnaCauldron/index.html.

Appendix

Figures

Script

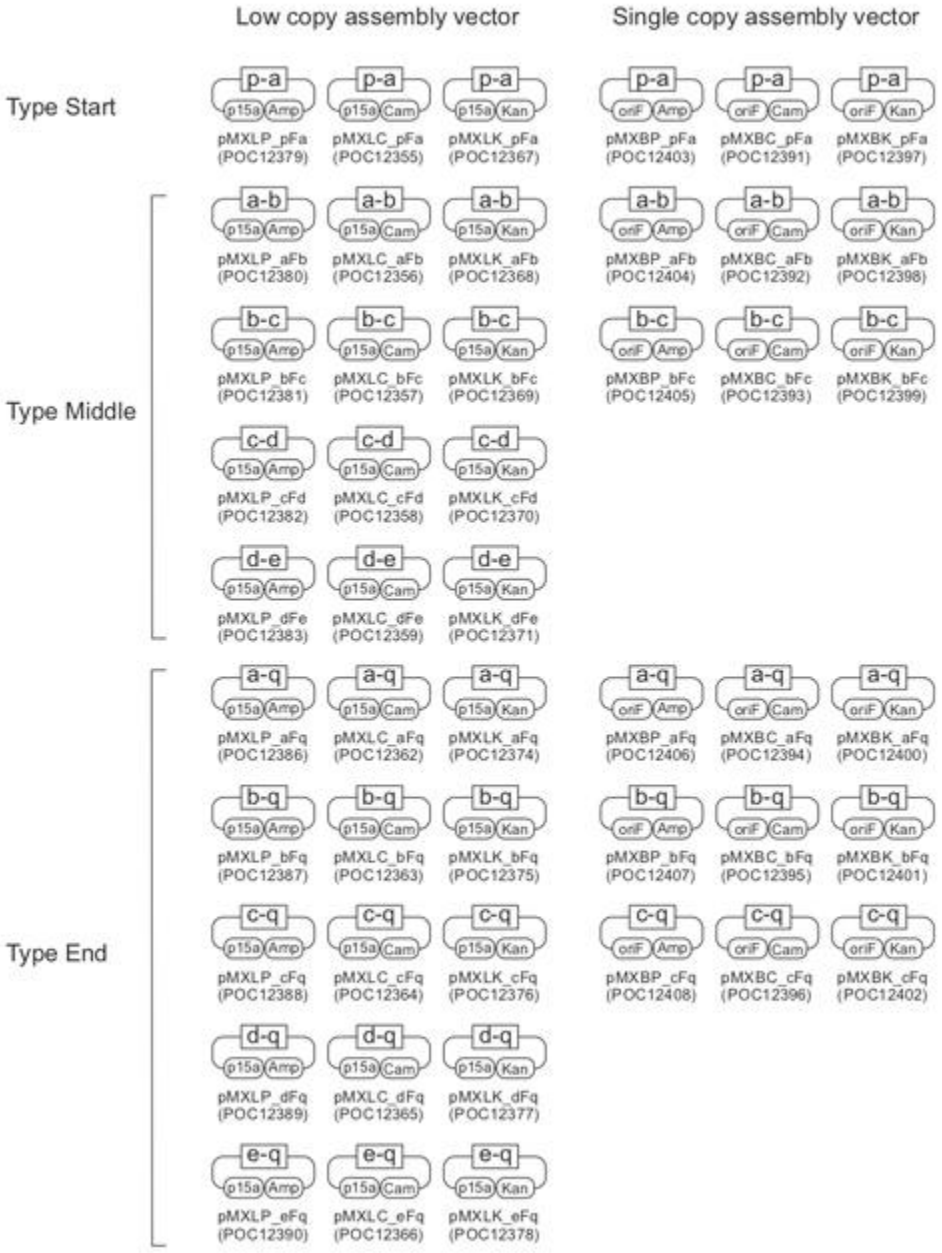


Figure 18 - Metclo Vector set *(Lin & O'Callaghan, 2018)**. The MetClo vector set contains a set of 30 low (p15a replication origin) and 18 high (F and ColEi replication origin) copy vectors. The vectors vary in antibiotic selection markers (ampicillin, kanamycin, and chloramphenicol) and adapter sets. These adapters, and subsequently the vector they belong to, determine the position of the parts. All vectors use LacZα is the negative selection marker.*

Edinburgh Genome Foundry

Overhang set report

⚠ MetClo

There are **7** overhangs in this set. The restriction enzyme used for this set is **BsaI**.

Overhangs in this set: GGAG, GCGA, AGGT, ACGG, TGTC, TGCT, CAAC.

Warning! **Weakly annealing overhang(s): CTCC/GGAG; CAAC/GTTG**

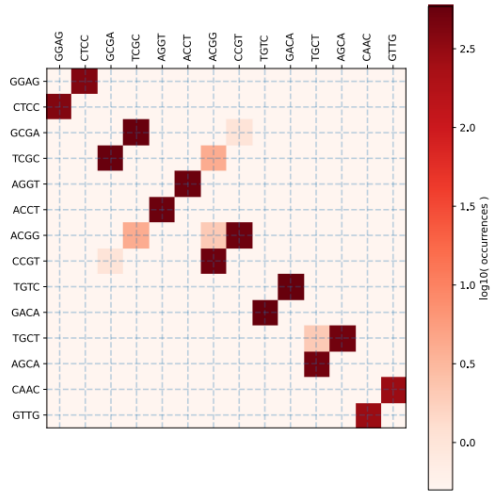
Warning! **The following may have self-misannealing issues: ACGG/CCGT; AGCA/TGCT**

No overhangs (including reverse complements) differ by fewer than 2 nucleotides.

Please see the Appendix on the last page for an explanation of details.

EGF's compendium of overhangs (version 0.1.6)

Tatapov annealing plot:



EGF's compendium of overhangs (version 0.1.6)

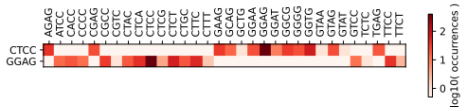
☑ CTCC GGAG

GC content: **75 %**.

Can form the following amino acids in 6 translation frames:

L[PLHRQ]
[CYPVDFILASNHRTG]S
[STAP][P]
G[VDAEG]
[KW*PVALMSRTEGQ]E
[RWG][SR]

Misannealing overhangs:



EGF's compendium of overhangs (version 0.1.6)

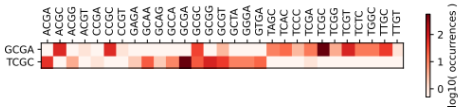
☑ GCGA TCGC

GC content: **75 %**.

Can form the following amino acids in 6 translation frames:

A[KIMSNRT]
[KW*PVALMSRTEGQ]R
[SRCG][DE]
S[PLHRQ]
[CYPVDFILASNHRTG]R
[VFIL][A]

Misannealing overhangs:



EGF's compendium of overhangs (version 0.1.6)

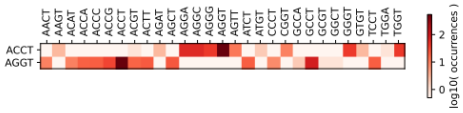
☑ ACCT AGGT

GC content: **50 %**.

Can form the following amino acids in 6 translation frames:

T[CWY*FLS]
[K*PVILASRTEGQ]P
[HNDY][L]
R[CWY*FLS]
[K*PVILASRTEGQ]G
[*KEQ][V]

Misannealing overhangs:



EGF's compendium of overhangs (version 0.1.6)

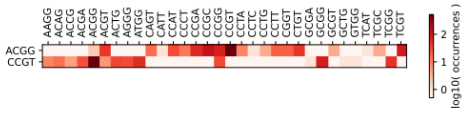
☑ ACGG CCGT

GC content: **75 %**.

Can form the following amino acids in 6 translation frames:

T[VDAEG]
[K*PVILASRTEGQ]R
[HNDY][G]
P[CWY*FLS]
[CYPVDFILASNHRTG]R
[STAP][V]

Misannealing overhangs:



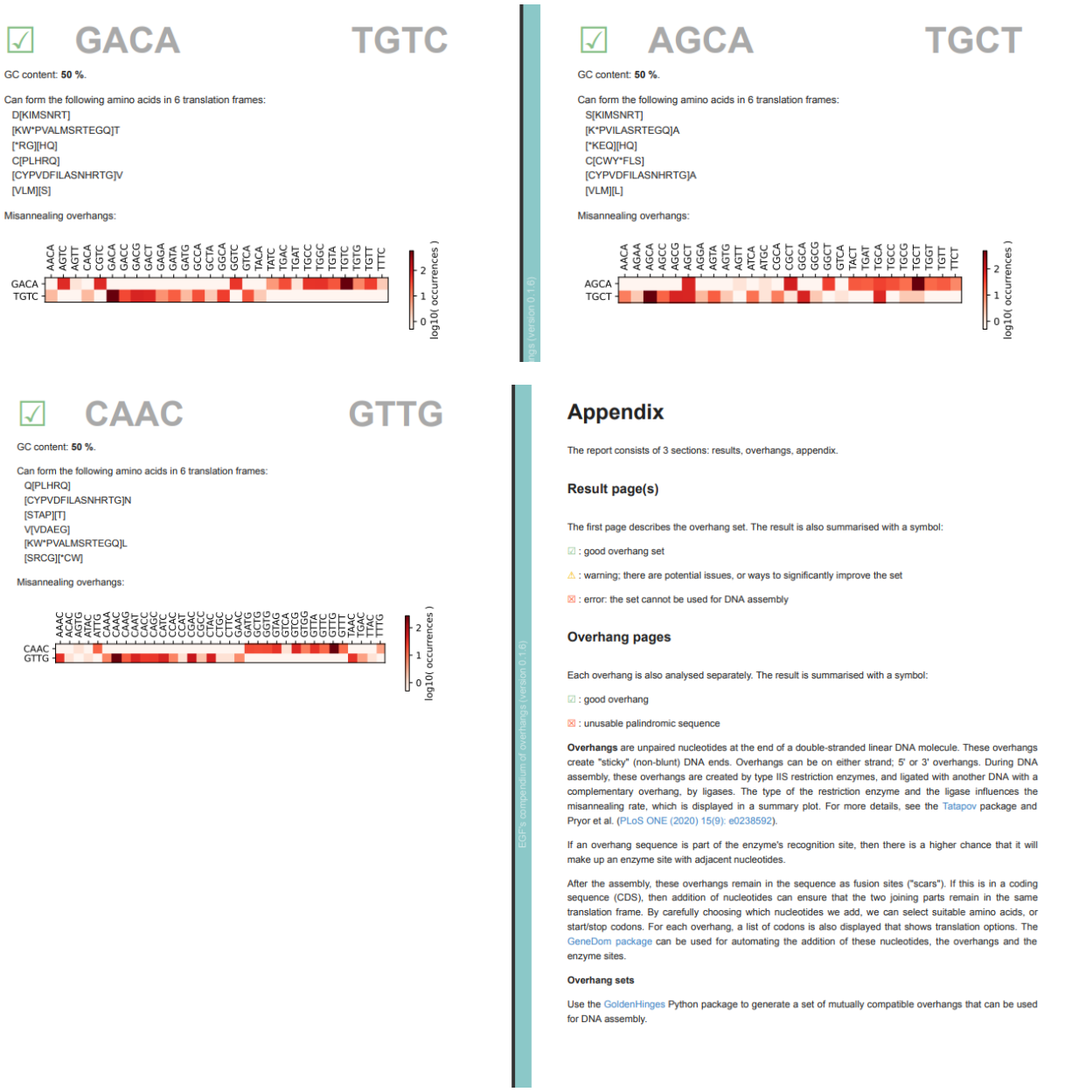


Figure 19 – Metclo's full overhang report generated from a simulated assembly with the use of BsaI restriction enzyme at 37°C for 1 hour within EGF's Overhangs (Vegh, 2022) – *The report summarises the findings of the assembly simulation. It provides a detailed report of each overhang, their interactions within the overhang group, and with itself. The figures within the report show the propensity (log_{10} occurrence) of the overhangs to ligate with each other and themselves. The strong ligation-efficiency of the correct overhang junction is represented by two dark red points while the weakest ligations interactions are white. Ideally, the adapter set should only ligate efficiently with their intended compliment, any deviation of this represents possible missannealing, self-missannealing or weak-annealing.*

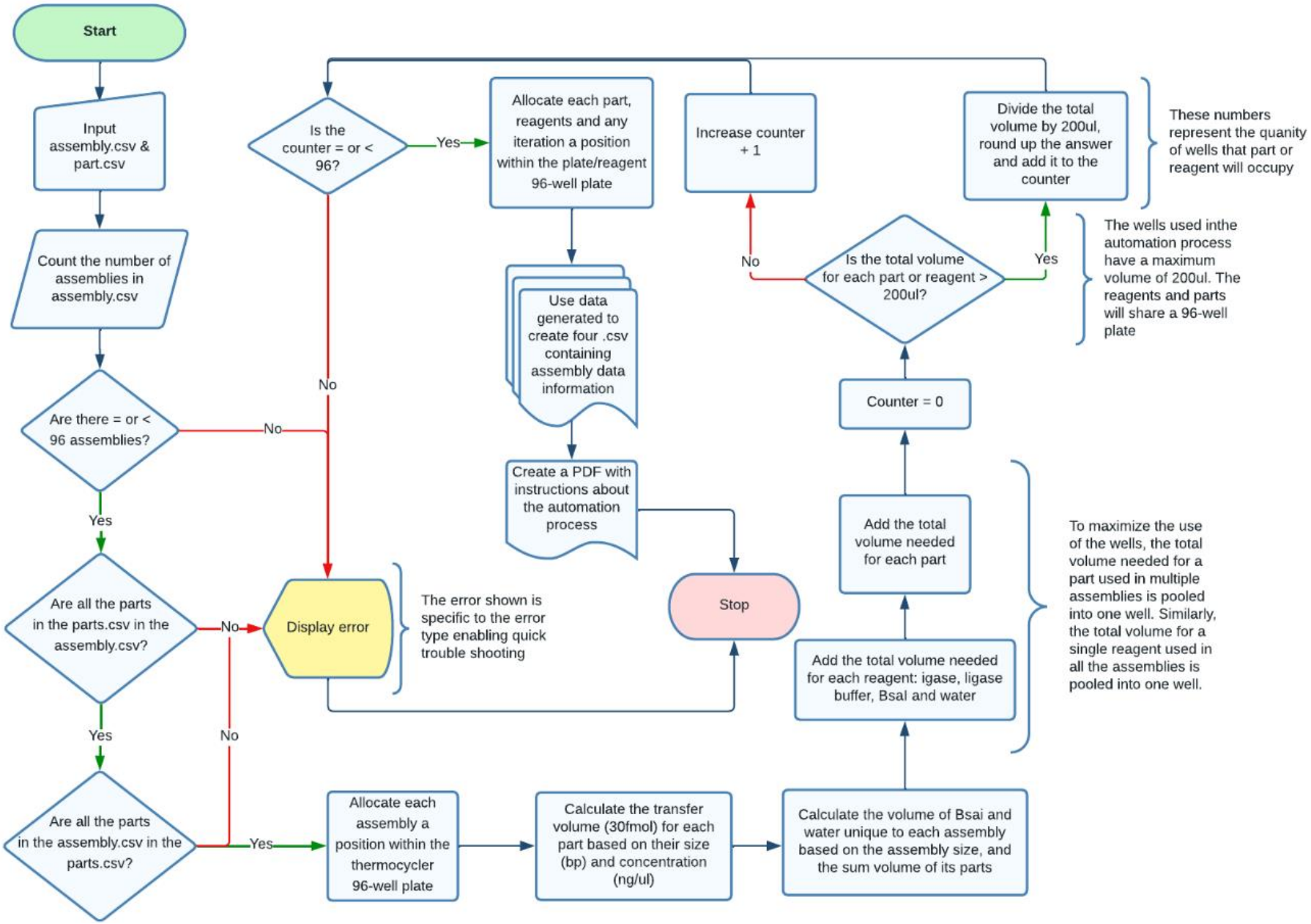

Start
Input assembly.csv & part.csv
Count the number of assemblies in assembly.csv
Are there = or < 96 assemblies?
No
Yes
Are all the parts in the parts.csv in the assembly.csv?
No
Display error
The error shown is specific to the error type enabling quick trouble shooting
Yes
Are all the parts in the assembly.csv in the parts.csv?
No
Yes
Allocate each assembly a position within the thermocycler 96-well plate
Calculate the transfer volume (30fmol) for each part based on their size (bp) and concentration (ng/ul)
Calculate the volume of Bsai and water unique to each assembly based on the assembly size, and the sum volume of its parts
Add the total volume needed for each reagent: igase, ligase buffer, Bsal and water
Add the total volume needed for each part
Counter = 0
Is the total volume for each part or reagent > 200ul?
No
Yes
Divide the total volume by 200ul, round up the answer and add it to the counter
These numbers represent the quanity of wells that part or reagent will occupy
The wells used inthe automation process have a maximum volume of 200ul. The reagents and parts will share a 96-well plate
To maximize the use of the wells, the total volume needed for a part used in multiple assemblies is pooled into one well. Similarly, the total volume for a single reagent used in all the assemblies is pooled into one well.
Increase counter + 1
Is the counter = or < 96?
Yes
No
Allocate each part, reagents and any iteration a position within the plate/reagent 96-well plate
Use data generated to create four .csv containing assembly data information
Create a PDF with instructions about the automation process
Stop

Figure 20 - metclo_plan.py algorithm flow chart detailing the step-by-step processes in the script. *Refer to Figure 7 for shape descriptions. The green and red arrows are 'yes' and 'n' responses to decisions.*

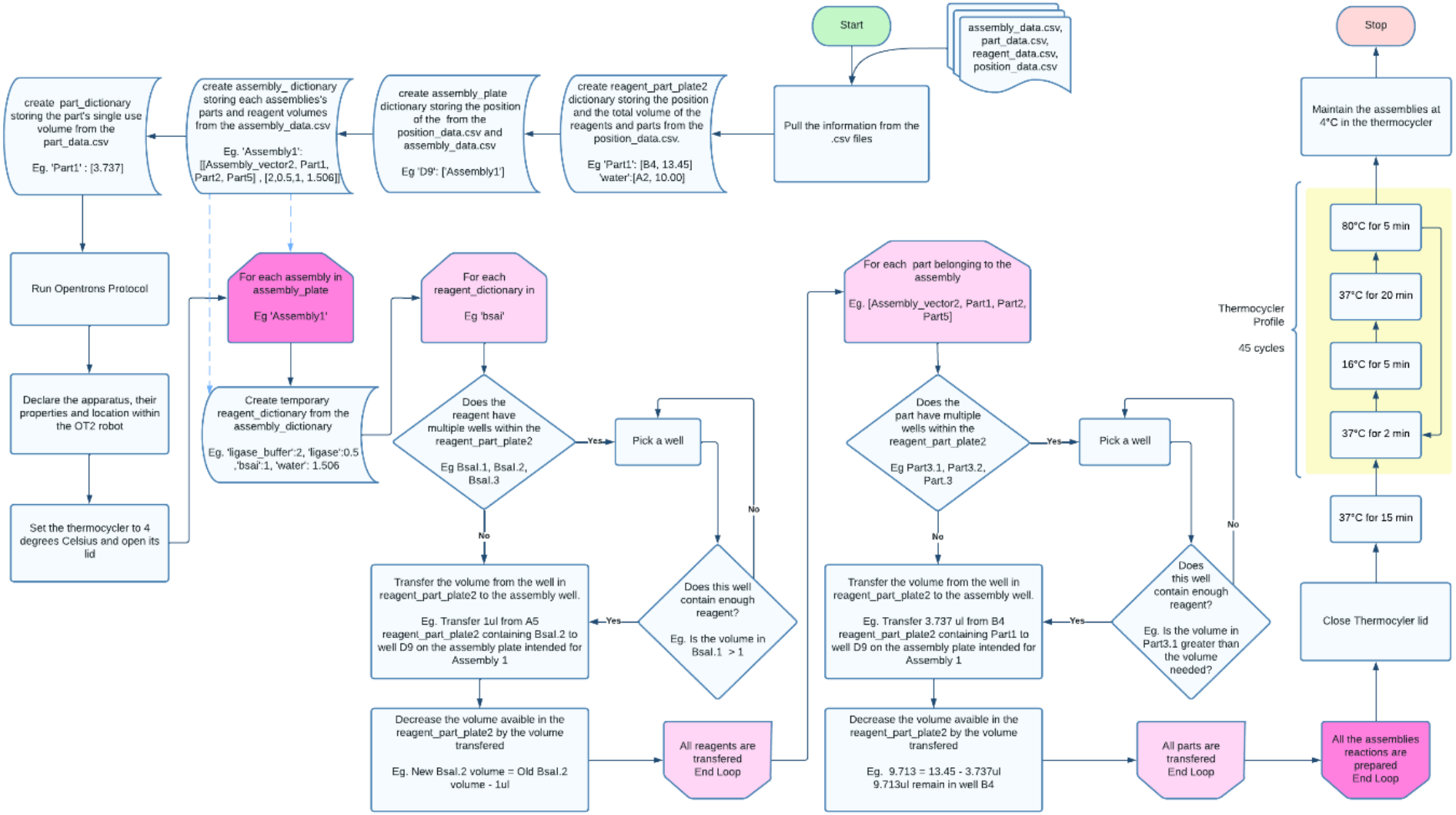

Start
assembly_data.csv, part_data.csv, reagent_data.csv, position_data.csv
Pull the information from the .csv files
create reagent_part_plate2 dictionary storing the position and the total volume of the reagents and parts from the position_data.csv.
Eg 'Part1': [B4, 13.45] 'water':[A2, 10.00]
create assembly_plate dictionary storing the position of the from the position_data.csv and assembly_data.csv
Eg 'D9': ['Assembly1']
create assembly_ dictionary storing each assemblies's parts and reagent volumes from the assembly_data.csv
Eg. 'Assembly1': [[Assembly_vector2, Part1, Part2, Part5] , [2,0.5,1, 1.506]]
create part_dictionary storing the part's single use volume from the part_data.csv
Eg. 'Part1' : [3.737]
Run Opentrons Protocol
Declare the apparatus, their properties and location within the OT2 robot
Set the thermocycler to 4 degrees Celsius and open its lid
For each assembly in assembly_plate
Eg 'Assembly1'
Create temporary reagent_dictionary from the assembly_dictionary
Eg. 'ligase_buffer':2, 'ligase':0.5 ,'bsai':1, 'water': 1.506
For each reagent_dictionary in
Eg 'bsai'
Does the reagent have multiple wells within the reagent_part_plate2
Eg Bsal.1, Bsal.2, Bsal.3
Yes
Pick a well
Does this well contain enough reagent?
Eg. Is the volume in Bsal.1 > 1
No
Yes
No
Transfer the volume from the well in reagent_part_plate2 to the assembly well.
Eg. Transfer 1ul from A5 reagent_part_plate2 containing Bsal.2 to well D9 on the assembly plate intended for Assembly 1
Decrease the volume avaible in the reagent_part_plate2 by the volume transfered
Eg. New Bsal.2 volume = Old Bsal.2 volume - 1ul
All reagents are transfered End Loop
For each part belonging to the assembly
Eg. [Assembly_vector2, Part1, Part2, Part5]
Does the part have multiple wells within the reagent_part_plate2
Eg Part3.1, Part3.2, Part.3
Yes
Pick a well
Does this well contain enough reagent?
Eg. Is the volume in Part3.1 greater than the volume needed?
No
Yes
No
Transfer the volume from the well in reagent_part_plate2 to the assembly well.
Eg. Transfer 3.737 ul from B4 reagent_part_plate2 containing Part1 to well D9 on the assembly plate intended for Assembly 1
Decrease the volume avaible in the reagent_part_plate2 by the volume transfered
Eg. 9.713 = 13.45 - 3.737ul
9.713ul remain in well B4
All parts are transfered End Loop
Thermocycler Profile
45 cycles
80°C for 5 min
37°C for 20 min
16°C for 5 min
37°C for 2 min
37°C for 15 min
Close Thermocyler lid
All the assemblies reactions are prepared End Loop
Maintain the assemblies at 4°C in the thermocycler
Stop

Figure 21 - ot2metclo.py algorithm flow chart detailing the step-by-step processes in the script. *Refer to Figure 7 for shape descriptions. The dotted blue arrow shows the movement of assembly data. Two light pink loop limits occur within a larger dark pink loop limit. The yellow area is the thermocycler steps.*

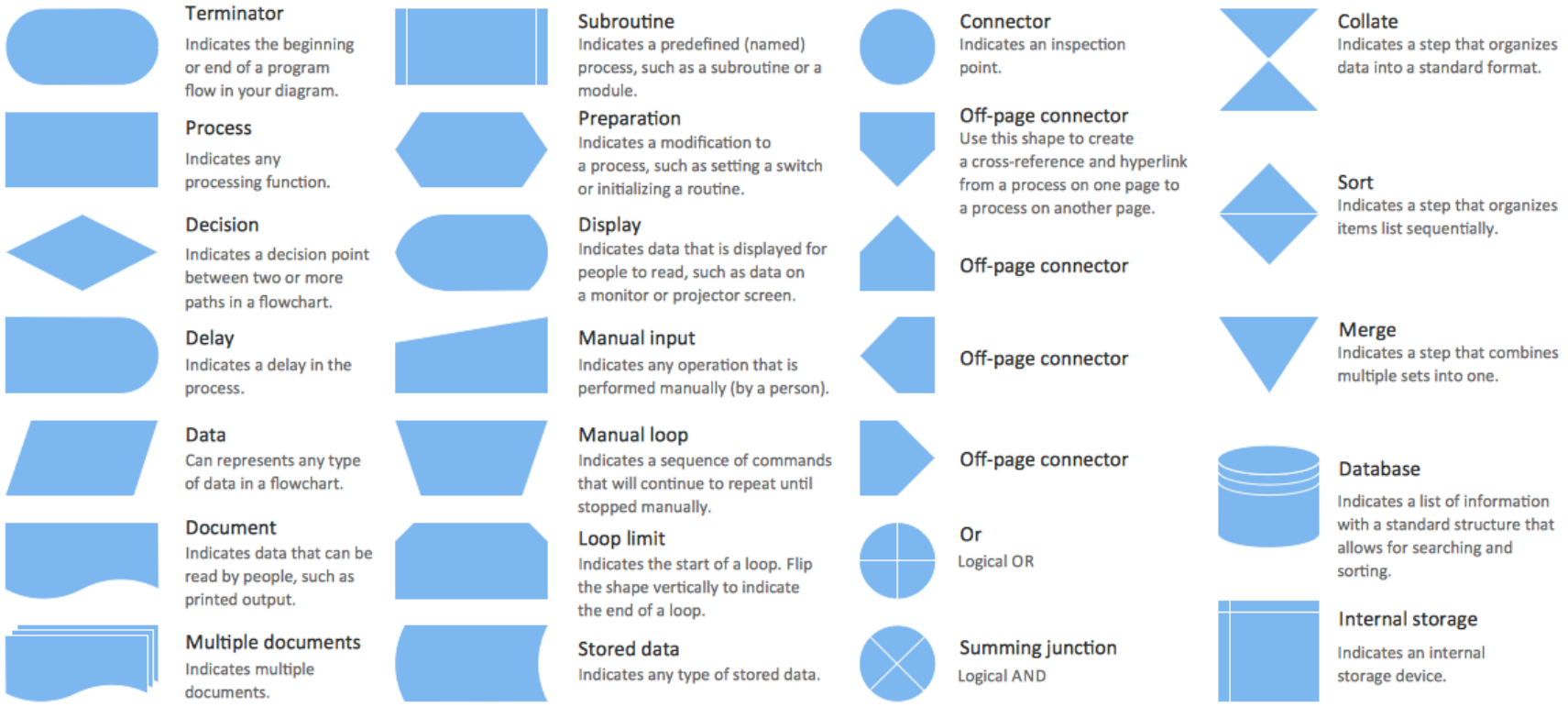


Figure 22 - Flow Cart Symbols and Descriptions *(ConceptDraw, n.d.).*

Automated MetClo Assembly Plan

OT2 Set-Up Instructions

Welcome to the MetcloOT2, plan specific to your assemblies!

This documentation is a userguide that instructs the setup of the OT2 robot for multiplexed MetClo based DNA assembly. Reading Lin & O'Callaghan (2018 & 2020) is highly recomended.

MATERIALS

> Software:

- OpenTrons OT-2 App
- Python 3

> FILES within metclo_plan_files:

- assembly_data.csv
- part_data.csv
- position_data.csv
- reagents_data.csv

> Hardware:

- OpenTron OT-2
- OpenTrons P20 Single Channel Electronic Pipette
- Opentrons Thermocycler
- Opentrons 96 Tip Rack 20ul

> Consumables and reagents

- 30fmol of all part containing insert plasmids and assembly vectors.
- T4 ligase buffer
- T4 DNA ligase
- BsaI-HFv2
- ddH20

PROTOCOL

1. Within the terminal, simulate the OpenTrons OT-2 protocol: ot2metclo.py. Review and confirm the output of the simulation. https://support.opentrons.com/s/article/Simulating-OT-2-protocols-on-your-computer?)

2. Follow the OpenTrons OT-2 get-started guidelines to prepare the Opentron OT2. https://support.opentrons.com/s/ot2-get-started).

3. Prepare the insert plasmids and assembly vectors as described in Lin & O' Callaghan (2020).

4. As mentioned in the README.md, dilute the plasmids and vectors where necessary.

5. Set up the OpenTrons OT-2 deck as depicted in the "OT2 LAYOUT" section. Additionally attach the P20 Single Channel Electronic Pipette to the left mount.

6. The "REAGENT PLATE LAYOUT (ul)" depicts the positioning and volume of the reagents and the parts needed for the assembly. Manually pipette the volume of reagents and parts to their respective well. Example [A1 ('ligase_buffer.1',7.2)] means that 7.2ul of T4 ligase buffer needs to be allocated to A1.

7. Assure that the metclo_plan_files produced from metclo_plan.py and the ot2metclo.py are within the same file. Transfer this file to the computer which is connected to the OT2 robot and has the downloaded OpenTrons OT-2 App.

8. Run the ot2metclo.py protocol on the OpenTrons OT-2 App. For assistance refer to OT - 2 : Getting Started. https://support.opentrons.com/s/ot2-get-started).

Automated MetClo Assembly Plan

9. The final assemblies will be stored within the thermocycler module at 4 degrees Celsius. The positions of the assemblies are depicted in "THERMOCYCLER PLATE WITH ASSEMBLIES". These assemblies can be used for further applications.

NOTES

- The "Assemblies" section provides a summary of the assemblies' size, recommended transformation method based on size, parts and reagent volumes.

- The "Parts" section provides a summary of the parts' size, concentration (ng/ul), number of assemblies in which the part was used, the volume containing 30fmol, and the total volume of the part needed for the protocol.

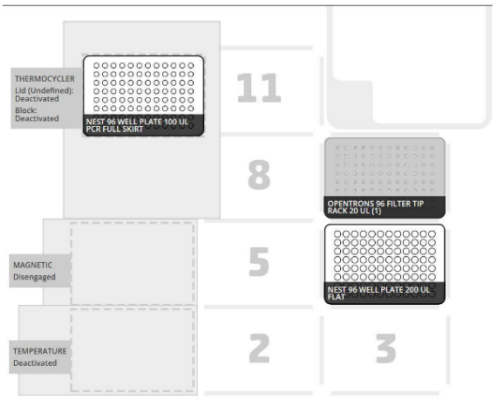
REFERENCE

Lin, D. & O'Callaghan, C.A. (2018) MetClo: methylase-assisted hierarchical DNA assembly using a single type IIS restriction enzyme. Nucleic Acids Research. 46 (19), e113. doi:10.1093/nar/gky596.

Lin, D. & O'Callaghan, C.A. (2020) Hierarchical Modular DNA Assembly Using MetClo. In: S. Chandran & K.W. George (eds.). DNA Cloning and Assembly. Methods in Molecular Biology. New York, NY, Springer US. pp. 143-159. doi:10.1007/978-1-0716-0908-8_9.

Automated MetClo Assembly Plan

OT2 Layout



Automated MetClo Assembly Plan

Reagent Plate Layout (ul)

A1 ('ligase_buffer', 7.2)	B1 ('ligase', 1.8)	C1 ('bsai', 2.4)	D1 ('water', 1.807)
A2 ('bq', 7.163)	B2 ('bc', 5.147)	C2 ('cq', 12.569)	D2
A3	B3	C3	D3
A4	B4	C4	D4
A5	B5	C5	D5
A6	B6	C6	D6
A7	B7	C7	D7
A8	B8	C8	D8
A9	B9	C9	D9
A10	B10	C10	D10
A11	B11	C11	D11
A12	B12	C12	D12

E1 ('MpMXBP_pFa', 13.453)	F1 ('pa', 23.537)	G1 ('aq', 6.263)	H1 ('ab', 18.984)
E2	F2	G2	H2
E3	F3	G3	H3
E4	F4	G4	H4
E5	F5	G5	H5
E6	F6	G6	H6
E7	F7	G7	H7
E8	F8	G8	H8
E9	F9	G9	H9
E10	F10	G10	H10
E11	F11	G11	H11
E12	F12	G12	H12

Automated MetClo Assembly Plan

Thermocycler Plate with Assemblies

A1 2frag	B1 3frag	C1 4frag	D1
A2	B2	C2	D2
A3	B3	C3	D3
A4	B4	C4	D4
A5	B5	C5	D5
A6	B6	C6	D6
A7	B7	C7	D7
A8	B8	C8	D8
A9	B9	C9	D9
A10	B10	C10	D10
A11	B11	C11	D11
A12	B12	C12	D12

E1	F1	G1	H1
E2	F2	G2	H2
E3	F3	G3	H3
E4	F4	G4	H4
E5	F5	G5	H5
E6	F6	G6	H6
E7	F7	G7	H7
E8	F8	G8	H8
E9	F9	G9	H9
E10	F10	G10	H10
E11	F11	G11	H11
E12	F12	G12	H12

Automated MetClo Assembly Plan

3 Assemblies

Assembly Name: 2frag

Assembly Size: 21,949 (Recomend: Electroporation)

3 Parts

MpMXBP_pFa	pa	aq

Reagents (ul)

ligase_buffer	ligase	bsai	water
2	0.5	0.5	1.506

Assembly Name: 3frag

Assembly Size: 29,494 (Recomend: Electroporation)

4 Parts

MpMXBP_pFa	pa	ab	bq

Reagents (ul)

ligase_buffer	ligase	bsai	water
2	0.5	0.5	0

Assembly Name: 4frag

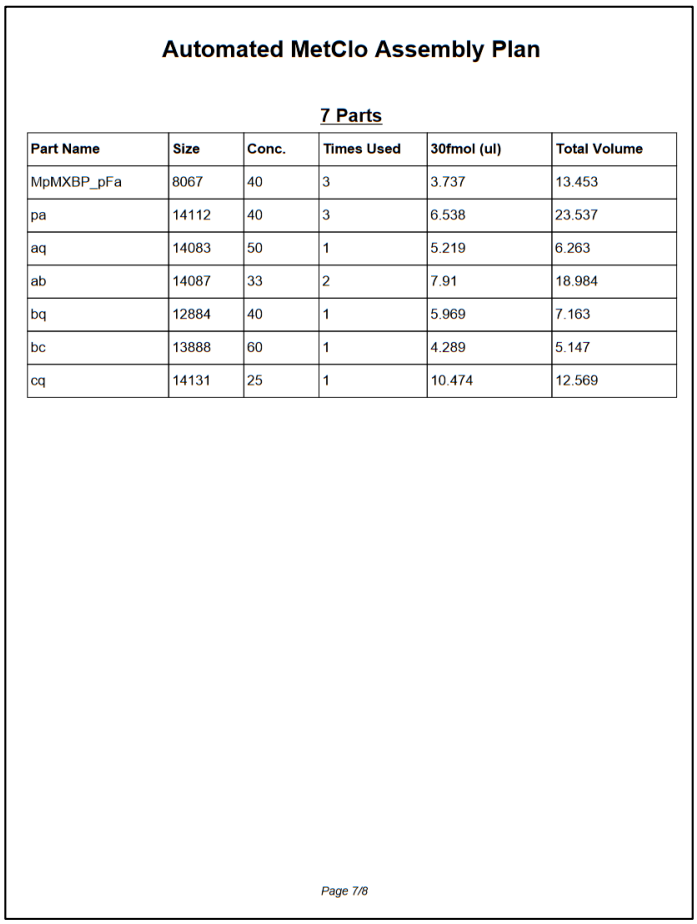
Assembly Size: 37,286 (Recomend: Electroporation)

5 Parts

MpMXBP_pFa	pa	ab	bc
cq			

Reagents (ul)

ligase_buffer	ligase	bsai	water
2	0.5	1.0	0

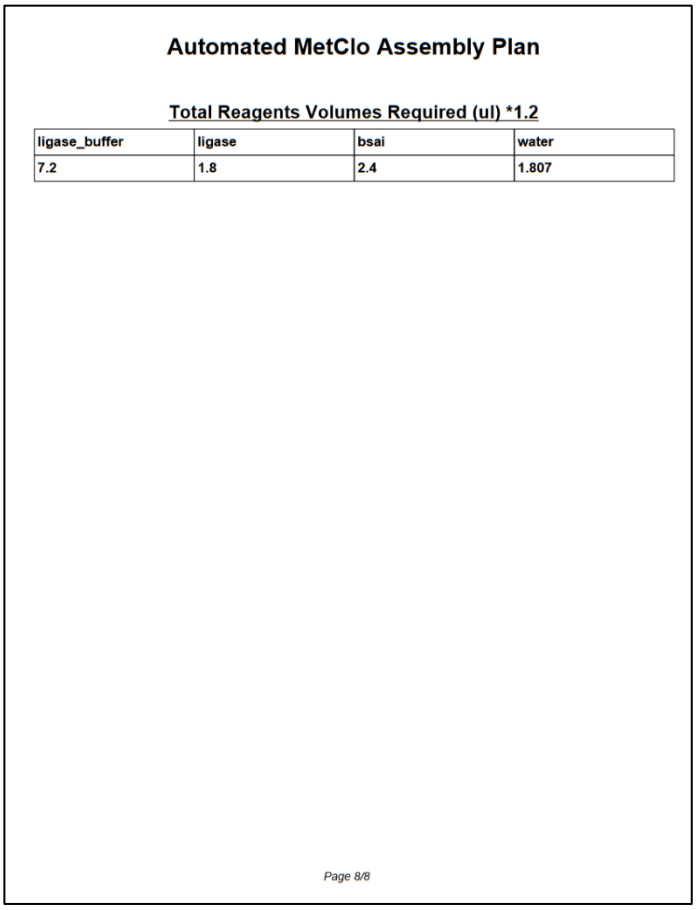


Automated MetClo Assembly Plan

7 Parts

Part Name	Size	Conc.	Times Used	30fmol (ul)	Total Volume
MpMXBP_pFa	8067	40	3	3.737	13.453
pa	14112	40	3	6.538	23.537
aq	14083	50	1	5.219	6.263
ab	14087	33	2	7.91	18.984
bq	12884	40	1	5.969	7.163
bc	13888	60	1	4.289	5.147
cq	14131	25	1	10.474	12.569

Page 7/8



Automated MetClo Assembly Plan

Total Reagents Volumes Required (ul) *1.2

ligase_buffer	ligase	bsai	water
7.2	1.8	2.4	1.807

Page 8/8

Figure 23 - metclo_plan.pdf contains instructions and steps for the user to perform an exemplary assembly protocol. *The plan contains the OT2 dock layout, the layout of the reagent/part and assembly plate, detailed information about each assembly, part and reagent.*

Figu Edinburgh-Genome-Foundry / **OT2Metclo** Private

ory page. The repository has the necessary

Code | Issues 1 | Pull requests | Actions | Projects | Wiki | Security | Insights | Settings

main | 1 branch | 0 tags | Go to file | Add file | Code

DanyMatute Update README.md e4ac05c on Aug 16 163 commits

doc	Add files via upload	4 months ago
example	conducted changes as per email	4 months ago
metclo	Many changes to file made. Can succesfully retreive xsl data and mani...	6 months ago
.gitignore	Initial commit	7 months ago
README.md	Update README.md	4 months ago
metclo_plan.py	comapred and checked to branch and coppies on computer	4 months ago
ot2metclo.py	Merge branch 'main' into egf	4 months ago

MetCloOT2: Automated MetClo assembly on OpenTrons OT-2 liquid handling robot

About MetCloOT2

Welcome to MetCloOT2. The purpose of this repository is to enable the automation of MetClo based assemblies within the Opentrons OT2 liquid handling robot. The program is based on Lin & O'Callaghan (2018) and Lin & O'Callaghan (2020). Reading these two papers prior to OT2Metclo implementation is highly recommended.

This repository is part of a dissertation project necessary to fulfil the requirements of the MSc Synthetic Biology and Biotechnology programme at the University of Edinburgh.

How to

1. Prior to using MetCloOT2, assure the quality of the assembly by means of computer aided designs and simulations, such tool can be found at the EGF's Cuba suite or at Edinburgh Genome Foundry's GitHub page.
2. Create 2 input .csv files, one containing information of the assemblies, and the other information about the part plasmids and assembly vector. Within the *example/* folder there are two sample files depicting the expected layout of the .csv files. If Simulate Golden Gate Assemblies was used to simulate the assemblies (as recommended in step 1), the assembly plan can be used as the assembly.csv in MetCloOT2 with a slight modification. OT2Metclo requires an assembly-size column to be added to the assembly plan.
3. Assure that the volume of 30fmol of all part plasmids and assembly vectors is at least 1µl. Use the following formula:

$$volume\ (\mu l) = \frac{30(617.69 * \boldsymbol{part\ length}[\boldsymbol{bp}])}{\boldsymbol{part\ concentration}\left[\frac{\boldsymbol{ng}}{\boldsymbol{\mu l}}\right] * 10^6}$$

   Or copy this formula into excel:

   *=round((18530.7 * [part length (bp)] + 1081.2) / ([part concentration (ng/ul)] * 10^6) ,3)*

   If the volume is lower than 1µl, then dilute the plasmid. If the volume is equal to or greater than 1µl, no dilution is necessary.

Example

	Donor plasmids 1	Donor Plasmid 2
size (bp)	14000	4500
concentration (ng/µl)	50	90
volume containing 30fmol (µl)	5.189	0.927

Plasmid 2 volume is too small to accurately be pipetted by the Opentron's P20 Gen 2 pipette. Manually adding and mixing 9.073 µl of a dilution solvent will increase the volume containing 30fmol to 10 µl. The pipette will be more accurate transferring this volume.
No dilution is necessary for plasmid 1 as 5.189 µl > 1 µl

4. Install or check for the presence of the following packages: csv, math, string, sys, fpdf, opentrons, string and sys. Confirm that Python 3 is installed. Confirm that OT2 APP is downloaded and UpToDate.
5. Clone this MetcloOT2 repository from GitHub.
6. Open the terminal, navigate to the location of the cloned repository. Run **metclo_plan.py** by typing `python`

README.md

- metclo_plan.pdf
- assembly_data.csv
- part_data.csv
- position_data.csv
- reagents_data.csv

8. Read and confirm the assembly and part information in `metclo_plan.pdf`.
9. Follow the instructions found in `metclo_plan.pdf`.

Supervisor

Peter Vegh

Author

Daniella Matute

files to run the ot2metclo.py and metclo_plan.py. It contains examples of files and outputs. A “Read me” is included and depicted, it should be read prior using the pipeline.

Script 1 – metclo_plan.py python script. *Purple areas contain commentary of the code.*

```
# Imports other software packages that will be needed in the code
import csv, math, string
import sys, os
from fpdf import FPDF
from fpdf.enums import XPos, YPos

# Counts the instances a part is used throughout all assemblies
def _countinstances(uncompressed_parts):
    count_parts = {}
    for i in uncompressed_parts:
        count_parts[i] = uncompressed_parts.count(i)
    return count_parts


# Calculates the volume with 30fmol of parts.
def _calcvolume(ngul, bp):
    volume = round(30 / (((ngul * 1e-9) / ((bp * 617.69) + 36.04)) * 1e15), 3)
    return volume


# Calculated the BsaI and water volume unique to each assembly
def _calcreagents(assembly_size, part_volumes):
    ligase_buffer = 2.0
    ligase = 0.5
    # The assembly needs more BsaI if it is over 30,000bp
    bsai = 1.0 if assembly_size > 30000 else 0.5
    sum_parts = sum([ligase_buffer,
                     ligase,
                     bsai,
                     part_volumes]
                    )
    # Water brings the reaction mixture up to a volume of 20 ul
    water = round(20 - sum_parts, 3) if sum_parts < 20 else 0
    return bsai, water


# Makes .csv files. These will serve as inputs for the ot2metclo.py which contains the OpenTrons
protocol
def _makecvs(doc, header, data):
    with open(doc, "w") as f:
        writer = csv.writer(f)
        writer.writerow(header)
        try:
            for i in data:
                row = [i]
                for j in data[i]:
                    row.append(j)
                writer.writerow(row)
```

```
        except:
            for i in data:
                row = [i, data[i]]
                writer.writerow(row)
    print(doc, " written successfully.")


# The maximum volume within well on the 96 well-plate is 200ul.
# The method checks if the volume needed is over 200ul, if so, the volume is divided into
different wells.
def _volumecheck(i, x):
    wellvolume = []
    plate = []
    count = 0
    if x / 200 < 1:
        plate.append(i)
        count += 1
        wellvolume.append(x)
    else:
        wells = math.ceil(x / 200)
        for j in range(wells):
            if x - 200 > 0:
                wellvolume.append(200)
                x = x - 200
            else:
                wellvolume.append(x)
        for t in range(wells):
            plate.append(i + "." + str(t + 1))
            count += 1
    return plate, wellvolume, count


# Checks if the parts within the two files are congruent with each other. Will flag an error if a
part is missing from the assembly and part file.
def _check_part_congruency(
    parts_concentration_size, part_count, part_file_name, assembly_file_name
):
    irregular_parts = []
    irregular_assembly = {}
    _check = False
    for i in parts_concentration_size:
        try:
            n = part_count[i[0]]
        except:
            irregular_parts.append(i[0])
    for i in part_count:
        check_ = False
        for j in parts_concentration_size:
            if i == j[0]:
                check_ = True
```

```
        if check_ == False:
            irregular_assembly[i] = []
            for k in assemblies:
                irregular_assembly[i].append(k[0])

    if len(irregular_parts) != 0:
        print(
            part_file_name, " has parts that are not found in ", assembly_file_name, ":"
        )
        for i in irregular_parts:
            print("\t", i)
        _check = True

    if len(irregular_assembly) != 0:
        print(
            assembly_file_name,
            " has assemblies with parts that are not found in ",
            part_file_name,
            ":",
        )
        for i in irregular_assembly:
            print("\t", i, " found in assemblies ", irregular_assembly[i])
        _check = True

    if _check == True:
        sys.exit(1)


# Creates a dictionary with the well label (eg A1, H11) as a key, and for each key a part or
reagent is allocated.
def _plate_dictionary_creator(reagent_total, part_dictionary):
    alpha = list(string.ascii_uppercase)[:8]
    plate_dictionary = {}
    reagent_list = list(reagent_total.items())
    part_list = list(part_dictionary.items())
    count = 0
    for j in range(12):
        for i in range(8):
            plate_dictionary[alpha[i] + str(j + 1)] = ""
    for i in plate_dictionary:
        if count < len(reagent_list):
            plate_dictionary[i] = reagent_list[count]
            count += 1
    count = 0
    for i in plate_dictionary:
        if plate_dictionary[i] == "" and count < len(part_list):
            plate_dictionary[i] = part_list[count][0], part_list[count][1][1]
            count += 1
    return plate_dictionary
```

```
# Variables declared to collect the .csv data and their related calculations
assemblies = []
uncompressed_parts = []
parts_concentration_size = []


# Input .csv files
assembly_path = input("Input Full Pathway of Assembly (.csv):\n")
part_path = input("Input Full Pathway of Parts (.csv):\n")
assembly_file_name = assembly_path.split("/")[-1]
part_file_name = part_path.split("/")[-1]


# Opens assembly file and extracts its information
try:
    with open(assembly_path, newline="") as csvfile:
        assembly_row = csv.reader(csvfile, delimiter=" ", quotechar="|")
        for row in assembly_row:
            list_row = [ele for ele in ((", ".join(row)).split(",")) if ele.strip()]
            assemblies.append(list_row)
             # A single part can be used in multiple assemblies. The
             number of times these parts are used is counted and stored.
            for j in list_row:
                if list_row.index(j) > 1:
                    uncompressed_parts.append(j)
        part_count = _countinstances(uncompressed_parts)
    # The number of assemblies is checked to be less than 96, as the
      plate is limited to 96 wells
    if len(assemblies) > 96:
        print(
            "Too many assemblies. Max number of assemblies = 96, ",
            assembly_file_name,
            " assemblies = " + str(len(assemblies)),
        )
        sys.exit(1)
except:
    print(assembly_file_name, " error.")
    sys.exit(1)


# Opens part file and extracts its information
try:
    with open(part_path, newline="") as csvfile:
        part_row = csv.reader(csvfile, delimiter=" ", quotechar="|")
        for row in part_row:
            list_row = [ele for ele in ((", ".join(row)).split(",")) if ele.strip()]
            parts_concentration_size.append(list_row)
except:
```

```
    print(part_file_name, " error.")
    sys.exit(1)

# Checks that the parts within the assembly file are present in the part file and vice versa
_check_part_congruency(
    parts_concentration_size, part_count, part_file_name, assembly_file_name
)

# Makes part dictionary, with the part name as the key, the key's value contains the single use volume (30fmol) and the total volume throughout all assemblies * 1.2.
# Additionally makes a dictionary containing the parts that have a total volume over 200 ul. The part name is the key and the value includes the number of wells that part will occupy, the single use volume (30fmol) and the total volume throughout all assemblies *1.2 (eg 'PartX':[3 wells, 30ul,430ul])
part_dictionary = {}
many_wells_parts = {}
for i in part_count:
    for j in parts_concentration_size:
        if i == j[0]:
            single_volume = _calcvolume(float(j[1]), float(j[2]))
            total_volume = round(part_count[i] * single_volume * 1.2, 3)
            plate, wellvolume, count = _volumecheck(j[0], total_volume)
            if len(plate) > 1:
                many_wells_parts[j[0]] = [
                    len(plate),
                    round(single_volume, 3),
                    sum(wellvolume),
                ]
            for q in range(len(plate)):
                part_dictionary[plate[q]] = [
                    round(single_volume, 3),
                    round(wellvolume[q], 3),
                ]


# Making assembly dictionary, the assembly name is the key, the key's value contains the size of the assembly (bp), assembly parts, and volumes of the reagents unique to the assembly. (eg 'Assembly 1' : [40000bp, [part1, part4, part5, part9], 2, 0.5, (bsai volume), (water volume)])
assembly_dictionary = {}
for i in assemblies:
    part_volume_sum = 0
    for j in i[2:]:
        count = 0
        for q in part_dictionary:
            if j == q.split(".", 1)[0]:
                count += 1
                volume = part_dictionary[q][0]
        if count == 1:
            part_volume_sum += round(volume, 3)
```

```
        else:
            part_volume_sum += round(volume, 3)
    part_volume_sum = round(part_volume_sum, 3)
    bsai, water = _calcreagents(int(i[1]), part_volume_sum)
    assembly_dictionary[i[0]] = [i[1], i[2:], 2, 0.5, bsai, water]

# Making reagent dictionary containing, where the key is the reagent's name and its value is the
total volume needed for all the assemblies *1.2
reagents = ["ligase_buffer", "ligase", "bsai", "water"]
reagent_total = dict.fromkeys(reagents, 0.0)
reagent_dictionary = {}
many_wells_reagents = {}
for i in assembly_dictionary:
    reagent_total["ligase_buffer"] += assembly_dictionary[i][2]
    reagent_total["ligase"] += assembly_dictionary[i][3]
    reagent_total["bsai"] += assembly_dictionary[i][4]
    reagent_total["water"] += assembly_dictionary[i][5]
for i in reagent_total:
    reagent_total[i] = round(reagent_total[i] * 1.2, 3)
    plate, wellvolume, count = _volumecheck(i, reagent_total[i])
    for q in range(len(plate)):
        reagent_dictionary[plate[q]] = round(wellvolume[q], 3)

# Checks that the number of wells needed for the parts and the reagents is less than 96 as the is
only 96 wells within the plate
if (len(part_dictionary) + len(reagent_total) > 96) == True:
    print(
        f"The sum of the parts and reagents wells needed
len(part_dictionary)+len(reagent_total)}is greater than 96. The parts and reagents will not fit in
the 96-well plate. Reduce the number of assemblies."
    )
    sys.exit(1)
else:
    # If the number of wells needed is 96 or less, then a dictionary
      is created that allocates a well to the reagents and parts.
    plate_dictionary = _plate_dictionary_creator(reagent_dictionary, part_dictionary)

# Makes a storage file in the user's computer to store the .csv files generated from this program
os.mkdir('metclo_plan_files')

# Declares variables necessary for the creation of the .csv files
header = [
    [
        "assembly name",
        "assembly size",
        "parts",
        "ligase buffer",
        "DNA ligase",
        "bsai",
```

```
        "water",
    ],
    ["part name", "volume with 30fmol", "sum*1.2"],
    ["reagent", "sum*1.2"],
    ["position", "solution", "well volume"],
]

doc = [
    "metclo_plan_files/assembly_data.csv",
    "metclo_plan_files/part_data.csv",
    "metclo_plan_files/reagents_data.csv",
    "metclo_plan_files/position_data.csv",
]
data = (
    assembly_dictionary,
    part_dictionary,
    reagent_dictionary,
    plate_dictionary
)

# Makes the .csv files what will serve as inputs for ot2metclo.py
for i in range(len(header)):
    _makecvs(doc[i], header[i], data[i])


# CREATES THE PDF WITH THE SPECIFICATIONS OF THE USER'S ASSEMBLY.
# Gives every page a title and a page number
class PDF(FPDF):
    def header(self):
        self.set_font('helvetica', 'B', 20)
        self.cell(
              0, 10,
              'Automated MetClo Assembly Plan',
              border = False,
              new_x = XPos.LMARGIN,
              new_y = YPos.NEXT, align='C'
        )
        self.ln(10)
    def footer(self):
        self.set_y(-15)
        self.set_font('helvetica', 'I', 10)
        self.cell(0, 10, f'Page {self.page_no()}/{{nb}}', align ='C')

# Creates the tiles of the different sections within the PDF
def __PDFtitle__(title):
    pdf.set_auto_page_break(auto = True, margin = 15)
    pdf.add_page()
    pdf.set_font('helvetica', 'BU', 16)
    pdf.cell(0, 10, title, border = False, new_x = XPos.LMARGIN, new_y= YPos.NEXT, align='C' )
```

```
    pdf.set_font('helvetica', 'B', 12)

# Creates the subtitles tiles within the PDF
def __PDFsubtitle__ (title):
    pdf.add_page()
    pdf.set_font('helvetica', 'B', 12)
    pdf.cell(0, 10, title, border = False, new_x = XPos.LMARGIN, new_y= YPos.NEXT )

# Creates the PDF section that summarizes the information for each assembly
def __PDFassembly__(i):
    w4 = (pdf.w) / 4.4
    pdf.set_font('helvetica', 'B', 14)
    pdf.cell(w4, 8, f'Assembly Name: {i}', border = 0, new_x = XPos.LMARGIN, new_y = YPos.NEXT)
    pdf.set_font('helvetica', '', 12)
    pdf.cell(
        w4,
        6,
        f'Assembly Size: {str("{:,}".format(int(assembly_dictionary[i][0])))}',
        border = 0,
        new_x = XPos.RIGHT
    )

      # Recommends the cloning technology based on the size of the final
      assembly
    if int(assembly_dictionary[i][0]) > 10000:
        protocol = 'Electroporation'
    else:
        protocol = 'Heat Shock'
    pdf.cell(w4, 6 , f'(Recomend: {protocol})', border = 0, new_x = XPos.LMARGIN, new_y =
YPos.NEXT)

    pdf.set_font('helvetica', 'B', 12)
    pdf.cell(
        0,
        6,
        f'{len(assembly_dictionary[i][1])} Parts',
        border = False,
        new_x = XPos.LMARGIN,
        new_y = YPos.NEXT
    )

    pdf.set_font('helvetica', '', 12)
    count = 0

    for j in assembly_dictionary[i][1]:
        if count <3:
            pdf.cell(w4, 6, j, border = True, new_x = XPos.RIGHT)
            count +=1
        else:
```

```
            pdf.cell(w4, 6, j, border = True, new_x = XPos.LMARGIN, new_y = YPos.NEXT)
            count = 0

    if len(assembly_dictionary[i][1]) != 4:
        pdf.cell(0, 5, '', border = False, new_x = XPos.LMARGIN, new_y = YPos.NEXT)

    pdf.set_font('helvetica', 'B', 12)
    pdf.cell(0 ,8 , 'Reagents (ul)', border = False, new_x = XPos.LMARGIN, new_y = YPos.NEXT)
    pdf.set_font('helvetica', '', 12)
    for x in reagent_total:
        pdf.cell(w4, 6, x, border = True, new_x = XPos.RIGHT)
    pdf.cell(0, 6, '', border = False, new_x = XPos.LMARGIN, new_y = YPos.NEXT)

    for j in assembly_dictionary[i][-4:]:
        pdf.cell(w4, 6, str(j), border = True, new_x = XPos.RIGHT)

    pdf.cell(0, 12, '', border = False, new_x = XPos.LMARGIN, new_y= YPos.NEXT)

# Creates the PDF section that summarizes the information for each part
def __PDFparts__(part_dictionary, parts_concentration_size, part_count):
    w5 = (pdf.w) / 6.6
    pdf.cell(w5+10, 10, 'Part Name', border = 1, new_x = XPos.RIGHT)
    pdf.cell(w5-10, 10, 'Size', border = 1, new_x = XPos.RIGHT)
    pdf.cell(w5-10, 10, 'Conc.', border = 1, new_x = XPos.RIGHT)
    pdf.cell(w5, 10, 'Times Used', border = 1, new_x = XPos.RIGHT)
    pdf.cell(w5+5, 10, '30fmol (ul)', border = 1, new_x = XPos.RIGHT)
    pdf.cell(w5+5, 10, 'Total Volume', border = 1, new_x = XPos.LMARGIN, new_y = YPos.NEXT)
    count = 0
    pdf.set_font('helvetica', '', 12)
    for i in part_count:
        pdf.cell(w5+10, 10, i, border = 1, new_x = XPos.RIGHT)
        for j in parts_concentration_size:
            if i == j[0]:
                pdf.cell(w5-10, 10, j[2], border = 1, new_x = XPos.RIGHT)
                pdf.cell(w5-10, 10, j[1], border = 1, new_x = XPos.RIGHT)
                pdf.cell(w5, 10, str(part_count[i]), border = 1, new_x = XPos.RIGHT)
                if i in many_wells_parts:
                    pdf.cell(w5+5, 10, str(many_wells_parts[i][1]), border = 1, new_x =
XPos.RIGHT)
                    pdf.cell(
                        w5+5,
                        10,
                        str(many_wells_parts[i][2]),
                        border = 1,
                        new_x = XPos.LMARGIN,
                        new_y = YPos.NEXT
                    )
                else:
                    pdf.cell(w5+5, 10, str(part_dictionary[i][0]), border = 1, new_x = XPos.RIGHT)
```

```
                    pdf.cell(
                        w5+5,
                        10,
                        str(part_dictionary[i][1]),
                        border = 1,
                        new_x = XPos.LMARGIN,
                        new_y = YPos.NEXT
                    )

# Creates the PDF section that summarizes the information for each reagent and part
def __PDFreagents__(reagent_total):
    w4 = (pdf.w) /4 .4
    for i in reagent_total:
        pdf.cell(w4, 8, i, border = True, new_x = XPos.RIGHT)
    pdf.ln(8)
    for i in reagent_total:
        pdf.cell(w4, 8, str(reagent_total[i]), border = True, new_x = XPos.RIGHT)

# Creates the PDF section that depicts the position of the reagents
def __PDFreagent_partplate__(plate_dictionary):
    w4 = (pdf.w) / 4.4
    count = 0
    pdf.set_font('helvetica', '', 8)
    for i in plate_dictionary:
        if list(i)[0] == 'A' or list(i)[0] == 'B' or list(i)[0] == 'C' or list(i)[0] == 'D':
            if count < 3:
                pdf.cell(
                    w4,
                    6,
                    i + '  '+ str(plate_dictionary[i]),
                    border = True,
                    new_x = XPos.RIGHT
                )
                count +=1
            else:
                pdf.cell(
                    w4,
                    6,
                    i + '  '+str(plate_dictionary[i]),
                    border = True,
                    new_x = XPos.LMARGIN,
                    new_y = YPos.NEXT
                )
                count =0
    pdf.cell(0, 10, '', border = False, new_x = XPos.LMARGIN, new_y = YPos.NEXT )
    for i in plate_dictionary:
        if list(i)[0] == 'E' or list(i)[0] == 'F' or list(i)[0] =='G' or list(i)[0] == 'H':
            if count < 3:
                pdf.cell(
```

```
                    w4,
                    6,
                    i + '  ' + str(plate_dictionary[i]),
                    border = True,
                    new_x = XPos.RIGHT
                )
                count +=1
            else:
                pdf.cell(
                    w4,
                    6,
                    i + '  ' + str(plate_dictionary[i]),
                    border = True,
                    new_x = XPos.LMARGIN,
                    new_y = YPos.NEXT )
                count =0

# Creates the PDF section that depicts the position of the assemblies
def __PDFtmcplate__(assembly_dictionary):
    plate_dictionary = {}
    alpha = list(string.ascii_uppercase)[:8]
    keysList = list(assembly_dictionary.keys())
    count = 0
    for j in range(12):
        for i in range(8):
            plate_dictionary[alpha[i] + str(j + 1)] = ''
    for i in plate_dictionary:
        if count < len(keysList):
            plate_dictionary[i] = keysList[count]
            count += 1
    count = 0
    w4 = (pdf.w) / 4.4
    pdf.set_font('helvetica', '', 8)
    for i in plate_dictionary:
        if list(i)[0] == 'A' or list(i)[0] == 'B' or list(i)[0] =='C' or list(i)[0] == 'D':
            if count <3:
                pdf.cell(w4, 6, i + '  ' + plate_dictionary[i], border = True, new_x = XPos.RIGHT)
                count += 1
            else:
                pdf.cell(
                    w4,
                    6,
                    i + '  ' + plate_dictionary[i],
                    border = True,
                    new_x = XPos.LMARGIN,
                    new_y = YPos.NEXT
                )
                count =0
    pdf.cell(0, 10, '', border = False, new_x = XPos.LMARGIN, new_y = YPos.NEXT)
```

```
    for i in plate_dictionary:
        if list(i)[0] == 'E' or list(i)[0] == 'F' or list(i)[0] =='G' or list(i)[0] == 'H':
            if count < 3:
                pdf.cell(w4, 6, i + '   ' + plate_dictionary[i], border = True, new_x = XPos.RIGHT)
                count += 1
            else:
                pdf.cell(
                    w4,
                    6,
                    i + '   ' + plate_dictionary[i],
                    border = True,
                    new_x = XPos.LMARGIN,
                    new_y = YPos.NEXT
                )
                count = 0

# Starts creating the PDF document by using the assembly information generated by the above code and by calling PDF methods created above.
pdf = PDF('P','mm','Letter')
__PDFtitle__(f'OT2 Set-Up Instructions')
pdf.set_font('helvetica','', 10)

with open('doc/ins.txt', 'r') as f:
    for i in f:
        pdf.multi_cell(0,3,i)

__PDFsubtitle__('OT2 Layout')
pdf.image('doc/OT2bench.JPG',45,50,150)
__PDFsubtitle__('Reagent Plate Layout (ul)')
__PDFreagent_partplate__(plate_dictionary)
__PDFsubtitle__('Thermocycler Plate with Assemblies')
__PDFtmcplate__(assembly_dictionary)
__PDFtitle__(f'{str(len(assembly_dictionary))} Assemblies')

for i in assembly_dictionary:
    __PDFassembly__(i)

__PDFtitle__(f'{str(len(part_dictionary))} Parts')
__PDFparts__(part_dictionary, parts_concentration_size,part_count)
__PDFtitle__(f'Total Reagents Volumes Required (ul) *1.2')
__PDFreagents__(reagent_total)

try:
    pdf.output('metclo_plan_files/metclo_plan.pdf')
    print('metclo_plan.pdf written successfully.')
except:
    print('metclo_plan.pdf not written')
    sys.exit(1)
```

Script 2 – ot2metclo.py python protocol script. *Purple areas contain commentary of the code.*

```
# Imports other software packages that will be needed in the code
import profile, string, csv, sys, math
from opentrons import protocol_api

# Open the .csv files produced my metclo_plan.py. Produces an error if the files are not
successfully opened.
def __openfile__(file):
    try:
        with open(file, newline="") as csvfile:
            rows = csv.reader(csvfile)
            header = next(rows)
            if header != None:
                data = []
                for j in rows:
                    data.append(j)
                return data
    except:
        print("File error.", file)
        sys.exit(1)

# The maximum volume within well on the 96 well-plate is 200ul.
# The method checks if the volume needed is over 200ul, if so, the volume is divided into
different wells.
def __volumecheck__(i, x, count, plate):
    if x / 200 < 1:
        plate[count] = [i[0], float(i[1])]
        count += 1
    else:
        wells = math.ceil(x / 200)
        for t in range(wells):
            plate[count] = [i[0] + "." + str(t + 1), float(i[1])]
            count += 1
    return plate, count

# Creates a dictionary with keys representing the well positions within a 96 well plate (A1 –
H12). The keys have empty values.
def __makeplate__():
    plate_dictionary = {}
    alpha = list(string.ascii_uppercase)[:8]
    for j in range(12):
        for i in range(8):
            plate_dictionary[alpha[i] + str(j + 1)] = ""
    return plate_dictionary


# Variables are declared containing the information from the .csv files
assembly_data = __openfile__("metclo_plan_files/assembly_data.csv")
part_data = __openfile__("metclo_plan_files/part_data.csv")
```

```
reagent_data = __openfile__("metclo_plan_files/reagents_data.csv")
position_data = __openfile__("metclo_plan_files/position_data.csv")
# Dictionaries are created to accommodate the information form the .csvs for easy manipulation
reagent_part_plate2 = {}
assembly_plate = {}
count = 0
for i in position_data:
    if len(i) == 3:
        reagent_part_plate2[i[1]] = [i[0], float(i[2])]
    try:
        assembly_plate[i[0]] = [assembly_data[count][0], 0]
        count += 1
    except:
        assembly_plate[i[0]] = ["", 0]

assembly_dictionary = {}
for i in assembly_data:
    parts_ = []
    for t in (i[2][1:-1]).split(","):
        t = (t.strip(" "))[1:-1]
        parts_.append(t)
    reagents_ = []
    for j in i[3:]:
        reagents_.append(float(j))
    assembly_dictionary[i[0]] = parts_, reagents_

part_dictionary = {}
for i in part_data:
    part_dictionary[i[0]] = float(i[1])

reagents = ["ligase_buffer", "ligase", "bsai", "water"]


# The metadata provides the Opentrons App necessary information about the protocol
metadata = {
    "apiLevel": "2.3",
    "protocolName": "Metclo Assembly - hardcoded with one assembly that cna change in size",
    "author": "Daniella Matute <daniella.l.matute@gmail.com",
    "description": "OT-2 protocol that allows for methylase DNA assembly",
}


# Instructs the OT-2 robot where to start the automation
def run(protocol: protocol_api.ProtocolContext):

##############################################################################
    # OT-2 APPARATUS ARE DECLRED
    # Modules
    temp_module = protocol.load_module("temperature module", 1)
    mag_module = protocol.load_module("magnetic module gen2", 4)
```

```
    tc_mod = protocol.load_module("thermocycler module")

    # Labware
    tr_20 = protocol.load_labware("opentrons_96_tiprack_20ul", 9)
    part_plate = protocol.load_labware("nest_96_wellplate_200ul_flat", 6)
    tc_plate = tc_mod.load_labware("nest_96_wellplate_100ul_pcr_full_skirt")

    # Instrument
    p_20 = protocol.load_instrument("p20_single_gen2", "left", tip_racks=[tr_20])

    ###############################################################################
    # PROTOCOL TO BE FOLLOWED BY THE ROBOT
    # Set temperature for the thermocycler
    tc_mod.set_lid_temperature(4)
    tc_mod.set_block_temperature(4)
    tc_mod.open_lid()

    # For each assembly, transfer the appropriate volume for all the reagents and
      parts corresponding to the assembly.
    for i in assembly_plate:
        #Checks that the position in the plate has an allotted assembly
        if assembly_plate[i][0] != "":
            assembly_name = assembly_plate[i][0]
            parts = assembly_dictionary[assembly_name][0]
            reagent_dictionary = {}
            for j in range(len(assembly_dictionary[assembly_name][1])):
                reagent_dictionary[reagents[j]] = assembly_dictionary[assembly_name][1][j]
            for j in reagent_dictionary:
                # Checks that a single instance of the reagent needed is in the
                  plate
                if j in reagent_part_plate2:
                    # Checks that there is enough reagent in the well required
                      for the assembly
                    if (
                        reagent_part_plate2[j][1] != 0
                        and reagent_part_plate2[j][1] > reagent_dictionary[j]
                    ):
                        p_20.transfer(
                            reagent_dictionary[j],
                            part_plate[reagent_part_plate2[j][0]],
                            tc_plate[i],
                        )
                        # Decreases the reagent volume found in the well
                        reagent_part_plate2[j][1] = round(
                            reagent_part_plate2[j][1] - reagent_dictionary[j], 3
                        )

                else:
                    # If more than one instance of the reagent is found in the
```

```
                plate (as in the case the total volume of the reagent is >200
                and needs to be divided into multiple wells). The first well
                of that reagent is checked to contain enough reagent needed
                for that assembly. If it does not have enough reagent, the
                second well is checked and so forth, until a well containing
                enough volume if found and the reagent is transferred.
                check = False
                for k in reagent_part_plate2:
                    if check == False:
                        val = reagent_part_plate2[k]
                        k_ = k.split(".")[0]
                        if (
                            k_ == j
                            and val != 0
                            and check == False
                            and reagent_dictionary[k_] != 0
                            and reagent_part_plate2[k][1] > reagent_dictionary[j]
                        ):
                            p_20.transfer(
                                reagent_dictionary[j],
                                part_plate[reagent_part_plate2[k][0]],
                                tc_plate[i],
                            )
                            # Decreases the reagent volume found in the well
                            reagent_part_plate2[k][1] = round(
                                reagent_part_plate2[k][1] - reagent_dictionary[j], 3
                            )
                            check = True

            for j in parts:
                # Checks that a single instance of the reagent needed is in the plate
                if j in part_dictionary:
                    p_20.transfer(
                        part_dictionary[j],
                        part_plate[reagent_part_plate2[j][0]],
                        tc_plate[i],
                    )
                    # Decreases the reagent volume found in the well
                    reagent_part_plate2[j][1] = round(
                        reagent_part_plate2[j][1] - part_dictionary[j], 3
                    )
                else:
                    # If more than one instance of the reagent is found in the plate (as in the
                    case the total volume of the reagent is >200 and needs to be divided into
                    multiple wells). The first well of that reagent is checked to contain enough
                    reagent needed for that assembly. If it does not have enough reagent, the
                    second well is checked and so forth, until a well containing enough volume if
                    found and the reagent is transferred.
                    check = False
```

```
                    for k in part_dictionary:
                        if check == False:
                            k_ = k.split(".")[0]
                            val = part_dictionary[k]
                            total = reagent_part_plate2[k][1]
                            if k_ == j and check == False and val != 0 and total > val:
                                p_20.transfer(
                                    val,
                                    part_plate[reagent_part_plate2[k][0]],
                                    tc_plate[i],
                                )
                                # Decreases the reagent volume found in the well
                                reagent_part_plate2[k][1] = round(
                                    reagent_part_plate2[k][1] - val, 3
                                )
                                check = True

    # Thermocycler beings cyclic temperature change for assembly creation
    protocol.comment("Assembly reaction ongoing")
    tc_mod.set_lid_temperature(85)
    tc_mod.set_block_temperature(37, hold_time_minutes=15, block_max_volume=20)
    tc_mod.close_lid()
    profile = [
        {"temperature": 37, "hold_time_minutes": 2},
        {"temperature": 16, "hold_time_minutes": 5},
        {"temperature": 37, "hold_time_minutes": 20},
        {"temperature": 80, "hold_time_minutes": 5},
    ]
    tc_mod.execute_profile(steps=profile, repetitions=45, block_max_volume=20)
    tc_mod.set_lid_temperature(4)
    tc_mod.set_block_temperature(4)
    protocol.comment(
        "Metclo assembly done. Assembly is incubating at 4 degrees Celsius."
    )
```