Smithy: A Bioinformatics Web Service for the Design and Comparison of Different DNA Cloning Procedures

Henri-Morgan Thomas

A Thesis

In the Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of

Master of Applied Science (Electrical and Computer Engineering) at

Concordia University

Montreal, Quebec, Canada

August 2022

Concordia University

School of Graduate Studies

This is to certify that the thesis

prepared by: Henri-Morgan Thomas

Entitled: *Smithy*: A Bioinformatics Web Service for the Design and Comparison of Different DNA Cloning Procedures

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Electrical and Computer Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the Examining Committee:

_____ Chair

Wahab Hamou-Lhadj

_____ Examiner

Steve Shih, PhD

_____ Examiner

Malcolm Whiteway, PhD

_____ Examiner

Wahab Hamou-Lhadj, PhD

_____ Supervisor

Nawwaf Kharma, PhD

Approved by _____

Yousef R. Shayan, PhD, PEng

_____ 2022 _____

Mourad Debbabi, PhD

Abstract

Smithy: A Bioinformatics Web Service for the Design and Comparison of Different DNA Cloning Procedures

Henri-Morgan Thomas

The field of Synthetic Biology has seen many recent developments in DNA assembly protocols, software tools in aide of assembly design and experiment planning, standardization of parts, and automation of complex and high-throughput DNA assembly and cloning processes. These software tools reduce human-introduced errors and the need for highly-skilled labor, cut-down the total cost of assembly experiments, and diminish the overall time expended in designing, planning and executing reliable cloning experiments. Shortcomings of these efforts, however, exist in their usability and accessibility, support for major cloning methods, the incorporation of both part repositories and direct-synthesis DNA sequences in the design process, extensibility of the software, and the lack of a comparison of assembly design options offered by different assembly methods. Here, *Smithy* is presented: a web-application automating DNA assembly project design for scar-less assemblies, incorporating existing sequences from major repositories (e.g., AddGene and iGEM) and synthetic DNA sequences into the assembly design process. Currently, the supported cloning methods are Gibson, Golden Gate, PCR-SOE, SLIC and BioBricks. *Smithy* software architecture utilizes a new DNA cloning methods *abstraction hierarchy,* which facilitates effective and extendible application implementation and extendibility. *Smithy*'s graphical user

interface and overall user experience are simpler and more informative than existing DNA assembly design tools. As a service, *Smithy* is open-source, free, and widely accessible for users of all experience levels. The core software features exist as independent modules that are portable to future automation projects. Furthermore, an analytical feature for creating *assembly bundles*, has been developed. Assembly bundles comprise of multiple assembly solutions for the same target construct; this facilitates quick comparison and selection of the most appropriate assembly solution by and for a particular user. To exemplify the usefulness of *Smithy*, two detailed case studies are presented: a single Gibson assembly for a riboswitch-modulated fluorescent genetic circuit, and another bundle assembly for an expanded riboswitch-modulated fluorescent genetic circuit, employing Gibson, Golden Gate and PCR-SOE methods. Finally, *Smithy* is a highly usable, future-oriented tool, which can be improved and expanded in response to future developments in cloning methodologies, standardization, and software tools, in the highly promising field of Synthetic Biology.

Acknowledgments

The completion of this project would not be possible without the contributions, challenge, and support from several sources. First, I would like to acknowledge my advisor Nawwaf Kharma for guiding and challenging me through the graduate research and thesis writing, during my journey in the NSERC Create SynBioApps graduate training program at Concordia University. Next, I would like to thank Jonathan Ouellet at Monmouth University for partnering with my professor and me, serving critically as an advisor for biological aspects of my thesis and performing validation experiments for my work. Lastly, I would like to thank my close friends and family, all around the world. They know who they are, and I present them with my sincerest, highest appreciation for supporting and encouraging me along the way, as I pursued this graduate degree.

Table of Contents

List of Figures

List of Tables

List of Terms and Acronyms

Acronym/Term	Definition
PCR	Polymerase chain reaction
MCS	Multiple cloning site
RE	Restriction endonuclease enzyme
Type 2S RE	Type 2S restriction enzyme
BsaI	Default Golden Gate Type 2S RE
EcoRI	Restriction enzyme used in BioBricks cloning
XbaI	Restriction enzyme used in BioBricks cloning
SpeI	Restriction enzyme used in BioBricks cloning
PstI	Restriction enzyme used in BioBricks cloning
ssDNA	Single-stranded DNA
dsDNA	Double-stranded DNA
nt	Nucleotide
CLI	Command line interface

Chapter 1: Introduction

This chapter serves as an introduction to the research context of Smithy. It begins by detailing the fundamental concepts behind DNA cloning (Sections 1.1-3). Then, the general methods of DNA cloning are outlined (Section 1.4). Finally, the six major cloning methods used in the field and supported by Smithy are presented in order of their publication times (Section 1.5).

1.1 DNA

Deoxyribonucleic Acid (DNA) is a large, polymer that holds heritable information, influencing all aspects of function and development of living organisms. The four molecular compounds that genetic information is written in are adenine (A), thymine (T), cytosine (C), and guanine (G). DNA exists in two major forms: double-stranded DNA (dsDNA) and single-stranded DNA (ssDNA). Double stranded DNA is composed of two complementary polymer strands with a specific directionality. This directionality is based on the molecular structure of the strands' sugar-phosphate backbones: the start of a strand being the *5' end* and the end of a strand the *3' end*. Complementary strands are formed through the bonding of adenine to thymine and cytosine to guanine. dsDNA is the form in which genes and plasmids exist, and single-stranded DNA is the form in which PCR primers exist. In the context of synthetic biology, novel DNA sequences are rationally designed and created by obtaining sequences from organisms or through chemical synthesis. These new genes can then be introduced to organisms for augmentation of existing processes or the addition of completely new features.

1.2 Primers

Primers are short sequences of ssDNA used for targeting specific sequences of DNA or RNA for amplification via a polymerase chain reaction (PCR) described in Section 1.3. For a typical PCR experiment, a dsDNA is targeted by two primers: one called the *forward primer*, the other called the *reverse primer*. These sequences of ssDNA are short, from about 20-50nt long. The forward primer will partially or wholly bind to the beginning of the target sequence and the reverse primer will bind to its end.

1.3 PCR

The polymerase chain reaction (PCR) is an effective and reliable process to amplify a target sequence of DNA exponentially for a diverse range of applications. The DNA is quickly amplified through thermocycling using a DNA polymerase enzyme. Thermocycling will cause the target DNA complementary strands to unbind and allow for the primers to bind as temperature cools. Then, the polymerase will extend the bound forward and reverse primer sequences to complete the target dsDNA sequence. This process is repeated until the desired target dsDNA concentration is achieved[1,2]. PCR is crucial to the field of synthetic biology as it is a fundamental procedure for the early stages of most DNA cloning projects.

1.4 Cloning

Molecular DNA cloning has been an existing biological experimental technique for several decades, since the 70s and 80s. There are several differing methods for performing cloning experiments, and they each use a different set of enzymes to extend, cut, and ligate DNA to facilitate *in vitro* creation of rationally designed genetic constructs that are later introduced to host cells.

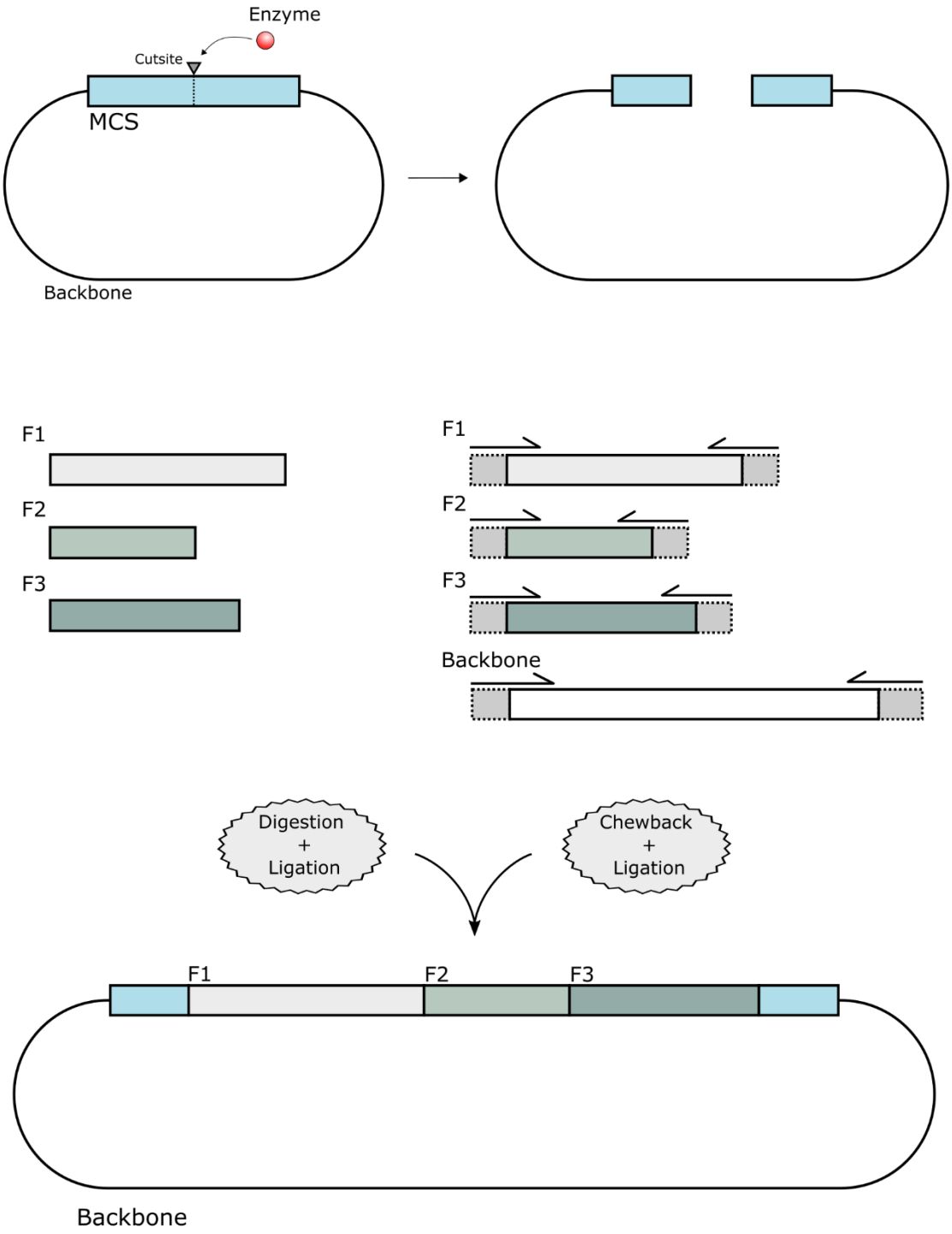


Figure 1: General cloning procedure. A plasmid backbone is shown with an MCS (blue) that is linearized by an enzyme. Three fragments (F1, F2, and F3) to be inserted into this backbone are

shown with assembly extension sequences added. The three fragments are then shown inserted into the backbone through, in general terms, a digestion-ligation or chewback-ligation procedure.

These methods are used to take sequences of DNA from external sources or organisms and assemble them together *in vitro* into a coherent genetic construct, often referred to as an "insert". DNA inserts created in this fashion are called recombinant DNA. With a genetic construct created, the experimenter will insert this new sequence into another large sequence of DNA that serves as a chassis for expression in the host cell, and this chassis is referred to as a "plasmid backbone"[3,4]. Backbone plasmids are a way to contain the DNA in a structure that allows reliable and repeated introduction into test cells/organisms, and these are selected based on the desired host organism and specifications of the experimental research study. Two crucial, required features of backbones are a replication of origin and selectable markers. After assembly of the insert and its incorporation into a backbone, a completed plasmid is achieved that is ready for expression. This complete construct is then introduced into a selected host organism that will express the assembled sequence. The introduction of a genetic construct into a host organism is called *transformation* or *transfection*.

There exist several sources of DNA for cloning endeavors. Organisms, synthetic DNA manufacturers, plasmid repositories selling scientifically published sequences, and a lab's local "database" (a large collection of freezers) of previously used and catalogued sequences from past experiments can all be used. When setting out to design a cloning experiment the researcher must acquire all sequences necessary for their genetic construct while also optimizing and minimizing the cost (and risk) of securing the sequences and assembling them together. The different sources of DNA sequences have different costs. For example, using DNA from a home lab is cheaper than

using sequences from a plasmid repository such as AddGene, and using DNA from a plasmid repository is cheaper and faster than purchasing completely synthesized DNA sequences.

DNA cloning spans an effective and important suite of experimental methods and tools for contemporary synthetic biology applications, including but not limited to: Gibson Assembly[5], Golden Gate Assembly[6], MoClo[7], SLIC[8,9], BioBricks[10,11], and PCR-SOE[12–14]. The applications of DNA cloning involve novel, rational designs, and modifications to the genomes of the host organisms. Additionally, these methods are independent of the genetic constructs' function and use, thus applicable to a broad range of research projects. The array of cloning experimental techniques began with what's called "Traditional Cloning" which uses restriction enzymes to "cut-and-paste" a single insert sequence into a single plasmid backbone. However, the methodologies have evolved and improved since to accomplish large, complex, and scalable assemblies that have ultimately led to the development of high throughput genome foundries[15,16]. Furthermore, not only have these methods evolved for scale, but they have also evolved for increased and improved standardization of parts, plasmid entry vectors, plasmid backbones, and experimental methodologies[4–7,17–22].

1.5 Contemporary Cloning Methods

1.5.1 Restriction Endonuclease Cloning – 70s/80s

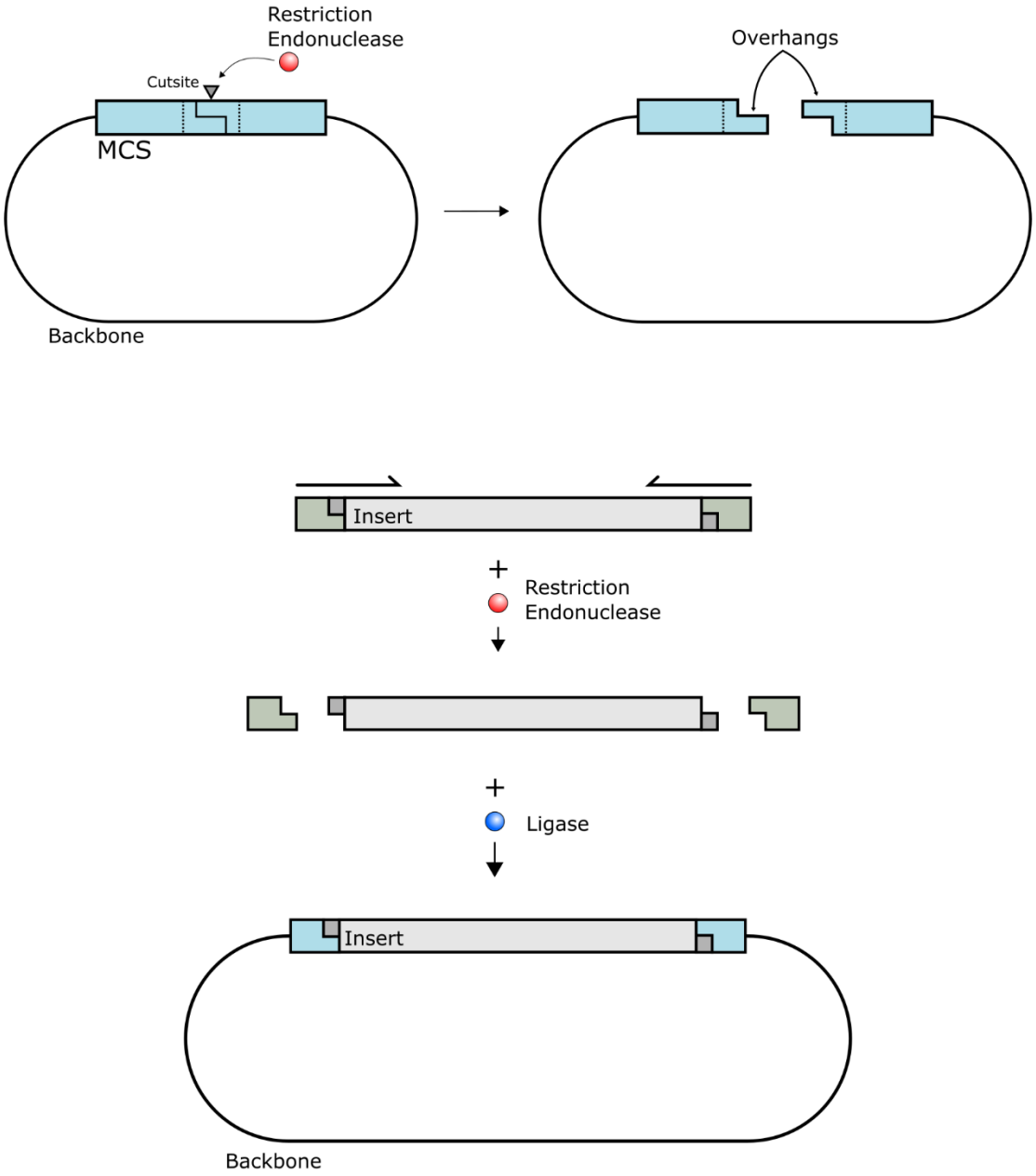


Figure 2: Restriction endonuclease cloning

Restriction endonuclease cloning involves the combination of a single insertion DNA sequence with a plasmid backbone. The insert sequence is digested with two unique restriction

endonucleases (RE), one for each end of the DNA molecule. Using different restriction endonucleases is necessary as it ensures the exposed overhang sites, resulting from the digestions, will be different from one another. In the same fashion as the insert fragment, the plasmid backbone is digested using the same two restriction endonucleases at its Multiple Cloning Site (MCS). An MCS is a specialized part of a plasmid that contains a large set of cut-sites, for a usually popular set of REs.

Once the insert and plasmid backbone are digested, the plasmid backbone is linearized which allows the insert sequence to be added. The insert is added through complementary binding of the cut-site overhangs facilitated by a ligase. With successful ligation, the insert is contained in the backbone and the newly constructed plasmid can be transformed/transfected into a host cell for expression[23].

It is useful to understand traditional cloning before covering newer methods because these contemporary methods all build and iterate upon the same fundamental ideas and methodologies present in this strategy. The core methodologies, specifically, are the use of insert sequences, a plasmid backbone to contain the genetic construct, and various enzymes that facilitate the cloning assembly process. Newer methods have become dominant because they not only improve on the strategies of traditional cloning but also scale up the capacity of experiments to large, multi-sequence assemblies of complex genetic constructs.

1.5.2 Polymerase Chain Reaction Splicing by Overlap Extension – 90s

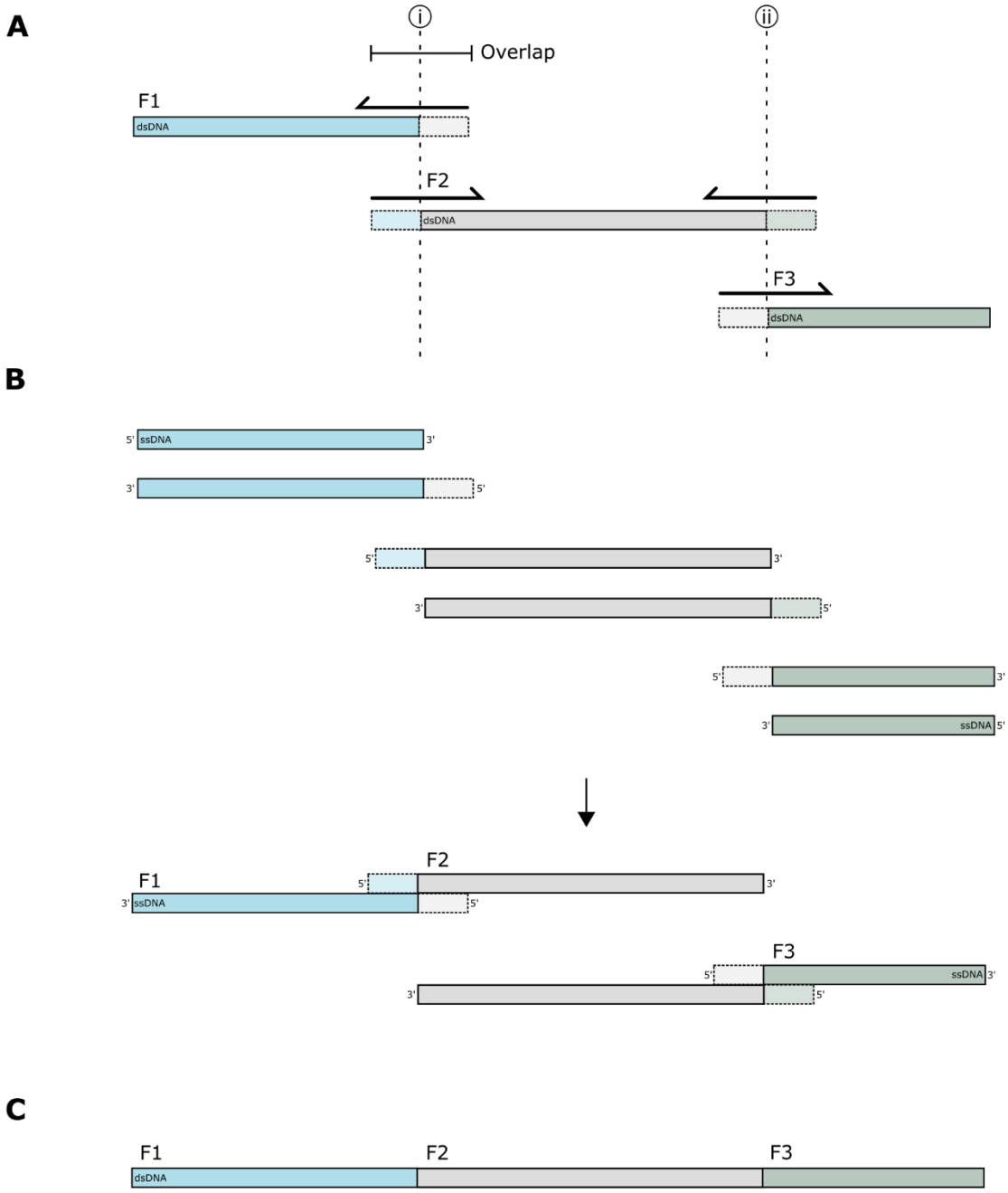


Figure 3: PCR cloning for three fragments: F1, F2, and F3. (A) Each fragment is extended using PCR to add overlaps at junctions for a total overlap length (i and ii). (B) The extended fragments are denatured and then annealed together based on the overlap sequences at the junctions. (C) After cycles of PCR the full construct is achieved.

After the development of traditional cloning, a competing method for the creation of recombinant DNA sequences was created that takes advantage of PCR. This new method is commonly known as PCR-SOE[12]. With this new method, two insert fragments are combined during PCR using primers that have extension sequences adding homology (shared nucleotide sequences) between the two insert fragments. During the PCR process, the primers first add homology to each insert fragment through their extension sequences. After the homologies are added, the two sequences then anneal together and act as large primers for each other, leading to a final and complete sequence. Initially PCR-SOE was only able to combine two fragments at once, however over time the methodology has been improved to allow up to four (4) large, several thousand base long sequences to be recombined[24].

1.5.3 Sequence and Ligation Independent Cloning – 2007

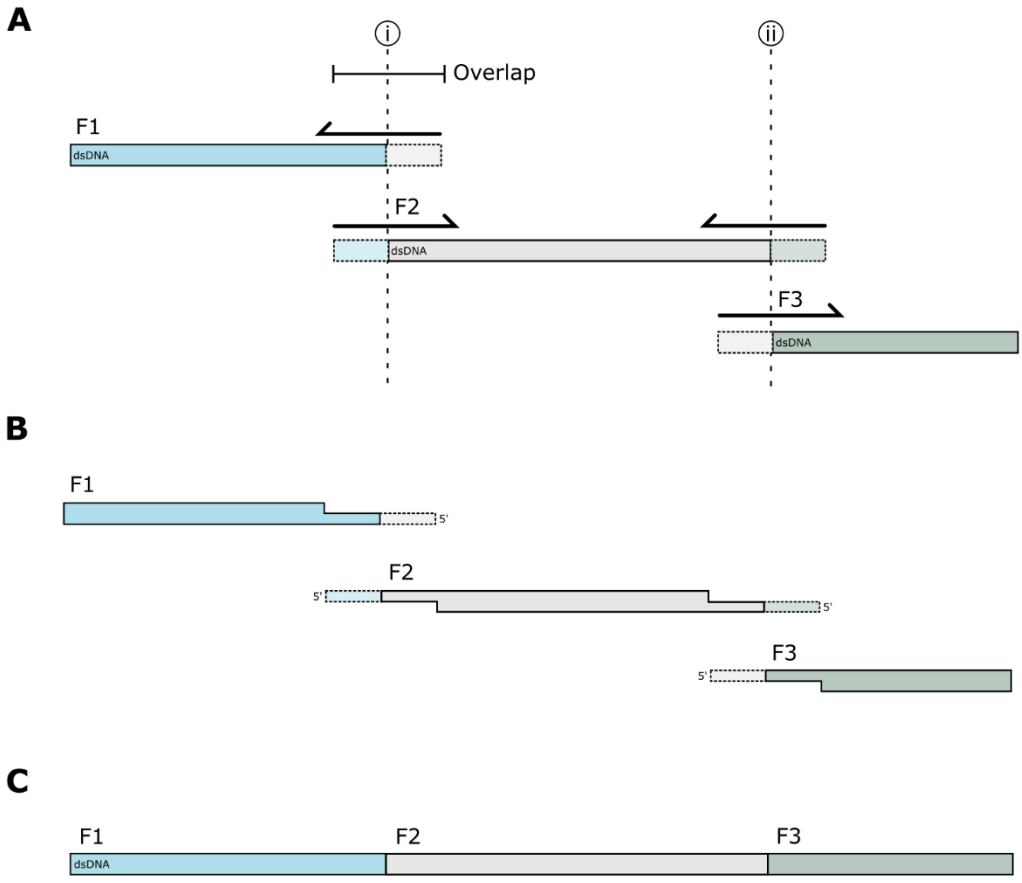


Figure 4: SLIC assembly for three fragments: F1, F2, and F3. (A) Each fragment is extended using PCR to add overlaps at junctions for a total overlap length (i and ii). (B) Fragments undergo exonuclease chewback to reveal 5' single stranded sequence overlaps that can be ligated together. (C) After ligation and repair of the chewback reaction the final construct is achieved.

A newer cloning method created in the late 2000s is called sequence and ligation-independent cloning, known as SLIC[8,9]. This method is unique compared to traditional and PCR-SOE cloning. It allows for multiple (5-10) insert fragments to be recombined *in vitro* using exonuclease reactions and incubation-initiated recombination. The recombination of insert fragments is produced through the exonuclease chewing back the ends of each fragment to expose homologies with left

and right neighboring fragments for every fragment in the assembly. Similar to PCR-SOE, the homology sequences are added during a set of PCR extension reactions and are roughly 30nt in length. The original study states that this method is a way that the researchers were able to mimic *in vivo* homologous recombination *in vitro* for complex genetic assemblies[8,9].

1.5.4 Golden Gate Assembly – 2008

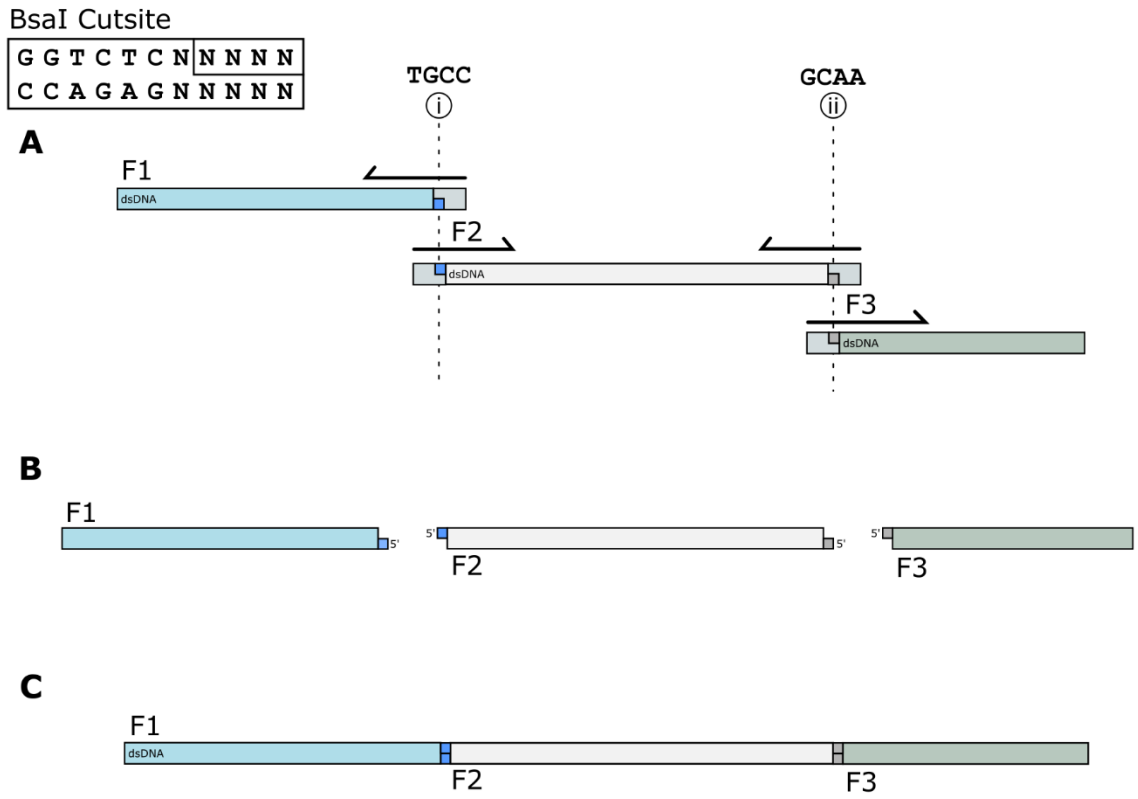


Figure 5: Golden Gate assembly for three fragments: F1, F2, and F3. (A) Each fragment is extended using PCR to add the BsaI cut-site and selected 4 nt overhangs at junctions (i and ii). (B) The three fragments are digested using BsaI to expose the 4 nt overhangs. (C) A ligase is used to recombine the fragments at the correct junctions at the overhang sequences.

Golden Gate cloning is a breakthrough method published in 2008[6], one that has seen widespread use because of its efficiency and precision. This cloning method utilizes Type 2S restriction

enzymes and ligases to facilitate the recombination of one or multiple DNA insert sequences and their insertion into a backbone plasmid. The restriction enzyme, typically BsaI, will digest the fragments to expose sequence overhangs, and the ligase will repair the nicks, joining the sequences into the desired construct.

Type 2S restriction enzymes are advantageous in Golden Gate cloning because they (as with most restriction enzymes) cut DNA to produce overhangs, which can be exploited for sequence recombination. However, these enzymes cut outside of their recognition sequence to expose a 4 nt 5' or 3' overhangs, depending on the orientation of the cut-site. In addition, the 4 nt overhang sequence is independent of the enzyme's recognition sequence and is thus customizable. These customizable overhangs are used to design assemblies where one or several insert sequences can be assembled together, and then into the backbone plasmid, in an effective and predictable way. Additionally, Golden Gate cloning is performed in a one-pot reaction, compared to earlier methods requiring multiple reaction volumes. Therefore, large assemblies of several fragments can be performed in the span of 1-6 hours, depending on the insert count[6].

1.5.5 Gibson Assembly – 2009

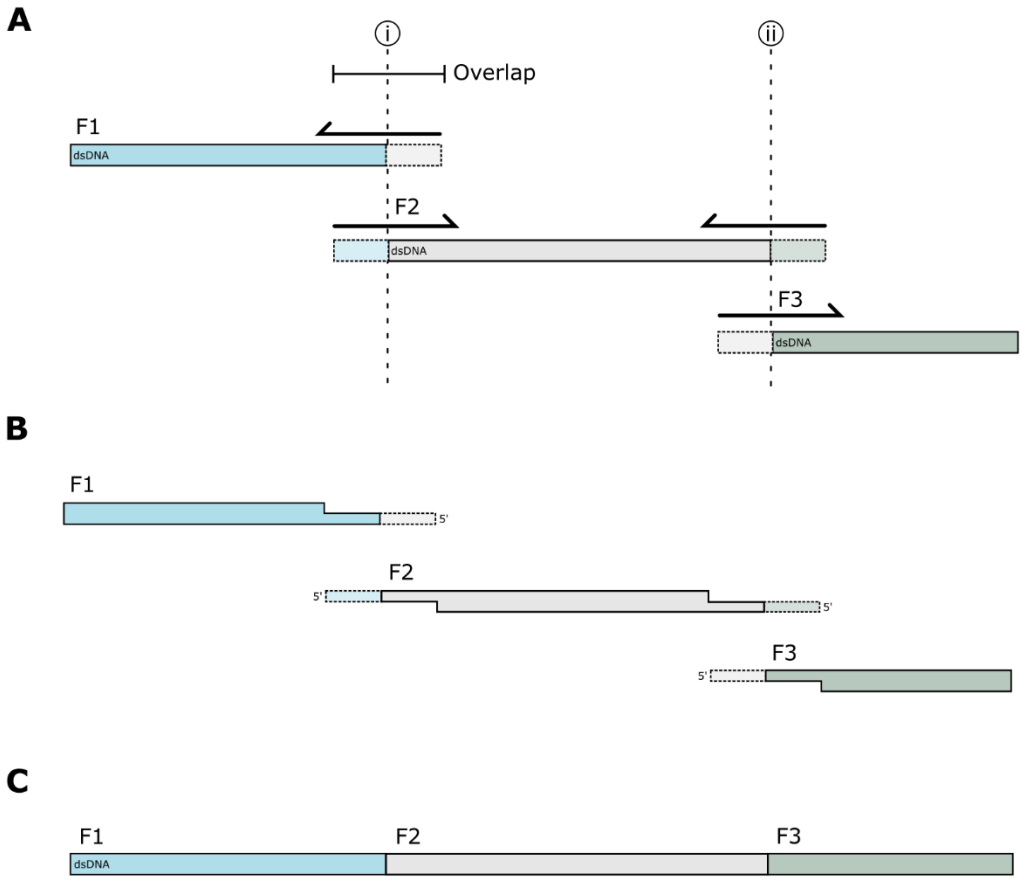


Figure 6: Gibson assembly for three fragments: F1, F2, and F3. (A) Each fragment is extended using PCR to add overlaps at junctions for a total overlap length (i and ii).

Gibson cloning is similar to SLIC but uses different enzymes for assembling several fragments. Like Golden Gate cloning, it is also a one-pot assembly reaction method. Fragments are recombined *in vitro* by using a T5 exonuclease to chew-back the sequences, exposing 3' single-stranded DNA that is homologous with neighboring fragments. As with PCR-SOE and SLIC, homology sequences are added during a set of PCR extension reactions for each insert fragment. The fragments then anneal together according to their shared homologies, finishing with a filling and repair of gaps and nicks in the recombined sequences by *Phusion* polymerase and *Taq* ligase,

respectively. The entire procedure for Gibson cloning is an isothermal reaction, taking place at 50°C and in a single pot. This renders it another efficient and convenient method for large, complex assemblies of up to several 100kb[5].

1.5.6 BioBricks Assembly – 2000s

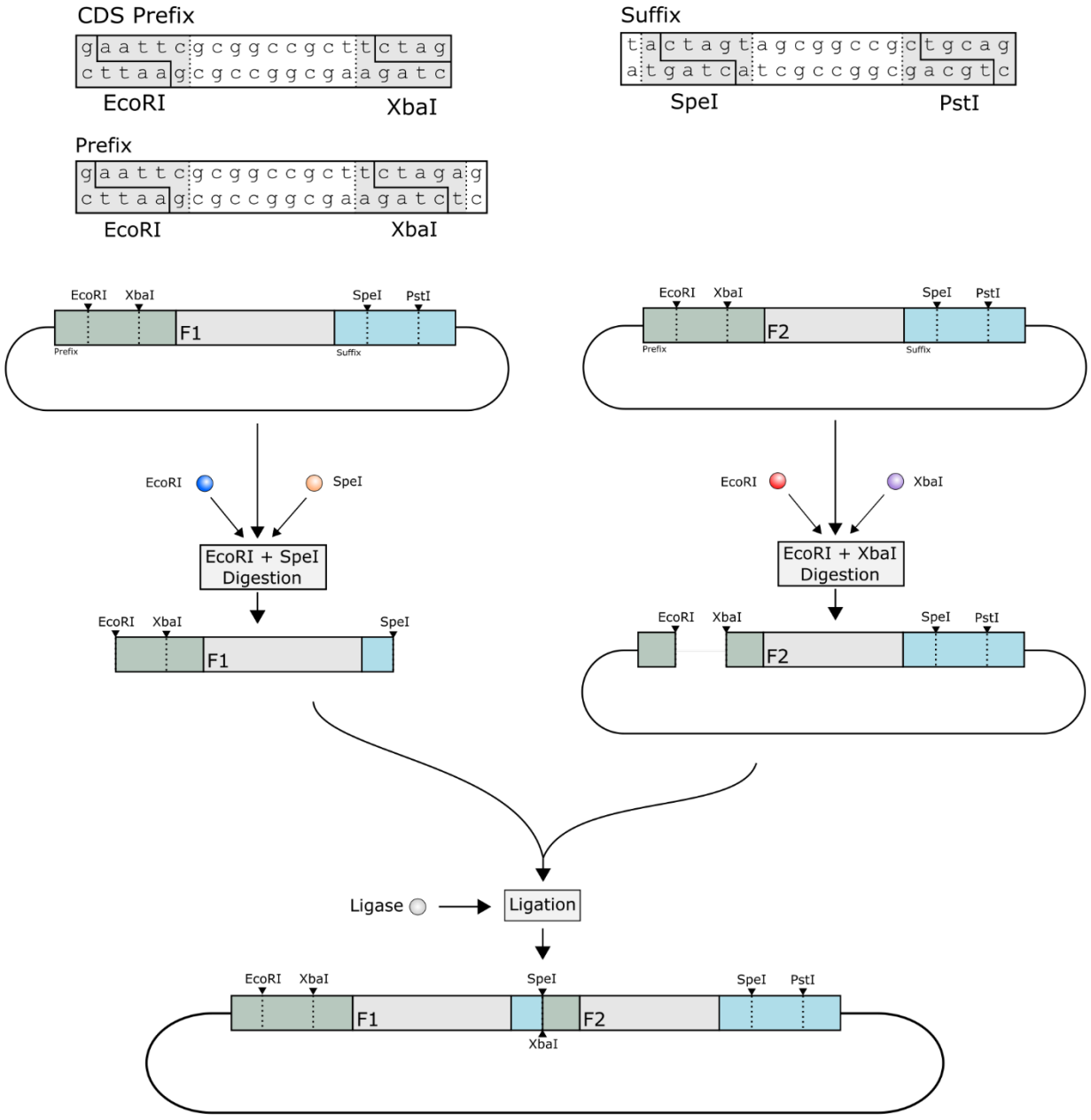


Figure 7: BioBricks assembly for two fragments: F1 and F2

In an attempt to standardize assembly parts and procedures, researchers at Massachusetts Institute of Technology (MIT) created the BioBricks assembly standard for pairwise, iterative assembly of insert sequences[10]. This method uses a standardized pair of flanking sequences for every used part. The creators of this method call these flanking sequences the "prefix" and "suffix" of each part. Insert fragments are flanked by these prefix and suffix sequences in entry vectors. The prefix contains two restriction enzyme cut-sites for EcoRI and XbaI, and the suffix contains one restriction cut-site for each of SpeI and PstI. Following a specific protocol for digestion-ligation reactions (similar to traditional cloning and Golden Gate) these cut-sites allow for pairwise recombination of two insert sequences. This results in the recombined sequence being flanked by the same prefix and suffix sequences. Thus BioBricks allows for iterative, pairwise assembly of several insert fragments into a reusable destination vector.

Chapter 2: Review of Existing DNA Assembly Design Tools

A handful of design tools for cloning projects have been developed over the past few decades. These tools span a broad range of applications, from code libraries that can be imported into software development projects, to full programs seeking to optimize designs and protocols for cloning experiments. As cloning methods became more efficient, robust, and scalable, DNA assembly design automation tools also evolved, for similar reasons. What follows is an overview of the major cloning design automation and aid tools that inspired or were utilized in the development of this project.

2.1 Biopython – 2000

```
>>> from Bio.Seq import Seq
>>> from Bio.Alphabet import generic_dna
>>> gene = Seq("ATGAAAGCAATTTTCGTACTG"
...            "AAAGGTTGGTGGCGCACTTGA",
...            generic_dna)
>>> print gene.transcribe()
AUGAAAGCAAUUUUCGUACUGAAAGGUUGGUGGCGCACUUGA
>>> print gene.translate(table=11)
MKAIFVLKGWWRT*
```

Figure 8: Sample Biopython code execution from publication[25]. This short script shows the use of a Biopython Seq class instance with simple molecular biology functions executed on it: transcription and translation.

Biopython is a set of bioinformatics tools for use in Python, and it has been updated continually since its initial release in 2000[25]. There are several useful features in Biopython to bioinformatics projects, however a select few have been utilized for the purposes of *Smithy*. These features are the local BLAST interface and BLAST query results datastructures. The rest of the capabilities of this library cover sequence records, input/output interfaces for popular bioinformatics file types, sequence alignments, population genetics, and structural bioinformatics. Biopython is a useful tool for using, recording, and managing large amounts of biological data.

With BLAST+[26] installed on the local machine with created sequence databases, Biopython has built-in tools that allow a programmer to call BLAST queries from a Python script using simple syntax. Additionally, there is a query results parser that will translate BLAST results into usable datastructures within Python so that these data can be utilized *Smithy*. The data structure for a given query result holds all of the information produced by the BLAST query in the Biopython Alignment object. For the purposes of *Smithy*, only the DNA sequence query command is used, but all of the BLAST+ commands are available for use.

2.2 Pydna – 2015

```
In [14]: cyc1_prd = pydna.pcr((p1,p2), cyc1)
In [15]: gfp_prd = pydna.pcr((p3,p4), gfp)
In [16]: cyc1_prd
Out[16]: Amplicon(380)
In [17]: cyc1_prd.figure()
Out[17]:
                                  5ATGACTGAATTCAAGG...GAAAAAAGCCTGTGAG3
                                                      |||||||||||||||| tm 41.4 (dbd) 53.0
                                                    3CTTTTTTCGGACACTCTACAGATTTCCACTTCTT5
5ATTATCTACTTTTTACAACAAATATAAAACCAAAAATGACTGAATTCAAGG3
                                   |||||||||||||||| tm 38.7 (dbd) 49.7
                                  3TACTGACTTAAGTTCC...CTTTTTTCGGACACTC5
In [18]: cyc1_prd.program()
Out[18]:
Taq (rate 30 nt/s) 35 cycles             |380bp
95.0°C    |95.0°C                |      |SantaLucia 1998
|_________|_____          72.0°C  |72.0°C|SaltC 50mM
| 03min00s|30s  \        _______|______|
|         |      \ 50.0°C/ 0min11s| 5min |
|         |       \_____/         |      |
|         |         30s           |      |4-12°C
```

Figure 9: Sample Pydna code execution from publication[27]. This sample code demonstrates the Pydna in silico PCR simulation capabilities for two sequences: CYC1 and GFP. For The two sequences, each is amplified using pydna.pcr(). Then the PCR amplicon for CYC1 is used to print its figure and program, which respectively show the primer bindings and a recommended PCR procedure.

The Pydna project is another Python code library that expands greatly on the efforts of Biopython into the realm of DNA assembly projects[27]. Pydna's main focus is on rough simulations and project documentation in a transmissible form for programmers and biologists alike. DNA sequence records can be stored in linear or circular formats, common molecular biology functions such as reverse complements and transcriptions can be performed on sequences, restriction enzyme digestions, and more can be done in a very simple fashion within a Pydna script. Additionally, a suite of tools is available for primer design, PCR simulation, and overlap-extension based assembly simulations are also possible in Pydna. A major contribution of the Pydna project is its ability to allow biologists and bioinformaticians to transmit project ideas and designs in a highly repeatable, reliable, and amendable fashion[27].

This library has been used extensively in the development of *Smithy*, notably the primer design capability, which uses a customizable multiparameter melting temperature function, and the various sequence record objects available.

2.3 J5 – 2012

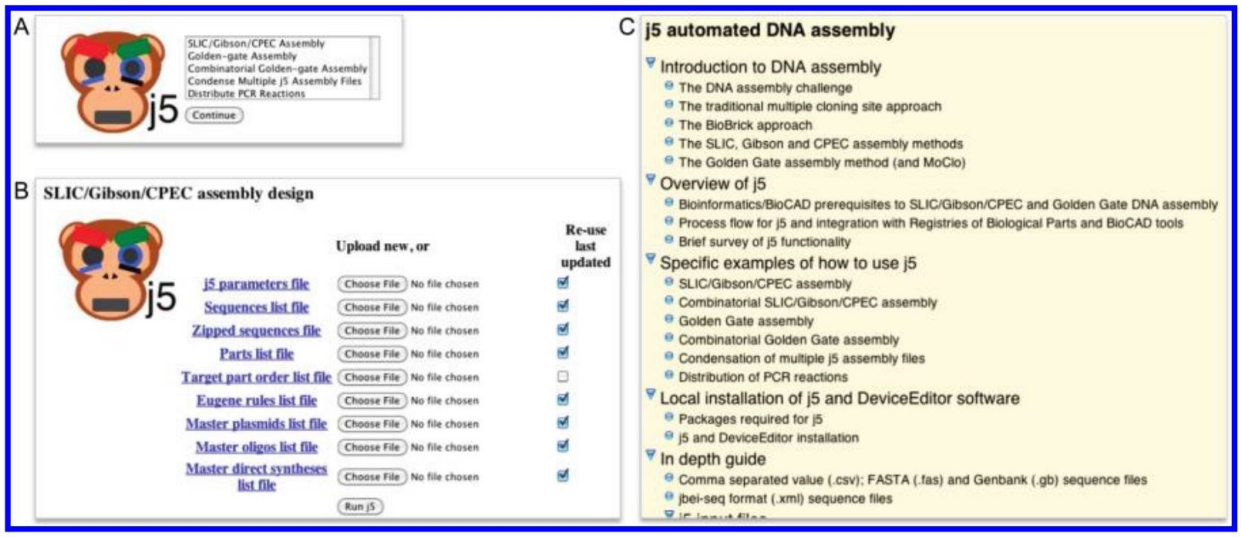


Figure 10: Figure 1 from j5 paper showing user interface[28]. (A) Selection of the desired cloning method. (B) The file inputs for every design aspect of the assembly for the j5 software. (C) User manual for the software.

First published in 2011, j5 provides a web service for design automation of SLIC, Gibson, CPEC, and Golden Gate assemblies[28]. With user specified design rules, j5 will help create scar-less assembly designs for a selected cloning method. Furthermore, the application will suggest direct part synthesis and hierarchical assembly strategies where necessary. An assembly design is generated after a user submits a collection of input files for parameters, sequences, design rules, and more. Results of the solution generation are found in another set of output files with extensive details of the assembly strategy determined by the software. The application employs 5 core algorithms to accomplish its tasks, which cover determining optimal PCR strategies, assembly

primer extension sequences (flanking homology sequences for Gibson Assembly, for example), finding incompatible assembly parts, and cost-optimized sequence sets for assembly[28]. J5 presented a useful tool for aiding assembly designs as cloning objectives and methodologies grew in scale and complexity.

2.4 REPP – 2020

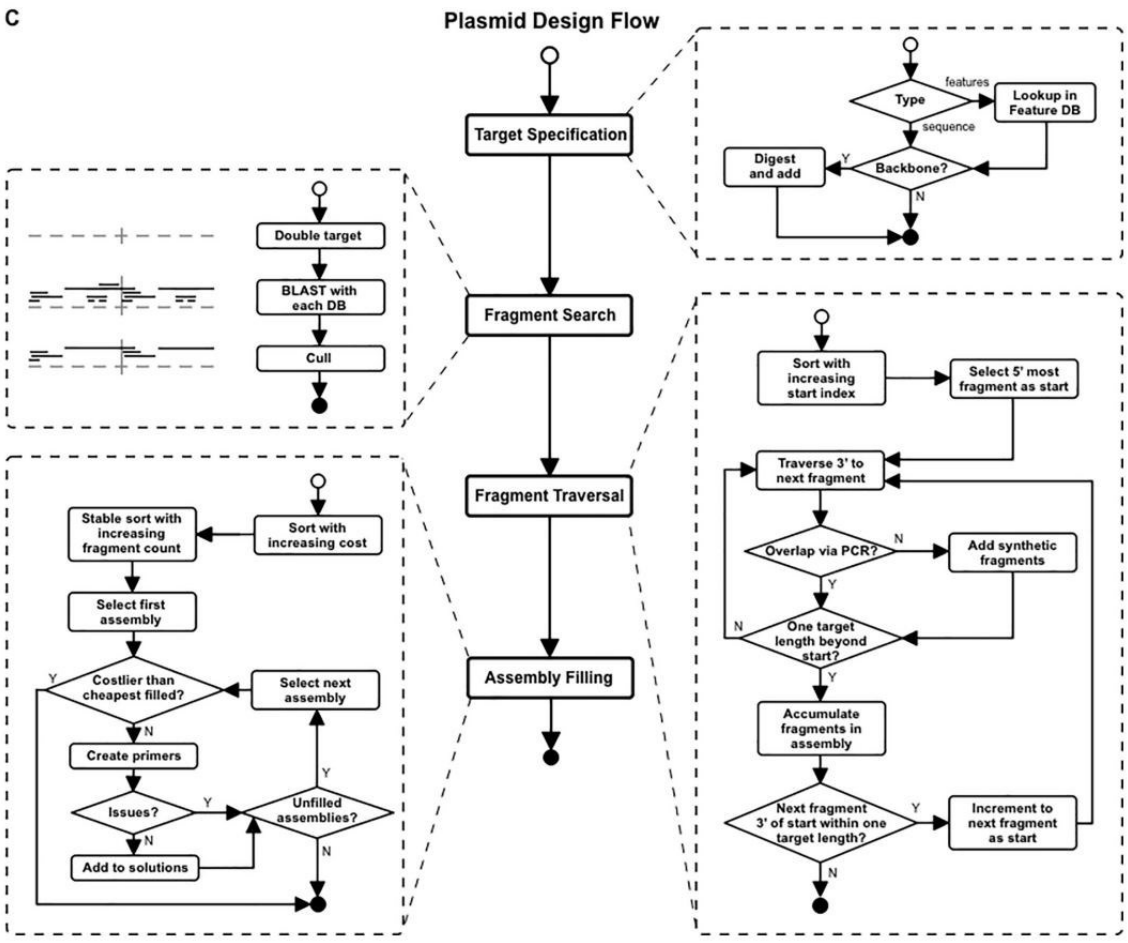


Figure 11: REPP algorithmic flowchart from Figure 1-C of publication[29] detailing how assembly solutions are designed.

Repository-based plasmid design (REPP) is a recently published project that, like j5, provides cloning design automation for Gibson assemblies only[29]. The software is a command line tool that

takes a different approach to design generation than j5. While j5 provides a large suite of information and protocol suggestions for different types of assembly methods, it requires user-submitted sequences. With REPP, the objective plasmid is submitted by either its full sequence or by a list of feature names. In response, the program will find existing sequences for the plasmid, using BLAST queries, in available local copies of major DNA databases, being AddGene, iGEM, and DNASU. Then, given the set of sequences found for the plasmid, gaps in the query results will be filled in with sequences to be purchased through direct synthesis. Cost optimization between synthetic and database parts is then performed and the rest of the Gibson assembly designed. Similar to j5, REPP utilizes a collection of several algorithms for its sequence queries, synthetic sequence determinations, cost optimizations, and more. This project offers a unique approach to design automation for Gibson assemblies by using queries on local copies of popular sequence repositories and performing cost optimizations based on synthetic sequence count, repository sequence count, overall cost, and total amount of assembly fragments[29].

2.5 Benchling

Benchling is a powerful cloud-based platform for academic and industrial research and development efforts[30]. It is a centralized service providing several different applications for users to take advantage of, such as project collaboration, project management, analytics, and experimental resource tracking. Part of their Molecular Biology toolset includes primer design aid and *in silico* execution of PCR amplification along with digestion-ligation, Gibson assembly, and Golden Gate assembly protocols[30]. Also included are primer secondary structure prediction and sequence design management. Paired with the overall feature set of Benchling, these primer and assembly features provide convenient and reliable design functionalities for contemporary, complex Synthetic Biology projects.

2.6 Cloning Methodology Study & Taxonomy

A core theoretical component of the *Smithy* project is the *taxonomy of cloning*. This taxonomy groups and organizes cloning methods into a hierarchy based on abstract attributes of each method. The concept arose from a motivation to limit the amount of algorithmic implementation of supported cloning methods for *Smithy*. After careful study and analysis of several different methods and reviews of the methodologies[3,4,18,21,31,32], developed over the entire timeline of the field, the taxonomy was discovered.

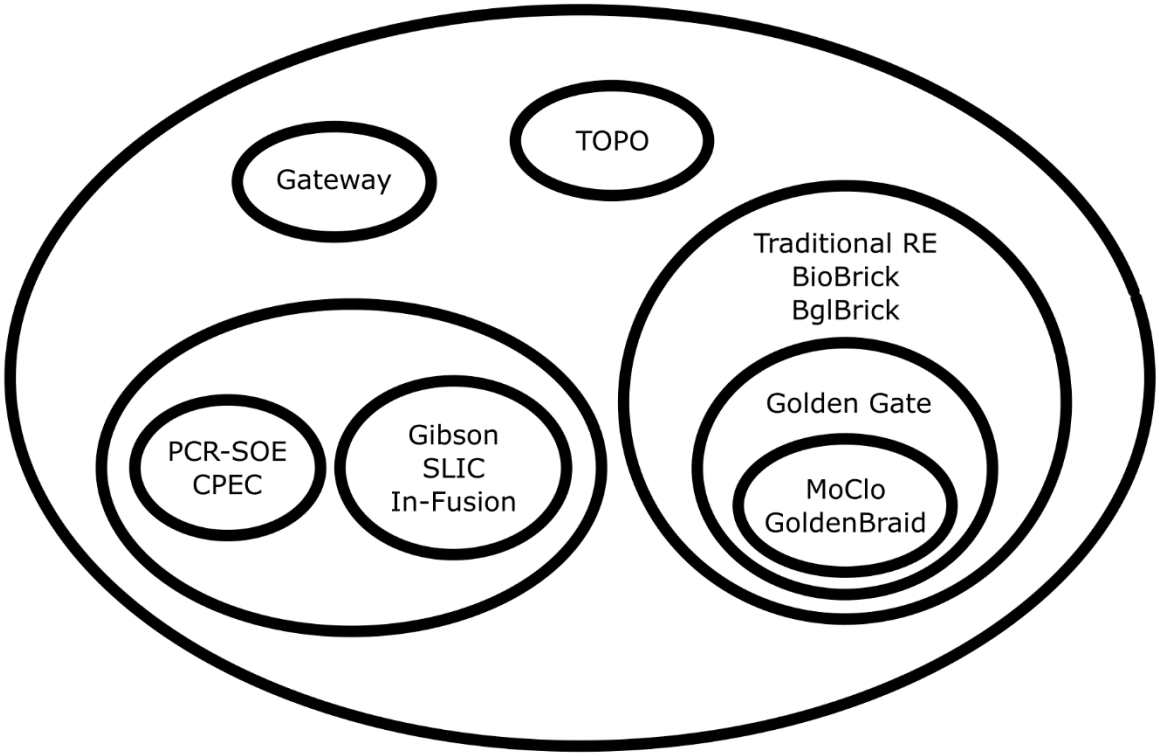


Figure 12: Cloning methods abstraction and mapping

The first group contains restriction enzyme (RE) based methods and begins with the traditional cloning and BioBricks methods. The next level of this category is Golden Gate assembly, as this method expands on the traditional RE method to use Type2S REs for more control, efficiency,

scalability, and reliability of assembly constructs. Finally, the last level is built on top of the Golden Gate category and is composed of the MoClo and GoldenBraid methods. These two methods expand further on those seen in Golden Gate for greater standardization, scalability, and complexity of assembly endeavors through hierarchical cloning strategies via additional Type2S enzymes[7,19,20].

The next major group is concerned with overlap-based assemblies that rely on sequence homologies between neighboring DNA fragments. The first sub-category developed contains methods that use enzymes to chew back, ligate, and repair multiple DNA sequences for assembly based on shared sequence homologies. The most well-known method of this type is Gibson assembly. Its relatives in the group are SLIC and In-fusion[33], which operate under the same methodological concepts and approach. The next sub-category does not contain methods utilizing chew back, ligation, and repair but rather overlap extension and ligation through polymerase chain reaction (PCR) procedures. Contained in this group are the methods PCR-SOE and CPEC[34].

The last two groups are independent of each other, one containing TOPO cloning and the other Gateway cloning[35,36]. These two methods are not included in the application, but they were added to the taxonomy, so it covers all popular methods. It is, however, possible to add them to *Smithy*, and efficiently, without disturbing *Smithy*'s overall software architecture.

As stated, not all of the methods under study for the taxonomy are implemented in *Smithy*, however they were mentioned because they were crucial to developing the categories of the hierarchy. This taxonomy is a fundamental design aspect of features in *Smithy*, namely the *Assembler* software class hierarchy, that are described later.

2.7 *Smithy*

We've created *Smithy*, a web application for automated design of complex DNA assembly experiments. The application is written in Python and uses the Django web application framework library. Several other Python libraries are used for efficient and precise automated design, along with the BLAST+ CLI program[26] for sequence alignment queries, JavaScript and the UI/UX tools of Bootstrap5 and Chart.js[37,38]. The important core functionalities of *Smithy* are contained in seven algorithms, and they will be described in order of processing, starting with BLAST queries and concluding with assembly primer design.

The first two algorithms involve performing BLAST queries using the BLAST+ command line interface (CLI) through Python for obtaining DNA sequence alignments from DNA databases. These are what we call *single-entry* and *multi-entry* queries, each having a separate unique algorithm for running the BLAST query and organizing the results properly. Single-entry queries use a FASTA input file containing one entry for the entire insert sequence to query over, while multi-entry queries also use a single input file, but with multiple entries of the insert's subsequences in proper order. Results of these two query modes are structured differently, so they need different organizational approaches for the data, approaches that are usable in the rest of the application's pipeline. The user, at the input, decides whether to use a single- or multi-query format.

Next, a single algorithm has been developed to obtain BLAST query results and build solutions. Here a solution is any ordered set of BLAST alignments that result in the exact same DNA sequence as the input query with no errors; effectively a rebuild of the input sequence. Solutions are created using the *FragmentTree* class that has been developed for the *Smithy* project. This class uses the entire set of query results from BLAST to construct a graph network, which contains every

possible solution for a given insert in a tree data structure. Any solution obtained from this network can be used for running specific assembly method design algorithms to produce a completed solution that facilitate experimental construction of the genetic assembly.

There are three algorithms used for the assembly design, given a particular solution from the previous step. Only one is used in each assembly method's design procedures as they describe different primer design approaches. Thus there is one for overlap extension assemblies, Gibson, PCR, and SLIC, which is found in the *OverlapExtensionAssembler* class. Then there is a single algorithm for Golden Gate assembly primers found in the *GoldenGateAssembler* class, and there is a single algorithm for BioBricks assemblies found in the *BioBrickAssembler* class. These primer design algorithms take a solution built by the *FragmentTree* and generate complete assembly primers for each part in the solution including the backbone. As a result, the full solution is comprised of the parts, sourcing listing, and the complete set of primers that needed for sequence extension and *in vitro* experimentation. Additionally, thermodynamic analysis through Primer3 is added to each primer and solution-level metadata is computed for analytical charts to be presented to the user. These charts determine the predicted approximate cost, risk, and time associated with the experimental DNA assembly[39]. Lastly, the design functions used by the assembler classes are described. These perform the full design procedure for a single solution generated by the software.

More on these algorithms with be described in the *Computational and Biological Methods* chapter, along with further detailing the front-end, middleware, and backend architecture of the project. The various databases used in the software will also be described, those being the BLAST DNA sequence databases and SQL tables, for assemblies, parts, and primers, used for the Django web application component of the project.

After critical review and analysis of relevant design tools in the field, shortcomings and gaps in features provided by existing tools, and new contributions in these areas providable by *Smithy* were identified. J5 is presented in its publication as a software tool available online. However, when navigating to its original link (j5.jbei.org) no assembly design tool is available, and visitors are referred to various other tools and publications for biological parts registries and formats[40,41], a deprecated tool called VectorEditor[42], a linked tool to j5 called DeviceEditor that is inaccessible[43], and a new plasmid editing tool called Open Vector Editor[44]. In addition, while j5 proposes strong design capabilities for popular cloning methods, there was no feature allowing for comparison of multiple assembly strategies. REPP presents itself as a strong DNA assembly design tool utilizing BLAST sequence searches, however major shortcomings were identified when it was installed for use. Following the instructions of the publication and online documentation, the full advertised feature set for the various ways a target construct could be submitted and designed was not completely functional. Designing a plasmid from feature names or insert subsequences did not function[29,45]. As a command line interface (CLI) tool, its use is prone to user errors and difficulties unless users are already familiar with the tool or working in the command line of a computer. Additionally, the installation and execution of the software was unsuccessful when attempted on various systems. Furthermore, the design software is implemented in the Go programming language[46] and only supports Gibson assemblies. The Go programming language is not one seen widely within scientific and academic communities and strong, highly developed scientific software libraries exist within other software communities, particularly Python and R. Having support for only Gibson assemblies greatly limits the range of researchers who will be attracted to use the tool. Pydna provides excellent DNA sequence manipulation, management, and annotation features while including *in silico* PCR and general overlap extensions assembly designs. However,

like REPP, having support for only a single DNA assembly methodology limits the applications of the software library, where assembly design is concerned. The shortcomings and lacking features of these existing tools provide several opportunities to contribute improvements and expansions.

Smithy addresses the opportunities identified in these projects in various ways. First, it addresses the issues in j5 by existing as an openly accessible, simple-to-use web application with minimal user inputs. Additionally, *Smithy* allows comparison of multiple assembly options for single constructs through a feature called *assembly bundles*. These allow users to make informed decisions on the optimal cloning methodology to use for a given target construct. The shortcomings of REPP are addressed by avoiding a CLI tool altogether by providing a webservice runnable by any common browser. REPP only support Gibson assemblies, but in *Smithy* Gibson, Golden Gate, PCR-SOE, SLIC, and BioBricks assemblies are implemented. These five methods were realized, amongst the greater set of existing cloning methods, based on a few key criteria. First, they hold significant popularity within the synthetic biology cloning community. Second, compared to the whole collection of cloning methods studied, these all showed opportunity for design aid that would benefit researchers in their cloning projects. Third, apart from BioBricks, these methods do not operate under existing, strict standardization in the methodology for primer designs and entry vectors. The methods of MoClo and GoldenBraid have very clearly defined plasmid entry vectors, overhang sequences to use, and overall procedures for assembling complex target constructs. In sum, we focused the efforts of implementing methods on Gibson, Golden Gate, SLIC, PCR-SOE, and BioBricks, though others, particularly MoClo and GoldenBraid, could be implemented in the future.

The software of *Smithy* is also written in Python, which is broadly accessible to many different groups in industry and academia with a highly diverse open-source community for scientific libraries. The use of Biopython[25], Pydna[27], and Primer3-py[39] in *Smithy* demonstrate this point. The assembly design features seen in Pydna are expanded upon in the assembly design classes implemented in *Smithy*.

With regards to Benchling, while their toolset and quality are thorough and high-quality, there are limitations to the assemblies they provide and differences in approach for assembly designs. First, only RE cloning, Gibson assembly, and Golden Gate assembly methods are supported. These three are also present in *Smithy*, but in addition, the PCR-SOE, SLIC, and BioBricks methods are also implemented. Second, the assembly design approach of Benchling utilizes plasmid constructs created by users within the application, without BLAST queries, synthetic fragment gap-filling, or solution candidate generation. Smithy implements these strategies for providing optimal target construct options to users. In addition, Benchling only offers only a limited set of its tools in a free version, which does include their Molecular Biology suite. *Smithy*, however, is open-source and free, with reusable software modules for other Python Bioinformatics and Synthetic Biology software developers.

Table 1: Comparison of different cloning design tools

	Biopython	**Pydna**	**J5**	**REPP**	**Benchling**	**Smithy**
Sequence sourcing	No	No	No	Yes	No	Yes
Cloning method support	No	No	Yes	No	Yes	Yes
Comparison of methods	No	No	No	No	No	Yes
Reusability	Yes	Yes	No	Yes	No	Yes

Extensibility	Yes	Yes	No	Yes	No	Yes
Simple UI/UX	No	No	No	No	Yes	Yes

Overall, *Smithy* collects a range of advantageous features for DNA cloning design automation seen in different projects, such as support for multiple methods, BLAST database sequence queries, openly accessible interfaces, and implements them into a single product that is not bound to simply being a web application. *Smithy* provides complete and usable primer designs, sequence sourcing from large DNA databases, thorough analytics for assembly solutions, and comparison of multiple assembly method options for individual solutions. The core functionality of *Smithy* can be imported into other synthetic biology software projects apart from the Django web application implementation. Furthermore, these features are expanded upon with reusable, extendible, and future-oriented software that can handle the addition of new cloning methods and features with ease.

Smithy is available online at: rs-loy-smithy.concordia.ca

Chapter 3: Computational and Biological Methods

In this chapter, the complete workflow of *Smithy* for assembly design generation is presented (Figure 13). The chapter begins by explaining the inputs required for creating assembly designs (Section 3.1). Assembly selection and input forms are described in detail. Second, the software that processes user inputs for generating target assembly construct designs, is explained (Section 3.2). Here, the Python classes that perform assembly designs, the databases containing DNA sequences and the web-application's data, and the full suite of design algorithms for generating comprehensive assembly designs are described. These components of *Smithy* are responsible for accepting the user inputs (mainly, desired assembly), and producing all the information needed for the (wet lab) construction of a complete DNA assembly solution. Third, how assembly solutions and their information are presented to the user on the web pages of the application is detailed (Section 3.3). Fourth, the key feature of *assembly bundles* is presented*;* it generates multiple design solutions, utilizing different assembly methods, leading to the same target construct (Section 3.4).

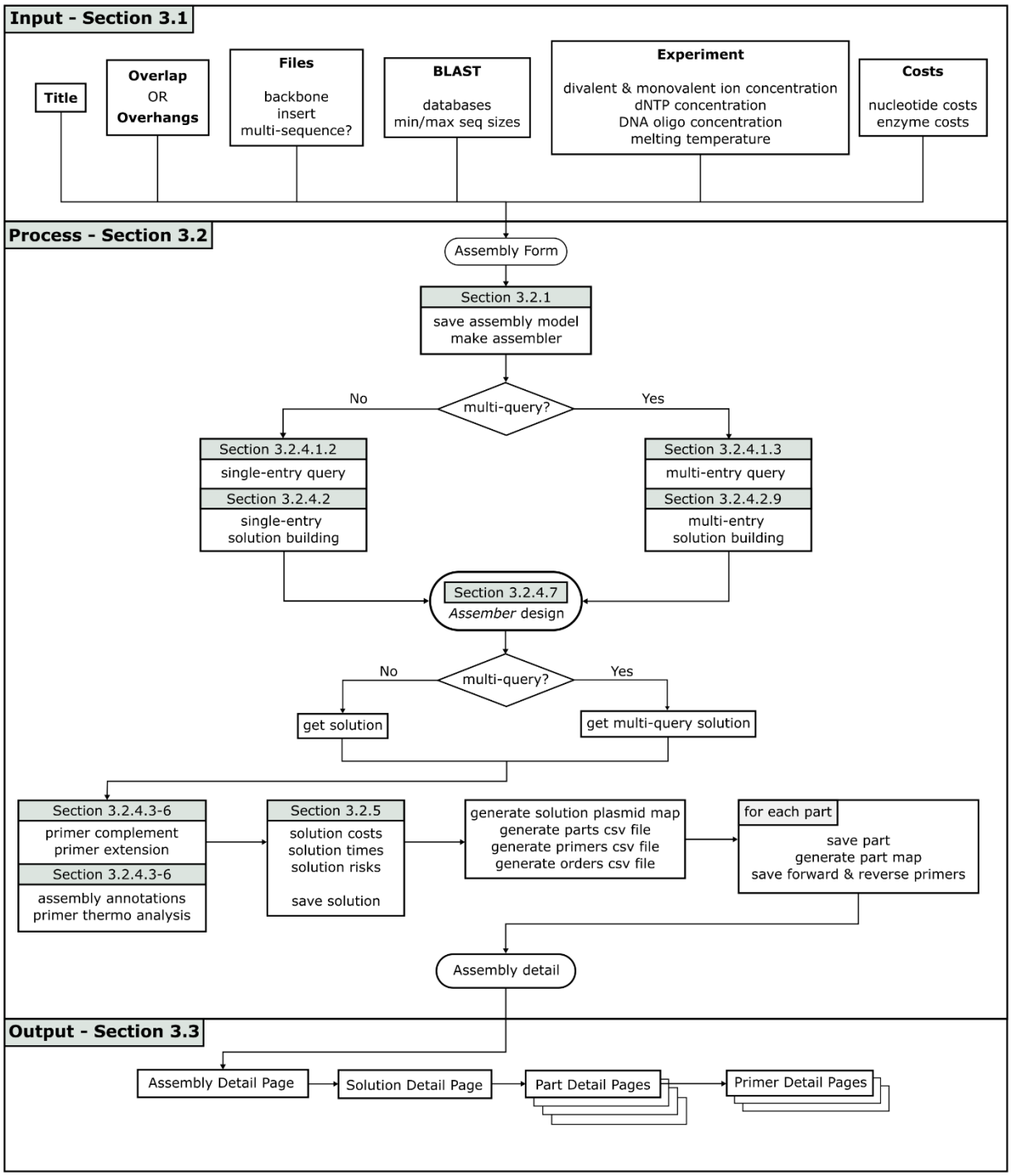


Figure 13: Visual representation of the Smithy assembly design process

3.1 Input

This section covers the input requirements and procedures for *Smithy*. The description will start at selecting a desired assembly, then the assembly form inputs are detailed. Differences between the input forms for different assemblies are also clarified.

3.1.1 Assembly Selection

The web page for assembly selection can be accessed from the home page and global site navigation bar. A page for selecting the assembly method is loaded, and used for project type selection. Current assembly options are Gibson, Golden Gate, PCR-SOE, BioBricks and assembly bundles. Once a selection is made, an input form is displayed, which requests the information needed for the assembly and for solution building.

3.1.2 Assembly Form and Submission

The input forms are similar across all supported cloning methods;

- Title
- Backbone sequence FASTA file
- Insert sequence FASTA file, either single- or multi-entry
- Boolean selection for single- or multi-entry BLAST query
- Overlap amount, if using Gibson, SLIC, or PCR-SOE
- High-fidelity overhang set, if using Golden Gate assembly
- Boolean selection of BLAST databases to query (AddGene, DNASU, iGEM)
- Minimum and maximum BLAST sequence sizes
- Minimum and maximum synthetic sequence sizes
- Monovalent and divalent ion concentrations (for Primer3 calculations)

- dNTP concentration (for Primer3 calculations)
- DNA oligo concentration (for Primer3 calculations)
- Target melting temperature (Celsius)
- Primer cost per nucleotide (0-100nt)
- Part cost per nucleotide (100-1000nt)
- Gene cost per nucleotide (>1000nt)

Some differences between the forms do exist, reflecting different methodological requirements and materials costs. Overlap extension assemblies (Gibson, SLIC, and PCR-SOE) ask for overlap lengths. Golden Gate assembly requires selection of a high-fidelity overhang set of which there are four[47]. BioBricks does not ask for either of these parameters or any other unique input for assembly design. Apart from these special design inputs, there are additional cost parameters for the methods' enzyme requirements. Users select each enzyme they need to perform the experiments, and only when selected, will a dollar cost value be inputted. The different enzyme cost inputs available for each assembly type are:

- Gibson: exonuclease, ligase, and polymerase
- Golden Gate: BsaI and ligase.
- PCR: polymerase
- SLIC: exonuclease and ligase
- BioBricks: EcoRI, XbaI, SpeI, PstI

Once the input form is complete, a single *Assemble* button is clicked and the user is shown a simple loading screen while *Smithy* runs the design pipeline (Section 3.2.4). When the design procedure is complete, results will be available through the assembly detail pages.

3.2 Assembly Design

Each major aspect of the solution generation process is explained in this section. Section 3.2.1 describes the Python classes that implement the full design procedure of *Smithy*. Sections 3.2.2-3 provides an overview of the databases for the application. One of these is the BLAST DNA sequence database used for target construct alignment searches, and the other is the Django application's SQL database where complete cloning assembly solution data is stored. Section 3.2.4 explores the seven core algorithms used for generating solution designs. Section 3.2.5 details the assembly solution cost, time, and risk estimates that are useful in determining whether or not a particular solution and its methodology is to be carried out in real experiments.

3.2.1 Assembler Classes

This cloning method class structure has a base class named *Assembler*, seen in Figure 24. The *Assembler* class implements the core functionality for achieving the objectives of *Smithy*, to automate the design pipeline of DNA assembly projects. *Assembler* contains the majority of the code, parameters, functions, algorithms, and variables. This class implements the following functionalities and data items:

- Building and running of BLAST queries
- Collecting and ordering BLAST results
- Building assembly insert solutions with BLAST results
- Assembly primer designs
- Primer thermodynamic analysis
- Parts annotations
- Analytical metadata calculation for assembly solutions

Two main child classes are inherited from the *Assembler* class, and they are used to further implement more child classes that are used within the *Smithy* application. These two classes are *TraditionalREAssembler* and *OverlapExtensionAssembler*. *TraditionalREAssembler* begins the branch from base functionality to specify restriction endonuclease based cloning methods: traditional cloning, Golden Gate, and BioBricks. *OverlapExtensionAssembler* begins another branch off base functionality to implement overlap based cloning methods: PCR-SOE, SLIC, and Gibson. With these two main branches in the class hierarchy, all the major cloning methods used for the project were implemented in Python, as their own child classes via minimal additional code. These child classes control initial parameters (restriction enzyme type, overlap amount, overhang sets, etc.), the primer extension algorithms, and solution design procedures (Figure 24).

Starting with *TraditionalREAssembler*, this class extends the base class to allow for the addition of two restriction enzymes, a ligase, and a polymerase. Two functions are implemented for adding cut-sites to assembly parts, one for adding a single RE cut-site to both ends of a part, and another for adding two unique cut-sites to a part: one for the forward and one for the reverse primer. Additionally, this class implements four basic cloning functions for different traditional cloning approaches: three different single digest functions, and one double digest function[48]. These functions allow this class to operate on its own for traditional cloning designs, using a single insert sequence.

The first child class of *TraditionalREAssembler* is the *GoldenGateAssembler* class. It has two main modes of operation that are chosen for any assembly, whether utilizing scar-less or non-scar-less assembly primer design. The overhangs used within the *GoldenGateAssembler* class originate from a comprehensive high-fidelity profiling of 4 nt overhang sequences, for Type 2S restriction enzyme cloning methods, that avoid assembly mismatches as much as possible.[47]. Four different

sets are available for use: the first containing 15 overhangs at 98.5% fidelity, the second with 20 overhangs for 98.1% fidelity, the third with 25 overhangs for 95.8% fidelity, and the last with 30 overhangs for 91.7% fidelity[47]. Utilizing these profiled overhang sets allows users to select the sets that best match the number of expected insert fragments to use and fulfill expectations of assembly accuracy. These design algorithms are detailed in Section 3.2.4.5. Additionally, this class has its own design algorithm for complete (forward and reverse) assembly primer design, for each part, in an assembly solution. The last implemented child class of *TraditionalREAssembler* is the *BioBrickAssembler* class. As with the other assembly classes, it designs appropriate forward and reverse primers for each part, to perform a BioBricks assembly experiment (Section 3.2.4.6). Primer extensions are static, according to the BioBricks standards: the same suffix sequence for reverse primers and two prefix sequences for forward primers, one for coding sequences and one for all others. The Golden Gate and BioBricks assembler classes are the only restriction enzyme-based classes in *Smithy*.

Moving on from the *TraditionalREAssembler* branch is the *OverlapExtensionAssembler* class and its child classes. After studying the methodologies of overlap-based cloning methods, it was determined that they ultimately do not require much unique implementation. So *OverlapExtensionAssembler* implements all the current features for its child classes, which are *PCRAssembler*, *SLICAssembler*, and *GibsonAssembler*. Similar to the Golden Gate assembler class, *OverlapExtensionAssembler* takes all parts in a solution, and adds primer extensions for assembly, but based on overlap extensions, using the overlap length parameter provided by users (Section 3.2.4.4). The Gibson, SLIC, and PCR classes do not add more features/functionality on top of what is already defined in their parent class. These three additional assembler classes complete the suite of cloning methods supported by *Smithy*.

3.2.2 BLAST Sequence Databases

With a local installation of BLAST+, BLAST databases can either be downloaded from the NCBI site, or created using multi-entry FASTA files[26]. *Smithy* makes use of the latter feature of BLAST+ to implement the local sequence databases for solution building from three popular DNA sequence and plasmid providers: AddGene, DNASU, and iGEM. Large FASTA files containing entries for the current state of these repositories' datasets were created. Then, the *makeblastdb* command was executed for each repository's FASTA file to generate local, queryable BLAST databases for the individual repositories. The sequence data for each provider was obtained upon direct request and is the most current available datasets as of 2021 - though regular updates should be made. Once created, these databases are accessible through the BLAST+ functionality used extensively in *Smithy* (Section 3.2.4.1).

3.2.3 SQL Database Tables and Models

The Django application framework for *Smithy* relies on a collection of SQLite database tables containing unique entries for each assembly with their parts, primers, and solutions. Bundled assemblies are also entered into the database when created by users. Each of these model types have their own SQL tables for each type of supported cloning method. For example, Gibson assemblies have a two-part table for solutions, one part for parts, and one for primers; this pattern applies to all assembly methods including: Golden Gate, SLIC, PCR-SOE, and BioBricks. See Figure 14 for this model relationship pattern. Every entry in these tables represents a unique component of a single assembly created by the user.

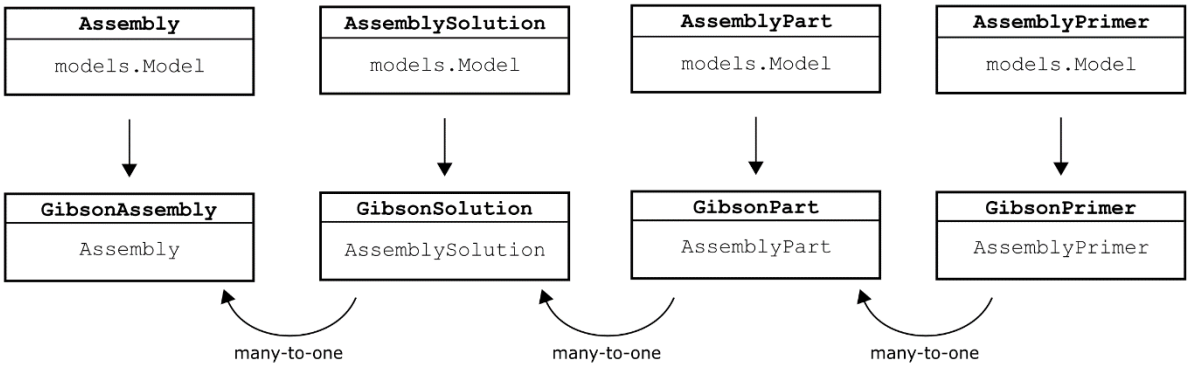


Figure 14: Smithy SQL models breakdown for Gibson assemblies. The model types are in bold in the box headers with their parent classes in lighter text within the box.

The hierarchical order of the database model types used in the application is: bundles, assemblies, solutions, parts, and primers. An assembly is created, and any number of solutions for that assembly is then associated with it in the database. Each solution has any number of parts added to its associations, which in total compose the full solution. Likewise, each part for a solution has primers associated with it, which are added to the database; *Smithy* only utilizes two primers: a forward one and a reverse one. When assembly bundling is chosen, the organization of data within each assembly is the same, but an *AssemblyBundle* model is created for each assembly selected by the user, using many-to-many relationships; see Figure 15 for the bundle relationships.

To reduce new code, the Django models used for these assembly components were first designed and implemented abstractly, both theoretically and practically. Django allows for database models to be defined abstractly without the creation of actual tables; tables are only created from new model class definitions, which inherit from the abstract class. This inheritance structure is seen in Figure 14. Thus, for the assembly, solution, part, and primer models are first defined using this abstract model class feature, named *Assembly*, *AssemblySolution*, *AssemblyPart*, and

AssemblyPrimer. Hence, every cloning method type can be defined using these abstract model classes for their own model. For example, for Gibson assemblies its models are *GibsonAssembly*, *GibsonSolution*, *GibsonPart*, and *GibsonPrimer*-- all inherited from their respective parent classes (Figure 14).

3.2.3.1 Assemblies

Assembly models are defined through inheritance from the *Assembly* abstract model class. This class defines most attributes, for all other child assembly models, of supported cloning methodologies. The attributes of this abstract class are:

- Title
- Date created
- Backbone sequence
- Insert sequence
- Multi-query (true/false), used for determining if multi-query functionality is used
- AddGene (true/false), for selecting this BLAST database for query
- iGEM (true/false), for selecting this BLAST database for query
- DNASU (true/false), for selecting this BLAST database for query
- The minimum BLAST sequence size
- The maximum BLAST sequence size
- The minimum synthetic sequence size
- The maximum synthetic sequence size
- Monovalent ion concentration ([Na+])
- Divalent ion concentration ([Mg2+])

- dNTP concentration
- DNA oligo concentration
- Melting temperature (°C)
- The backbone record file
- The insert record file

This abstract class does not result in the creation of a SQL table. When child classes are defined, these are used for creating tables and saving unique model entries. When child classes inherit from this base class, they contain all parent attributes, and thus only need to specify new attributes unique to the assembly type. This idea applies to all other abstract parent and child models, defined for the application. The assemblies created using the *Assembly* parent class are the supported cloning methods of *Smithy*: *GibsonAssembly*, *GoldenGateAssembly*, *PCRAssembly*, *SLICAssembly*, and *BioBricksAssembly*. These classes are used to implement the assembly forms at user input, and when submitted these forms allow the unique instances of the *Assembler* class to be created, used, and saved to the SQL database. After the bundle model, which is only used for specific situations, these assembly classes are the highest level of relationships within the SQLite database.

3.2.3.2 Solutions

The design and implementation of assembly solution models follows the same approach as the *Assembly* models. An abstract base model class is defined for child solution classes called *AssemblySolution*. The specific classes simply need to define their relations to specific *Assembler* classes they are associated with, as the attributes needed for *Smithy* are predefined in *AssemblySolution*. For example, the Gibson solution model is called *GibsonSolution* and it inherits from this base class. It defines a foreign key relationship with the *GibsonAssembly* model for a

many-to-one relationship of solutions to assemblies. This implementation style is also seen for the other cloning methods' solution models. Due to this many-to-one relationship, several different solutions can be designed for a single assembly model, solutions that are all accessible to each other via Django and SQL functionalities.

Solution models hold unique data about a single solution, as designed by the assembler class, using the design methodologies described in Section 3.2.4.

The direct, descriptive information on a solution are:

- Name
- Date created
- Backbone sequence
- Query sequence
- Solution sequence (can be different from the query sequence)
- Parts count
- Primers count

In addition, the following analytical attributes are defined:

- BLAST nucleotide match percentage
- Synthetic nucleotide percentage
- A graphical plasmid map
- Required restriction enzymes for assembly
- Average part length
- Average primer length
- Longest part length

- Shortest part length
- Average primer melting temperature
- Number of BLAST database parts
- Number of synthetic parts

Each of the assembler models defined for the supported cloning methods, has one implemented solution model, which is inherited from the parent model class. As such, *GibsonSolution* is defined for *GibsonAssembly*, *GoldenGateSolution* for *GoldenGateAssembly*, and so on.

3.2.3.3 Parts and Primers

As with the other models, the database models for parts and primers are defined in the same fashion. First, abstract parent classes are defined, and then unique child classes for the individual cloning methodologies are defined. The abstract class for parts is called *AssemblyPart*, while the one for primers is called *AssemblyPrimer*.

The attributes defined for *AssemblyPart* are:

- Name
- Date created
- Origin database
- Original length
- Assembly extension length
- Original nucleotide sequence
- Assembly extension sequence
- Positional index in the solution
- BLAST query start index

- BLAST query end index
- BLAST subject start index
- BLAST subject end index
- Graphical part map
- Number of Type2S restriction enzyme cuts (Golden Gate only)
- Locations of Type2S restriction enzyme cuts (Golden Gate only)

The attributes for *AssemblyPrimer* are:

- Name
- Date created
- Primer type (forward or reverse)
- Full sequence
- Annealing sequence
- Extension sequence
- Total melting temperature (Celsius)
- Annealing sequence melting temperature (Celsius)
- GC percentage
- Hairpin found (true/false)
- Hairpin melting temperature
- Hairpin delta Gibbs
- Hairpin delta H (enthalpy)
- Hairpin delta S (entropy)
- Homodimer found (true/false)

- Homodimer melting temperature
- Homodimer delta Gibbs
- Homodimer delta H (enthalpy)
- Homodimer delta S (entropy)

3.2.3.4 Bundle

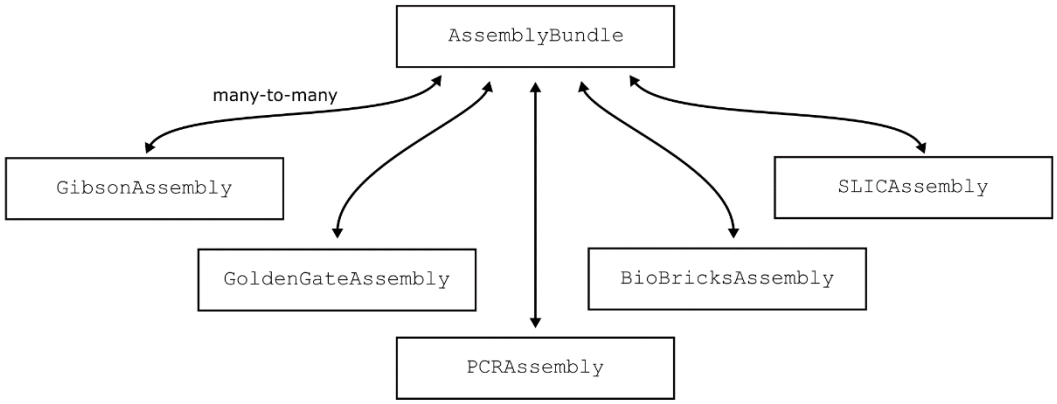


Figure 15: Assembly bundle many-to-many model relationships

The assembly bundle is a feature of *Smithy* used to consolidate into a single procedure, all of the computation needed for the creation of multiple (user-selected) assemblies. The model class is used for a many-to-many relationship between selected assemblies and the *AssemblyBundle* model (Figure 15). A new bundle is created with relationships to the selected assemblies, which all follow the standard relational hierarchy. The attributes of *AssemblyBundle* are:

- Title
- Date created
- Description
- Gibson, a many-to-many attribute for *GibsonAssembly*

- Golden Gate, a many-to-many attribute for *GoldenGateAssembly*
- SLIC, a many-to-many attribute for *SLICAssembly*
- PCR, a many-to-many attribute for *PCRAssembly*
- BioBricks, a many-to-many attribute for *BioBricksAssembly*

3.2.4 Algorithms

Here, *Smithy's* core algorithms for generation of assembly solutions are presented. Section 3.2.4.1 explains the two different methods of performing BLAST sequence alignment queries for finding DNA sequences of the target assembly construct. The target construct sequence is provided and DNA sequence alignments from selected databases are obtained to be used for creating solutions. The way in which *Smithy* uses BLAST alignments for generating these candidate solutions is shown in Section 3.2.4.2. All of the alignments from the BLAST queries are used to generate a comprehensive set of combinations of sequences that compose candidate solutions, reconstructing the target assembly construct sequence. Section 3.2.4.3 describes the beginning stages of assembly primer design for a single solution candidate where non-extension primer sequences are created using a target melting temperature. These three main algorithms are used in each DNA cloning methodology's software implementation.

The next three sections, Sections 3.2.4.4-6, present the three different primer extension sequence design procedures for the different cloning methods supported by *Smithy*. Using a single solution with its non-extension primers, the primer extension sequences that facilitate cloning assemblies during experiments for specific methodologies (e.g., Gibson or Golden Gate assembly) are created. The three primer extension design algorithms cover overlap extension assemblies (Gibson, PCR-SOE, SLIC), Golden Gate assemblies, and BioBrick assemblies.

Lastly, Section 3.2.4.7 shows how the algorithms of Sections 3.2.4.1-6 are used to execute the full assembly solution design process of *Smithy*.

3.2.4.1 Blast Single-Entry and Multi-Entry Queries

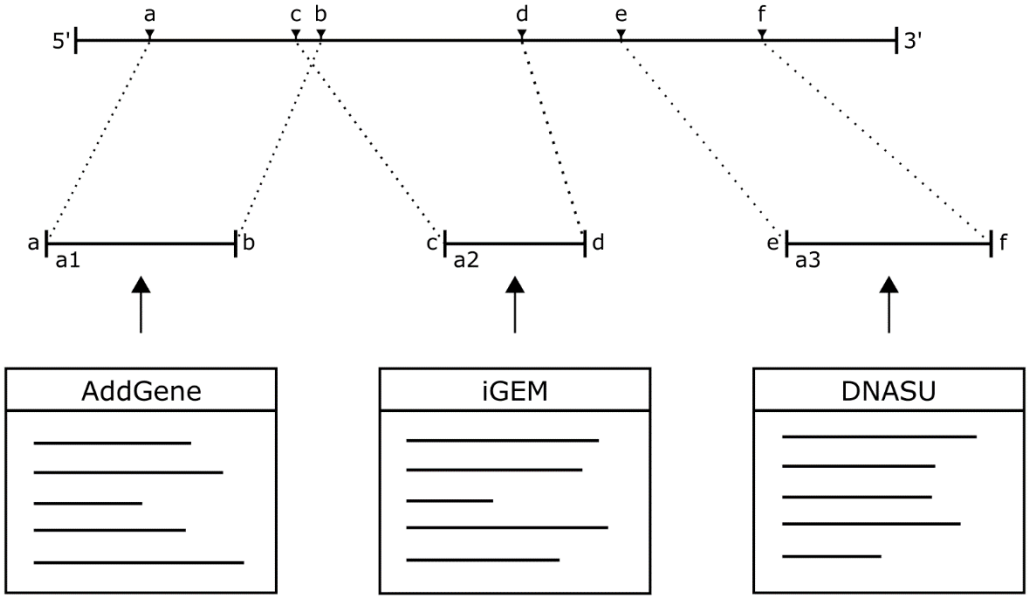


Figure 16: BLAST single-entry query with three hypothetical alignments: a1, a2, and a3. The first (a1) is from AddGene and spans the query sequence within the a and b indices. The second (a2) is from iGEM and spans the query sequence within the c and d indices. The last alignment (a3) is from DNASU and covers the query sequence within the e and f indices

The single-entry and multi-entry queries use BLAST+ for sequence alignment searches of (user-selected) local copies of DNA databases. The databases used in the current version of *Smithy* are the AddGene, iGEM, and DNASU sequence repositories-- with the possibility of extension to lab-specific DNA sequence databases. A new Python class, named *Blaster*, has been developed to manage execution of the BLAST queries using the Biopython wrapper functionality. This functionality of Biopython allows rapid command building, via a Python script, for purpose of local BLAST queries. Given the focus of *Smithy*, only the *blastn* tool is used for performing

nucleotide alignment searches. To run queries, an instance of the *Blaster* class is created, with user parameters provided through the assembly input form.

The *Blaster* class provides an interface used by the *Assembler* classes for several aspects of the assembly design pipeline. First, the building of parameterized queries for each user-selected database, according to the *blastn* format and execution of these queries on each database. Second, the collection and combination of the query results, stored as Biopython *Alignment* objects, and filtered according to user preferences (such as minimum and maximum sequence lengths). Lastly, the combined and filtered query results are provided to the calling script, an instance of the *Assembler* classes—though the *Blaster* functionality can be used independently.

In addition to this interface provided by the Blaster class, there are two different modes of querying available. These query modes are for single-entry or multi-entry FASTA input files. Single- and multi-entry queries differ in their input format and management of results, but both use a FASTA file for the input sequence(s). A single-entry query uses an input file with one single FASTA entry, for the whole insert DNA sequence. The multi-query uses one input file, but with multiple FASTA entries representing the insert's contiguous subsequences, in proper order. These two query types are important, as they dictate how the provided insert is used for performing and managing query results from BLAST.

To begin the procedure, a simple and limited set of inputs is required from the user, gathered in the assembly creation form on the site, for creating a *Blaster* object. These values are the selected databases to query, a FASTA input file for the insert sequence, a Boolean value for design around a single- or multi-entry query and the minimum and maximum fragment sizes (to allow past the query). With these inputs, the local BLAST queries can be performed.

3.2.4.1.1 Query Building

Creating the query commands is a process performed separately for each selected database. Given a list of databases, a set of parameters necessary for blastn queries is created for each database. Values for these parameters include the database's name, a file path for the FASTA query sequence(s), declaring an XML output format, and standard BLAST+ parameters guiding the query algorithms. This procedure of building the parameter sets for the queries is done within a single Python function, which returns these parameters, for use by the Biopython *blastn* command line wrapper functions.

3.2.4.1.2 Single-Entry Query

With the query commands properly created and formatted, a single-entry query can be done. Each BLAST local database for the assembly is queried through the Biopython *blastn* interface. In addition to this interface, for BLAST queries the Biopython library has a separate tool for parsing BLAST results (in XML format) to place them in their Python *Alignment* objects. Each of these *Alignment* objects contains all the information provided by BLAST but accessible through a Python object-oriented datastructure form. This information includes the score, alignment indices for the query and subject sequence, length, and database origin. When the query is completed and parsed, a large list of these alignment results is obtained for each database query.

In brief, a standard BLAST sequence alignment query will find matches to the provided input sequence, within the databases selected for the problem at hand. When a match is found between a sequence in the database on the input sequence, the individual result will contain where on each of these sequences the match was found (e.g., indices *a* and *b* in Figure 16). There are various parameters that can be used to determine matching strength. Mismatches can be tolerated if the user and/or project permits them. Sequence matches also do not necessarily span the entire query

sequence but can be subsequences. Given this, results from these queries will contain sequence alignments of various sizes and, ideally, of 100% homology between input and output sequences. The size variance seen in query results is important to note for the discussion of the solution graph network (Section 3.2.4.2) and is constrained to the user-provided minimum and maximum sized sequences (Section 3.1.2). The graph network will generate all possible combinations of sequence alignments that reconstruct the submitted insert, while compensating for gaps between alignments by adding filler sequences (indicating spans of the insert that will be purchased through direct DNA synthesis). These sequence combinations are then sorted by total BLAST alignment matching, and hence sequences from the databases that cover the most of the target construct are prioritized

After each query is performed and results collected, they must be combined into a single list, for use in latter stages of the design pipeline. To merge the results, each list of *blastn* alignments is combined, to form a single list of all the results for each database. This merged list is then cleared of unusable sequences.

Filtering the query alignments employs three simple attributes: sequence redundancy (according to the query sequence), minimum sequence size, and maximum sequence size. The minimum and maximum size values are provided by the users via the input forms. These values are important, as they signify the range of sequence sizes a user will tolerate for their lab assembly experiment(s). When filtering alignments, sequences are checked for redundancy in the results set, based on their start and end indices on the query sequence. For example, given two sequence alignments that match the same sequence and location on the query, only the first alignment is retained. After redundancy checks, alignments are checked for valid size within the minimum-maximum range specified. Finally, the sequence alignments are sorted in ascending order according to their start

indices on the query sequence. This is done so that during the solution generation protocol sequences at the start of the query input sequence are encountered first.

Once the query results are completed for single-entry queries, they can be used in the creation of the solution graph network. This network is built from these alignments, using their start and end indices, and has all valid candidate solutions for the assembly, embedded within it.

3.2.4.1.3 Multi-Entry Query

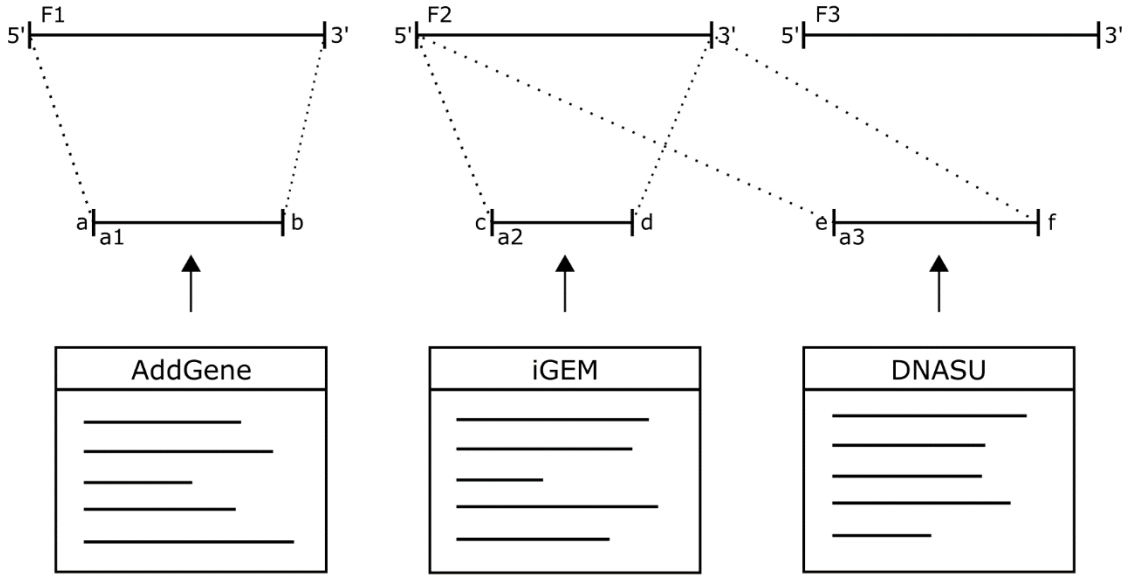


Figure 17: BLAST multi-entry query for three hypothetical insert sequences: F1, F2, and F3. The first insert sequence, F1, has one alignment from AddGene (a1). The second, F2, has two alignments: one from iGEM and another from DNASU (a2 and a3). It is shown in this figure that the alignments span the entire insert sequences.

The multi-entry query procedure requires a different input format and manages results differently than those seen in the single-entry procedure. With the input file containing multiple FASTA entries, each of these subsequence entries are queried individually with *blastn*. The results are then parsed in the same way followed for single-entry results, but with separate lists containing

alignments for each of the subsequence entries. Results from each of the databases are then combined, according to the subsequences of the query. The alignments are filtered based on full matches with the query's subsequences (Figure 17). Alignments that do not match a subsequence 100% are discarded. The crucial difference between the two query modes is the structure of the results with alignments for each insert subsequence stored in separate lists. Since there are strict bounds on the sequences used for composing the assembly insert, variations on sequence alignments cannot be tolerated, and only full matches with the insert subsequences can be used. This changes the solution design process significantly, because multi-entry query assemblies do not use the solution graph network, used by single-entry queries, that assumes BLAST alignments are varied in length and location on the single, full target construct sequence. While the approach to gathering and storing the alignments differs, the class that implements the solution network (*FragmentTree*, Section 3.2.4.2.2) stores BLAST results in a general way that is the same for single- or multi-entry queries. These multi-entry assemblies are advantageous for users who require strict bounds on assembly sequences, such as when Golden Gate scars can only occur at certain junctions between sub-fragments.

3.2.4.2 Solution Graph Network

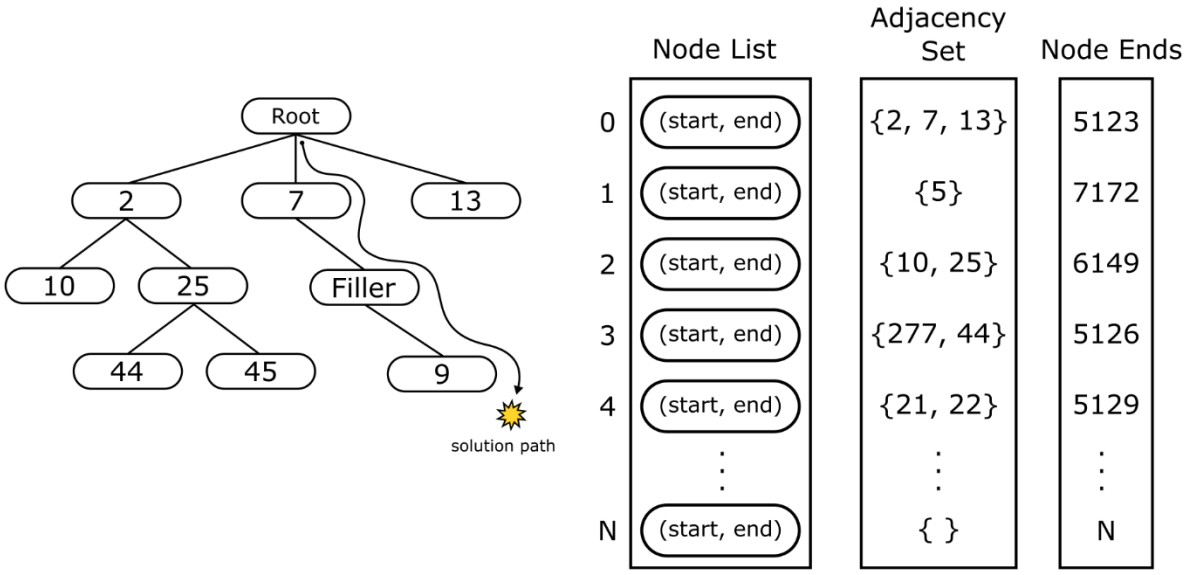


Figure 18: Assembly solution graph network with depth-first-search shown from the root node to node 9 (Root, 7, Filler, and 9). The Root node is an empty node that provides starting points for sequence alignments at the start of the target construct, shown as nodes 2, 7, and 13. Filler nodes contain sequence data not found in BLAST searches (Section 3.2.4.2.6). Node List is the full set of FragmentNodes, containing individual BLAST alignments, used for creating the solution network. Adjacency Set contains all adjacencies between nodes of Node List, using the index of each node in the set. Node Ends is a set of all alignment end indices of entries in Node List for avoiding redundant filler node additions (Section 3.2.4.2.7). The tree network structure seen here is created through a depth-first-search over the Adjacency Set.

The graph network is a critical component of the *Smithy* design process and requires the most computational resources. At the highest level, the network is designed using standard concepts from computer science. A graph network, implemented by the *FragmentTree* class, is comprised of network nodes, implemented by the *FragmentNode* class that contains individual BLAST alignment data used for establishing connections between nodes. Connections between

FragmentNodes in the network are used to find assembly solutions through a depth-first-search process (Figure 18).

Each node is checked with every other node, in pairs, in the *FragmentNode* set used for creating assembly solutions. When a valid link between any two nodes is found a network adjacency is added to the *FragmentTree* adjacency set (Figure 19, Pseudocode 1). Adjacencies are set between nodes when their sequences are able to reconstruct portions of the insert query in correct sequential order. With all adjacencies found between nodes, a vast graph network is created. Traversing this tree in a depth-first-search fashion will allow all valid solutions contained within the tree to be found (Figure 18, Pseudocode 2).

As mentioned with the BLAST single- and multi-entry modes, the *FragmentTree* class has two types of control flow and logic for these two query types. These two query methods have different approaches to utilizing sequences acquired from the various databases. The single-entry queries require the creation of the *FragmentTree* network, while the multi-entry queries do not. For the multi-entry query mode, the FragmentTree is still used, but only for storage and retrieval of alignments. The multi-entry queries have lower complexity than single-entry query solution building, when considering the requirements for obtaining alignment results, and using them for assembly solutions. A graph network is not created in this method, and instead query results for each insert sub-sequence are collected into individual lists, with the same format as that of a multi-sequence FASTA input file. Then, solution sequences are retrieved from each of these lists in the same order, retrieving the alignments, so a correctly ordered solution can be generated.

3.2.4.2.1 FragmentNode

The *FragmentNode* class is responsible for containing information about a single BLAST sequence alignment. These data are limited and used for access to essential information on a particular alignment, when constructing the *FragmentTree* network. These data include:

- **data:** a variable containing the actual Biopython BLAST Alignment object of a query result
- **start:** the start index of the alignment on the query sequence
- **end:** the end index of the alignment on a query sequence
- **score:** the BLAST score value, a one-to-one scoring for the number of nucleotide matches in an alignment (100% matches used) (this value can be tuned with advanced query parameters)
- **node_id:** a simple name for identifying the given node
- **db:** the name of the database from which a given node originates
- **query_seq:** BLAST query sequence for the FragmentNode
- **subject_seq:** BLAST subject sequence for the FragmentNode
- **coordinates:** a Python tuple of the *FragmentNode's* start and end values

Further information about the BLAST alignment can be accessed through the node's *data* attribute. The *FragmentNode* class, as it currently stands, contains little functionality apart from data fetching of the node's data.

3.2.4.2.2 FragmentTree

The *FragmentTree* class is responsible for building assembly solution candidates from the BLAST insert query results, which are contained in the *FragmentNodes*. This class utilizes a few input

parameters from the user assembly form to guide the solution tree network building algorithms. These parameters are:

- query sequence, being the insert's full DNA sequence
- length of the query sequence
- minimum nucleotide size of synthetic fragments for assembly designs
- maximum nucleotide size of synthetic fragments for assembly designs
- assembly costs
- assembly experiment probabilities of success

The query length is used for the solution generation stage, which checks the end of each fragment set acquired by the network for its completion of the insert query. Based on the last node's index, if the solution ends before the end of the query sequence, then a filler sequence is added to the solution to complete the query sequence (Section 3.2.4.2.8). It is important to note that filler sequences are only added to solution ends if the length of the missing segment fulfills the minimum and maximum synthetic DNA size parameters, provided by the user.

In order to fully describe the functionality of the *FragmentTree* class, it helps to list out its class attributes. These are crucial to building and reporting solutions, and they are:

- **node_list:** a list of FragmentNodes for building a solution tree
- **min_synth:** minimum nucleotide size of synthetic fragments for assembly designs
- **max_synth:** maximum nucleotide size of synthetic fragments for assembly designs
- **adjacency_set:** a set of adjacencies between FragmentNodes for building insert solutions
- **node_end:** a set of FragmentNode ends used for avoiding redundant filler nodes
- **visited:** a list of visited nodes to use when building solution

- **solutions:** a list of lists, where each list is a collection of indices of FragmentNodes in the nodes list that make an ordered solution for the assembly
- **scores:** a list of scores for each respective solution. Indices of the scores list correspond to those of the solutions list.
- **query_len:** length of the query sequence
- **query:** the insert's full DNA sequence
- **max:** the maximum total score of all solutions
- **part_costs:** list of nucleotide costs for part size groups
- **pcr_polymerase_cost:** per-reaction cost for PCR polymerase
- **polymerase_cost:** per-reaction cost for assembly polymerase
- **exonuclease_cost:** per-reaction cost for assembly exonuclease
- **ligase_cost:** per-reaction cost for assembly ligase
- **biobricks_digest_cost:** per-reaction cost for BioBricks assembly digestion
- **restriction_enzyme_cost:** per-reaction cost for Golden Gate assembly Type2S RE
- **parts_preference:** user parts count preference
- **cost_preference:** user cost preference
- **pcr_ps:** PCR probability of success
- **gibson_ps:** Gibson assembly probability of success
- **goldengate_ps:** Golden Gate assembly probability of success
- **ligation_ps:** Ligation reaction probability of success
- **biobricks_digest_ps:** BioBricks digestion probability of success
- **slic_exonuclease_ps:** SLIC exonuclease reaction probability of success

3.2.4.2.3 Network Building

To build the fragment network several procedures take place. First, the BLAST alignments are used to build the *node_list* and *node_end* lists for the *FragmentTree* instance. Then the node list is extensively used to determine adjacencies between every node in the list and add filler nodes (Figure 19), indicating synthetic sequences for the assembly, where and when needed. Adjacencies are stored within the *adjacency_set* and are represented by indices of nodes within *node_list*. With adjacencies found and collected a depth-first-search is performed on the adjacency set to find solutions and store them with their individual scores in *solutions* (Pseudocode 2). This solution search builds a tree network representing all possible unique solutions, which are contiguous sets of insert subsequences (Figure 18).

With the solution search complete, the max solution score, as determined by total database nucleotide matches, is found by accessing the maximum score value in the *solutions* list. Solutions are then sorted, in descending order, according to their scores. This ensures that the better solutions are used first during assembly design. Hence, solutions that do not complete the submitted insert sequence are completed, using filler fragments, respecting the user-provided maximum and minimum synthetic sequence values. Finally, those solutions that fully represent the insert sequence are retained; all others are deleted.

3.2.4.2.4 Node Adjacency

Here a single function is described, *build_edge_set()*, which is the class's core algorithm for determining the network tree adjacencies.

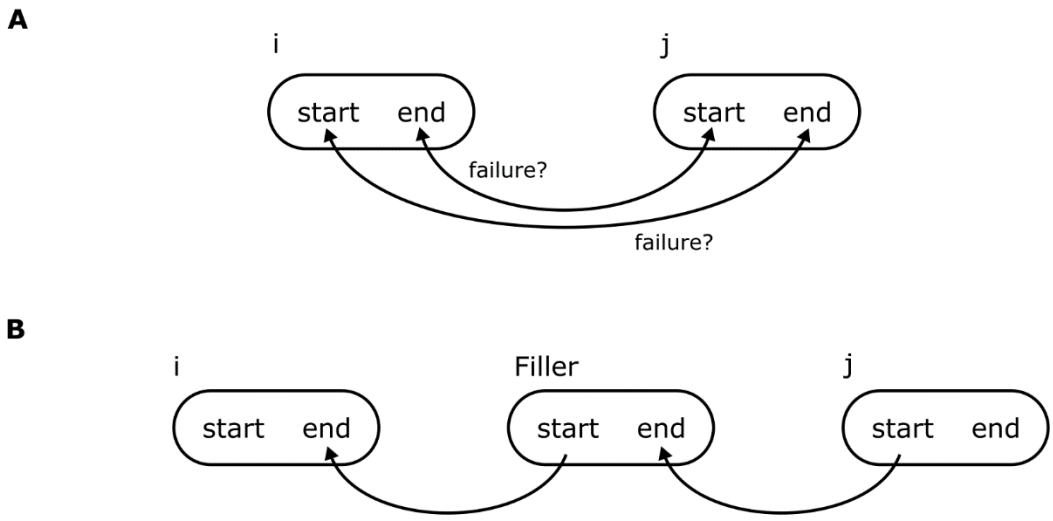


Figure 19: Determining node adjacencies with two FragmentNodes i and j. (A) representation of the start and end indices of each node being checked for a filler node gap (Pseudocode 1). (B) An example of inserting a filler node between nodes i and j.

Pseudocode 1: Node Adjacency

Inputs:

NONE

Variables:

node_A: The first node in the pair for comparison

node_B: The second node in the pair for comparison

start_index: The node's starting index on the query sequence

end_index: The node's ending index on the query sequence

min_synth_seq: The length of the smallest tolerable synthetic sequence

max_synth_seq: The length of the largest tolerable synthetic sequence

```
1   build_edge_set():
2       for every i, j combination in node_list
3           get node_A start_index, end_index from node_list at index i
```

```
4           get node_B start_index, end_index from node_list at index j
5
6           if node_B start_index > node_A end_index + min_synth_seq
7               and node_B start_index < node_A end_index + max_synth_seq
8                   and node_B start_index is not in node_ends
9               create filler node using node_A end_index + 1 and node_B start_index - 1
10              add node_B start_index – 1 to node_ends
11
12
13          if node_A start_index > node_B end_index + min_synth_seq
14              and node_A start_index < node_B end_index + max_synth_seq
15                  and node_A start_index is not in node_ends
16              create filler node using node_B end_index + 1 and node_A start_index - 1
17              add node_A's start – 1 to node_ends
18          endif
19      endfor
20
21      for every i, j combination in node_list
22          get node_A start_index, end_index from node_list at index i
23          get node_B start_index, end_index from node_list at index j
24
25          if node_B start_index equals node_A end_index
26              and node_A does not already have an adjacency to node_B
27              add adjacency from node_A to node_B
28          endif
29          if node_A start_index equals node_B end_index
30              and node_B does not already have an adjacency to node_A
31              add adjacency from node_B to node_A
32          endif
33      endfor
```

The algorithm begins by finding all valid filler nodes to create and add to *node_list*. Given the node list, all pairwise combinations of nodes are inspected for possible filler node gaps (Figure 19-A, Pseudocode 1 lines 2-19). If a valid filler node is added to *node_list*, this new node represents the sequence between the original node pair. Each pair is tested against every pair. Given two nodes, A and B, node A is tested against node B and vice versa. The conditional logic for these two checks is similar, and the references are simply swapped.

Next, the algorithm sets node adjacencies. With all valid and necessary filler nodes added, adjacencies between all possible pairwise combinations of the fragments in *node_list* can be found

(Figure 19-B, Pseudocode 1 lines 21-33). Similar to the filler node steps, each pair has two *FragmentNodes*, A and B, that are tested against each other. With two nodes A and B, if B's start is equal to A's end without being a repeated adjacency add an adjacency from A to B.

With these two procedures complete, all filler nodes for the network are created and all possible valid adjacencies between nodes would have been determined.

3.2.4.2.5 Normal Node

A typical FragmentNode contains a small but crucial set of attributes necessary for building the network. These values are used for identification, adjacency searching, database origin, BLAST score value, and the complete BLAST result information. Specifically, this information is:

- the BLAST alignment data, stored in a Biopython *Alignment*
- start index on the query sequence
- end index on the query sequence
- score, determined by the BLAST alignment match
- descriptive title and id

3.2.4.2.6 Filler Node

Filler nodes contain similar data to normal nodes, though it is more limited. The creation and addition of filler nodes to *node_list* is straight forward. First, an empty Biopython *Alignment* instance is created, and then it is filled with the proper filler node values: *start* and *end*. The data stored in this alignment object are:

- descriptive title and id, beginning with "Synthetic"
- length
- score, which is 0 since it is compensating the lack of a BLAST alignment

- insert subsequence extracted from the insert query sequence using *start* and *end* as index bounds
- start index
- end index

Once the *Alignment* instance is completed, it is stored in a new *FragmentNode's data* attribute, and this new node is added to the *node_list*. The attributes used to create the *FragmentNode* for a filler node are similar to those of normal nodes, but particular to the filler at hand.

3.2.4.2.7 Solution Searching

Pseudocode 2: Solution Searching

Inputs:

node: The starting node index of *node_list* for the depth-first-search

path: The current list of *node_list* indices for a solution path in the *FragmentTree*

Variables:

visited: A list of visited nodes to use when building solution

adjacency_set: A set of adjacencies between *FragmentNodes* for building insert solutions

scores: A list of scores for each respective solution. Indices of the scores list correspond to those of the solutions list

solutions: A list of lists, where each list is a collection of indices of *FragmentNodes* in the nodes list that make an ordered solution for the assembly

```
1   dfs(node, path)
2       visited[node] ← True
3       append node to path
4
5       if empty adjacency_set for node
6           score ← solution score for current path
```

```
7           append score to scores
8           p ← copy path
9           prepend score to p
10          add p to solutions
11      else
12          for each i in adjacency_set at node
13              if visited[i] is False
14                  dfs(i, path)
15              endif
16          endfor
17
18      pop last node from path
19      visited[node] ← False
```

The *adjacency_set* is an abstract representation of the nodes in *node_list*. It does not contain any node data, just integers of node indices of *node_list*, stored in sets for each node's one-to-one adjacency mapping. This approach reduces space and time for building the solution tree, as the actual node data is separated from the network building logic. In the initial stages of creating this approach, the entire *FragmentNode* object was used within the network instead of index-based abstract representation. Using full *FragmentNodes* in this way caused their data to be copied whenever multiple adjacencies existed for a single node, severely consuming memory resources and causing the code to run unacceptably slow.

The networks created can be large and complex, containing hundreds of solution paths. The search for solutions uses a depth-first-search (DFS) algorithm that visits each path of adjacencies in the network, down to its leaf nodes (*FragmentNodes* with no remaining adjacencies). Each path is composed of the node indices gathered from traversing the adjacencies. When a node is visited, its index is appended to the current path. When a leaf node is found, the path to the leaf is added to *solutions*, and the solution's score is added to the *scores* list (Pseudocode 2 lines 5-10). The score value is calculated by summing the number of parts and probability-adjusted cost of the assembly for the solution, each weighted by coefficients provided by users that express their weight of

consideration for these components of the calculation. In the input forms, users see a *cost preference* and *parts count preference* fields to enter these weightings. The cost value is calculated by summing the PCR amplification costs for each part with the cost of the assembly experiments on a per-reaction cost basis, all adjusted by their probabilities of success which are also user-provided values. For example, the score for a Gibson assembly solution candidate would sum the number of parts in the solution, scaled by the user's parts count preference, with the sum of the PCR amplification cost with the one-pot assembly cost, scaled by the user's cost preference. In sum, the solutions are scored by their parts count and total assembly cost, scaled by the users' weighting of these attributes of the calculation. Solution searching is complete when all nodes have had their adjacencies exhausted by the DFS search.

After the solution search, the maximum score is found from *scores*, and the *solutions* list is sorted in descending order, according to their individual scores. With these two procedures complete, the solutions can be inspected to determine completion (or not).

3.2.4.2.8 Completing Solutions

During the solution completion procedure, each solution's end index, found through its terminal node, is used to determine if it can be completed according to user specifications. Every solution of the Blaster instance is tested by taking their end index and subtracting it from the length of the query sequence. If this difference falls in the user's synthetic sequence minimum-maximum range, a filler node is added to the end of the solution. In this case, the filler node's start and end indices would be the original end index and the query sequence length, respectively. However, if a solution fails, this test its index in *solutions* is added to a deletion list, which contains all the indices to be removed from *solutions*. This is done after all the (candidate) solutions have been vetted.

3.2.4.2.9 Multi-Entry Query Solutions

Solutions for multi-entry queries are implemented quite differently from those of single-entry queries. This is because the *blastn* results are structured differently and thus require different handling.

First, the blast results are organized in a list of lists called *multi_query_node_list*. Each list here contains the alignments for each subsequence in the FASTA input file (provided by the user). If there happened to be no results for a given subsequence, then a filler node is created (see Completing Solutions). Otherwise, each alignment is used to create a *FragmentNode* to hold its query result information.

Next, a large set of solutions are generated using *multi_query_node_list* with a size determined by an inputted maximum size value. No *FragmentTree* is created for multi-entry queries and instead, solutions are generated by sampling nodes from each list in the *multi_query_node_list*. This is done to provide an alignment for each subsequence's portion of the query. These solutions have the same form as that of single-entry queries, but with each index in the solution indicating a node in the respective *multi_query_node_list* sub-list. Additionally, solutions here are added to a different list in the Blaster instance, this being *multi_query_solutions*. When solutions are sampled, they appear like those of single-entry queries. Solutions here form a compact lists of node indices, which are used to obtain alignment data.

***

The Blaster class is designed in such a way that the use of single- or multi-entry query solution building produces solutions in identical format. This is useful because the higher-level scripts, in the case of *Smithy* the use of the Blaster in the *Assembler* objects, the distinction between these

two solutions generating modes do not affect the way solutions appear. Thus, both types of solutions can be used interchangeably without downstream barriers in the assembly solution design process. Once candidate solutions for an assembly have been generated, through either of the two modes, primer design can take place. For primer design, a single solution is retrieved from the *FragmentTree* and used to obtain the sequences from each *FragmentNode* in the solution. These sequences are then stored in Pydna *Dseqrecord* objects, which are convenient for the primer design procedures, as they make extensive use of Pydna's features[27].

The full set of solutions are not outputted to a script, but rather individual solutions are retrieved and used for assembly design within a cloning method-specific *Assembler* class instance, such as *GibsonAssembler* or *GoldenGateAssembler*. The assembler will first design primer complementary sequences for each fragment in a solution and then add the appropriate extension sequences to these primers that facilitate proper assembly. The method for generating the complementary primer sequences is the same for all assembler classes, but the extension sequence design is specific for each class. For example, the primer extension design for a Gibson assembly is quite different than that of Golden Gate assembly.

3.2.4.3 Primer Complements

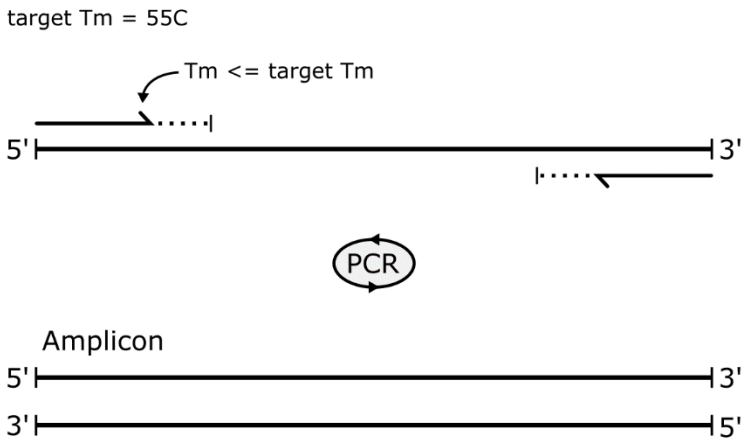


Figure 20: Primer complements design for a single fragment using a target T_m of 55°C. The forward and reverse primers are shown to be designed up to the target T_m. Then the resulting amplicon dsDNA sequence from PCR with these primers is drawn.

Pseudocode 3: Primer Complement Design

Inputs:

fragments: A list of the assembly insert subsequences

backbone: The assembly backbone sequence

Variables:

pcr_fragments: A list of the PCR amplified assembly insert subsequences

pcr_backbone: The PCR amplified assembly backbone

```
primer_complement(fragments, backbone)
    pcr_fragments ← call Pydna primer_design() for each fragment in fragments using
        melting temperature and custom melting temperature function
    pcr_backbone ← call Pydna primer_design() for backbone using melting temperature
        and custom melting temperature function
    return pcr_fragments, pcr_backbone
```

This initial step in the primer design process creates non-extension primers for all the fragments and the backbone of an assembly. The procedure uses the Pydna *primer_design* function[27], with user provided target melting temperature, and a customized melting temperature function, which is initialized with user parameters. The target melting temperature is a single Celsius value, and the parameters used for the custom melting temperature function are: DNA oligo concentration, monovalent ion concentration, divalent ion concentration, and dNTP concentration. Using this

primer design function will generate forward and reverse PCR primers, with the target melting temperature (Figure 20). When *primer_design* is used it will output a Pydna *Amplicon* object containing the simulated PCR amplified fragment, with its forward and revers primers, as Pydna *Primer* class instances[27]. When each fragment's and backbone's simulated amplicons are generated (Pseudocode 3), they are used in the next stage of the assembly design, to create primer extension sequences particular to the chosen cloning method. The collection of amplicons generated at this step serve as a strong base for the following primer extension design.

3.2.4.4 Primer Overlap Extensions

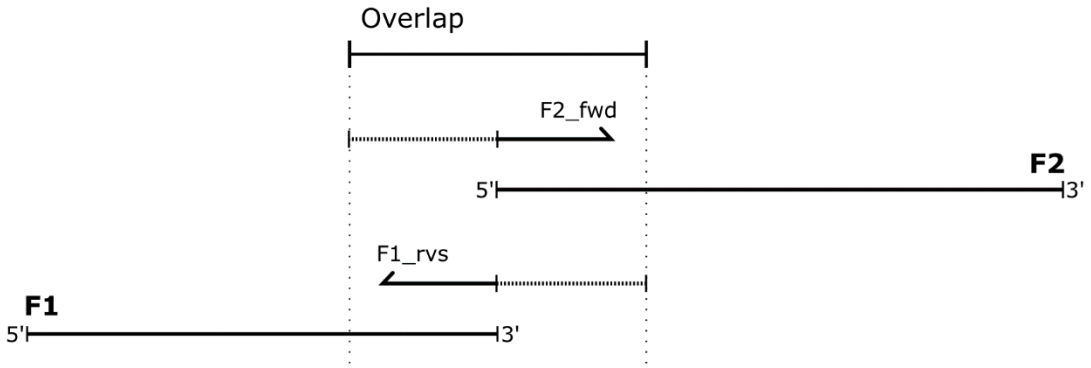


Figure 21: Primer overlap extensions design with the overlap length spanning the junctions between the two fragments F1 and F2 (Pseudocode 4). The overlap extensions are added to these sequences by extending the forward primer of F2 (F2_fwd) and the reverse primer of F1 (F1_rvs).

Pseudocode 4: Overlap Extensions

Inputs:

fragments_pcr: A list of Pydna *Amplicons* used for a given assembly solution

backbone_pcr: The backbone *Amplicon* for a given assembly

Variables:

Forward_overlap: The overlap sequence for the forward primer extension

reverse_overlap: The overlap sequence for the reverse primer extension

```
1   overlap_extensions(fragments_pcr, backbone_pcr)
2       assembly ← empty list
3       append backbone_pcr to fragments_pcr
4       end_index ← length of fragments_pcr – 1
5       overlap ← total_overlap / 2
6
7       for i, amplicon in enumerated fragments_pcr
8           if i equals 0
9               forward_overlap ← last sequences of watson strand of
10                  fragments_pcr[end_index] with length overlap
11          else
12              forward_overlap ← last sequences of watson strand of
13                  fragments_pcr[i - 1] with length overlap
14          endif
15
16          if i equals end_index
17              reverse_overlap ← last sequences of crick strand of
18                  fragments_pcr[0] with length overlap
19          else
20              reverse_overlap ← last sequences of crick strand of
21                  fragments_pcr[i + 1] with length overlap
22          endif
23
24          extended_amplicon ← call add_extensions with forward_overlap,
25              reverse_overlap, and amplicon
26          append extended_amplicon to assembly
27      endfor
```

The overlap extension based cloning methods currently supported by *Smithy* are Gibson, PCR-SOE, and SLIC assemblies. Since these methods share similar sequence homology overlap based assembly methodologies, their primer extension design is implemented within the *OverlapExtensionAssembler* class and shared across these three methods' assembler class implementations. These specific class implementations are *GibsonAssembler*, *SLICAssembler*, and *PCRAssembler*. They are defined uniquely by inheriting from *OverlapExtensionAssembler*, but so far only specify default parameters unique to the individual methodologies. For the purpose of primer overlap extensions the parameter of interest is *overlap*. The default values for *overlap* differ across the three overlap extension assemblers, but its purpose is singular: to specify the amount of sequence homology shared by assembly fragments. This value is fundamental for designing the overlap extensions.

To generate the overlap extensions, each amplicon has sequences of its neighbors at their junctions added to its ends (Figure 21). Half of the user's overlap value is used to obtain sequences so that the resulting overlap length at any junction totals the length defined. So, from a given amplicon's left neighbor half of *overlap* is used to copy sequences from this neighbor's 3' end and add them to the amplicon's forward primer. Similarly, with the right neighbor half of *overlap* is used to copy sequences from this neighbor's 5' end and add them to the amplicon's reverse primer. Next, the current amplicon is used for re-amplification using Pydna, using the new extension sequences. As a result, a new *Amplicon* is created where the sequence has been extended. This procedure occurs for every amplicon in the assembly solution as well as the backbone. Each new amplicon is added to a new Python list and hence, returned to the calling script, to conclude assembly design (Pseudocode 4).

3.2.4.5 Primer Golden Gate Extensions

Depending on the user's selection for non-scar-less (default) (Figure 22) or scar-less (Figure 23) Golden Gate assembly, one of two routes for primer extension design is chosen. Another aspect of the design controlled by the user is the overhangs set used. There are four sets to choose from and they differ in number of overhangs and their correlated fidelity (for proper ligation during assembly). These overhang sets were obtained from an overhang sequence profiling study conducted by Potapov et al[47]. By default, these Golden Gate extensions are designed for BsaI restriction enzyme assembly.

3.2.4.5.1 Default Non-Scar-less Extensions

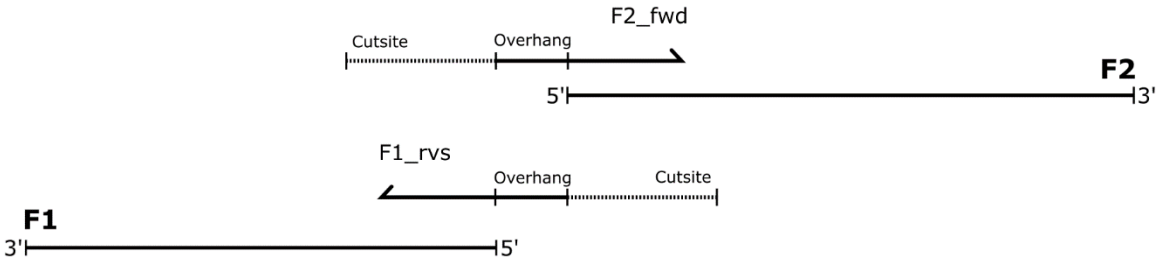


Figure 22: Primer Golden Gate non-scar-less extensions design for two fragments, F1 and F2.

Pseudocode 5: Non-Scar-less Golden Gate Extensions

Inputs:

fragments_pcr: A list of Pydna *Amplicons* used for a given assembly solution

backbone_pcr: The backbone *Amplicon* for a given assembly

Variables:

forward_overhang: The overhang sequence for the forward primer extension

reverse_overhang: The overhang sequence for the forward primer extension

forward_site: The full cut-site with overhang for the forward primer

reverse_site: The full cut-site with overhang for the reverse primer

extended_amplicon: The full PCR extended amplicon ready for Golden Gate assembly

```
goldengate_extensions(fragments_pcr, backbone_pcr)
    assembly ← empty list
    append backbone_pcr to fragments_pcr
    set_size ← length of fragments_pcr

    for i, amplicon in enumerated fragments_pcr
        forward_overhang ← overhang_set[i]
       if i equals set_size – 1
           reverse_overhang ← reverse complement of overhang_set[0]
       else
           reverse_overhang ← reverse complement of overhang_set[i + 1]
       endif

       forward_site ← cut-site + ‘n’ + forward_overhang
       reverse_site ← cut-site + ‘n’ + reverse_overhang
       extended_amplicon ← call add_cut-sites with forward_site, reverse_site,
           and amplicon

       append extended_amplicon to assembly
    endfor
```

As with the overlap extensions procedure the fragments and backbone are iterated over once in a circular fashion. For each new junction between parts, the overhang used is taken from the overhangs set selected. An overhang sequence is selected for the forward primer from the set, then an overhang sequence, for the reverse primer, is selected from the next sequence in the set. For the last part in the solution, its reverse primer overhang is the first overhang in the set, so that circular plasmid assembly is achieved. Finally, the fragments and backbone have their forward and reverse

primer extensions added to them through reamplification by simulated PCR (courtesy of Pydna) (Pseudocode 5).

3.2.4.5.2 Scar-less Extensions

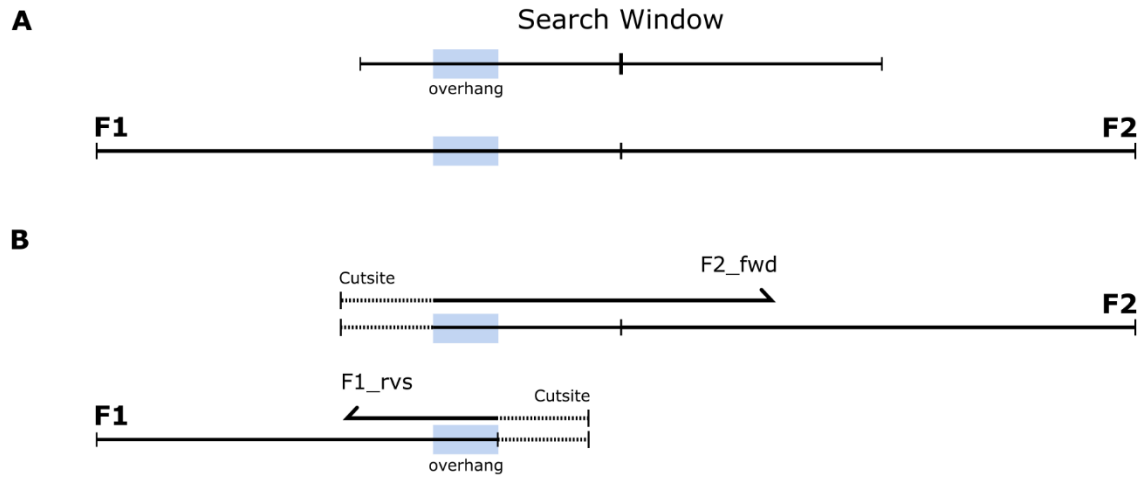


Figure 23: Primer Golden Gate scar-less extensions design for two fragments, F1 and F2, following Pseudocode 6. (A) The overhang is shown to be found within the search window on F1. (B) Adjustments of the two fragments is shown through extension of F2 with F1 sequences (F2_fwd) and the shortening of F1 equivalent to the F2 extension (F1_rvs).

Designing Golden Gate scar-less extensions requires more control over the primer design by *GoldenGateAssembler* and adds new functionality. The new functionality is independent primer complement and extension design for facilitation of scar-less Golden Gate assembly. Most of the critical work for this design approach is handled by the new primer complement design procedure (Pseudocode 6).

Pseudocode 6: Scar-less Golden Gate Complements

Inputs:

fragments: A list of fragment sequences for the assembly

backbone: The backbone sequence

interval: Half the length of the search window, the amount of each sequence used for finding scar-less overhangs

Variables:

pcr_intervals: A list of lists, each list containing the updated start and end locations for PCR for each sequence of the assembly

extensions: A list of lists, each list containing forward and reverse primer extension sequences for PCR for each sequence of the assembly

i: The index in fragments for the current fragment to inspect

i_next: The index in fragments for the current fragment to inspect

test_sequence: A search window of nucleotide sequences within which an overhang from *overhang_set* is attempted to be found

test_overhang: An overhang found in *test_sequence* to check for being in the *overhang_set*

overhang_start: The start index for the overhang

overhang_end: The end index for the overhang

```
1   goldengate_scar-less_complements(fragments, backbone, interval)
2       append backbone to fragments
3       pcr_intervals ← list of [0, record length] for every record in fragments
4       overhang_set ← overhang sequences from assembler
5       extensions ← list of lists of two empty strings for each fragment
6
7       for i in range of length of fragments
8           found ← False
9           if i equals length of fragments – 1
10              i_next ← 0
11          else
12              i_next ← i + 1
```

```
        endif
        test_sequence ← subsequence of last nucleotides in watson strand of
            fragments[i] with length interval + subsequence of first nucleotides in
            watson strand of fragments[i_next] with length interval

        for j in range of length of test_sequence
            test_overhang ← four nucleotides from test_sequence starting at j

            if test_overhang in overhang_set
                found ← True
                remove test_overhang from overhang_set
                break
            endif

        if not found
            j ← interval – 4
        endif

        if j is greater than interval – 1
            overhang_start ← j – interval
            overhang_end ← (j + 4) – interval

            pcr_intervals[i_next][0] ← overhang_start

            extensions[i][1] ← BsaI cut-site + 'N' + last sequences of crick strand
                of fragments[i_next] of length overhang_end
            extensions[i_next][0] ← BsaI cut-site + 'N'
        elif j equals interval -1
            overhang_start ← -(interval – j)
            overhang_end ← -(interval – (j + 4))

```

```
44              pcr_intervals[i_next][0] ← overhang_end
45
46              extensions[i][1] ← BsaI cut-site + ‘N’ + last sequences of crick strand
47                  of fragments[i_next] of length overhang_end
48              extensions[i_next][0] ← BsaI cut-site + ‘N’
49          else
50              overhang_start ← -(interval - j)
51              overhang_end ← -(interval – (j + 4))
52
53              if overhang_end not 0
54                  pcr_intervals[i][1] ← overhang_end
55              endif
56
57              extensions[i][1] ← BsaI cut-site + ‘N’
58              extensions[i_next][0] ← BsaI cut-site + ‘N’ + first sequences of watson
59                  strand of fragments[i] of length overhang_start
60          endif
61      endfor
62      for each fragment in fragments
63          call primer_design using new pcr_intervals for fragment, target_tm, and
64              tm_function
65      endfor
```

Every neighboring pair of solution sequences uses a sequence window as a search space for 4nt overhang sequences. An equal number of nucleotides is taken from each of the sequences at the junction, summing up to a search space size parameter, which defaults to 40 nt (Pseudocode 6 lines 14-16). When the search window is obtained, it is iterated over in 4nt groups to check each of these subsequences for a match in the user’s selected overhang set. When an overhang is found in either the current fragment or its right neighbor, the indices of the overhang are recorded (Pseudocode 6

lines 18-25). If no overhang sequence is found, the search defaults to using the last four bases of the current fragment for the overhang sequence (Pseudocode 6 lines 27-29). These indices are used for extracting sequences that extend either the current or neighboring fragment with the overhang at the end (Pseudocode 6 lines 31-61). Indices of the overhangs are also used for properly amplifying each fragment if it must be trimmed to allow for the neighboring fragment to extend into the trimmed sequences, while the removed sequence will be added to the neighbor. This will possibly lead to the found overhang within the search window to shift the junction between the two sequences. (Figure 23, Pseudocode 6 lines 62-65). Designing overhangs in this way ensures that 4 nt scars are not added at part junctions which may result in mutant proteins, due to (unwanted) additional nucleotides within exons of the transcript.

Pseudocode 7: Scar-less Golden Gate Extensions

Inputs:

pcr_fragments: A list of Pydna *Amplicons* used for a given assembly solution that are not extended yet and adjusted for scar-less assembly

extensions: A list of lists for each assembly fragment's scar-less Golden Gate extension sequences: one for the forward primer extension, the other for the reverse primer extension

Variables:

extended_amplicon: A list of scar-less Golden Gate assembly prepared PCR amplicons

```
1   goldengate_scar-less_extensions(pcr_fragments, extensions)
2       assembly ← empty list
3
4       for extension, amplicon in zipped (extensions, pcr_fragments)
5           extended_amplicon ← call add_cut-sites with extension[0], extension[1], and
6               amplicon
```

```
7           append extended amplicon to assembly
8       endfor
9
10      return assembly
```

Once the overhang search is complete, each fragment is re-amplified *in silico* with the edits determined by the search, to provide new Pydna *Amplicons* and *Primers* for each assembly fragment. Additionally, given the new junctions, the sequences are then extended with the Type2S RE cut-site, the default case being for BsaI (*GGTCTC*). The extensions will either be just the RE cut-site or the shifted subsequence at the junction plus the RE cut-site. As with the other assemblers' extension methods, each assembly part will be re-amplified so all the parts are complete hence ready for assembly (Pseudocode 7).

3.2.4.6 Primer BioBricks Extensions

Assembly extensions from the *BioBricksAssembler* are created in a simple way, following the method's protocols. The BioBricks methods defines specific 5' and 3' additions to a given sequence, named prefixes and suffixes, respectively. The prefix sequence has two options, one for coding sequences and one for any other part. The coding sequence prefix is *gaattcgcggccgcttctag*, and the default prefix is *gaattcgcggccgcttctagag*. Prefix sequences contain EcoRI and XbaI restriction endonuclease sites. The suffix is constant for any assembly part, and its sequence is *ctgcagcggccgctactagta*. Suffixes contain SpeI and PstI restriction endonuclease sites. Every assembly sequence in a solution will have these prefix and suffix sequences[10,11].

Generating these BioBricks extensions involves iterating over each amplicon of the assembly's solution, adding either the coding sequence prefix or the standard prefix, adding the suffix, and then reamplifying the amplicon so it obtains the extensions. As with the other extension

procedures, this occurs for every amplicon in the assembly solution and the backbone. The rest of the assembly design process occurs after these extensions are added.

3.2.4.7 Assembler Design Function

3.2.4.7.1 Overview

Each *Assembler* subclass has a *design* function, for executing a complete design procedure, given a provided solution. This function is a collection of method calls and logic, which implement these algorithms (Section 3.2.4) of the different cloning methods. Each of these algorithms have been implemented as single functions within their *Assembler* classes. As such, each function is a container for each of these algorithms to be used for a single solution. Within that function, a solution is fetched from the *FragmentTree*, and primer complements are created, assembly primer extensions are designed for the given cloning methodology, annotations are added for each part, and analytical thermodynamic calculations are performed using Primer3[39] for every primer (Pseudocode 8 and 9). Golden Gate assemblies differ only by a Boolean control variable, indicating if scar-less assembly design is required. If true, the function will utilize the Golden Gate scar-less primer complement and extension design functions (Pseudocode 9 lines 10-14). Upon completion, a completed assembly solution of parts and primers is returned to be saved and presented to the user through the Django framework. All of the solution components are translated to SQL database models, to be stored, and are then retrieved for detailed viewing on the *Smithy* website.

Pseudocode 8: Assembler Design Function – Gibson Cloning

Inputs:

solution: The solution in *solution_tree* of the assembler to design

Variables:

fragments: List of sequences for the assembly solution that comprise the construct

nodes: List of *FragmentNodes* for each sequence in sequences that contain BLAST alignment data

pcr_fragments: List of PCR amplicons of fragments ready for assembly extensions

pcr_backbone: PCR amplified backbone ready for assembly extensions

assembly: The completed assembly solution

```
1   design(solution)
2       if multi_query
3           fragments ← call get_multi_query_solution with solution
4           nodes ← call solution_tree multi_query_solution_nodes function with solution
5       else
6           fragments ← call get_solution with solution
7           nodes ← call solution_tree solution_nodes function with solution
8       endif
9
10      pcr_fragments, pcr_backbone ← call primer_complement with fragments and backbone
11      assembly ← call primer_extension with pcr_fragments and pcr_backbone
12      assembly ← call annotations with assembly and nodes
13      assembly ← call assembly_thermo with assembly, monovalent ion concentration,
14          divalent ion concentration, DNA oligo concentration, and Tm function
15
16      return assembly and nodes
```

3.2.4.7.2 Solution Network

To begin, a solution's sequences and *FragmentNodes* are extracted from the solution tree (Pseudocode 8 lines 2-8, Pseudocode 9 lines 2-8). The solution sequences are used during the

primer design procedures using Pydna[27]. The nodes are for extracting BLAST data (database origin, start index, end index, etc.) from the sequence alignments and adding this data to annotations for each part after assembly design is complete. Both of these sets of assembly solution data are used later, when translating to Django webapp data.

3.2.4.7.3 Primer Complements

Using the solution and backbone sequences, primer complements are designed and created (Pseudocode 8 line 10, Pseudocode 9 lines 11 or 16). These complements must satisfy the user-supplied experimental parameter for melting temperature(s). Here the Pydna melting temperature calculation function is used. Thus for each sequence, non-extension primers are created and *in silico* PCRs are performed to provide amplicons, which are used in latter stages of extension primer design.

Pseudocode 9: Assembler Design Function – Golden Gate Cloning

Inputs:

solution: The solution in *solution_tree* of the assembler to design

Variables:

scar-less: Boolean control variable whether to run scar-less design

extensions: Scar-less assembly extension sequences

fragments: List of sequences for the assembly solution that comprise the construct

nodes: List of *FragmentNodes* for each sequence in sequences that contain BLAST alignment data

pcr_fragments: List of PCR amplicons of fragments ready for assembly extensions

pcr_backbone: PCR amplified backbone ready for assembly extensions

assembly: The completed assembly solution

```
design(solution)
    if multi_query
        fragments ← call get_multi_query_solution with solution
        nodes ← call solution_tree multi_query_solution_nodes function with solution
    else
        fragments ← call get_solution with solution
        nodes ← call solution_tree solution_nodes function with solution
    endif

    if scar-less
        pcr_fragments, extensions ← call primer_complement_scarless with
            fragments, backbone, and interval
        assembly ← call primer_extension_scarless with pcr_fragments and
            extensions
    else
        pcr_fragments, pcr_backbone ← call primer_complement with fragments and
            backbone
        assembly ← call primer_extension with pcr_fragments and pcr_backbone
    endif

    assembly ← call annotations with assembly and nodes
    add cut-site annotations for the assembly
    assembly ← call assembly_thermo with assembly, monovalent ion concentration,
        divalent ion concentration, DNA oligo concentration, and Tm function

    return assembly and nodes
```

3.2.4.7.4 Primer Extensions

With the complement primers, extension primer sequences are added to every amplicon for assembly. If Gibson, SLIC, or PCR-SOE has been selected, then the overlap extension method is used from *OverlapExtensionAssembler* (Pseudocode 8 line 11). If Golden Gate is selected, then the Golden Gate extensions method is used in *GoldenGateAssembler* performing either scar-less or non-scar-less assembly design (Pseudocode 9 lines 13 or 18). If BioBricks is selected, then the BioBricks method is used in *BioBricksAssembler*. After completion of the assembly extensions, the solution contains all parts in assembly-ready form.

3.2.4.7.5 Annotations

Once an assembly solution has been designed each *FragmentNode* is given annotations from the BLAST alignment data of each node (Pseudocode 8 line 12, Pseudocode 9 lines 21-22). These annotations are useful and necessary for creating SQL database entries that will eventually be presented to the user. The specific annotations added to each part are:

- database origin
- BLAST *query_start*
- BLAST *query_end*
- BLAST *subject_start*
- BLAST *subject_end*

These annotations for each part are further expanded in the next stage of primer thermodynamic analysis.

3.2.4.7.6 Primer Thermodynamic Analysis

Here the annotated assembly and user-provided concentrations of monovalent ions, divalent ions, and DNA oligos are used for computing primer thermodynamic values via Primer3-py[39]. Hairpin and homodimer secondary structures and thermodynamic information are computed for each part's forward and reverse primers. Total and "footprint" (primer sequences binding to the non-extended sequence) melting temperatures are found, along with GC nucleotide percentage. Each part's annotations are updated by adding this information to the forward and reverse primers (Pseudocode 8 lines 13-14, Pseudocode 9 lines 23-24). Overall, the following data are created for each primer:

- Total melting temperature
- Footprint melting temperature
- GC percentage
- Boolean value for a hairpin structure
- Hairpin melting temperature
- Hairpin Gibbs free energy
- Hairpin enthalpy
- Hairpin entropy
- Boolean value for a homodimer structure
- Homodimer melting temperature
- Homodimer Gibbs free energy
- Homodimer enthalpy
- Homodimer entropy

With these calculations concluded and annotations completed, the complete assembly-ready solution can be utilized by the rest of the software.

Pseudocode 10: Assembly Create Service – Gibson Cloning

Inputs:

NONE

Variables:

results: A list of BLAST query alignments for the insert

error: Error messages from BLAST if errors occurred

assembly: The fully designed assembly solution

fragments: A list of FragmentNodes to extract more information about each assembly sub-sequence

```
create gibson assembler with user form inputs

if multi_query
    results, error ← call assembler run_multi_query function
    call assembler multi_query_solution_building function with results
else
    results, error ← call assembler query function
    call assembler solution_building function with results
endif

assembly, fragments ← call assembler design function with solution (default 0)

save all assembly solution data to SQL database
```

3.2.5 Time, Cost, and Risk Calculations

This section explains how assembly cost, time, and risk estimates are calculated for a single solution after the design procedures of Section 3.2.4 are complete.

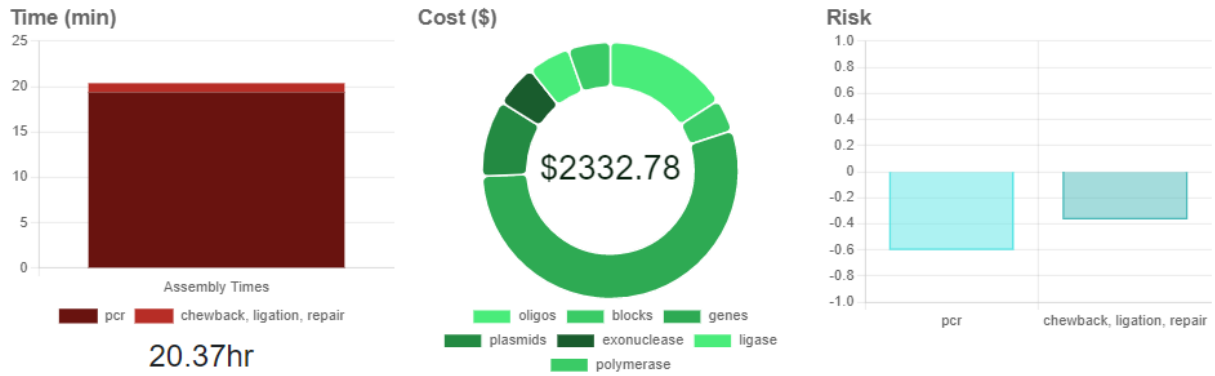


Figure 24: An example of the time, cost, and risk estimates charts for a Gibson assembly solution. The total time estimate is 20.37 hours, and the total cost estimate is $2332.78. PCR log odds of failure is -0.602, and for the assembly reactions (chewback, ligation, and repair) the log odds of failure is -0.368.

Each solution for an assembly has values for cost, time, and risk calculated using user-provided inputs and calculated attributes of the assembly (Figure 24). The time values show users expected experimental times for major steps in their assembly, such as PCR amplification, RE chewback, and ligation for Golden Gate assemblies. Cost values are important for indicating an expected cost for assemblies, accounting for small, medium, and large DNA sequence costs and enzymes used in the experiments (e.g., BsaI or ligase for Golden Gate). Risk values are also computed for major steps in the assembly experiments, representing *log odds of failure*, based on the log-based odds of success formula[49]. These analytical computations are then displayed as charts on the front-end (UI) of the web service. The log-based odds of success is calculated using the formula, with the different $P_{success}$ values for different procedures (Table 1):

$$risk = \log\left(\frac{1 - P_{success}}{P_{success}}\right)$$

Time values are split into PCR amplification and assembly method-specific experiment times. The PCR time is calculated for all assembly types using Q5 DNA polymerase in minutes, uses nucleotide lengths of all subsequences of the assembly, groups lengths of assembly sequences in ranges of 1kb, and is based on general PCR amplification protocols[1,2]. Grouping the assembly sequences into clusters allows a practical PCR time to be computed. With the clusters created, a single time can be determined that expresses the time for parallel PCR amplification of all sequences in that cluster. Sequences in the size range of 0-1000 nt are in cluster 1, 1001-2000 in cluster 2, etc. The total PCR time is the summation of the times needed to PCR the clusters of the sequences. The exact form of calculation follows:

$$step = 0.5 * cluster$$

$$denature = step + 0.4$$

$$anneal = step + 0.4$$

$$polymerization = step + 0.4$$

$$time = 1.0 + 30 * (denature + anneal + polymerization) + 10.0$$

In addition to PCR time, specific times for each assembly type are added. Golden Gate assemblies have the digestion-ligation experiment times added, using the New England Biolabs (NEB) protocol[50]. Gibson assemblies have chewback-ligation-repair time added according to the NEB[51]. SLIC assemblies have individual chewback and ligation times calculated, according the published methodology[8,9]. PCR assemblies do not have additional time calculations. BioBrick assemblies have digestion and ligation experiment times added using the Ginkgo Bioworks and NEB guidelines[52]. Once the PCR time and method-specific times are calculated, they are summed up to

provide the total expected time for a particular assembly, for each assembly method. This result is presented as a stacked bar chart on the solution's front-end detail page (Figure 24). The total height of a reflects the total time, and the heights of its subdivisions reflect the times of the components' experimental procedures (e.g., PCR).

Assembly costs are in US dollars are determined by:

- Short, medium, and long length DNA sequences
- Repository plasmids
- Enzyme costs, particular to each assembly type

Repository plasmid costs are currently a flat $75, enzyme costs are added by users only where needed, via custom cost inputs, and DNA sequence costs are for primers and assembly sequences not found in the BLAST databases (Pseudocode 11). We define short DNA sequences as those ≤ 100 nt and called *oligos*. Medium length sequences are >100 nt and ≤ 1000 nt and called *blocks*. Sequences > 1000 nt are considered large and are called *mega-blocks*. Nucleotide lengths for each of the assembly sub-sequences have their individual costs calculated and summed according to these three categories, with the cost per nucleotide for each category provided by the users. The total cost of the assembly solution is the summation of the costs for plasmids and other DNA sequences of varying length, plus the costs of enzymes. This grand total and its components are displayed on the solution's front-end detail page in a donut chart (Figure 24). The costs of the various components are shown as slices of the donut, with relative sizes proportional to their relative original dollar costs.

Pseudocode 11: Assembly Cost Calculations

```
Inputs:
nt_costs: list of individual user-submitted per-nucleotide costs for oligos, blocks,
    and genes, ex. [0.1, 0.2, 0.2]
nt_lengths: list of lengths of non-repository plasmid insert sequences
plasmid_count: number of repository plasmid insert sequences
enzyme_costs: list of user-submitted costs for assembly experiment enzymes

Variables:
nt_totals: list of individual total costs for oligos, blocks, genes, plasmids.
    Initialized to: [0.0, 0.0, 0.0, 0.0]

costs(nt_costs, nt_lengths, plasmid_count, enzyme_costs)
    for length in nt_lengths
        if length <= 100
            add length * nt_costs[0] to nt_totals[0]
        elif length > 100 and length <= 1000
            add length * nt_costs[1] to nt_totals[1]
        else
            add length * nt_costs[2] to nt_totals[2]
    endfor
    add plasmid_count * 75.0 to nt_totals[3]
    total_cost ← sum of nt_totals + sum of enzyme_costs
    individual_costs ← combined lists of nt_totals and enzyme_costs
```

Risk values are calculated for each major experiment of an assembly type, similar to time values. These values are *log odds of failure*, determined by taking the logarithm of the quotient of a given reaction type's probability of failure divided by the probability of success. These probability values are determined by a rough estimate of the number of successful experiments in a set of ten trials

(Table 1). This approach provides values in the range of -1 to 1, with -1 containing the least risk and 1 the most risk. PCR amplification is present in all assemblies. PCR assemblies have no risk values besides the risk of PCR amplification. Gibson assembly has risk values for one-pot chewback, ligation, and repair. Golden Gate has risk values for one-pot digestion and ligation. BioBrick assemblies have one value for digestion and another for ligation reactions. SLIC assemblies have a risk value for chewback and another for ligation. The risks for a solution's reactions are displayed on the solution's front-end detail page, as a bar chart ranging from -1 to 1, where the risk of each type of reaction has its own (separate) bar (Figure 24).

Table 2: Listing of $P_{success}$ for experiment types of each supported cloning method

Method	**PCR**	**Chewback**	**Digestion**	**Ligation**	**Digestion, Ligation**	**Chewback, Ligation, Repair**
Gibson	0.8	-	-	-	-	0.7
Golden Gate	0.8	-	-	-	0.9	-
SLIC	0.8	0.8	-	0.8	-	-
BioBricks	0.8	-	0.9	0.8	-	-
PCR-SOE	0.8	-	-	-	-	-

These cost, time, and risk calculations are useful when inspecting a single assembly solution but are critical when choosing between different assembly methods for the realization of a construct. This is the purpose of assembly bundles: to provide the user with information that would assist him/her in making the best choice. When an assembly bundle is viewed, each different assembly in the bundle is displayed in a column with their cost, time and risk charts shown first. The charts have a structure that allows the user to compare the costs, times, and risks of the various assembly methods, by quick visual inspection.

3.3 Output

Detail pages for the site present all the models' data fields in an organized and attractive manner for users. This section describes these front-end pages for assemblies (3.3.1), solutions (3.3.2), parts (3.3.3), and primers (3.3.4), where various interactive and static elements allow users to explore all aspects of assemblies and their solutions.

3.3.1 Assembly Detail

Assembly detail pages first display general, BLAST query, and experimental information that is mostly a reflection of input values. Next, the solution for the assembly is displayed, first with its time, cost, and risk charts. Below these charts is the access to the solution itself. Within the page element displaying the solution, general analytical information is shown for informative points such as average primer melting temperature, average primer size, number of parts (Figure 2S).

3.3.2 Solution Detail

The solution pages contain most content of all the pages across the site for assembly details (Figure 3S). Extensive details about the solution's parts and primers are displayed. Starting with parts, a table listing their general information for the solution is shown for (Figure 5S):

- Database
- Length
- Extended length
- BLAST query-start and query-end values
- BLAST subject-start and subject-end values
- Solution positional index

Below this table is a plasmid graphical map showing a high-level representation of the theoretical solution construct with part labels (Figure 4S). After this plasmid map each of the parts are concisely displayed in individual HTML elements with links to their individual detail pages (Figure 6S).

3.3.3 Part Detail

Detail pages for solution parts show exact sequence data for a single part (Figure 9S). General info is displayed first for date created, database origin, length, and extended length. A part map, using the same plotting library as the solution plasmid maps, is then displayed that shows the part within the plasmid construct with its neighbors and assembly primers. This map is useful for providing users with an intuitive idea of how each part fits into the construct. Under the graphic the part's assembly and non-assembly sequences are displayed. Forward and reverse primer sequences are also shown after the part sequences, and their detail page links are found at the top of this part detail page.

3.3.4 Primer Detail

Primer detail pages have the same approach as the part pages, displaying general info for date created primer type (forward or reverse), sequence, and their individual thermodynamic analysis calculations from Primer3 (Section 3.2.4.7.6) (Figure 10S). Solution primers are displayed in a similar format as the parts, with a table listing shown first (Figure 7S). As with the parts, this table contains general information with Primer3 thermodynamic calculations:

- Melting temperature
- GC%
- Hairpin presence (true/false)

- Hairpin melting temperature
- Hairpin free Gibbs energy
- Homodimer presence (true/false)
- Homodimer melting temperature
- Homodimer free Gibbs energy

As with the parts, each primer is then displayed below this table with links to their individual detail pages (Figure 8S). Summary information is also added to the individual primer elements for primer type, melting temperature, and presence of hairpin or homodimer structures that potentially need inspection.

3.4 Assembly Bundles

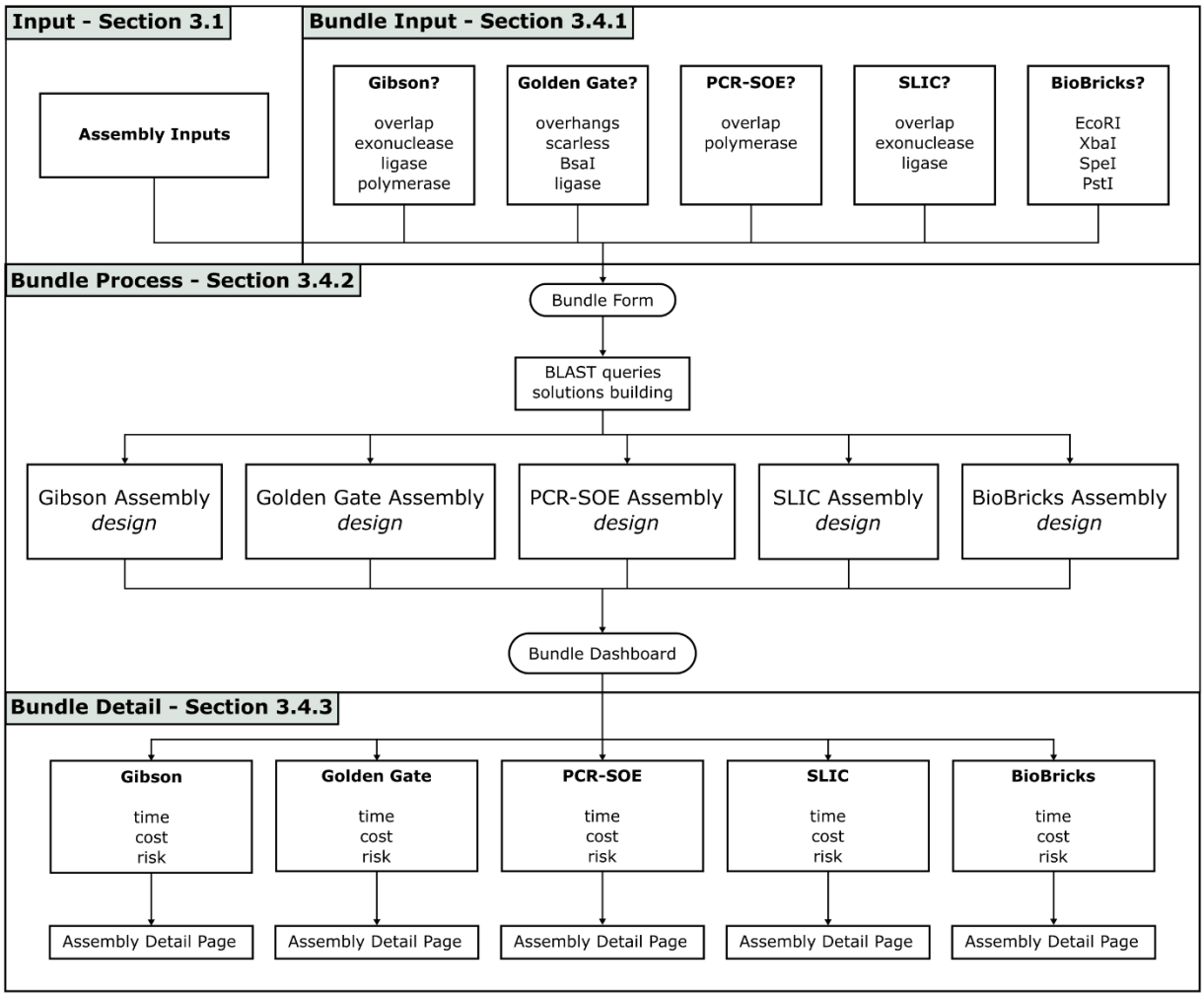


Figure 25: Visual organization of the assembly bundle feature and its workflow

Within this section, the creation, execution, and outputs of the assembly bundles are described. It is shown how the input form for this approach gathers identical data as the single assemblies. The internal procedures for creating these collections of independent assemblies with single solutions, following the algorithms in Section 3.2.4, for comparison and optimal selection is then described. Lastly, the way in which these multiple assembly solutions are presented in a dashboard is shown.

3.4.1 Bundle Input

Assembly bundle forms are essentially identical to the other assemblies (Section 3.1.2), however without the assemblies' specifics. An added section is at the beginning of the form where users will select the methods they desire for inclusion in the bundle. When specific assemblies are, only then will their unique parameter inputs be added to the end of the form. Otherwise, the assemblies and their unique parameters are ignored.

3.4.2 Bundle Creation

Assembly bundles utilize all the functionality described in Sections 3.2 and 3.3. Using input data collected by the bundle input form, an instance of the *Assembler* class is created for BLAST queries and solution building. An instance of *Assembler* is created since it will not be used for cloning method-specific designs. Then, following the methodology in Sections 3.2.4.1 and 3.2.4.2, the appropriate single- or multi-entry BLAST query and solutions generation will take place. With the sequence alignments gathered and solution candidates created, an *AssemblyBundle* SQL database model is created (Section 3.2.3.4). This bundle object is used for association of the selected assemblies' *Assembly* SQL models (Section 3.2.3.1) that will be created. Next, for each assembly method selected their individual *design* procedures following Sections 3.2.4 and 3.2.5 are run in series. Following a single example if Gibson assembly was included in a bundle, first, the appropriate *Assembly* model (*GibsonAssembly*) and *Assembler* class instance is created (*GibsonAssembler*) that will utilize the solutions generated from the first base *Assembler* created earlier in the bundle procedure. Next, the top scoring solution is used for the *design* procedure to create assembly primers, Primer3 thermodynamic analysis, and part annotations (Section 3.2.4.7). Then, the cost, time, and risk estimates are computed for the solution according to the methods in Section 3.2.5. For this example, all the data created for the assembly and its solution will be

contained within the *GibsonAssembly* model instance. Finally, this *GibsonAssembly* model is associated to the *AssemblyBundle* model created. This full solution design procedure is repeated for the other selected assembly methods, following their specific *design* processes. Once complete, there will be multiple *Assembly* SQL models associated to the single *AssemblyBundle* model created for this procedure (Figure 15), each containing complete information for their individual assembly and solution that will be presented to the user at the output.

3.4.3 Bundle Detail

Bundles are displayed with a dashboard approach for the collection of candidate assemblies created by the user. The dashboard is organized in a column-wise fashion, with each column containing a single assembly. For each assembly's column their time, cost, and risk charts are displayed in rows that allow for quick comparison between the different assemblies. For example, each of the time charts are within the same row and so on for the cost and risk charts. The assembly detail pages are accessible through links at the tops of each column, and their solutions through links below the charts (Sections 3.3.1 and 3.3.2). With the assembly bundles organized in this way users can decide based on experience, resources, and information of the dashboard which assembly method would be most advantageous for their objectives.

Chapter 4: Bioinformatics Contributions; Claims and Case Studies

This chapter describes and establishes the significance of *Smithy* as a practical tool in the field of Synthetic Biology and Bioinformatics (Table 2). Each major contribution of the web application will be justified in its own section (Sections 4.1-7). Next, the two case studies performed that exemplify *Smithy's* strengths are described in detail (Sections 4.8-9), where their inputs, internal processing, and outputs for their assembly solutions are thoroughly reported. Finally, a discussion of *Smithy* as a whole is given in Section 4.10.

Table 3: Summary of advantages

Claim	Substantiation
1. Categorization and mapping of cloning methods to an abstraction hierarchy that facilitates effective and extendible software design	**Section 4.1** • Implementation of cloning taxonomy (Section 2.5) • The assembler class hierarchy is designed around the abstractions embedded into these groupings • The assembler class and its descendants used for specific cloning method implementations • The Assembler class can be seen as a natural addition to the context of assembly design code libraries for Synthetic Biology • Smooth development of existing methods • Additions of unsupported methods, and the creation of new class definitions for breakthrough methods • OOP class-based design for assemblers which combine all of the major contributions of the project that can be extended with minimal code • New components and features of the Django application can be added with ease
2. Simplified UI/UX over the j5 and REPP approaches	**Section 4.2** **UI**: Unified inputs for all methods **UX**: No download and installation necessary Solution results are presented in highly

Unique inputs, and input types are only present in forms where necessary Minimal, essential user input requirements: only two file uploads and a simple form unlike j5 requiring several input files and configurations Bundles are a strong feature for quickly running multiple assembly types without redundant input requirements	structured web-page organization for assemblies, solutions, parts, primers All solution data is downloadable in files (parts, primers, materials orders) that users can take and use for running experiments

3. Improved UI/UX over the j5 and REPP approaches

Section 4.2

UI	**UX**
Usage of contemporary UI/UX front-end tools for implementing web pages: Bootstrap5, JavaScript, Chart.js Interactive pages for viewing solution data No CLI requirements like REPP that are prone to difficulties and errors	An openly accessible website Solutions have analytical charts for cost, time, and risk estimate to determine feasibility and strength of solutions Form submission checks for correct and usable input values, correct values saved, and correctable values are alerted to the user when form is returned

4. Support for major cloning methods

Section 4.3

- Like j5 but an improvement on REPP: Gibson, Golden Gate, PCR, SLIC, BioBricks have been implemented

5. Comprehensive assembly solutions for optimal choices

Section 4.4

- The fragment network can take nodes and organize 100s of solutions is fractions of a second
- *FragmentTree* (Section 3.2.4.2) contains every possible solution combination of fragments in the query results set

	• Assembly solutions are executable in the lab
6. Open and accessible tool	**Section 4.5** • Free, open source, and hosted on GitHub • Simple, widely accessible website • User accounts can be added later for extra features and exclusivity of assembly solutions
7. Analytical dashboard for comparison of assembly candidates for user-selected cloning methods	**Section 4.6** • The *assembly bundle* (Section 3.4) and its dashboard • Multiple assembly types are run in succession to provide different solution types for a single set of assembly fragments • The results of the bundle are combined, individual assemblies that contain all the same data seen in individual assemblies • Bundle results are shown in the bundle dashboard with the initially displayed data being the cost, time, and risk charts for quick analysis and comparison (Section 3.4.3)
8. Reusability of core features	**Section 4.7** • Each major component of the project (*Blaster*, A*ssembler*, and primer analysis) are all designed, implemented, and used as independent Python modules • The Django *Smithy* sub-application is also, by nature of Django, portable to other existing Django applications • These major algorithmic components can be used in any Python software project outside of a web-application

4.1 Cloning Methods Abstraction Hierarchy

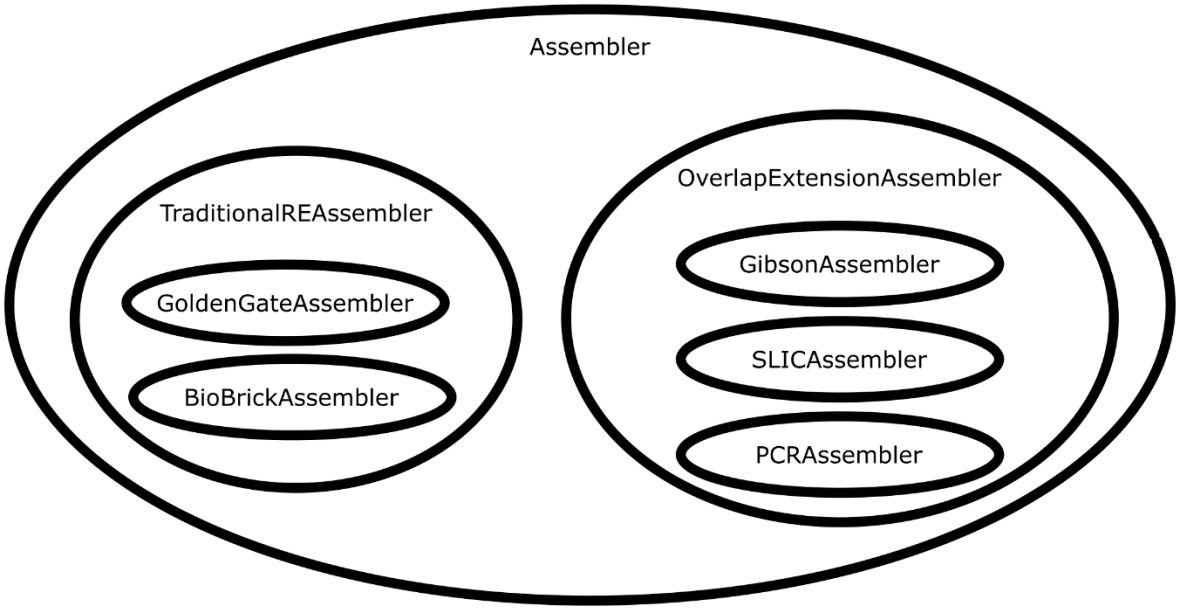


Figure 26: Assembler class hierarchy

The detailed study of contemporary cloning methods described in Chapter 2 led to an abstract grouping of cloning methods according to previous reviews and personal analysis (Figure 26). The groupings were made to satisfy relationships between cloning methods from experimental and procedural perspectives. Once the grouping of the studied cloning methods (e.g., Traditional Cloning, PCR-SOE, SLIC, Gibson, Golden Gate, GoldenBraid, and MoClo) was complete, the group were utilized to implement a robust class hierarchy in Python, following object-oriented design principles. This software implementation of the abstract groupings allows extension and modification of existing classes. The creation of new classes to describe newly invented cloning methods can be done with ease and minimal coding, while avoiding redundant code, as much as possible. At the core, the motivation is to follow the programming principle of "don't repeat yourself" (DRY) to the fullest.

4.2 Simplified and Improved UI/UX

Compared to the j5 and REPP projects, *Smithy* provides a simplified and improved user interface and experience.

The user interface has been simplified in several ways. First, inputs for all cloning method types have been unified. Unique inputs and input types are only present in individual assembly type forms, as appropriate, such as overlap amount for Gibson cloning or overhangs' set for Golden Gate cloning. Otherwise, similar forms exist for the different assembly methods. User inputs are also minimized with only two file uploads needed, one for an insert sequence and one for a backbone sequence. This is, unlike j5, where several input and configuration files are required. Additionally, the assembly bundle is a strong feature, which allows the execution of multiple assembly types, following a single input procedure.

Smithy provides improvements to the kinds of user interfaces seen in the previous projects. This is done through the use of contemporary UI/UX front-end tools for implementing webpages: Bootstrap5, JavaScript, Chart.js. Furthermore, there is no command line interface (CLI) requirements for initiating assemblies, such as REPP's. CLIs are generally more permissive to input errors and are not as easy to use as graphical UIs.

In contrast to REPP and j5, *Smithy's* user experience has been simplified in multiple ways. First, no application need be downloaded and installed, as seen with REPP. Second, assembly solution results are presented in a structured webpage organization for assemblies, solutions, parts, and primers. Third, all solution data for parts, primers, and materials orders are downloadable in files, which users can utilize for running actual experiments. In addition, *Smithy* also has other, more subtle, improvements to user experience: it provides an openly accessible web application, which is not the case for REPP and is no longer the case for j5; solutions for single and bundle assemblies

have analytical charts for cost, time, and risk attributes, to assist users in selecting between multiple solutions; finally, form submissions are always verified for correct and usable input values. This last embellishment is standard practice in web application development, but is important for ensuring that researchers submit assembly parameters that yield usable solutions.

4.3 Support for Major Contemporary Cloning Methods

Similar to j5, but an improvement to REPP (supporting only Gibson assemblies), *Smithy* supports the major contemporary cloning methods used in synthetic biology: Gibson, Golden Gate, PCR-SOE, SLIC, and BioBricks. With support for these experimental methods, *Smithy* appeals to a broad range of researchers in synthetic biology and can be useful for many different projects.

4.4 Optimal and Comprehensive Assembly Solutions

Smithy's core algorithms generate assembly solutions that are comprehensive and optimal. Selected databases are used for sequence queries using the submitted insert and all alignments are returned by these queries for use in constructing candidate solutions. Searching for existing sub-sequences of a desired construct using BLAST ensures that users can utilize as many pre-existing sequences as possible to reduce cost. Given the full set of BLAST query results for a given insert, all possible combinations of fragments, including gap-filling synthetic fragments, are contained in the *FragmentTree*. The solution tree is also created in fractions of a second, providing hundreds of solutions, and time bottlenecks are dominated by the time to perform BLAST queries for each sequence database.

4.5 Open and Easily Accessible Tool

Smithy is free to use, is open source, and fully downloadable from GitHub[53] for those who wish to make contributions to the project. *Smithy* also exists as a simple web application openly accessible

to any researcher with internet access. The project is written in Python using popular bioinformatics libraries, so other synthetic or computational biologists, who are also developing similar applications, can contribute to and improve upon the software with relative ease.

4.6 Bundle Dashboard for Comparison of Methods

The bundling feature and its resulting dashboard allows for comparison of multiple assembly solutions, respecting different assembly methods. Multiple cloning method types are run in succession, to provide different solutions for making one construct, using a single set of assembly fragments. A bundle is run similarly to any other assembly, but in addition, the user must select the cloning methods he/she is interested in. In response, the application displays requests for parameter values, particular to each and every chosen method. Bundle results are displayed in the bundle dashboard, whose front page includes cost, time, and risk charts, for a quick view of and comparison between the different assembly methods – all leading to the same final construct.

4.7 Portability of Core Features

Smithy has been designed for portability of its core features. Each major component of the project, *Blaster*, *Assembler*, *Primer Analysis*, are designed as independent Python modules. These modules can also be used in other Python programs outside of web application development, and independently of each other (e.g., using the *Blaster* class for sequence queries). The Django "*smithy"* sub-application is also, by nature of Django, portable to other existing Django applications.

4.8 Case Study 1: Riboswitch Gibson Assembly

The objective of this case study is to demonstrate *Smithy*'s ability to generate all necessary information, including repository and synthetic DNA insert sequences, to build a target construct,

following a single assembly method, chosen from a menu of available methods. In this study, a riboswitch-modulated fluorescent gene circuit solution was generated for construction via Gibson assembly.

4.8.1 Input

The target construct is composed of mCherry, a theophylline riboswitch (theoRs), and GFP, in that order, within a pUC18 plasmid backbone (sequences in Table 3). The mCherry will provide a constant fluorescent signal with the GFP signal modulated by the riboswitch once activated. For the assembly design, the pUC18 backbone sequence was inputted in a FASTA file and the insert sequences (mCherry, theophylline riboswitch, and GFP) in a multi-entry FASTA file (*insert.fasta* and *pUC18.fasta*, in supplementary information). It is important to note that backbone sequences are expected to be in 5' to 3' form, already linearized at the location, where the construct is to be inserted.

Table 4: Input sequences for the Gibson riboswitch fluorescence construct

Part	**Sequence**
mCherry	ATGGTGAGCAAGGGCGAGGACGACAACATGGCCATCATCAAGGAGTTCATGCGCTTC AAGGTGCACATGGAGGGCTCCGTGAACGGCCACGAGTTCGAGATCGAGGGCGAGGG CGAGGGCCGCCCCTACGAGGGCACCCAGACCGCCAAGCTGAAGGTGACCAAGGGCG GCCCCCTGCCCTTCGCCTGGGACATCCTGTCCCCTCAGTTCATGTACGGCTCCAAGGC CTACGTGAAGCACCCCGCCGACATCCCCGACTACTTGAAGCTGTCCTTCCCCGAGGG CTTCAAGTGGGAGCGCGTGATGAACTTCGAGGACGGCGGCGTGGTGACCGTGACCCA GGACTCCTCCCTGCAGGACGGCGAGTTCATCTACAAGGTGAAGCTGCGCGGCACCAA CTTCCCCTCCGACGGCCCCGTAATGCAGAAGAAGACCATGGGCTGGGAGGCCTCCTC CGAGCGGATGTACCCCGAGGACGGCGCCCTGAAGGGCGAGATCAAGCAGAGGCTGA AGCTGAAGGACGGCGGCCACTACGACGCCGAGGTCAAGACCACCTACAAGGCCAAG AAGCCCGTGCAGCTGCCCGGCGCCTACAACGTCAACATCAAGCTGGACATCACCTCC CACAACGAGGACTACACCATCGTGGAACAGTACGAGCGCGCCGAGGGCCGCCACTC CACCGGCGGCATGGACGAGCTGTACAAGTAA
theoRs	GGTGATACCAGCATCGTCTTGATGCCCTTGGCAGCACCCTGCTAAGGTAACAACAAG
GFP	ATGAGTAAAGGAGAAGAACTTTTCACTGGAGTTGTCCCAATTCTTGTTGAATTAGATG GTGATGTTAATGGGCACAAATTTTCTGTCAGTGGAGAGGGTGAAGGTGATGCAACAT ACGGAAAACTTACCCTTAAATTTATTTGCACTACTGGAAAACTACCTGTTCCATGGCC AACACTTGTCACTACTTTCTCTTATGGTGTTCAATGCTTTTCCCGTTATCCGGATCATA

TGAAACGGCATGACTTTTTCAAGAGTGCCATGCCCGAAGGTTATGTACAGGAACGCA
CTATATCTTTCAAAGATGACGGGAACTACAAGACGCGTGCTGAAGTCAAGTTTGAAG
GTGATACCCTTGTTAATCGTATCGAGTTAAAAGGTATTGATTTTAAAGAAGATGGAA
ACATTCTCGGACACAAACTCGAGTACAACTATAACTCACACAATGTATACATCACGG
CAGACAAACAAAAGAATGGAATCAAAGCTAACTTCAAAATTCGCCACAACATTGAA
GATGGATCCGTTCAACTAGCAGACCATTATCAACAAAATACTCCAATTGGCGATGGC
CCTGTCCTTTTACCAGACAACCATTACCTGTCGACACAATCTGCCCTTTCGAAAGATC
CCAACGAAAAGCGTGACCACATGGTCCTTCTTGAGTTTGTAACTGCTGCTGGGATTAC
ACATGGCATGGATGAGCTCTACAAATAA

pUC18

ATGACCATGATTACGAATTCGAGCTCGGTACCCGGGGATCCTCTAGAGTCGACCTGC
AGGCATGCAAGCTTGGCACTGGCCGTCGTTTTACAACGTCGTGACTGGGAAAACCCT
GGCGTTACCCAACTTAATCGCCTTGCAGCACATCCCCCTTTCGCCAGCTGGCGTAATA
GCGAAGAGGCCCGCACCGATCGCCCTTCCCAACAGTTGCGCAGCCTGAATGGCGAAT
GGCGCCTGATGCGGTATTTTCTCCTTACGCATCTGTGCGGTATTTCACACCGCATATG
GTGCACTCTCAGTACAATCTGCTCTGATGCCGCATAGTTAAGCCAGCCCCGACACCCG
CCAACACCCGCTGACGCGCCCTGACGGGCTTGTCTGCTCCCGGCATCCGCTTACAGAC
AAGCTGTGACCGTCTCCGGGAGCTGCATGTGTCAGAGGTTTTCACCGTCATCACCGA
AACGCGCGAGACGAAAGGGCCTCGTGATACGCCTATTTTTATAGGTTAATGTCATGA
TAATAATGGTTTCTTAGACGTCAGGTGGCACTTTTCGGGGAAATGTGCGCGGAACCC
CTATTTGTTTATTTTTCTAAATACATTCAAATATGTATCCGCTCATGAGACAATAACCC
TGATAAATGCTTCAATAATATTGAAAAAGGAAGAGTATGAGTATTCAACATTTCCGT
GTCGCCCTTATTCCCTTTTTTGCGGCATTTTGCCTTCCTGTTTTTGCTCACCCAGAAAC
GCTGGTGAAAGTAAAAGATGCTGAAGATCAGTTGGGTGCACGAGTGGGTTACATCGA
ACTGGATCTCAACAGCGGTAAGATCCTTGAGAGTTTTCGCCCCGAAGAACGTTTTCCA
ATGATGAGCACTTTTAAAGTTCTGCTATGTGGCGCGGTATTATCCCGTATTGACGCCG
GGCAAGAGCAACTCGGTCGCCGCATACACTATTCTCAGAATGACTTGGTTGAGTACT
CACCAGTCACAGAAAAGCATCTTACGGATGGCATGACAGTAAGAGAATTATGCAGTG
CTGCCATAACCATGAGTGATAACACTGCGGCCAACTTACTTCTGACAACGATCGGAG
GACCGAAGGAGCTAACCGCTTTTTTGCACAACATGGGGGATCATGTAACTCGCCTTG
ATCGTTGGGAACCGGAGCTGAATGAAGCCATACCAAACGACGAGCGTGACACCACG
ATGCCTGTAGCAATGGCAACAACGTTGCGCAAACTATTAACTGGCGAACTACTTACT
CTAGCTTCCCGGCAACAATTAATAGACTGGATGGAGGCGGATAAAGTTGCAGGACCA
CTTCTGCGCTCGGCCCTTCCGGCTGGCTGGTTTATTGCTGATAAATCTGGAGCCGGTG
AGCGTGGGTCTCGCGGTATCATTGCAGCACTGGGGCCAGATGGTAAGCCCTCCCGTA
TCGTAGTTATCTACACGACGGGGAGTCAGGCAACTATGGATGAACGAAATAGACAGA
TCGCTGAGATAGGTGCCTCACTGATTAAGCATTGGTAACTGTCAGACCAAGTTTACTC
ATATATACTTTAGATTGATTTAAAACTTCATTTTTAATTTAAAAGGATCTAGGTGAAG
ATCCTTTTTGATAATCTCATGACCAAAATCCCTTAACGTGAGTTTTCGTTCCACTGAG
CGTCAGACCCCGTAGAAAAGATCAAAGGATCTTCTTGAGATCCTTTTTTTCTGCGCGT
AATCTGCTGCTTGCAAACAAAAAAACCACCGCTACCAGCGGTGGTTTGTTTGCCGGA
TCAAGAGCTACCAACTCTTTTTCCGAAGGTAACTGGCTTCAGCAGAGCGCAGATACC
AAATACTGTTCTTCTAGTGTAGCCGTAGTTAGGCCACCACTTCAAGAACTCTGTAGCA
CCGCCTACATACCTCGCTCTGCTAATCCTGTTACCAGTGGCTGCTGCCAGTGGCGATA
AGTCGTGTCTTACCGGGTTGGACTCAAGACGATAGTTACCGGATAAGGCGCAGCGGT
CGGGCTGAACGGGGGGTTCGTGCACACAGCCCAGCTTGGAGCGAACGACCTACACCG
AACTGAGATACCTACAGCGTGAGCTTTGAGAAAGCGCCACGCTTCCCGAAGGGAGAA
AGGCGGACAGGTATCCGGTAAGCGGCAGGGTCGGAACAGGAGAGCGCACGAGGGAG
CTTCCAGGGGGAAACGCCTGGTATCTTTATAGTCCTGTCGGGTTTCGCCACCTCTGAC
TTGAGCGTCGATTTTTGTGATGCTCGTCAGGGGGGCGGAGCCTATGGAAAAACGCCA
GCAACGCGGCCTTTTTACGGTTCCTGGCCTTTTGCTGGCCTTTTGCTCACATGTTCTTT
CCTGCGTTATCCCCTGATTCTGTGGATAACCGTATTACCGCCTTTGAGTGAGCTGATA
CCGCTCGCCGCAGCCGAACGACCGAGCGCAGCGAGTCAGTGAGCGAGGAAGCGGAA
GAGCGCCCAATACGCAAACCGCCTCTCCCCGCGCGTTGGCCGATTCATTAATGCAGC
TGGCACGACAGGTTTCCCGACTGGAAAGCGGGCAGTGAGCGCAACGCAATTAATGTG

AGTTAGCTCACTCATTAGGCACCCCAGGCTTTACACTTTATGCTTCCGGCTCGTATGT
TGTGTGGAATTGTGAGCGGATAACAATTTCACACAGGAAACAGCT

With Gibson assembly selected as the method for *Smithy* to use, the assembly form is ready to fill in. The title for the assembly was *Gibson Riboswitch Assembly*, a multi-entry query solution was selected, and the insert file (*insert.fasta*; Figure 27) containing the mCherry, theophylline riboswitch, and GFP sequences, as individual entries, was named. The pUC18 backbone sequence was contained in another FASTA file (*pUC18.*fasta; Figure 27).

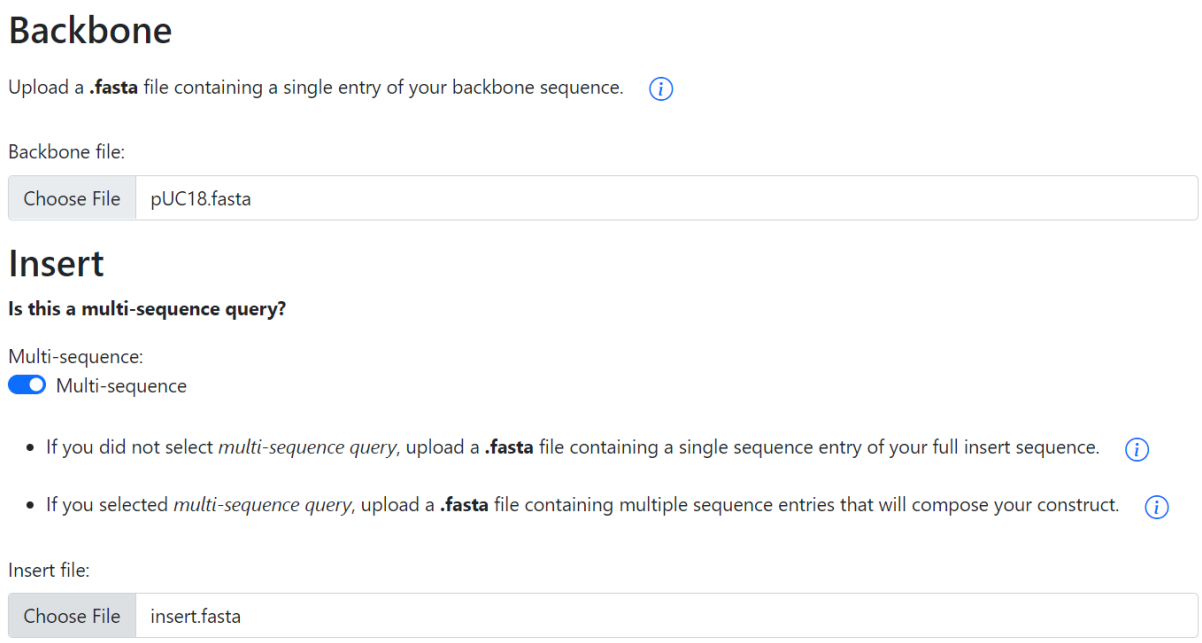


Figure 27: Assembly sequence inputs

All three supported BLAST DNA sequence databases (AddGene, iGEM, and DNASU) were selected (Figure 28). An overlap amount of 30 nt, a minimum BLAST sequence size of 50 nt, a maximum BLAST sequence size of 5000 nt, minimum synthetic sequence size of 50 nt, and a maximum synthetic sequence size of 5000 nt were entered for the *Overlap* and *BLAST Query* inputs in the form (Figure 28).

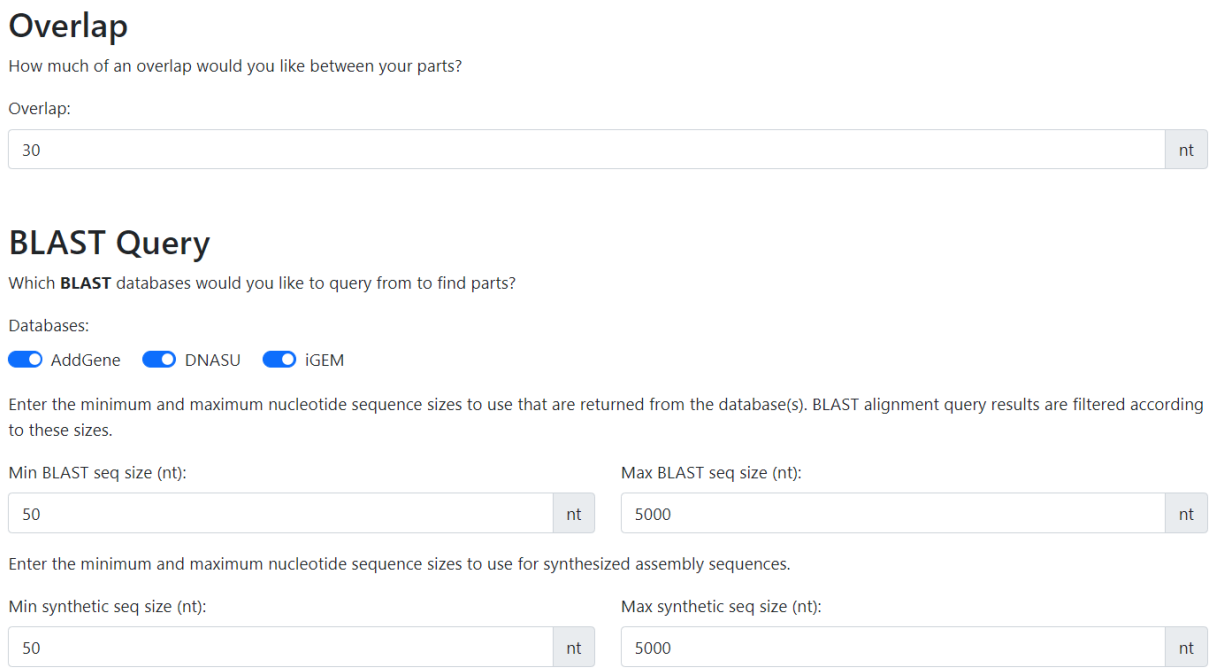


Figure 28: Overlap length and BLAST query parameters

Default *Experiment* inputs were used for this Gibson assembly: monovalent ion concentration of 50 nM, divalent ion concentration of 1.5 mM, a 0.8 mM dNTP concentration, 50 nM DNA oligo concentration, and a 60 °C target melting temperature (see Figure 29).

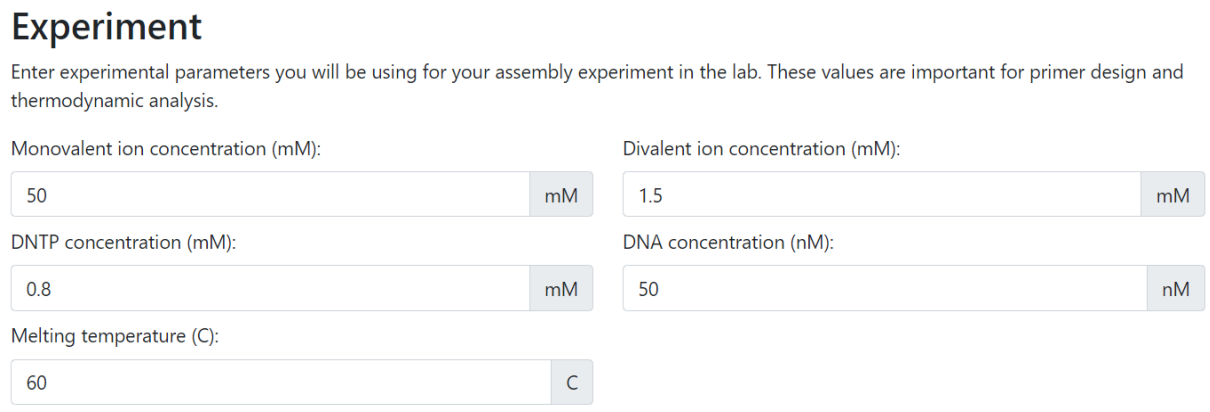


Figure 29: Experimental values and melting temperature input for primer design and Primer3 thermodynamic analysis

For calculating an estimate of assembly costs (Section 3.2.5) the cost for short oligo sequences was $0.80, for medium sequences $0.39, long sequences $0.65, and $125 for the enzymes (Figure 30).

Figure 30: Assembly cost inputs

Once the *Assemble* button is clicked (Figure 29) *Smithy* began the solution generation and design processes.

4.8.2 Process

First, a multi-entry BLAST query was performed over the individual insert sub-sequences, with their results collected independently according to the multi-entry query methodology in Section 3.2.4.1.3. For this particular solution, each of the three insert sequences were found in the BLAST databases: mCherry from AddGene, and the riboswitch and GFP from iGEM. As this was a multi-entry assembly design, no solution tree was created. From the three BLAST databases, 1726 sequence alignments were found for mCherry, 45 for the riboswitch, and 335 for GFP. For this case study's solution alignments (from BLAST), 50458-addgene corresponds to mCherry, BBa_J119317-igem to the theophylline riboswitch (theoRs), and BBa_M45123-igem to GFP.

With the BLAST alignments collected for each sub-sequence, primers were designed for the first solution created by *Smithy*. First, primer complementary (*footprint*) sequences for each insert and backbone sequence, as Pydna *Dseqrecord* objects, were created. A 60 °C target melting temperature as a constraint for forward and reverse primers was used, following the methodology in Section 3.2.4.3 (Table 4). This process produced Pydna *Amplicons* for each sequence, via *in silico* PCR simulations these amplicons are used for the rest of the design process[27]. Then, using the algorithm of Section 3.2.4.4, extension (*tail*) sequences were added to each primer for achieving a 30 nt combined overlap for each pair of contiguous sequences of the assembly (Table 4).

Table 5: Complementary and extension primer sequences for the Gibson assembly

Primer	**Complementary (footprint) Sequence**	**Extension (tail) Sequence**
mCherry_fwd (50458-addgene)	atggtgagcaagggc	acacaggaaacagct
mCherry_rvs (50458-addgene)	ttacttgtacagctcgtcc	gatgctggtatcacc
TheoRs_fwd (BBa_J119317-igem)	ggtgataccagcatcg	gagctgtacaagtaa
TheoRs_rvs (BBa_J119317-igem)	cttgttgttaccttagcag	ttctcctttactcat
GFP_fwd (BBa_M45123-igem)	atgagtaaaggagaagaacttttt	taaggtaacaacaag
GFP_rvs (BBa_M45123-igem)	ttatttgtagagctcatccat	cgtaatcatggtcat
pUC18_fwd	atgaccatgattacgaattcg	gagctctacaaataa
pUC18_rvs	agctgtttcctgtgtgaa	gcccttgctcaccat

For each part, annotations for BLAST data were then added and Primer3 thermodynamic analysis for each primer was calculated. Once Primer3 calculations were complete, the thermodynamic data for each primer of every part was added to the part's annotations (Table 6).

Lastly, the solution cost, time, and risk values were determined using user inputs and the solution.

4.8.3 Output

With the design complete for this construct, the assembly, solution, every part, part annotations, and every primer were saved to the Django SQL database for access by users. The assembly detail page first provides general information about the submitted construct: title, downloadable input files for the backbone and insert, and a listing of the form inputs the user entered (Figure 31). The solution for the assembly is accessible below this posted information.

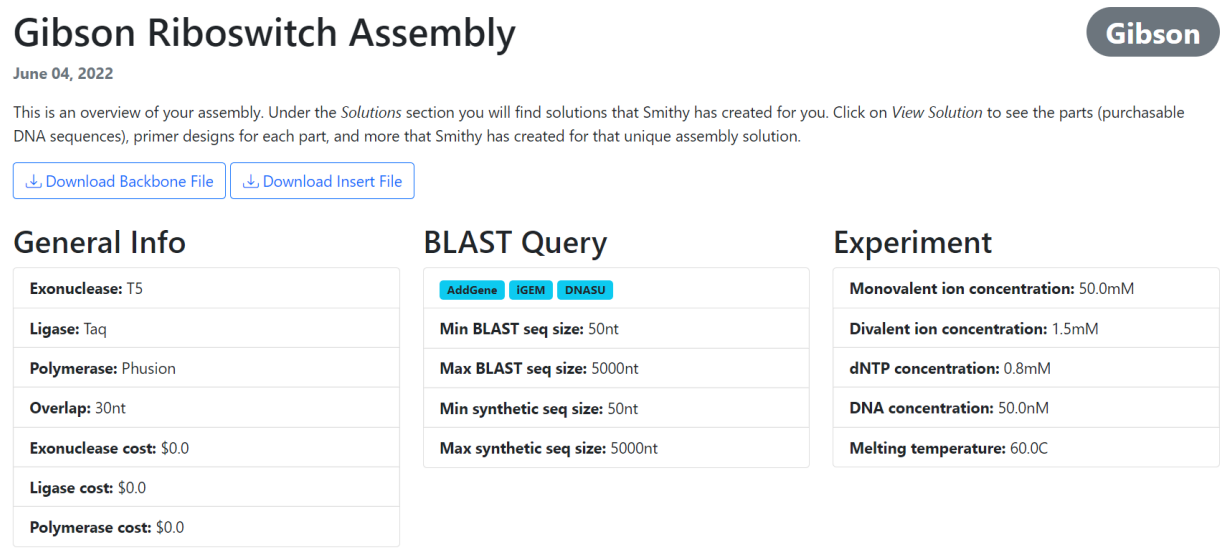


Figure 31: General information about the riboswitch assembly

The solution had a total expected time of 12.07 hrs: 11.07 hrs for PCR and 1.00 hrs for Gibson assembly. Total cost was $817.60: $219.20 for short DNA sequences, $225 for plasmids, $125 for exonuclease, $125 for ligase, and $125 for polymerase. PCR amplification risk is -0.602 (low risk) and -0.368 (moderate-low risk). See Figure 31 for the time, cost, and risk charts generated for these values for this assembly solution, seen on the assembly and solution detail pages.

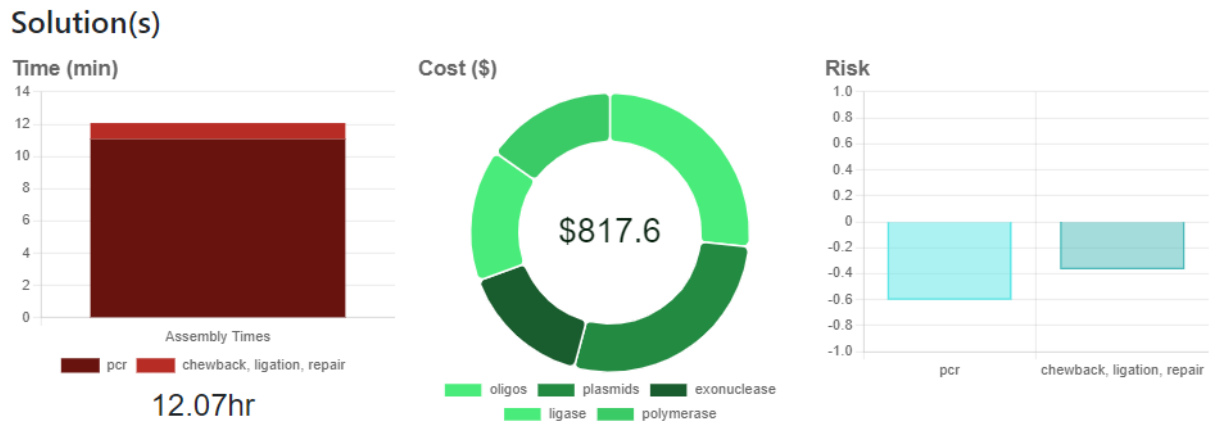


Figure 32: Time, cost, and risk charts for the riboswitch assembly

In the solution page, general information, much like that seen for the assembly detail page, is shown at the top: parts counts, average primer melting temperature, average primer length, and other information (Figure 33). For this example, the solution has 3 parts, 6 primers, an average primer T_m of 60.42 °C, and an average primer length of 34 nt. This portion of the page is also where the solution insert and primer sequence data files in CSV format can be downloaded. Another file is available for download via the *Download Order* button, a file that contains concise information about all repository plasmids, primers, and enzymes that need to be purchased for the assembly.

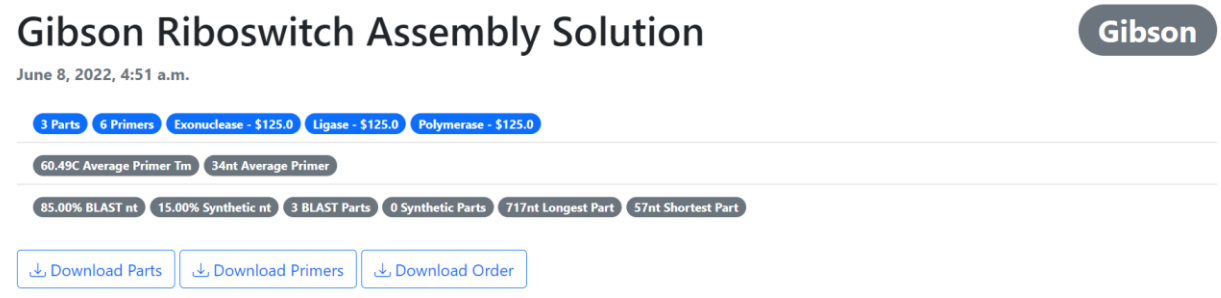
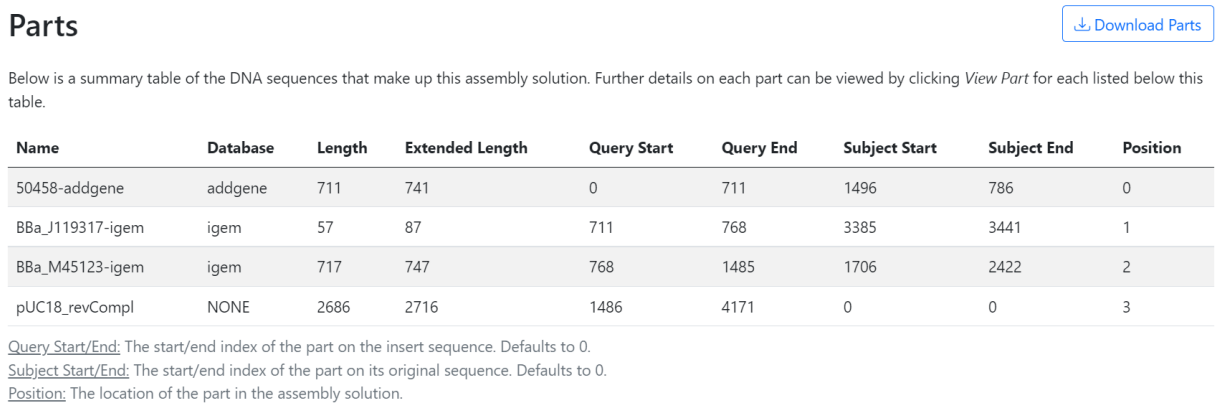


Figure 33: Assembly solution general information

Following this general information, is a display of the time, cost and risk charts, identical to those of Figure 32. Parts details are then listed, with an overview of their attributes shown in a table

containing: part names, BLAST database origin, original and assembly extension lengths, indexes on the construct that the parts fall into, and indexes on the originating plasmids where the sequences were found using BLAST. For this example assembly, each insert sequence was found in the BLAST databases: mCherry within the AddGene 50458 plasmid, the riboswitch in the iGEM BBa_J119317 sequence, and GFP in the iGEM BBa_M45123 sequence (Figure 34).



Parts

Download Parts

Below is a summary table of the DNA sequences that make up this assembly solution. Further details on each part can be viewed by clicking *View Part* for each listed below this table.

Name	Database	Length	Extended Length	Query Start	Query End	Subject Start	Subject End	Position
50458-addgene	addgene	711	741	0	711	1496	786	0
BBa_J119317-igem	igem	57	87	711	768	3385	3441	1
BBa_M45123-igem	igem	717	747	768	1485	1706	2422	2
pUC18_revCompl	NONE	2686	2716	1486	4171	0	0	3

Query Start/End: The start/end index of the part on the insert sequence. Defaults to 0.
Subject Start/End: The start/end index of the part on its original sequence. Defaults to 0.
Position: The location of the part in the assembly solution.

Figure 34: Summary table for assembly insert sequences (parts) for the solution

After this table of parts information, each part's detail page can be accessed. Summary information is displayed along with links to the part's primers, and a part mapping is shown that shows that part's neighboring sequences. Additionally, the sequences of the part and of its primers are presented. For example, the summary information for the AddGene 50458 plasmid part details page for mCherry is shown in Figure 35.

50458-addgene

Forward Primer | Reverse Primer

Date Created	Database	Length	Extended Length
May 26, 2022, 2:32 a.m.	addgene	711	741

Part Map

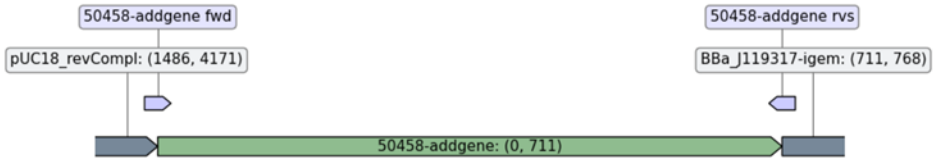


Figure 35: Summary information for the AddGene 50458 plasmid part for mCherry

The assembly's primers are then shown and as with the parts sections a summary table of primer information is given with individual primer detail pages. The table in Figure 36 shows the Primer3 thermodynamic calculations for each forward and reverse primer in the assembly.

Primers

Download Primers

Below is a summary table of the primers needed for each part of this assembly solution (shown above). Further details on each primer can be viewed by clicking *View Primer* for each listed below this table.

Name	Type	Tm (C)	GC%	Hairpin	hp Tm (C)	hp ΔG	Homodimer	hd Tm (C)	hd ΔG
50458-addgene forward primer	fwd	62.012	53.3	True	45.159	-612.597	True	3.513	-3469.324
50458-addgene reverse primer	rvs	61.920	50.0	True	39.281	-148.213	True	-0.026	-4952.691
BBa_J119317-igem forward primer	fwd	59.473	48.4	True	41.613	-271.143	True	-3.445	-4572.426
BBa_J119317-igem reverse primer	rvs	58.734	38.2	False	0.000	0.000	True	-33.069	-2810.310
BBa_M45123-igem forward primer	fwd	60.173	31.6	True	37.008	-1.130	True	-20.930	-3085.011
BBa_M45123-igem reverse primer	rvs	59.620	36.1	False	0.000	0.000	True	17.553	-7148.451
pUC18_revCompl forward primer	fwd	60.711	36.1	True	46.325	-668.448	True	25.367	-8370.901
pUC18_revCompl reverse primer	rvs	60.697	51.5	True	42.953	-493.427	True	-10.200	-4430.501

hp Tm (C): The melting temperature of the primer's hairpin (hp) structure.
hp ΔG: The Gibbs free energy of the primer's hairpin (hp) structure.
hd Tm (C): The melting temperature of the primer's homodimer (hd) structure.
hd ΔG: The Gibbs free energy of the primer's homodimer (hd) structure.

Figure 36: Summary table for assembly primers of the solution

Table 6: Complete primer designs with T_m for all sequences in the assembly case study

Primer	Sequence	T_m
mCherry_fwd (50458-addgene)	acacaggaaacagctatggtgagcaagggc	62.01
mCherry_rvs (50458-addgene)	gatgctggtatcaccttacttgtacagctcgtcc	61.92
TheoRs_fwd (BBa_J119317-igem)	gagctgtacaagtaaggtgataccagcatcg	59.47
TheoRs_rvs (BBa_J119317-igem)	ttctcctttactcatcttgttgttaccttagcag	58.73
GFP_fwd (BBa_M45123-igem)	taaggtaacaacaagatgagtaaaggagaagaacttttt	60.17
GFP_rvs (BBa_M45123-igem)	cgtaatcatggtcatttatttgtagagctcatccat	59.62
pUC18_fwd	gagctctacaaataaatgaccatgattacgaattcg	60.71
pUC18_rvs	gcccttgctcaccatagctgtttcctgtgtgaa	60.70

The full primer sequences designed by *Smithy* for this riboswitch fluorescence assembly are listed in Table 5, together with their melting temperatures (calculated with Primer3). As with each part in the assembly solution, each primer has a detail page where its unique data can be inspected. The forward primer for the AddGene 50458 plasmid part for mCherry is shown in Figure 37, and all other primer detail pages are in the same format.

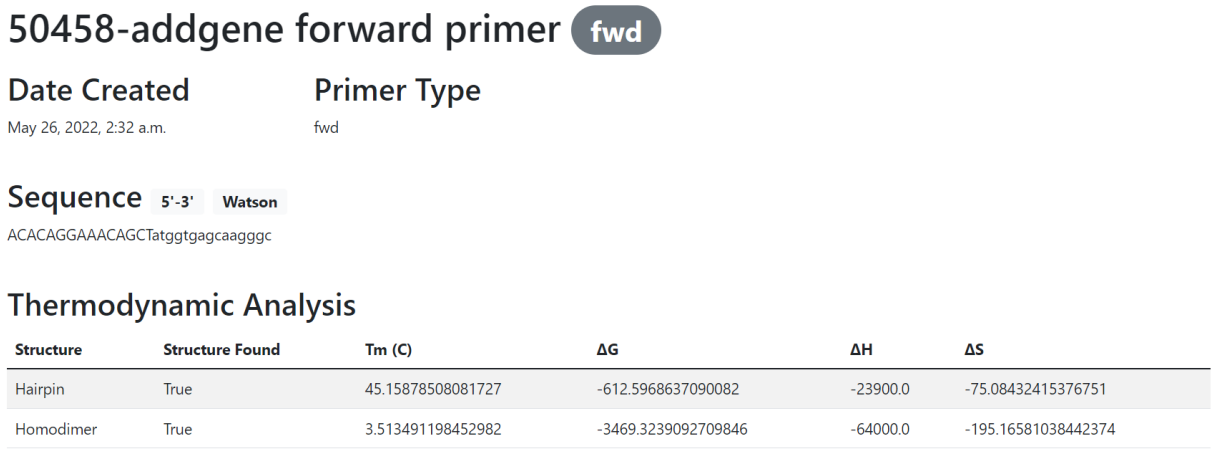


Figure 37: Detail page for the forward primer of the AddGene 50458 plasmid sequence for mCherry

Table 7: Primer3 thermodynamic computations for the Gibson assembly primers. HP is short for hairpin, and HD is short for homodimer.

T_m Total	GC	Hairpin	HP T_m	HP ΔG	HP ΔH	HP ΔS	Homodimer	HD Tm	HD ΔG	HD ΔH	HD ΔS
76.62	53.30	TRUE	45.16	-612.60	-23900.00	-75.08	TRUE	3.51	-3469.32	-64000.00	-195.16
75.18	50.00	TRUE	39.28	-148.21	-20300.00	-64.97	TRUE	-0.02	-4952.69	-46200.00	-132.99
72.66	48.40	TRUE	41.61	-271.14	-18500.00	-58.77	TRUE	-3.44	-4572.42	-44300.00	-128.09
70.79	38.20	FALSE	0.00	0.00	0.00	0.00	TRUE	-33.07	-2810.31	-28800.00	-83.80
70.53	31.60	TRUE	37.00	-1.13	-43400.00	-139.93	TRUE	-20.93	-3085.01	-35400.00	-104.19
70.98	36.10	FALSE	0.00	0.00	0.00	0.00	TRUE	17.55	-7148.45	-60800.00	-172.98
69.84	36.10	TRUE	46.32	-668.45	-22900.00	-71.68	TRUE	25.37	-8370.90	-73000.00	-208.38
77.32	51.50	TRUE	42.95	-493.43	-26200.00	-82.88	TRUE	-10.20	-4430.50	-37800.00	-107.59

Primer	T_m Footprint
50458-addgene-fwd	62.01
50458-addgene-rvs	61.92
BBa_J119317-igem-fwd	59.47
BBa_J119317-igem-rvs	58.73
BBa_M45123-igem-fwd	60.17
BBa_M45123-igem-rvs	59.62
pUC18_revCompl-fwd	60.71
pUC18_revCompl-fwd	60.70

4.8.4 Biological Results

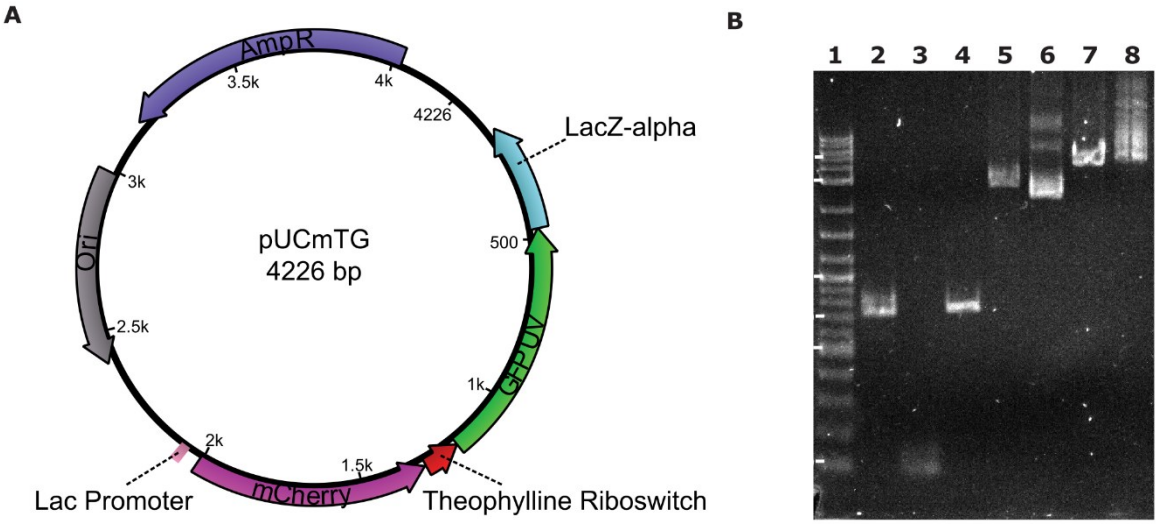


Figure 38: Case Study 1 results. A) The inserts mCherry, Theophylline riboswitch and GFP were inserted by the Gibson assembly after the pUC18 Lac promoter, immediately before the sequence corresponding to the regular AUG initiation codon from LacZa. B) 1% agarose gel of the PCR products where each DNA part was amplified with Smithy-designed primers containing the compatible tails for Gibson assembly. Lanes 1: Quickload Purple 1Kb plus DNA ladder (NEB), 2: mCherry, 3: Theophylline riboswitch, 4: GFP_UV, 5: PCR-amplified pUC18, 6:

supercoil/circular pUC18, 7: EcoRI-linearized Gibson Assembly plasmid and 8: supercoil/circular Gibson Assembly plasmid.

Table 8: Sizes of DNA parts, with and without Gibson amplification.

DNA Part	Size (nt)	Gibson-Amplified Size
mCherry	711	741
Theophylline Riboswitch	124	87
GFP_UV	717	747
pUC18	2686	2716
Gibson Assembly	4226	

4.8.5 Biological Methods

The PCR reactions to add the corresponding overlapping tails on each DNA parts and vector were performed using synthetic DNA oligonucleotides (Integrated DNA Technologies). The primer sequences and lengths were exclusively determined by *Smithy*. PCR reactions were carried out using Q5 DNA polymerase (New England Biolabs), with 100 μM primers and 0.5 ng of plasmid (Addgene) for 30 cycles with an annealing temperature ranging from 55 °C to 63 °C. The size and homogeneity of the PCR products were verified on agarose gel electrophoresis, followed by extraction and silica gel column purification (IBI Scientific). The optical density of the gel-purified PCR products was determined with a nanodrop to calculate its concentration.

The Gibson assembly of the tailed-vector (pUC18) and 3 tailed-DNA parts (mCherry, Theophylline riboswitch and GFP_UV) were performed according to the NEB procedures where 75 ng of plasmid and 0.5 pmol of each DNA parts were mixed together with the NEB's Gibson Assembly Master mix. It is recommended to incubate at 50 °C for 1 hour when 4 pats or more are used and for 15 minutes for less than 3 DNA parts. The mixture was incubated for 30 minutes.

The assembled plasmid DNA were transformed into NEB 5-alpha Competent *E. coli* (New England Biolabs) according to the manufacturer, followed by 1 hour incubation while shaking at

37 °C and then plated onto IPTG-spread LB-agar ampicillin petri dish for overnight incubation at 37 °C.

Pink colonies were picked and put in 3 ml LB culture overnight with shaking. The plasmids were extracted from the bacteria using silica gel column mini-prep kit (IBI Scientific). The plasmids were sequenced at Genewiz.

4.8.6 Summary

This example demonstrates *Smithy*'s abilities to efficiently produce DNA cloning assembly designs for a target construct, on that implements a theophylline riboswitch-modulated fluorescent circuit. The target construct, the plasmid backbone, and parameters for guiding the solution design were taken from user input and a comprehensive collection of necessary information was generated, information that allows users to perform successful assembly experiments in the lab.

4.9 Case Study 2: Modified Riboswitch Bundle Assembly: Gibson, Golden Gate, PCR-SOE

This second case study is intended to demonstrate the assembly bundle features of *Smithy* for comparison of three different assembly options for a single construct. The theophylline riboswitch modulating fluorescent gene circuit from Case Study 1 was the target construct, and had an added lactose operator and modifications to the original riboswitch. Gibson, Golden Gate, and PCR-SOE assembly methodologies were selected for the exercise.

4.9.1 Input

The target construct is composed of a lactose operator (LacO), mCherry, a theophylline riboswitch, and GFP within a pBR322 plasmid backbone (Table 7). All sequences excepting the new lactose operator are the same as in Case Study 1. The backbone sequence is inputted in an individual

FASTA file, and the insert sequences in a separate multi-entry FASTA file (*theoRs_LacO.fasta* and *pBR322.fasta*).

Table 9: Input sequences for the modified riboswitch fluorescence construct for the assembly bundle

Part	Sequence
LacO	GTGAGCTGATACCGCTCGCCGCAGCCGAACGACCGAGCGCAGCGAGTCAGTGAGCGAG GAAGCGGAAGAGCGCCCAATACGCAAACCGCCTCTCCCCGCGCGTTGGCCGATTCATT AATGCAGCTGGCACGAC
mCherry	ATGGTGAGCAAGGGCGAGGACGACAACATGGCCATCATCAAGGAGTTCATGCGCTTCA AGGTGCACATGGAGGGCTCCGTGAACGGCCACGAGTTCGAGATCGAGGGCGAGGGCG AGGGCCGCCCCTACGAGGGCACCCAGACCGCCAAGCTGAAGGTGACCAAGGGCGGCC CCCTGCCCTTCGCCTGGGACATCCTGTCCCCTCAGTTCATGTACGGCTCCAAGGCCTAC GTGAAGCACCCCGCCGACATCCCCGACTACTTGAAGCTGTCCTTCCCCGAGGGCTTCAA GTGGGAGCGCGTGATGAACTTCGAGGACGGCGGCGTGGTGACCGTGACCCAGGACTCC TCCCTGCAGGACGGCGAGTTCATCTACAAGGTGAAGCTGCGCGGCACCAACTTCCCCTC CGACGGCCCCGTAATGCAGAAGAAGACCATGGGCTGGGAGGCCTCCTCCGAGCGGATG TACCCCGAGGACGGCGCCCTGAAGGGCGAGATCAAGCAGAGGCTGAAGCTGAAGGAC GGCGGCCACTACGACGCCGAGGTCAAGACCACCTACAAGGCCAAGAAGCCCGTGCAGC TGCCCGGCGCCTACAACGTCAACATCAAGCTGGACATCACCTCCCACAACGAGGACTA CACCATCGTGGAACAGTACGAGCGCGCCGAGGGCCGCCACTCCACCGGCGGCATGGAC GAGCTGTACAAGTAA
TheoRs	GGTGATACCAGCATCGTCTTGATGCCCTTGGCAGCACCCTGCTAAGGTAACAACAAGAT GAGTAAAGGAGAAGAACTTTTCACTGCTGGAGTTGTCCCAATTCTTGTTGAAT
GFP	ATGAGTAAAGGAGAAGAACTTTTCACTGGAGTTGTCCCAATTCTTGTTGAATTAGATGG TGATGTTAATGGGCACAAATTTTCTGTCAGTGGAGAGGGTGAAGGTGATGCAACATAC GGAAAACTTACCCTTAAATTTATTTGCACTACTGGAAAACTACCTGTTCCATGGCCAAC ACTTGTCACTACTTTCTCTTATGGTGTTCAATGCTTTTCCCGTTATCCGGATCATATGAA ACGGCATGACTTTTTCAAGAGTGCCATGCCCGAAGGTTATGTACAGGAACGCACTATAT CTTTCAAAGATGACGGGAACTACAAGACGCGTGCTGAAGTCAAGTTTGAAGGTGATAC CCTTGTTAATCGTATCGAGTTAAAAGGTATTGATTTTAAAGAAGATGGAAACATTCTCG GACACAAACTCGAGTACAACTATAACTCACACAATGTATACATCACGGCAGACAAACA AAAGAATGGAATCAAAGCTAACTTCAAAATTCGCCACAACATTGAAGATGGATCCGTT CAACTAGCAGACCATTATCAACAAAATACTCCAATTGGCGATGGCCCTGTCCTTTTACC AGACAACCATTACCTGTCGACACAATCTGCCCTTTCGAAAGATCCCAACGAAAAGCGT GACCACATGGTCCTTCTTGAGTTTGTAACTGCTGCTGGGATTACACATGGCATGGATGA GCTCTACAAATAA
pBR322	CTTATCATCGATAAGCTTTAATGCGGTAGTTTATCACAGTTAAATTGCTAACGCAGTCA GGCACCGTGTATGAAATCTAACAATGCGCTCATCGTCATCCTCGGCACCGTCACCCTGG ATGCTGTAGGCATAGGCTTGGTTATGCCGGTACTGCCGGGCCTCTTGCGGGATATCGTC CATTCCGACAGCATCGCCAGTCACTATGGCGTGCTGCTAGCGCTATATGCGTTGATGCA ATTTCTATGCGCACCCGTTCTCGGAGCACTGTCCGACCGCTTTGGCCGCCGCCCAGTCC TGCTCGCTTCGCTACTTGGAGCCACTATCGACTACGCGATCATGGCGACCACACCCGTC CTGTGGATCCTCTACGCCGGACGCATCGTGGCCGGCATCACCGGCGCCACAGGTGCGG

TTGCTGGCGCCTATATCGCCGACATCACCGATGGGGAAGATCGGGCTCGCCACTTCGGG
CTCATGAGCGCTTGTTTCGGCGTGGGTATGGTGGCAGGCCCCGTGGCCGGGGGACTGTT
GGGCGCCATCTCCTTGCATGCACCATTCCTTGCGGCGGCGGTGCTCAACGGCCTCAACC
TACTACTGGGCTGCTTCCTAATGCAGGAGTCGCATAAGGGAGAGCGTCGACCGATGCC
CTTGAGAGCCTTCAACCCAGTCAGCTCCTTCCGGTGGGCGCGGGGCATGACTATCGTCG
CCGCACTTATGACTGTCTTCTTTATCATGCAACTCGTAGGACAGGTGCCGGCAGCGCTC
TGGGTCATTTTCGGCGAGGACCGCTTTCGCTGGAGCGCGACGATGATCGGCCTGTCGCT
TGCGGTATTCGGAATCTTGCACGCCCTCGCTCAAGCCTTCGTCACTGGTCCCGCCACCA
AACGTTTCGGCGAGAAGCAGGCCATTATCGCCGGCATGGCGGCCGACGCGCTGGGCTA
CGTCTTGCTGGCGTTCGCGACGCGAGGCTGGATGGCCTTCCCCATTATGATTCTTCTCGC
TTCCGGCGGCATCGGGATGCCCGCGTTGCAGGCCATGCTGTCCAGGCAGGTAGATGAC
GACCATCAGGGACAGCTTCAAGGATCGCTCGCGGCTCTTACCAGCCTAACTTCGATCAT
TGGACCGCTGATCGTCACGGCGATTTATGCCGCCTCGGCGAGCACATGGAACGGGTTG
GCATGGATTGTAGGCGCCGCCCTATACCTTGTCTGCCTCCCCGCGTTGCGTCGCGGTGC
ATGGAGCCGGGCCACCTCGACCTGAATGGAAGCCGGCGGCACCTCGCTAACGGATTCA
CCACTCCAAGAATTGGAGCCAATCAATTCTTGCGGAGAACTGTGAATGCGCAAACCAA
CCCTTGGCAGAACATATCCATCGCGTCCGCCATCTCCAGCAGCCGCACGCGGCGCATCT
CGGGCAGCGTTGGGTCCTGGCCACGGGTGCGCATGATCGTGCTCCTGTCGTTGAGGACC
CGGCTAGGCTGGCGGGGTTGCCTTACTGGTTAGCAGAATGAATCACCGATACGCGAGC
GAACGTGAAGCGACTGCTGCTGCAAAACGTCTGCGACCTGAGCAACAACATGAATGGT
CTTCGGTTTCCGTGTTTCGTAAAGTCTGGAAACGCGGAAGTCAGCGCCCTGCACCATTA
TGTTCCGGATCTGCATCGCAGGATGCTGCTGGCTACCCTGTGGAACACCTACATCTGTA
TTAACGAAGCGCTGGCATTGACCCTGAGTGATTTTTCTCTGGTCCCGCCGCATCCATAC
CGCCAGTTGTTTACCCTCACAACGTTCCAGTAACCGGGCATGTTCATCATCAGTAACCC
GTATCGTGAGCATCCTCTCTCGTTTCATCGGTATCATTACCCCCATGAACAGAAATCCC
CCTTACACGGAGGCATCAGTGACCAAACAGGAAAAAACCGCCCTTAACATGGCCCGCT
TTATCAGAAGCCAGACATTAACGCTTCTGGAGAAACTCAACGAGCTGGACGCGGATGA
ACAGGCAGACATCTGTGAATCGCTTCACGACCACGCTGATGAGCTTTACCGCAGCTGCC
TCGCGCGTTTCGGTGATGACGGTGAAAACCTCTGACACATGCAGCTCCCGGAGACGGT
CACAGCTTGTCTGTAAGCGGATGCCGGGAGCAGACAAGCCCGTCAGGGCGCGTCAGCG
GGTGTTGGCGGGTGTCGGGGCGCAGCCATGACCCAGTCACGTAGCGATAGCGGAGTGT
ATACTGGCTTAACTATGCGGCATCAGAGCAGATTGTACTGAGAGTGCACCATATGCGGT
GTGAAATACCGCACAGATGCGTAAGGAGAAAATACCGCATCAGGCGCTCTTCCGCTTC
CTCGCTCACTGACTCGCTGCGCTCGGTCGTTCGGCTGCGGCGAGCGGTATCAGCTCACT
CAAAGGCGGTAATACGGTTATCCACAGAATCAGGGGATAACGCAGGAAAGAACATGT
GAGCAAAAGGCCAGCAAAAGGCCAGGAACCGTAAAAAGGCCGCGTTGCTGGCGTTTTT
CCATAGGCTCCGCCCCCCTGACGAGCATCACAAAAATCGACGCTCAAGTCAGAGGTGG
CGAAACCCGACAGGACTATAAAGATACCAGGCGTTTCCCCCTGGAAGCTCCCTCGTGC
GCTCTCCTGTTCCGACCCTGCCGCTTACCGGATACCTGTCCGCCTTTCTCCCTTCGGGAA
GCGTGGCGCTTTCTCATAGCTCACGCTGTAGGTATCTCAGTTCGGTGTAGGTCGTTCGCT
CCAAGCTGGGCTGTGTGCACGAACCCCCCGTTCAGCCCGACCGCTGCGCCTTATCCGGT
AACTATCGTCTTGAGTCCAACCCGGTAAGACACGACTTATCGCCACTGGCAGCAGCCAC
TGGTAACAGGATTAGCAGAGCGAGGTATGTAGGCGGTGCTACAGAGTTCTTGAAGTGG
TGGCCTAACTACGGCTACACTAGAAGGACAGTATTTGGTATCTGCGCTCTGCTGAAGCC
AGTTACCTTCGGAAAAAGAGTTGGTAGCTCTTGATCCGGCAAACAAACCACCGCTGGT
AGCGGTGGTTTTTTTGTTTGCAAGCAGCAGATTACGCGCAGAAAAAAAGGATCTCAAG
AAGATCCTTTGATCTTTTCTACGGGGTCTGACGCTCAGTGGAACGAAAACTCACGTTAA
GGGATTTTGGTCATGAGATTATCAAAAAGGATCTTCACCTAGATCCTTTTAAATTAAAA
ATGAAGTTTTAAATCAATCTAAAGTATATATGAGTAAACTTGGTCTGACAGTTACCAAT
GCTTAATCAGTGAGGCACCTATCTCAGCGATCTGTCTATTTCGTTCATCCATAGTTGCCT
GACTCCCCGTCGTGTAGATAACTACGATACGGGAGGGCTTACCATCTGGCCCCAGTGCT
GCAATGATACCGCGAGACCCACGCTCACCGGCTCCAGATTTATCAGCAATAAACCAGC
CAGCCGGAAGGGCCGAGCGCAGAAGTGGTCCTGCAACTTTATCCGCCTCCATCCAGTCT
ATTAATTGTTGCCGGGAAGCTAGAGTAAGTAGTTCGCCAGTTAATAGTTTGCGCAACGT
TGTTGCCATTGCTGCAGGCATCGTGGTGTCACGCTCGTCGTTTGGTATGGCTTCATTCAG
CTCCGGTTCCCAACGATCAAGGCGAGTTACATGATCCCCCATGTTGTGCAAAAAAGCG

GTTAGCTCCTTCGGTCCTCCGATCGTTGTCAGAAGTAAGTTGGCCGCAGTGTTATCACT
CATGGTTATGGCAGCACTGCATAATTCTCTTACTGTCATGCCATCCGTAAGATGCTTTTC
TGTGACTGGTGAGTACTCAACCAAGTCATTCTGAGAATAGTGTATGCGGCGACCGAGTT
GCTCTTGCCCGGCGTCAACACGGGATAATACCGCGCCACATAGCAGAACTTTAAAAGT
GCTCATCATTGGAAAACGTTCTTCGGGGCGAAAACTCTCAAGGATCTTACCGCTGTTGA
GATCCAGTTCGATGTAACCCACTCGTGCACCCAACTGATCTTCAGCATCTTTTACTTTCA
CCAGCGTTTCTGGGTGAGCAAAAACAGGAAGGCAAAATGCCGCAAAAAAGGGAATAA
GGGCGACACGGAAATGTTGAATACTCATACTCTTCCTTTTTCAATATTATTGAAGCATTT
ATCAGGGTTATTGTCTCATGAGCGGATACATATTTGAATGTATTTAGAAAAATAAACAA
ATAGGGGTTCCGCGCACATTTCCCCGAAAAGTGCCACCTGACGTCTAAGAAACCATTAT
TATCATGACATTAACCTATAAAAATAGGCGTATCACGAGGCCCTTTCGTCTTCAAG

For this assembly bundle Gibson, Golden Gate, and PCR-SOE methods were selected. The title for the assembly was *LacO Riboswitch Bundle* and a description of *Thesis Case Study 2*.

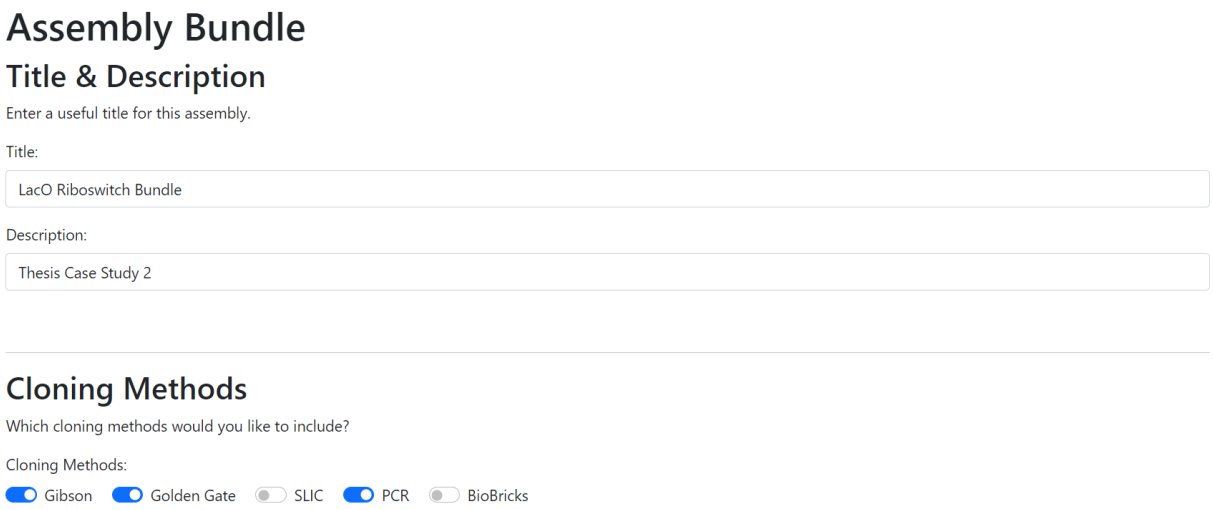


Figure 39: Beginning form entries for the assembly bundle: title, description, and method selection

A multi-entry query solution was selected, the insert sequences (LacO, mCherry, theoRs, and GFP) were contained in *theoRs_LacO.fasta*, and the backbone sequence contained in *pBR322.fasta*.

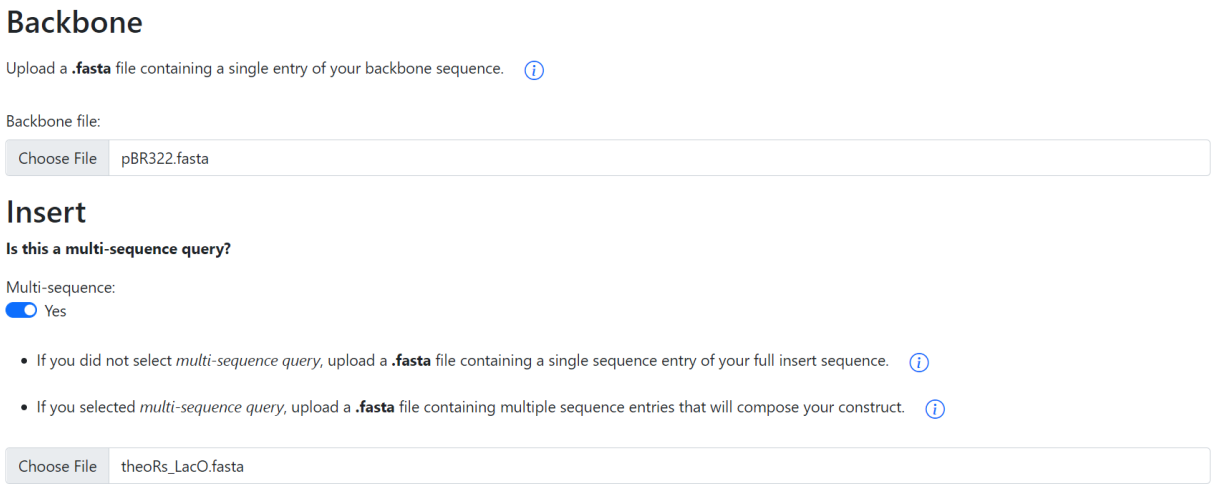


Figure 40: Backbone and insert sequence inputs for the bundle

All three BLAST databases were selected: AddGene, DNASU, and iGEM. As with Case Study 1, a minimum BLAST sequence size of 50 nt, a maximum BLAST sequence size of 5000 nt, minimum synthetic sequence size of 50 nt, and a maximum synthetic sequence size of 5000 nt were entered for the *BLAST Query* inputs of the bundle form. Default *Experiment* inputs were used for this assembly bundle: monovalent ion concentration of 50 nM, divalent ion concentration of 1.5 mM, a 0.8 mM dNTP concentration, 50 nM DNA oligo concentration, and a 60 °C target melting temperature. Assembly cost estimate parameters (Section 3.2.5) were $0.80 for short sequences, for medium sequences $0.39, and for long sequences $0.65. Enzyme prices were entered at $125. Form components for the BLAST query, experiment, and nucleotide inputs are identical to those Case Study 1 and of individual assemblies.

The selection of the three cloning methods at the beginning of the form (Figure 37), reveals the form components for each of these methods, including their unique inputs. Gibson parameters were then entered for an overlap length of 30 nt, and enzyme costs of $125 for exonuclease, ligase, and polymerase.

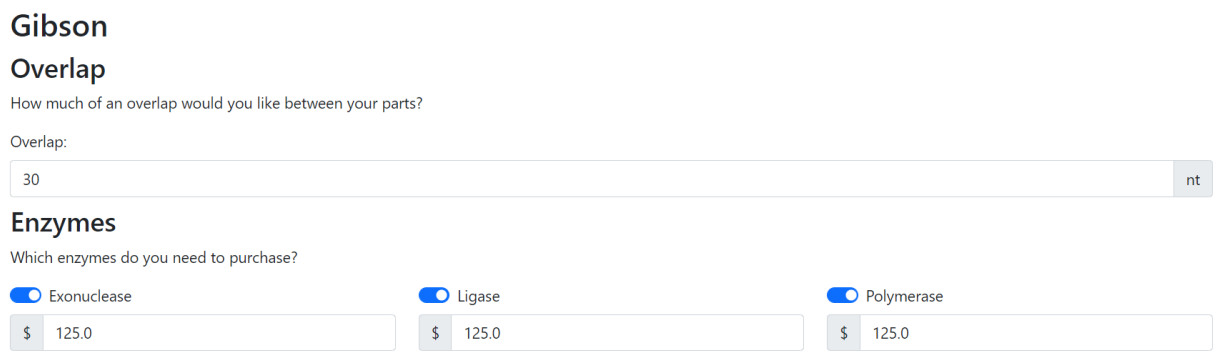


Figure 41: Gibson inputs for the bundle

With the Gibson component of the form completed, the Golden Gate inputs were entered. The 98.5% fidelity overhang set[47] and scar-less primer design were selected. Enzyme costs for BsaI and ligase were entered at $125.

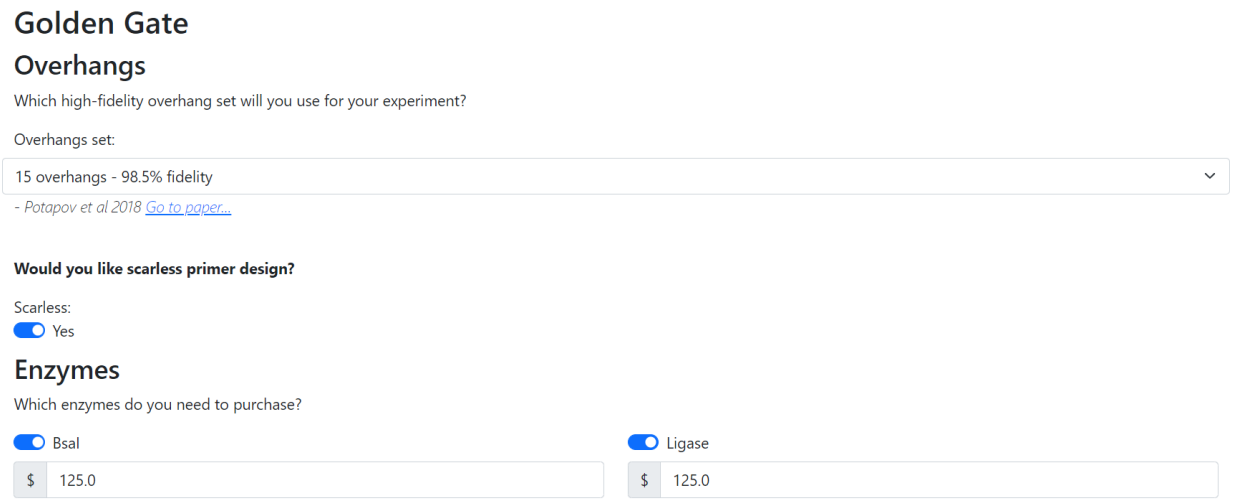


Figure 42: Golden Gate inputs for the bundle

Inputs for PCR-SOE were then entered. This cloning method uses the same overlap amount as Gibson assemblies: 30 nt. The cost for polymerase was entered at $125.

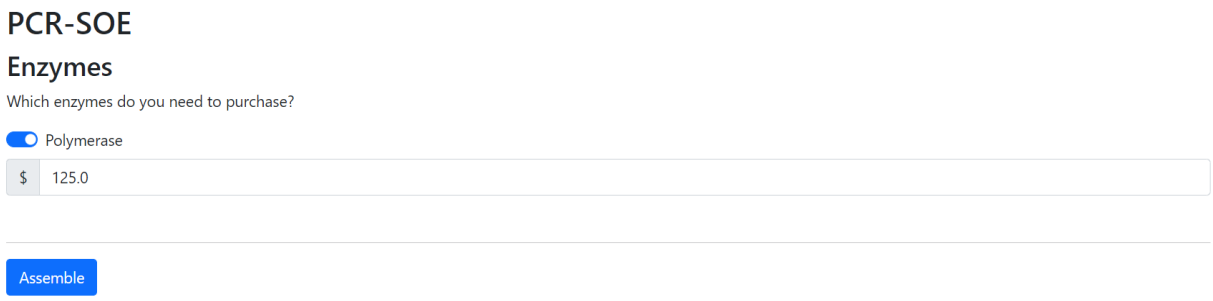


Figure 43: Gibson inputs for the bundle

The bundle design procedure began when the *Assemble* button was clicked.

4.9.2 Process

A shared multi-entry BLAST query was performed for the bundle insert sequences, following Section 3.2.4.1.3. Each of the sequences except the modified riboswitch were found in the public databases. The count of BLAST sequence alignments, for each part, coming from the three databases were: 2231for LacO, 1726 for mCherry, no alignments for the riboswitch, and 335 alignments for the GFP. For the particular solution of this case study, the BLAST alignment names for the insert sequences are: 120399-addgene for LacO, 37760.1-addgene for mCherry, Synthetic-1.112 for the riboswitch (not found in BLAST databases), 99831-addgene for GFP. These sequence query results were used for each of the Gibson, Golden Gate, and PCR-SOE assembly designs.

With these BLAST sequences, Gibson primers were designed. Primer complementary sequences for the insert and backbone sequences were created using a target melting temperature of 60 °C (Section 3.2.4.3). Then, extension sequences were generated for a 30 nt combined overlap between each of the assembly's pairs of contiguous sequences (Section 3.2.4.4).

Table 10: Complementary and extension primer sequences for the bundle's Gibson assembly

Primer	Complementary (footprint) Sequence	Extension (tail) Sequence
120399-addgene-fwd	gtgagctgataccgct	cctttcgtcttcaag
120399-addgene-rvs	gtcgtgccagctgc	gcccttgctcaccat
37760.1-addgene-fwd	atggtgagcaagggc	tgcagctggcacgac
37760.1-addgene-rvs	ttacttgtacagctcgtcc	gatgctggtatcacc
Synthetic-1.112-fwd	ggtgataccagcatcg	gagctgtacaagtaa
Synthetic-1.112-rvs	attcaacaagaattgggaca	ttctcctttactcat
99831-addgene-fwd	atgagtaaaggagaagaactttt	caattcttgttgaat
99831-addgene-rvs	ttatttgtagagctcatccat	cttatcgatgataag
pBR322-fwd	cttatcatcgataagctttaatgc	gagctctacaaataa
pBR322-rvs	cttgaagacgaaagggc	gcggtatcagctcac

Next, Golden Gate assembly primers were designed. Primer complementary sequences for the insert and backbone sequences were created using a target melting temperature of 60 °C (Section 3.2.4.3). Then, extension sequences were generated for scar-less overhang ligation between each of the assembly's sequences (Section 3.2.4.5.2).

Table 11: Complementary and extension primer sequences for the bundle's Golden Gate assembly

Primer	Complementary (footprint) Sequence	Extension (tail) Sequence
120399-addgene-fwd	cgaacgaccgagcg	ggtctcn
120399-addgene-rvs	gaatcggccaacgc	ggtctcn
37760.1-addgene-fwd	atggtgagcaagggc	ggtctcnattcattaatgcagctggcacgac

37760.1-addgene-rvs	cggttgagtggcgg	ggtctcn
Synthetic-1.112-fwd	ggtgataccagcatcg	ggtctcnaccggcggcatggacgagctgtacaagtaa
Synthetic-1.112-rvs	attcaacaagaattgggaca	ggtctcncctttactcat
99831-addgene-fwd	aaggagaagaacttttcactg	ggtctcn
99831-addgene-rvs	gctcatccatgccatg	ggtctcn
pBR322-fwd	cttatcatcgataagctttaatgc	ggtctcngagctctacaaataa
pBR322-rvs	cttgaagacgaaagggc	ggtctcnttcggctgcggcgagcggtatcagctcac

Lastly, PCR primers were designed. Primer complementary sequences for the insert and backbone sequences were created using a target melting temperature of 60 °C (Section 3.2.4.3). Then, overlap extension sequences were generated for 30 nt overlaps between each of the assembly's sequences, identical to the Gibson assembly (Section 3.2.4.4).

Table 12: Complementary and extension primer sequences for the bundle's PCR assembly

Primer	**Complementary (footprint) Sequence**	**Extension (tail) Sequence**
120399-addgene-fwd	gtgagctgataccgct	cctttcgtcttcaag
120399-addgene-rvs	gtcgtgccagctgc	gcccttgctcaccat
37760.1-addgene-fwd	atggtgagcaagggc	tgcagctggcacgac
37760.1-addgene-rvs	ttacttgtacagctcgtcc	gatgctggtatcacc
Synthetic-1.112-fwd	ggtgataccagcatcg	gagctgtacaagtaa
Synthetic-1.112-rvs	attcaacaagaattgggaca	ttctcctttactcat
99831-addgene-fwd	atgagtaaaggagaagaactttt	caattcttgttgaat
99831-addgene-rvs	ttatttgtagagctcatccat	cttatcgatgataag
pBR322-fwd	cttatcatcgataagctttaatgc	gagctctacaaataa

pBR322-rvs	cttgaagacgaaagggc	gcggtatcagctcac

For each of these individual assemblies, *Smithy* computes their cost, time, and risk estimates separately and displays them centrally in the assembly bundle dashboard.

4.9.3 Output

The design procedures finish by bringing the user to the dashboard page for the assembly bundle. First, the bundle's descriptive information is shown for: title, date created, and description (Figure 42). From this dashboard page, the bundle charts are seen, each assembly is accessible through links, and each of the assemblies' solutions are directly linked for convenience.

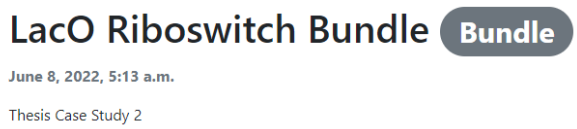


Figure 44: The title section of the assembly bundle dashboard

The time, cost, and risk estimates charts are the main feature of the dashboard, and they are displayed as the first components of the page (Figure 44). The horizontal axes of the time charts have a maximum value equal to the greatest time estimate of all the assemblies. For the Gibson assembly, the time estimate was 16.22 hrs: 15.22 hrs for PCR, 1 hr for assembly. The Golden Gate time estimate was 16.32 hrs: 15.22 hrs for PCR, 1.1 hrs for assembly. Time for PCR-SOE was 15.22 hrs for PCR reactions only. The cost values are normalized, to display the costs relative to the highest cost estimate. The total cost for the Gibson assembly was $911.68, for Golden Gate $798.68 and for PCR-SOE $661.68. Risk estimates for Gibson assembly were -0.602 for PCR and -0.368 for assembly (chewback, ligation, and repair). Golden Gate risk values were -0.602 for PCR and -0.954 for assembly digestion-ligation. Risk values for the PCR-SOE assembly were -0.602 for PCR (Section 3.2.5).

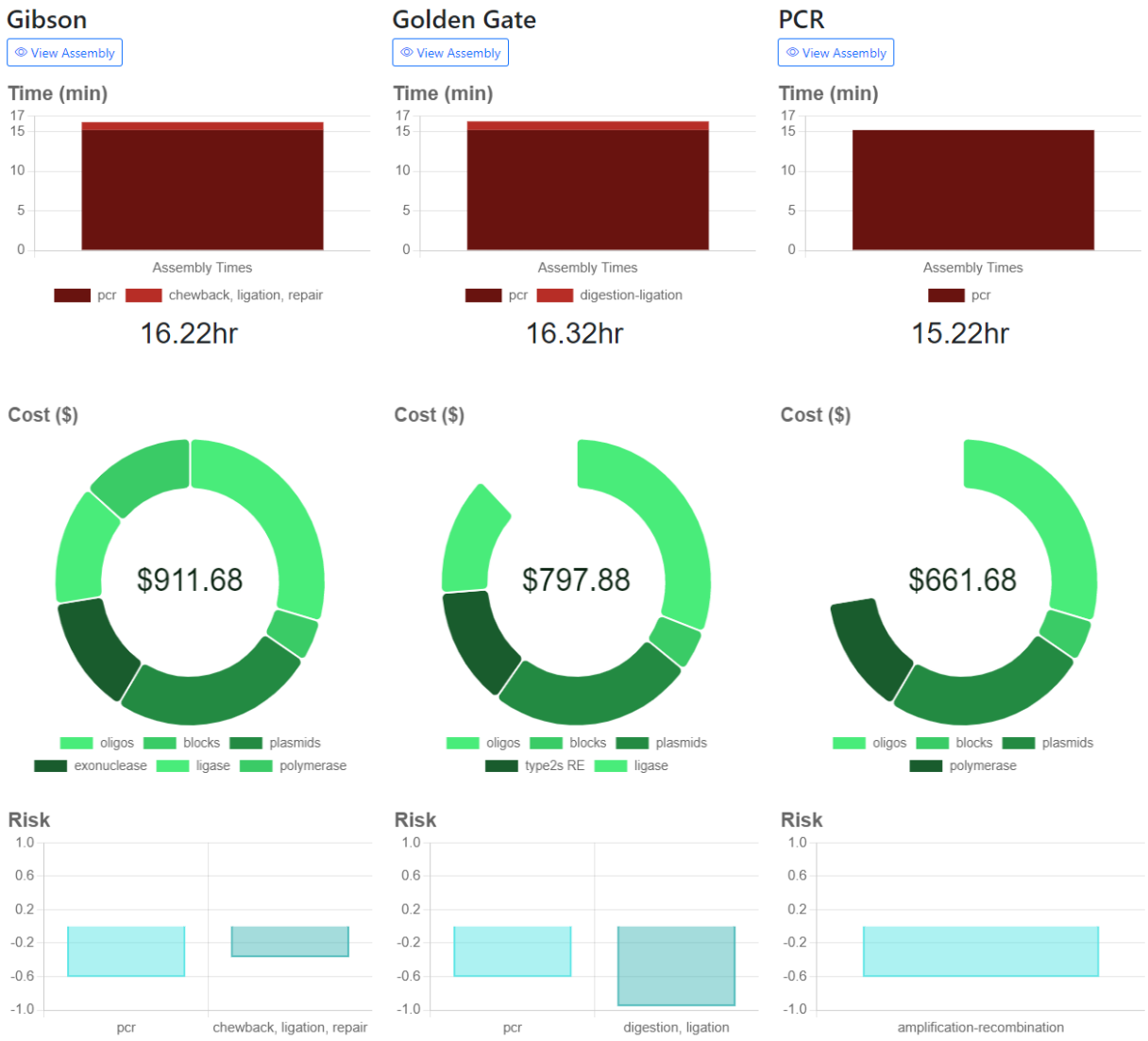


Figure 45: Time, cost, and risk charts of the bundle dashboard for Gibson, Golden Gate, and PCR-SOE assemblies

Comparison of each assembly's solution using these charts is what allows users to make informed decisions about the most suitable assembly method for their particular needs and background. The solution details for each assembly method are accessible directly below the charts display. The format of these pages is identical to those seen in Case Study 1. Each solution detail page provides all the information on the solution, with individual part and primer detail pages accessible via its components. Golden Gate solutions show a yellow badge indicating the presence of internal restriction enzyme cut-sites in the assembly sequences. In each of the solution detail pages, the parts, primers, and orders files for the assembly can be downloaded. These files are where users obtain full sequences for the insert fragments, sequences for each primer with its thermodynamic properties, and order files of materials to purchase for actual wet lab assembly.

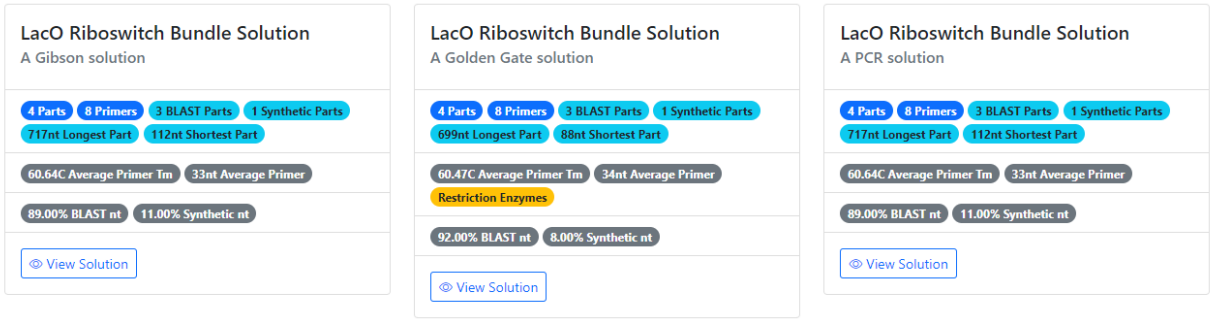


Figure 46: The Gibson, Golden Gate, and PCR-SOE solution components of the dashboard providing access to the individual solution pages

Table 13: Complete primer designs with T_m for all sequences in the bundle's Gibson assembly

Primer	Sequence	T_m
120399-addgene-fwd	cctttcgtcttcaaggtgagctgataccgct	60.47
120399-addgene-rvs	gcccttgctcaccatgtcgtgccagctgc	61.68
37760.1-addgene-fwd	tgcagctggcacgacatggtgagcaagggc	62.01

37760.1-addgene-rvs	gatgctggtatcaccttacttgtacagctcgtcc	61.92
Synthetic-1.112-fwd	gagctgtacaagtaaggtgataccagcatcg	59.47
Synthetic-1.112-rvs	ttctcctttactcatattcaacaagaattgggaca	60.13
99831-addgene-fwd	caattcttgttgaatatgagtaaaggagaagaactttt	60.17
99831-addgene-rvs	cttatcgatgataagttatttgtagagctcatccat	59.62
pBR322-fwd	gagctctacaaataacttatcatcgataagctttaatgc	60.80
pBR322-rvs	gcggtatcagctcaccttgaagacgaaagggc	60.06

Table 14: Complete primer designs with T_m for all sequences in the bundle's Golden Gate assembly

Primer	**Sequence**	**T_m**
120399-addgene-fwd	ggtctcncgaacgaccgagcg	60.90
120399-addgene-rvs	ggtctcngaatcggccaacgc	59.84
37760.1-addgene-fwd	ggtctcnattcattaatgcagctggcacgacatggtgagcaagggc	62.01
37760.1-addgene-rvs	ggtctcncggttgagtggcgg	62.66
Synthetic-1.112-fwd	ggtctcnaccggcggcatggacgagctgtacaagtaaggtgataccagcatcg	59.47
Synthetic-1.112-rvs	ggtctcnccttactcatattcaacaagaattgggaca	60.13
99831-addgene-fwd	ggtctcnaaggagaagaacttttcactg	59.78
99831-addgene-rvs	ggtctcngctcatccatgccatg	59.84
pBR322-fwd	ggtctcngagctctacaaataacttatcatcgataagctttaatgc	60.80
pBR322-rvs	ggtctcnttcggctgcggcgagcggtatcagctcaccttgaagacgaaagggc	60.06

Table 15: Complete primer designs with T_m for all sequences in the bundle's PCR assembly

Primer	Sequence	T_m
120399-addgene-fwd	cctttcgtcttcaaggtgagctgataccgct	60.47
120399-addgene-rvs	gcccttgctcaccatgtcgtgccagctgc	61.68
37760.1-addgene-fwd	tgcagctggcacgacatggtgagcaagggc	62.01
37760.1-addgene-rvs	gatgctggtatcaccttacttgtacagctcgtcc	61.92
Synthetic-1.112-fwd	gagctgtacaagtaaggtgataccagcatcg	59.47
Synthetic-1.112-rvs	ttctcctttactcatattcaacaagaattgggaca	60.13
99831-addgene-fwd	caattcttgttgaatatgagtaaaggagaagaactttt	60.17
99831-addgene-rvs	cttatcgatgataagttatttgtagagctcatccat	59.62
pBR322-fwd	gagctctacaaataacttatcatcgataagctttaatgc	60.80
pBR322-rvs	gcggtatcagctcaccttgaagacgaaagggc	60.06

4.9.4 Summary

This second case study example demonstrates Smithy's bundle features for generating solutions for multiple user-selected assembly methodologies, for one construct. Three independent assembly solutions were generated, each with comprehensive data, with the addition of cost, time, and risk estimates in the bundle dashboard. This allows quick visual inspection and evaluation of the pros and cons (from the user's perspective) of using different methods of assembly (i.e., Gibson, Golden Gate, and PCR-SOE) of the final construct. As such, the dashboard feature facilitates selection of the most appropriate assembly method by and for a given user.

4.10 Discussion

Smithy builds and improves on previous DNA cloning automation design tools in synthetic biology for new and advanced researchers in DNA cloning. Given the way *Smithy*'s core *Assembler* has

been designed for abstraction and extendibility, *Smithy* does not only have wide appeal but is also flexible enough to handle many future demands, such as new assembly methods or improvements to existing method. The assembly bundle feature is not seen in the other assembly design tools and aids synthetic biologists in choosing the most appropriate assembly method for them. Utilization of popular DNA sequence repositories implemented as BLAST databases, especially with the capacity to host user lab sequence datasets, ensures the minimization of synthetic sequence orders, which reduces overall cost. Smithy is open source, and its portability provides opportunities for its core features (Section 3.2, Section 4.7) to be useful for a diverse array of other projects and research endeavors.

The cloning method taxonomy and resulting *Assembler* class hierarchy generated out of it provide a new way to view relationships between the contemporary cloning methods and how these methodologies can be implemented in code for design tools. New and modified cloning methods can be categorized according to the taxonomic groups (Figure 12) based on their approach to DNA sequence recombination. Then, if a new *Assembler* class is needed for generating designs for these novel methodologies, its definition in software can be guided by the *Assembler* class hierarchy for optimal development. For example, GoldenBraid cloning[19,20] is not yet implemented in *Smithy* but is related to the Golden Gate and restriction endonuclease (RE) assembly methods. When designing a new *Assembler* class for GoldenBraid it could be defined as a child class of the *TraditionalREAssembler* class to limit the amount of new code to any necessary attributes and functions not already seen in the parent class. This dynamic of the *Smithy Assembler* classes shows how experimental methodologies in biology can be abstracted for translation to software methodologies.

The implementation of this project in Python provides a way for other researchers and developers to make use of it or expand and improve upon it. Python is already a popular programming language with a multitude of different applications in industry and academia with a generally smooth learning curve. There also exist a vast amount of code libraries, which can be incorporated into this project for any new or improved features, for new assembly design automation or statistical computations. For researchers and programmers, *Smithy* exists as a readily accessible tool for direct use in other projects, or for expansions and/or modification.

A comprehensive set of solutions is generated for each assembly by the design software, often numbering in the hundreds, which is advantageous for creating several options for assembly subsequence sets. However, a limitation arises from the fact that *Smithy* in its current form only uses the first, most optimal solution created by the fragment solution tree. This limits the solution generation, as other valid but sub-optimal (from *Smithy's* perspective) may be more optimal for a given user, in a given context. Conversely, although only one solution at the present time is being used, it is the best solution found given the set of BLAST results that the software produces. Solutions also benefit from the thermodynamic parameters computed for each part's forward and reverse primers (via the Primer3 python package)[39]. The full suite of thermodynamic results of each primer is calculated and provided by *Smithy* and accessible in detail on the site webpages and downloadable in a primers CSV file (Section 3.3.4). If a particular aspect of a primer or group of primers appears problematic, edits to the primer sequences can be executed or, if necessary, a complete change to the assembly approach can be adopted.

The assembly bundle feature is another way in which *Smithy* provides users with opportunities to make informed decisions, based on experience and preferences, about various assembly projects through comparison of different candidate methodologies for a single insert solution. This way of

presenting candidate assemblies through a dashboard where an optimal choice is made based on the dimensions of cost, time, and risk of the methodology has not been seen in previous studies and tools. Limitations of the bundle dashboard features exist for aspects of the calculations. The time estimates are simple projections based on published method protocols, strictly based on listed procedures, and do not account for times to perform experiments within a particular lab or for specific overhead times a user may experience. These values are ideal times assuming success and efficiency in the experiment without unforeseen problems. To ameliorate that optimistic prediction, risk is also calculated; these reflect the possibility of failure of certain types of experiments and hence, the need for experiment repetition (more time). However, risk estimates are based on subjective and anecdotal experience-based determinations from a few experienced colleagues. A more robust estimate for experimental probability of success would be to obtain and/or test for general expectations so that a more complete statistical model could be developed. Finally, cost values are a rough estimate for assembly materials. Sequence costs are determined by flat per-nucleotide costs for categories of nucleotide size and provided by users. More accurate estimates for DNA synthesis could be used by obtaining actual provider costs. Repository plasmid costs are also given a flat value of $75 when different repositories could have varied listing prices. These bundle dashboard features have several opportunities for improvement that would strengthen *Smithy's* capabilities of aiding researchers in making optimal decisions for their assembly projects.

Chapter 5: Conclusion and Future Work

Smithy is an extensible, abstracted DNA assembly design automation tool that facilitates more reliable target construct design and creation. It is based on existing theoretical and practical knowledge of DNA cloning, along with inspiration from previous DNA cloning design automation efforts. As a web application, it is widely accessible to any researcher, experienced or novice. For DNA assembly projects, Smithy decreases the risk of error when working with large sets of DNA sequences for complex cloning projects. Additionally, comprehensive solutions are generated that contain several dimensions of analytical calculations: thorough primer thermodynamics with assembly cost, time, and risk estimates for major experimental procedures. Furthermore, a comparison of different assembly options for single target constructs, utilizing the cost, time, and risk assessments, is available through the assembly bundling capabilities of *Smithy*. In sum, the combined features of *Smithy* facilitate optimal, informed decision-making for DNA cloning assemblies, and the two case studies presented exemplify its practicality as a tool in the field of synthetic biology. Finally, as the fields of synthetic biology and DNA assembly grow and improve, *Smithy* has been designed to adapt to novel contributions in these fields.

Several opportunities for expansion and improvement on *Smithy* exist. To start, unsupported DNA assembly methods could be added, and two are immediately clear: MoClo[7] and GoldenBraid[19,20] assemblies. These two methodologies implement strategies for achieving modular, high throughput assembly projects with clear parts standardization and specification. Incorporation of these would make *Smithy* a much more useful, broadly applicable tool for large DNA cloning assemblies. The way these could be added would be by creating new *Assembler* classes that extend the *GoldenGateAssembler* class: *MoCloAssembler* and *GoldenBraidAssembler*. These two *Assembler* classes would need to carefully implement the hierarchical assembly strategies of these

two methods, because MoClo operates on hierarchical plasmid extension with standardized entry vectors and GoldenBraid with cycling of the target construct through two different sets of assembly plasmids[7,19,20]. The addition of user profiles would allow users of the application to control access to their project, providing exclusivity to assembly solutions. Next, addition of local lab sequence databases, queryable only by members of the lab, would expand on the sequences queryable by *Smithy's* BLAST capabilities to incorporate into assembly solutions sequences that are directly available and not requiring purchase. Expanding the BLAST databases in this way would provide opportunities for decreased cost of assembly projects. Additionally, adding local lab sequences further builds the case for user profiles, because selection of these lab databases would only be available to users associated with the lab.

Several improvements to this project can also be made on various fronts. A better model for risk assessment could be developed that incorporates more extensive data about the odds of failure of assembly experiment types. For example, a side project could be made to perform several trials of each major experiment in an assembly (e.g., chewback, ligation, and repair for Gibson assemblies or digestion-ligation for Golden Gate assemblies) of the supported methods in *Smithy* to thoroughly evaluate the odds of success and failure for these experiments. Additionally, risk values could be crowd-sourced from the scientific community of authenticated cloners, and additional inputs to the assemblies in *Smithy* (after the experiments are complete) could be added that collect information about the assembly reactions' successes. This data collection could be used to grow a record of risk data that could be utilized in an evolving model to calculate constantly improving risk estimates for assemblies. These efforts would make the risk estimates more scientifically sound and reliable. Cost estimates for solutions are also open to improvement. Thus far, only three sequence size costs are implemented, however these could be significantly expanded, for a more

thorough breakdown of DNA synthesis costs. For example, the company Integrated DNA Technologies (IDT) has three different cost breakdowns for short oligos for various sizes and concentrations, and, in addition, a more detailed breakdown of larger DNA sequence synthesis costs in ranges of 25-500 nt, 501-1500 nt, 1501-3000 nt, 3001-5000 nt, and 5000+ nt[54]. Expanding the cost input fields in this way would allow *Smithy* to provide more accurate assembly cost estimates. Furthermore, a feature could be added to the software to automatically obtain current prices for DNA synthesis and suggest them to the user at the assembly creation forms. The estimates for assembly experiment times could also be improved. While the times are calculated following published methodologies for the supported cloning methods, much like the suggestion for improving risk estimates, an extensive suite of experiments could be performed to collect detailed data about actual expected times for the major assembly procedures in a project. Crowd-sourced data could also be collected, and actual experimental times inputted to *Smithy* for particular assemblies that add to an ever-improving model that calculates expected assembly times. Adding to *Smithy* in these ways would improve its abilities to facilitate informed and optimized decisions for assembly projects by researchers using the application.

If possible, parallelization of certain procedures of the software would decrease the times for generating assembly solutions. Currently, the BLAST and assembly bundle procedures perform their individual tasks (querying databases in BLAST or creating multiple assemblies with the bundle) in series. Though if querying BLAST databases and creating the individual bundle assemblies could be run as parallel and independent processes the overall solution generation time of *Smithy* could significantly improve, especially for BLAST queries as these consume the most time.

Finally, the assembly solutions designed by *Smithy* can be translated into the Synthetic Biology Open Language standard (SBOL)[22] which provides a framework for standardization of synthetic biology project designs that are broadly transferrable across applications. There also exists a Python implementation of the SBOL framework with a project called pySBOL[55]. This project is significant because it could be integrated into *Smithy* with ease as both projects are implemented in Python. Usage of SBOL in *Smithy* would broaden its capabilities as a practical tool in the synthetic biology field as its assembly designs could be contained in a standardized format that could be transferred to other modernized laboratory workflows, industrial applications, and software tools[55]. This potential for standardization is especially notable for *Smithy* as DNA cloning methodologies and projects continue to grow in scale and complexity.

References

(1) *PCR Cycling Parameters—Six Key Considerations for Success - CA*. https://www.thermofisher.com/ca/en/home/life-science/cloning/cloning-learning-center/invitrogen-school-of-molecular-biology/pcr-education/pcr-reagents-enzymes/pcr-cycling-considerations.html (accessed 2022-05-30).

(2) Lorenz, T. C. Polymerase Chain Reaction: Basic Protocol Plus Troubleshooting and Optimization Strategies. *J. Vis. Exp.* **2012**, No. 63, 3998. https://doi.org/10.3791/3998.

(3) Bertero, A.; Brown, S.; Vallier, L. Methods of Cloning. In *Basic Science Methods for Clinical Researchers*; Elsevier, 2017; pp 19–39. https://doi.org/10.1016/B978-0-12-803077-6.00002-3.

(4) Casini, A.; Storch, M.; Baldwin, G. S.; Ellis, T. Bricks and Blueprints: Methods and Standards for DNA Assembly. *Nat. Rev. Mol. Cell Biol.* **2015**, *16* (9), 568–576. https://doi.org/10.1038/nrm4014.

(5) Gibson, D. G.; Young, L.; Chuang, R.-Y.; Venter, J. C.; Hutchison, C. A.; Smith, H. O. Enzymatic Assembly of DNA Molecules up to Several Hundred Kilobases. *Nat. Methods* **2009**, *6* (5), 343–345. https://doi.org/10.1038/nmeth.1318.

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

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

(8) Li, M. Z.; Elledge, S. J. Harnessing Homologous Recombination in Vitro to Generate Recombinant DNA via SLIC. *Nat. Methods* **2007**, *4* (3), 251–256. https://doi.org/10.1038/nmeth1010.

(9) Li, M. Z.; Elledge, S. J. SLIC: A Method for Sequence- and Ligation-Independent Cloning. In *Gene Synthesis*; Peccoud, J., Ed.; Methods in Molecular Biology; Humana Press: Totowa, NJ, 2012; Vol. 852, pp 51–59. https://doi.org/10.1007/978-1-61779-564-0_5.

(10) Knight, T. Idempotent Vector Design for Standard Assembly of Biobricks. *MIT Artif. Intell. Lab. MIT Synth. Biol. Work. Group* **2003**.

(11) Shetty, R. P.; Endy, D.; Knight, T. F. Engineering BioBrick Vectors from BioBrick Parts. *J. Biol. Eng.* **2008**, *2* (1), 5. https://doi.org/10.1186/1754-1611-2-5.

(12) Horton, R. M.; Hunt, H. D.; Ho, S. N.; Pullen, J. K.; Pease, L. R. Engineering Hybrid Genes without the Use of Restriction Enzymes: Gene Splicing by Overlap Extension. *Gene* **1989**, *77* (1), 61–68. https://doi.org/10.1016/0378-1119(89)90359-4.

(13) Horton, R. M.; Cai, Z.; Ho, S. N.; Pease, L. R. Gene Splicing by Overlap Extension: Tailor-Made Genes Using the Polymerase Chain Reaction. *BioTechniques* **2013**, *54* (3), 129–133. https://doi.org/10.2144/000114017.

(14) Kadkhodaei, S.; Memari, H. R.; Abbasiliasi, S.; Rezaei, M. A.; Movahedi, A.; Shun, T. J.; Ariff, A. B. Multiple Overlap Extension PCR (MOE-PCR): An Effective Technical Shortcut to High Throughput Synthetic Biology. *RSC Adv.* **2016**, *6* (71), 66682–66694. https://doi.org/10.1039/C6RA13172G.

(15) Müller, K. M.; Arndt, K. M. Standardization in Synthetic Biology. In *Synthetic Gene Networks: Methods and Protocols*; Weber, W., Fussenegger, M., Eds.; Methods in Molecular Biology; Humana Press: Totowa, NJ, 2012; pp 23–43. https://doi.org/10.1007/978-1-61779-412-4_2.
(16) Chao, R.; Yuan, Y.; Zhao, H. Building Biological Foundries for Next-Generation Synthetic Biology. *Sci. China Life Sci.* **2015**, *58* (7), 658–665. https://doi.org/10.1007/s11427-015-4866-8.
(17) Ashwini, M.; Murugan, S. B.; Balamurugan, S.; Sathishkumar, R. Advances in Molecular Cloning. *Mol. Biol.* **2016**, *50* (1), 1–6. https://doi.org/10.1134/S0026893316010131.
(18) Ellis, T.; Adie, T.; Baldwin, G. S. DNA Assembly for Synthetic Biology: From Parts to Pathways and Beyond. *Integr. Biol.* **2011**, *3* (2), 109–118. https://doi.org/10.1039/c0ib00070a.
(19) Sarrion-Perdigones, A.; Falconi, E. E.; Zandalinas, S. I.; Juárez, P.; Fernández-del-Carmen, A.; Granell, A.; Orzaez, D. GoldenBraid: An Iterative Cloning System for Standardized Assembly of Reusable Genetic Modules. *PLoS ONE* **2011**, *6* (7), e21622. https://doi.org/10.1371/journal.pone.0021622.
(20) Sarrion-Perdigones, A.; Vazquez-Vilar, M.; Palaci, J.; Castelijns, B.; Forment, J.; Ziarsolo, P.; Blanca, J.; Granell, A.; Orzaez, D. GoldenBraid 2.0: A Comprehensive DNA Assembly Framework for Plant Synthetic Biology. *PLANT Physiol.* **2013**, *162* (3), 1618–1631. https://doi.org/10.1104/pp.113.217661.
(21) Chao, R.; Yuan, Y.; Zhao, H. Recent Advances in DNA Assembly Technologies. *FEMS Yeast Res.* **2014**, n/a-n/a. https://doi.org/10.1111/1567-1364.12171.
(22) Galdzicki, M.; Clancy, K. P.; Oberortner, E.; Pocock, M.; Quinn, J. Y.; Rodriguez, C. A.; Roehner, N.; Wilson, M. L.; Adam, L.; Anderson, J. C.; Bartley, B. A.; Beal, J.; Chandran, D.; Chen, J.; Densmore, D.; Endy, D.; Grünberg, R.; Hallinan, J.; Hillson, N. J.; Johnson, J. D.; Kuchinsky, A.; Lux, M.; Misirli, G.; Peccoud, J.; Plahar, H. A.; Sirin, E.; Stan, G.-B.; Villalobos, A.; Wipat, A.; Gennari, J. H.; Myers, C. J.; Sauro, H. M. The Synthetic Biology Open Language (SBOL) Provides a Community Standard for Communicating Designs in Synthetic Biology. *Nat. Biotechnol.* **2014**, *32* (6), 545–550. https://doi.org/10.1038/nbt.2891.
(23) Cohen, S. N.; Chang, A. C. Y.; Boyer, H. W.; Helling, R. B. Construction of Biologically Functional Bacterial Plasmids In Vitro. *Proc. Natl. Acad. Sci.* **1973**, *70* (11), 3240–3244. https://doi.org/10.1073/pnas.70.11.3240.
(24) Shevchuk, N. A. Construction of Long DNA Molecules Using Long PCR-Based Fusion of Several Fragments Simultaneously. *Nucleic Acids Res.* **2004**, *32* (2), 19e–119. https://doi.org/10.1093/nar/gnh014.
(25) Cock, P. J. A.; Antao, T.; Chang, J. T.; Chapman, B. A.; Cox, C. J.; Dalke, A.; Friedberg, I.; Hamelryck, T.; Kauff, F.; Wilczynski, B.; de Hoon, M. J. L. Biopython: Freely Available Python Tools for Computational Molecular Biology and Bioinformatics. *Bioinformatics* **2009**, *25* (11), 1422–1423. https://doi.org/10.1093/bioinformatics/btp163.
(26) *BLAST® Command Line Applications User Manual*; National Center for Biotechnology Information (US), 2008.
(27) Pereira, F.; Azevedo, F.; Carvalho, Â.; Ribeiro, G. F.; Budde, M. W.; Johansson, B. Pydna: A Simulation and Documentation Tool for DNA Assembly Strategies Using Python. *BMC Bioinformatics* **2015**, *16* (1), 142. https://doi.org/10.1186/s12859-015-0544-x.

(28) Hillson, N. J.; Rosengarten, R. D.; Keasling, J. D. J5 DNA Assembly Design Automation Software. *ACS Synth. Biol.* **2012**, *1* (1), 14–21. https://doi.org/10.1021/sb2000116.
(29) Timmons, J. J.; Densmore, D. Repository-Based Plasmid Design. *PLOS ONE* **2020**, *15* (1), e0223935. https://doi.org/10.1371/journal.pone.0223935.
(30) *Cloud-based platform for biotech R&D | Benchling*. https://www.benchling.com/ (accessed 2022-07-04).
(31) Ortega, C.; Abreu, C.; Oppezzo, P.; Correa, A. Overview of High-Throughput Cloning Methods for the Post-Genomic Era. In *High-Throughput Protein Production and Purification*; Vincentelli, R., Ed.; Methods in Molecular Biology; Springer New York: New York, NY, 2019; Vol. 2025, pp 3–32. https://doi.org/10.1007/978-1-4939-9624-7_1.
(32) Hughes, R. A.; Ellington, A. D. Synthetic DNA Synthesis and Assembly: Putting the Synthetic in Synthetic Biology. *Cold Spring Harb. Perspect. Biol.* **2017**, *9* (1), a023812. https://doi.org/10.1101/cshperspect.a023812.
(33) Zhu, B.; Cai, G.; Hall, E. O.; Freeman, G. J. In-Fusion^{TM} Assembly: Seamless Engineering of Multidomain Fusion Proteins, Modular Vectors, and Mutations. *BioTechniques* **2007**, *43* (3), 354–359. https://doi.org/10.2144/000112536.
(34) Quan, J.; Tian, J. Circular Polymerase Extension Cloning of Complex Gene Libraries and Pathways. *PLoS ONE* **2009**, *4* (7), e6441. https://doi.org/10.1371/journal.pone.0006441.
(35) Shuman, S. Novel Approach to Molecular Cloning and Polynucleotide Synthesis Using Vaccinia DNA Topoisomerase. *J. Biol. Chem.* **1994**, *269* (51), 32678–32684.
(36) Hartley, J. L. DNA Cloning Using In Vitro Site-Specific Recombination. *Genome Res.* **2000**, *10* (11), 1788–1795. https://doi.org/10.1101/gr.143000.
(37) Bootstrap; Mark Otto; Jacob Thornton. *Bootstrap*. Bootstrap · The most popular HTML, CSS, and JS library in the world. https://getbootstrap.com/ (accessed 2022-04-01).
(38) Evert Timberg; Jukka Kurkela; Ben McCann. *Chart.js | Open source HTML5 Charts for your website*. Chart.js. https://www.chartjs.org/ (accessed 2022-04-01).
(39) Untergasser, A.; Cutcutache, I.; Koressaar, T.; Ye, J.; Faircloth, B. C.; Remm, M.; Rozen, S. G. Primer3—New Capabilities and Interfaces. *Nucleic Acids Res.* **2012**, *40* (15), e115–e115. https://doi.org/10.1093/nar/gks596.
(40) Plahar, H. A.; Rich, T. N.; Lane, S. D.; Morrell, W. C.; Springthorpe, L.; Nnadi, O.; Aravina, E.; Dai, T.; Fero, M. J.; Hillson, N. J.; Petzold, C. J. BioParts—A Biological Parts Search Portal and Updates to the ICE Parts Registry Software Platform. *ACS Synth. Biol.* **2021**, *10* (10), 2649–2660. https://doi.org/10.1021/acssynbio.1c00263.
(41) Ham, T. S.; Dmytriv, Z.; Plahar, H.; Chen, J.; Hillson, N. J.; Keasling, J. D. Design, Implementation and Practice of JBEI-ICE: An Open Source Biological Part Registry Platform and Tools. *Nucleic Acids Res.* **2012**, *40* (18), e141–e141. https://doi.org/10.1093/nar/gks531.
(42) *JBEI Vector Editor*; Joint BioEnergy Institute, 2021.
(43) Chen, J.; Densmore, D.; Ham, T. S.; Keasling, J. D.; Hillson, N. J. DeviceEditor Visual Biological CAD Canvas. *J. Biol. Eng.* **2012**, *6* (1), 1. https://doi.org/10.1186/1754-1611-6-1.
(44) *Open Vector Editor*; Teselagen Biotechnology, Inc., 2022.
(45) *REPP GitHub*; Lattice Automation, 2022.
(46) *The Go Programming Language*. https://go.dev/ (accessed 2022-06-05).
(47) 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. Comprehensive Profiling

of Four Base Overhang Ligation Fidelity by T4 DNA Ligase and Application to DNA Assembly. *ACS Synth. Biol.* **2018**, *7* (11), 2665–2674. https://doi.org/10.1021/acssynbio.8b00333.
(48) *Traditional Cloning Basics - US*. //www.thermofisher.com/us/en/home/life-science/cloning/cloning-learning-center/invitrogen-school-of-molecular-biology/molecular-cloning/cloning/traditional-cloning-basics.html (accessed 2022-04-01).
(49) Rotella, J. Probability, Log-Odds, and Odds. 3.
(50) *Golden Gate Assembly | NEB*. https://www.neb.com/applications/cloning-and-synthetic-biology/dna-assembly-and-cloning/golden-gate-assembly (accessed 2022-05-30).
(51) *Gibson Assembly® Protocol (E5510) | NEB*. https://www.neb.com/protocols/2012/12/11/gibson-assembly-protocol-e5510 (accessed 2022-05-30).
(52) BioBrick Assembly Manual.
(53) Thomas, H. *Smithy*; 2022.
(54) *Integrated DNA Technologies | IDT*. Integrated DNA Technologies. https://www.idtdna.com/pages (accessed 2022-06-12).
(55) Bartley, B. A.; Choi, K.; Samineni, M.; Zundel, Z.; Nguyen, T.; Myers, C. J.; Sauro, H. M. PySBOL: A Python Package for Genetic Design Automation and Standardization. *ACS Synth. Biol.* **2019**, *8* (7), 1515–1518. https://doi.org/10.1021/acssynbio.8b00336.
(56) *The web framework for perfectionists with deadlines | Django*. https://www.djangoproject.com/ (accessed 2022-04-01).
(57) *Django 3.2 release notes | Django documentation | Django*. https://docs.djangoproject.com/en/3.2/releases/3.2/ (accessed 2022-06-14).
(58) Zulkower, V.; Rosser, S. *DNA Features Viewer, a Sequence Annotations Formatting and Plotting Library for Python*; preprint; Bioinformatics, 2020. https://doi.org/10.1101/2020.01.09.900589.
(59) *DNA Features Viewer*. https://edinburgh-genome-foundry.github.io/DnaFeaturesViewer/index.html# (accessed 2022-06-14).

Appendix

6.1 Application Architecture

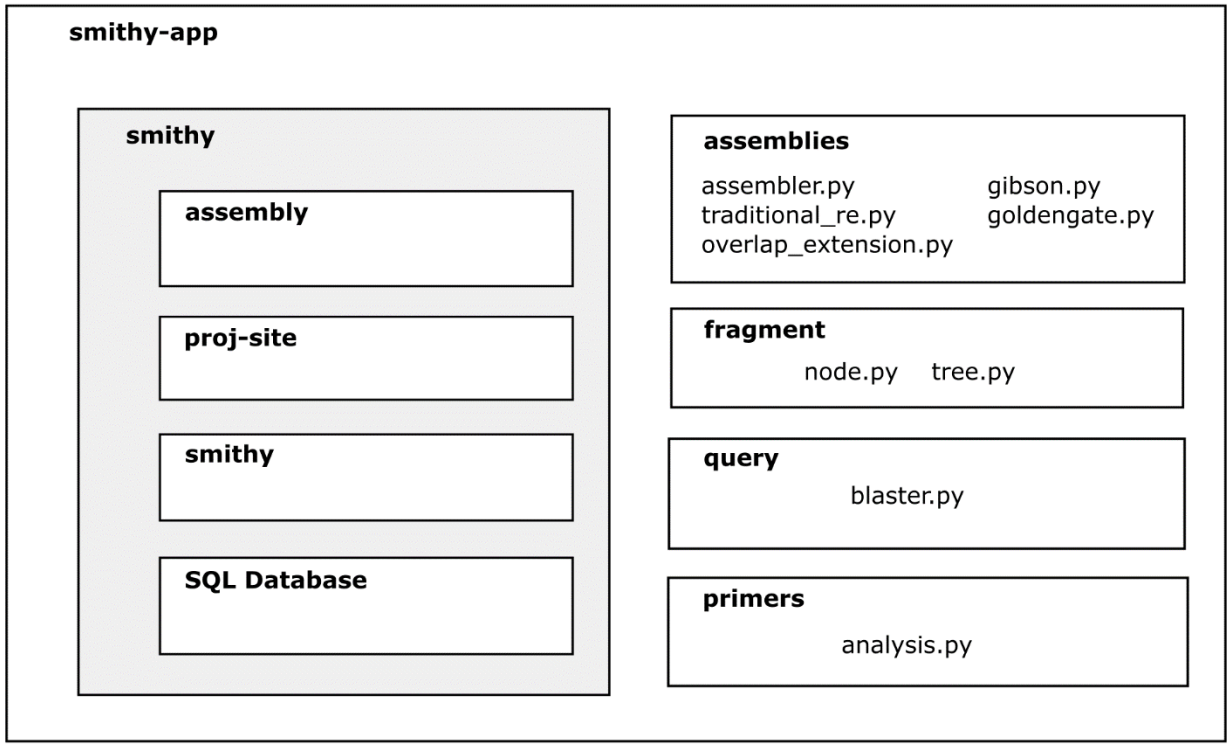


Figure 1S: An overview of application architecture for *Smithy*

The project in its entirety is composed of several interacting software components for establishing a programming environment, executing queries, generating assembly solutions, using solutions for assembly designs, running the website, and database handling. Each of these core aspects have been implemented in several Python modules. Full details and information on the software can be found by exploring the application repository on GitHub.

6.1.1 Virtual Environment

A Python virtual environment has been created for *Smithy* using the standard library's *venv* module. Before the *Smithy* application is started, first its virtual environment is activated then the application is started and run within. For a full listing of dependencies, view the *requirements.txt* file on the project GitHub repository. Virtual environments are advantageous for software projects

because they manage dependencies, external Python libraries downloaded and incorporated into the application, and allow a project to be easily reproducible. Additionally, virtual environments avoid downloading project dependencies to the global operating system environment which in many cases causes conflicts with other projects. Any package intended to be used for a given application is installed directly in the virtual environment and not any global environment space. The notable packages used for this project are:

- Django 3.2[56,57]
- Biopython 1.78[25]
- Pydna 3.1.3[27]
- Primer3-py 0.6.1[39]
- DNA Features Viewer 3.0.3[58,59]

6.1.2 Django & Model-View-Template

The Django framework is based on a fundamental principle of model-view-template, also known as MVT[56]. In brief, models define how the data is to be stored and structured, views manage data gathering and transmission to the templates, and templates dictate how such data is to be presented. Models will be represented in relational database tables, with individual entries represented by a model's class instance. Views perform create, update, read, or delete operations on models of the application while also fetching the appropriate HTML templates. Templates are HTML webpages presented to users containing the data delivered by the views. This encompasses the core approach of Django web application development, and these MVT principles are seen across the Django-based components of *Smithy*.

6.1.3 Django Apps

A central philosophy of the Django framework is the creation of a web application based on sub-applications for specific tasks.[56] These sub-applications are structured in such a way that they are portable to entirely different Django applications. This design philosophy has been applied to the development of *Smithy*, with three core sub-applications comprising the complete application for *Smithy*, named "*Smithy", "proj_site"*, and "*assembly"*. The first of these, *Smithy*, contains the core configurations and settings for the whole application, with connection to the two other sub-applications. The second, *proj_site*, provides the website's basic content, such as home, survey, and about webpages. Lastly, *assembly* contains all of the functionality and database handling procedures for the entire *Smithy* assembly design automation pipeline.

A. Core – *Smithy*

General settings and app configurations are found within the core *Smithy* sub-application; there exists no additional functionality besides: admin site access and connections to the other two sub-apps.

For the settings and configurations, found in *settings.py*, the information is not extensive. Here, one finds definitions of allowed server hosts, registrations of sub-applications, a list of middleware packages, configuration of the SQLite database, and HTML template directories. Other attributes such as time zone, language, template packs, media paths, and static content paths are also found there. All of these are necessary so the application framework can link all components, internal and external, in this one central file.

Access to the other sub-applications is possible through the Django URL patterns defined within *Smithy*. The *assembly* app is accessed through *Smithy*.fungalgenomics.ca/assembly, and *proj_site* by one of its defined URLs such as *Smithy*.fungalgenomics.ca/about. Since very little functionality

exists within the *Smithy* sub-application it is effectively a central access point to the complete feature set of the project.

B. General Site – *proj_site*

As previously mentioned, proj_site is a limited sub-application that contains simple features for general website content. The motivation for this component was to separate the assembly and basic site features from each other. From *proj_site* the application's home page, about page, Google forms survey, and glossary of terms are all accessed. JavaScript and CSS files for front-end needs are also contained within *proj_site*. These files are used for general site styling, front-end supplementary functionality, and assembly solution charts generation, which are especially important for assembly bundles.

C. Assembly Site – *assembly*

Almost all business logic for the Django web application exists here. All database models for the site are implemented, database migrations for the SQLite server are logged; the form for assembly bundles is defined; URL patterns for each cloning method are defined, Django views for each webpage are implemented; and the solution design scripting, with interfacing to the other software components of *Smithy*, are implemented. The database models define the project's SQL tables used for managing assembly, solution, part, primer, and bundle entries. There are several URL patterns that dictate which views to use for every defined URL of the site. These URLs deliver assembly creation forms, detail forms for every model, and list pages for accessing existing assemblies. The Django views, one for each URL, are responsible for either gathering the assembly database entries for display, taking a submitted assembly form and executing the design procedure, or displaying pre-existing assemblies for users to explore. Combined, these features of *assembly* provide the full design automation pipeline features for assembly experiments, apart from the

definitions of the external *Assembler*, *FragmentTree*, and *Blaster* modules. Additionally, the -*assembly* sub-application is defined independently of the full application's other components, so in principle it can be ported to other Django projects, if desired.

6.1.4 Assemblers, Fragments, and Queries

External modules to the web application aspects of *Smithy* are those responsible for BLAST queries, *Assembler* class definitions, *FragmentTree* implementation, *FragmentNode* definition, and Primer3 thermodynamic calculations. However, the rest of the software is an implementation of general web application design principles and features within this collection of external modules is the feature set of the core contributions of *Smithy*. Explicit implementation of these modules is found in the *service.py* script of the *assembly* sub-application.

6.2 Detail Pages

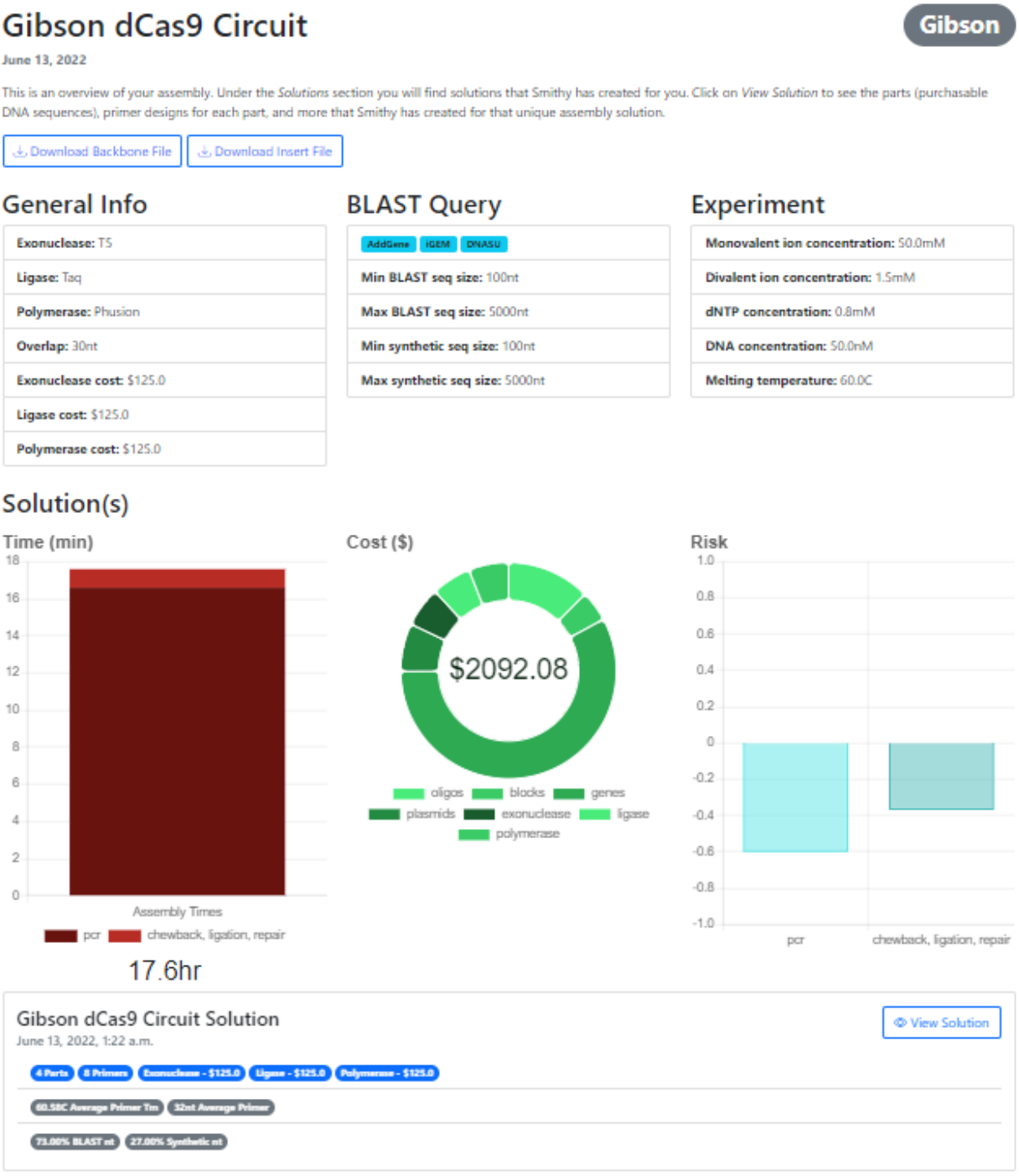


Figure 2S: Assembly detail page for an example Gibson assembly

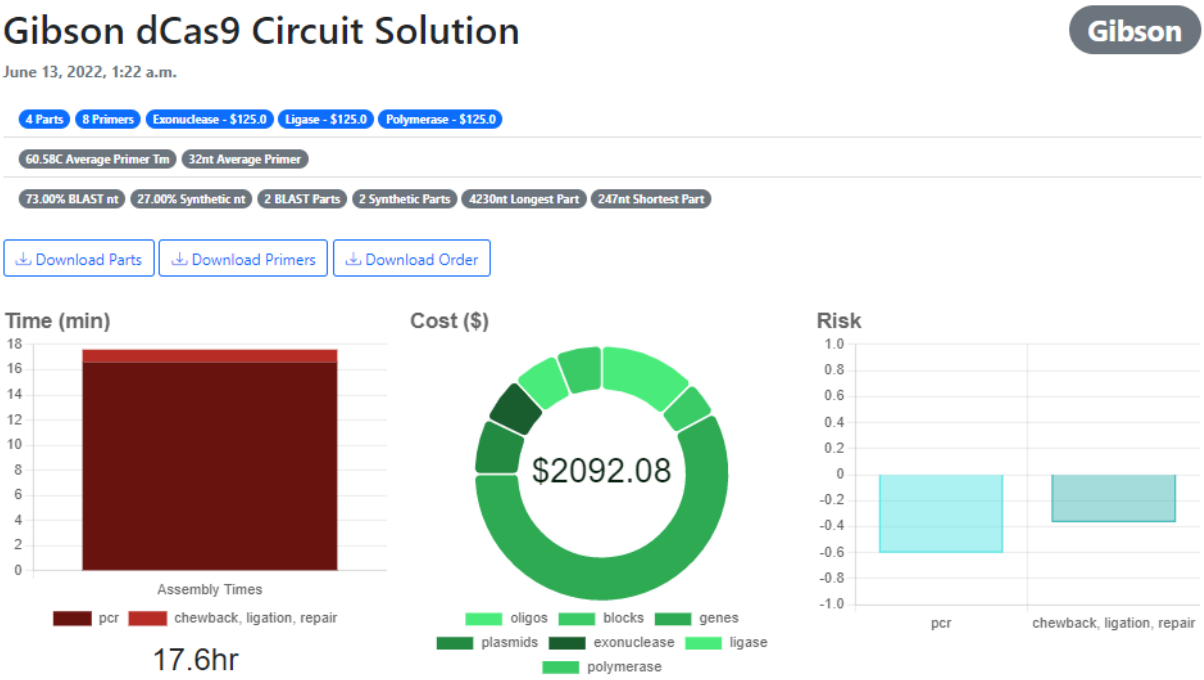


Figure 3S: Top components of the solution detail page for an example Gibson assembly

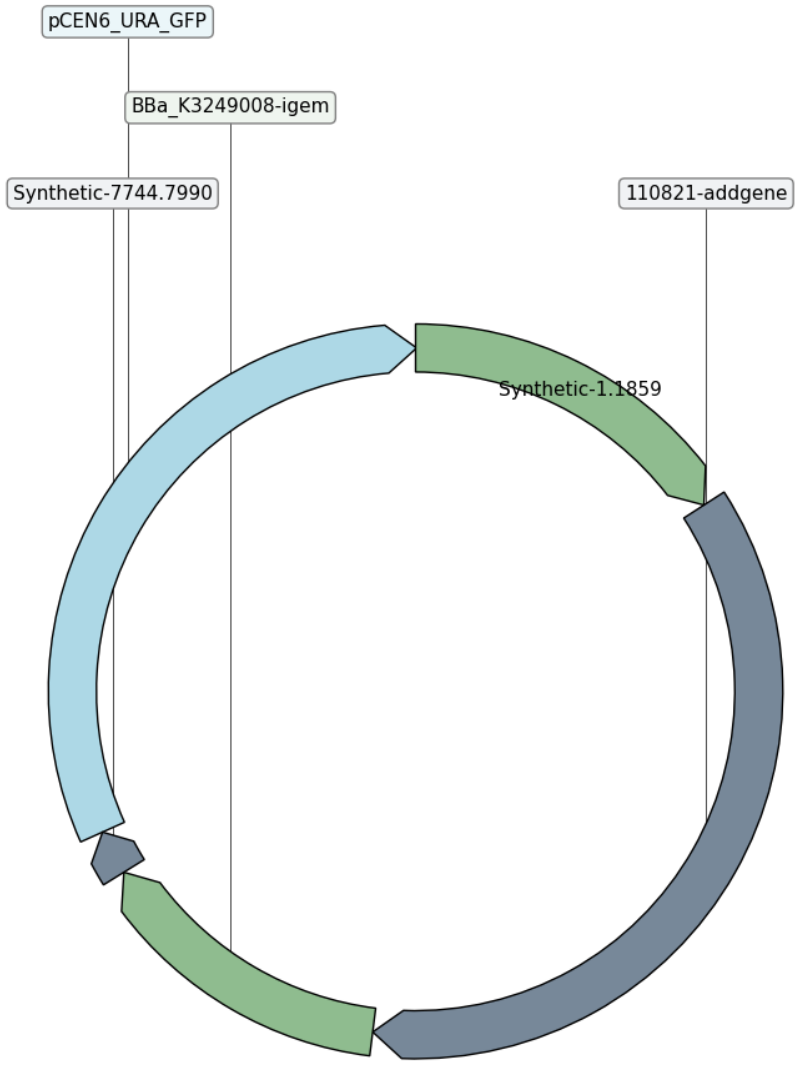


Figure 4S: A plasmid map generated for an example Gibson assembly

Parts

Download Parts

Below is a summary table of the DNA sequences that make up this assembly solution. Further details on each part can be viewed by clicking *View Part* for each listed below this table.

Name	Database	Length	Extended Length	Query Start	Query End	Subject Start	Subject End	Position
Synthetic-1.1859	NONE	1859	1889	1	1859	0	0	0
110821-addgene	addgene	4230	4260	1860	6083	655	4884	1
BBa_K3249008-igem	igem	1660	1690	6084	7743	1980	3639	2
Synthetic-7744.7990	NONE	247	277	7744	7990	0	0	3
pCEN6_URA_GFP	NONE	3721	3751	7991	11711	0	0	4

Query Start/End: The start/end index of the part on the insert sequence. Defaults to 0.
Subject Start/End: The start/end index of the part on its original sequence. Defaults to 0.
Position: The location of the part in the assembly solution.

Figure 5S: The parts table on the solution detail page for an example Gibson assembly

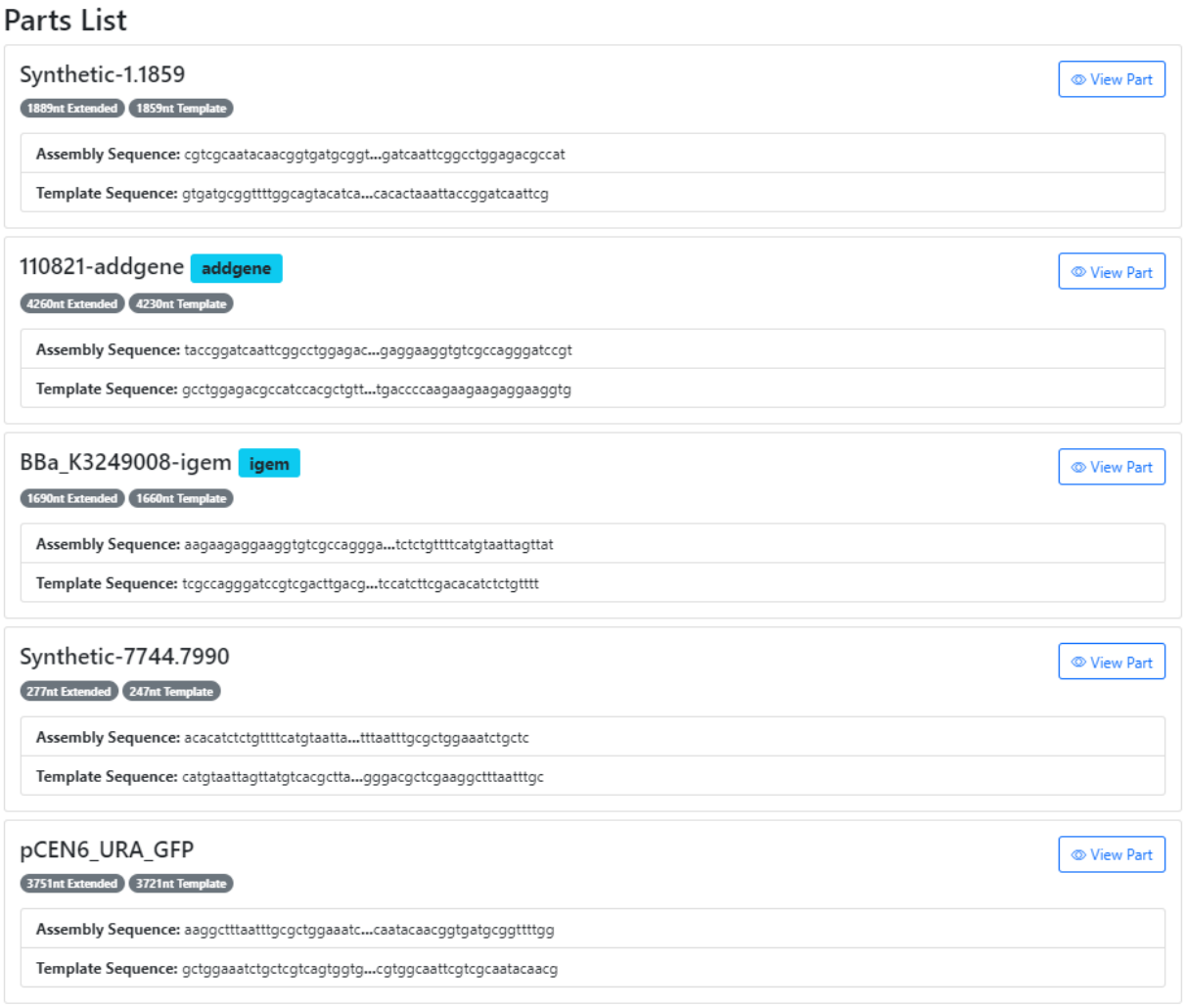


Figure 6S: The parts listing on the solution detail page for an example Gibson assembly

Primers

Download Primers

Below is a summary table of the primers needed for each part of this assembly solution (shown above). Further details on each primer can be viewed by clicking *View Primer* for each listed below this table.

Name	Type	Tm (C)	GC%	Hairpin	hp Tm (C)	hp ΔG	Homodimer	hd Tm (C)	hd ΔG
Synthetic-1.1859 forward primer	fwd	61.445	54.8	True	49.864	-2146.569	True	33.667	-9875.112
Synthetic-1.1859 reverse primer	rvs	61.989	48.6	True	53.459	-1526.924	True	-1.373	-5116.221
110821-addgene forward primer	fwd	61.493	60.7	True	49.036	-1027.356	True	9.289	-6977.121
110821-addgene reverse primer	rvs	62.874	57.1	True	40.045	-168.194	True	5.824	-5337.541
BBa_K3249008-igem forward primer	fwd	60.293	57.1	True	40.442	-199.774	True	1.671	-5148.061
BBa_K3249008-igem reverse primer	rvs	60.636	30.6	True	31.264	563.394	True	-4.293	-4949.301
Synthetic-7744.7990 forward primer	fwd	60.215	35.1	True	31.262	309.122	True	-14.519	-3088.401
Synthetic-7744.7990 reverse primer	rvs	59.450	47.1	False	0.000	0.000	True	11.319	-6774.121
pCEN6_URA_GFP forward primer	fwd	58.919	45.2	True	37.848	-80.179	True	9.581	-6347.431
pCEN6_URA_GFP reverse primer	rvs	58.515	51.6	True	53.031	-1592.340	True	21.136	-5501.412

hp Tm (C): The melting temperature of the primer's hairpin (hp) structure.
hp ΔG: The Gibbs free energy of the primer's hairpin (hp) structure.
hd Tm (C): The melting temperature of the primer's homodimer (hd) structure.
hd ΔG: The Gibbs free energy of the primer's homodimer (hd) structure.

Figure 7S: The primers table on the solution detail page for an example Gibson assembly

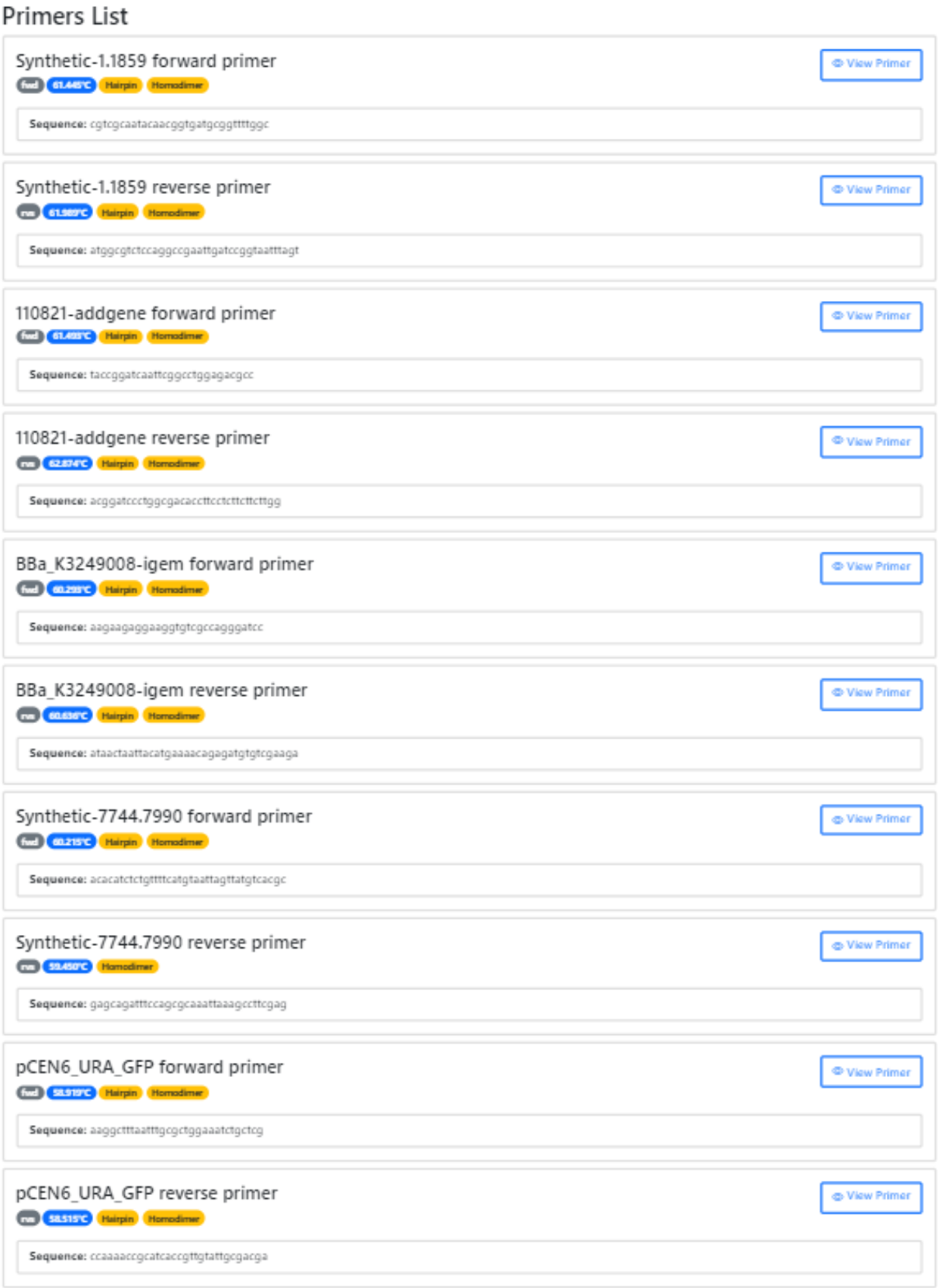


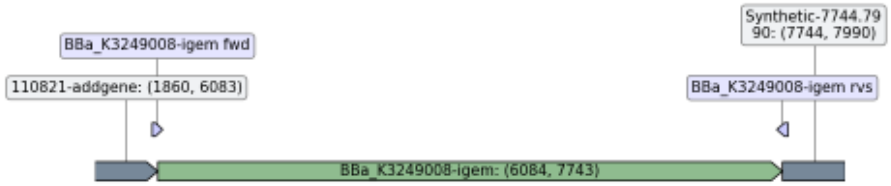
Figure 8S: The primers list on the solution detail page for an example Gibson assembly

BBa_K3249008-igem

Forward Primer | Reverse Primer

Date Created	Database	Length	Extended Length
June 13, 2022, 1:22 a.m.	igem	1660	1690

Part Map



View Assembly Sequence | View Sequence

BBa_K3249008-igem Assembly Sequence 5'-3' Watson

aagaagaggaaggtgtcgccagggatccgtcgacttgacgcgttgatatcaacaagtttgtacaaaaaagcaggctacaaagaggccagcggttccggacgggctgacgcattggacgattttgatctggatatgctgggaagtgacgccctc
gatgatttttgaccttgacatgcttggttcggatgcccttgatgacttttgacctcgacatgctcggcagtgacgcccttgatgatttcgacctggacatgctgattaactctagaagttccggatctccgaaaaagaaacgcaaagttggtagccagtacc
tgcccgacaccgacgaccggcaccggatcgaggaaaagcggaagcggacctacgagacattcaagagcatcatgaagaagtccccctttcagcggcccaccgaccctagacctccacctagaagaatcgccgtgcccagcagatccagcgcc
agcgtgccaaaacctgccccccagccttaccccttcaccagcagcctgagcaccatcaactacgacgagttccctaccatggtgttccccagcggccagatctctcaggcctctgctctggctccagccccttcctcaggtgctgcctcaggctcctgctc
ctgcaccagctccagccatggtgtctgcactggctcaggcaccagcacccgtgcctgtgctggctcctggacctccacaggctgtggctccaccagcccctaaacctacacaggccggcgagggcacactgtctgaagctctgctgcagctgcagt
tcgacgacgaggatctgggagccctgctgggaaacagcaccgatcctgccgtgttcaccgacctggccagcgtggacaacagcgagttccagcagctgctgaaccagggcatccctgtggcccctcacaccaccgagcccatgctgatggaata
ccccgaggccatcacccggctcgtgacaggcgctcagaggcctcctgatccagctcctgcccctctgggagcaccaggcctgcctaatggactgctgtctggcgacgaggacttcagctctatcgccgatatggatttctcagccttgctgggctct
ggcagcggcagccgggattccagggaagggatgtttttgccgaagcctgaggccggctccgctattagtgacgtgtttgagggccgcgaggtgtgccagccaaaacgaatccggccatttcatcctccaggaagtccatgggccaaccgcccac
tccccgccagcctcgcaccaacaccaaccggtccagtacatgagccagtcgggtcactgaccccggcaccagtccctcagccactggatccagcgcccgcagtgactcccgaggccagtcacctgttggaggatcccgatgaagagacgagcca
ggctgtcaaagcccttcgggagatggccgatactgtgattccccagaaggaagaggctgcaatctgtggccaaatggacctttcccatccgcccccaaggggccatctggatgagctgacaaccacacttgagtccatgaccgaggatctgaacct
ggactcacccctgaccccggaattgaacgagattctggataccttcctgaacgacgagtgcctcttgcatgccatgcatatcagcacaggactgtccatcttcgacacatctctgttttcatgtaattagttat

View Forward Primer | View Reverse Primer

BBa_K3249008-igem forward primer 5'-3' Watson

Sequence: aagaagaggaaggtgtcgccagggatcc

Footprint: tcgccagggatcc

Tail: aagaagaggaaggtg

BBa_K3249008-igem reverse primer 5'-3' Crick

Sequence: ataactaattacatgaaaacagagatgtgtcgaaga

Footprint: aaaacagagatgtgtcgaaga

Tail: ataactaattacatg

Figure 9S: A part detail page for an example Gibson assembly

Synthetic-1.1859 forward primer fwd

Date Created

June 13, 2022, 1:22 a.m.

Primer Type

fwd

Sequence 5'-3' Watson

cgtcgcaatacaacggtgatgcggttttggc

Thermodynamic Analysis

Structure	Structure Found	Tm (C)	ΔG	ΔH	ΔS
Hairpin	True	49.86404269587939	-2146.568909267873	-53900.0	-166.86581038443376
Homodimer	True	33.66684229994462	-9875.112364094617	-123400.0	-366.0322025984375

Figure 10S: The primer detail page for an example Gibson assembly