PUDU: Automated Protocols for Synthetic Biology Workflows.

Gonzalo Vidal,[*,†,‡] Oscar Rodriguez,[‡] Parker Ackerknecht,[‡] Luke Dysart,[‡] Matt Burridge,[†] David Markham,[†] Timothy J. Rudge,[†] and Chris Myers[*,‡]

†*Interdisciplinary Computing and Complex Biosystems, School of Computing, Newcastle University, Newcastle upon Tyne, NE1 7RU, United Kingdom*

‡*Electrical, Computer and Energy Engineering, University of Colorado Boulder, Boulder, Colorado 80309, USA*

E-mail: gonzalo.vidalpena@colorado.edu; chris.myers@colorado.edu

Documentation

PUDU provides a high-level abstraction for liquid handling robot control using a simple object-oriented programming approach in Python. Each object corresponds to a protocol template that can be easily customized changing its attributes. This allows the user to define the samples, volumes, pipette position, labware and more, but the protocol remains the same. To create a new class, or protocol template, users can inherit from existing ones and build on top of them making this process more straightforward, easy to update and reducing errors. For the purpose of this paper, let's define a protocol as the series of steps that the human and the machine has to follow. The steps that the human has to perform are encoded in natural language and include robot deck setup with modules and the modules setup with reagents. The steps that the machine has to perform are encoded in the script. To define a script, or OT-2 protocol, you need to create a run function that encodes the

labware type and position, as well as the liquid transfers. PUDU simplifies this process of making and using scripts to a few lines of code. Inside the script's run function, the user would need to instantiate a PUDU class and then call its own run method. Users can also add options such as creating an SBOL file, capturing metadata in a machine readable format, and/or adding the creation of an Excel file, capturing relevant information for deck setup, position of reagents and products in a human readable format. For a typical PUDU protocol, the user creates a script and simulates it. The simulation will output an Excel file with information about reagent position. Then, the user can load the script into the Opentrons App and get information to set up the OT-2 deck and run the script. These scripts automate different stages of the cloning process, the test setup and calibration (see Figure 1). All the code, protocols, examples and documentation are publicly available at `https://github.com/MyersResearchGroup/PUDU`.

DNA Assembly

Abstract Base Class Implementation:

The core functionality was abstracted into four primary base classes implementing the *Template Method* design pattern. The `BaseAssembly` class provides 21 configurable parameters including volume specifications (total reaction, parts, enzymes, buffers), hardware configuration (thermocycler labware, temperature module settings), tip management (rack positions, pipette selection), and execution parameters (aspiration/dispense rates, replication settings). Child classes inherit shared methods including `liquid_transfer()`, `setup_tip_management()`, `get_xlsx_output()`, and `run()`.

Factory Pattern Implementation:

The `LoopAssembly` factory class implements intelligent format detection through static analysis methods. The `_is_sbol_format()` method identifies SBOL assembly plan representa-

tions by detecting required keys (`Product`, `Backbone`, `PartsList`, `Restriction Enzyme`) and URI patterns (https:// prefixes), while `_is_manual_format()` validates manual assemblies by confirming the presence of a `receiver` key and absence of SBOL-specific structures. This enables seamless protocol switching without user intervention.

SBOL Format Processing:

The `SBOLLoopAssembly` class implements comprehensive support of SBOL assemblies parsing with URI-based component extraction. The `_extract_name_from_uri()` method processes SBOL URIs by extracting the penultimate URI segment and removing version numbers, handling both versioned (`/GFP/1`) and unversioned formats. The class maintains separate tracking sets for parts (`parts_set`), backbones (`backbone_set`), and restriction enzymes (`restriction_enzyme_set`), enabling independent reagent management and validation.

Combinatorial Assembly Generation:

The `ManualLoopAssembly` class implements sophisticated combinatorial expansion using Python's `itertools.product()`. The `_generate_combinations_for_assembly()` method converts role-based dictionaries into Cartesian products, supporting both single parts (strings) and multiple parts (lists) per role. Restriction enzyme selection follows the Odd/Even naming convention with automatic pattern matching using `fnmatch`: patterns `Odd*` trigger BsaI enzyme selection, while `Even*` patterns select SapI.

Domestication Protocol Implementation:

The `Domestication` class handles individual part insertion into universal acceptor backbones. The implementation validates single-enzyme, single-backbone constraints while supporting multiple parts through iterative processing. Each part undergoes independent assembly with the backbone, generating separate reaction wells with comprehensive tracking in `dict_of_parts_in_thermocycler`.

Advanced Tip Management Algorithm:

Implemented batch tip management supporting protocols requiring more tips than deck capacity. The algorithm calculates total tip requirements using protocol-specific formulas:

$$N_{tips,total} = (N_{reagent\_tips} + N_{part\_tips}) \times N_{assemblies} \times N_{replicates} \tag{1}$$

$$N_{racks,needed} = \lceil N_{tips,total}/96 \rceil \tag{2}$$

$$N_{batches} = \lceil N_{racks,needed}/N_{deck\_slots} \rceil \tag{3}$$

The system manages on-deck and off-deck tip racks, performing automatic batch swapping when `tips_used % tips_per_batch == 0`. The `_perform_tip_rack_batch_swap()` method coordinates labware movement using `protocol_api.OFF_DECK` staging, ensuring continuous tip availability for high-throughput protocols.

Volume Validation and Error Handling:

Each assembly class implements comprehensive validation including reagent capacity constraints (maximum 24 wells on temperature module), thermocycler well availability (96 wells minus starting position), and part count limitations. The validation logic accounts for constant reagents (water, ligase buffer, ligase, restriction enzymes) when calculating maximum supported parts:

$$N_{parts,max} = 24 - N_{reagents,constant} \tag{4}$$

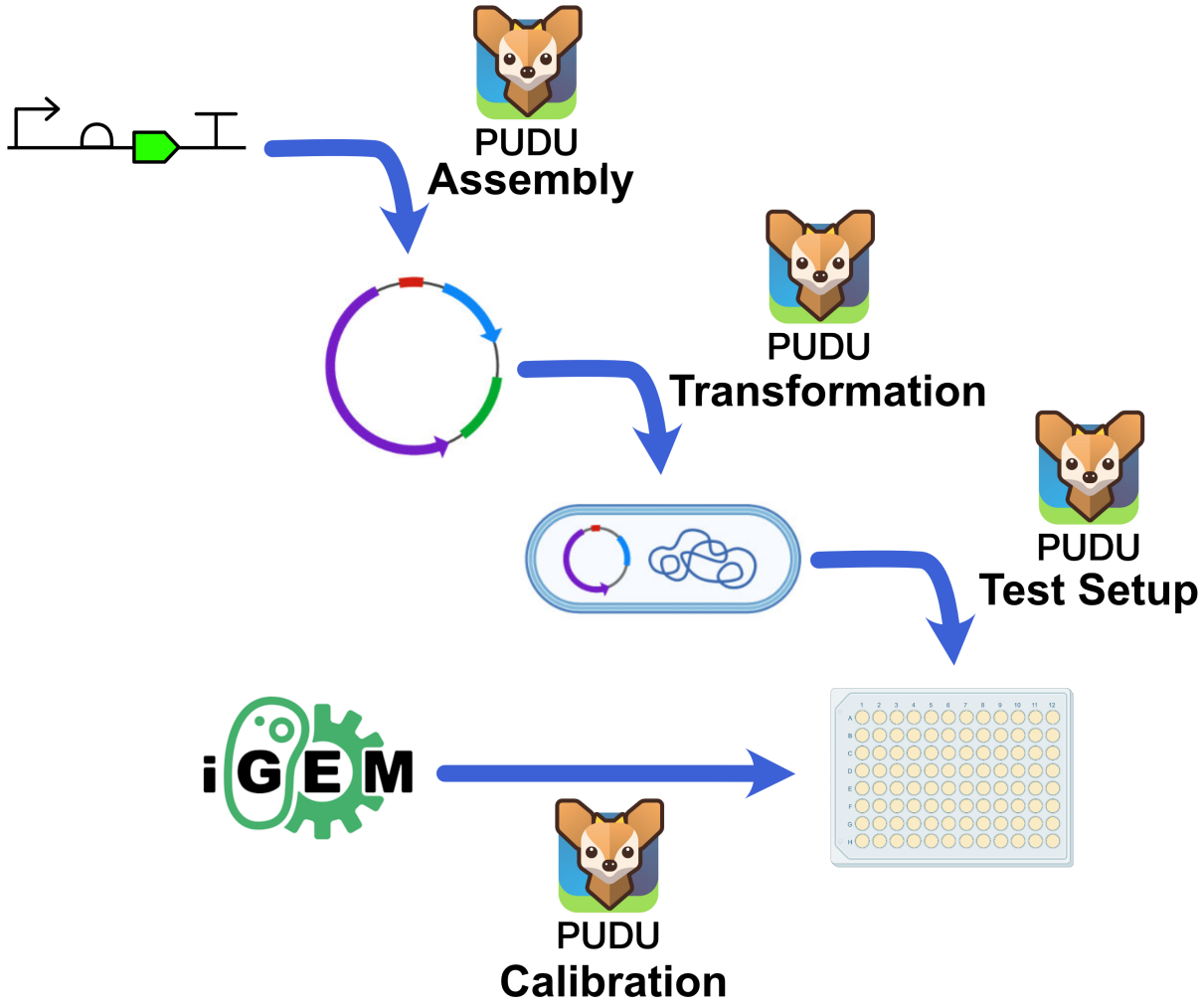

Figure 1: PUDU workflow diagram. Synthetic biology workflow automated using PUDU. The DNA Assembly class automates domestication and the assembly using a design in SBOL or a list of parts using Loop. This process can be automated using a a DNA Assembly script. Then, the Transformation class automates the transformation of bacteria with the assembled DNA. Finally, the Test Setup class automates the setup of a 96 well plate with the transformed bacteria under different conditions. Furthermore, The Calibration class automates the preparation of a 96 well plate to obtain calibrated data.

Transformation

Heat Shock Protocol Implementation:

The `HeatShockTransformation` class implements precise temperature-controlled bacterial transformation with configurable incubation profiles. Default parameters include: cold incubation 1 (4°C, 30 minutes), heat shock (42°C, 1 minute), cold incubation 2 (4°C, 2 minutes),

and recovery incubation (37°C, 60 minutes). The implementation supports custom temperature profiles through dictionary parameters. The implementation supports the DNA construct source to be provided using either individual tubes in the temperature module or can be sourced directly from the DNA assembly protocol in the 96-well pcr plate by setting the `use_dna_96plate` parameter to True.

Multi-Tube Reagent Management:

Advanced reagent management automatically calculates tube requirements based on volume constraints:.

$$N_{transformations,total} = N_{constructs} \times N_{replicates} \tag{5}$$

$$N_{tubes,competent} = \lceil N_{transformations,total} / \lfloor V_{tube} / V_{transfer} \rfloor \rceil \tag{6}$$

$$N_{tubes,media} = \lceil N_{transformations,total} / \lfloor V_{tube,media} / V_{transfer,media} \rfloor \rceil \tag{7}$$

The `_validate_protocol()` method ensures reagent requirements do not exceed temperature module capacity (24 wells), considering DNA constructs, competent cell tubes, and media tubes.

Optimized Liquid Distribution:

Competent cell and recovery media transfer utilize Opentrons' built-in `distribute()` method for efficiency. The implementation includes mixing parameters (`mix_before=(3,50)`) for competent cells and air gap settings (`air_gap=10`) for media to prevent contamination. DNA transfer employs the custom `liquid_transfer()` method with air bubble removal through repeated aspiration/dispense cycles.

Plating

Custom Labware Development:

Developed `nunc_omnitray_96grid` labware definition that maps standard Nunc OmniTray reservoir geometry to 96-well coordinate systems. This enables systematic colony placement using standard pipetting protocols while maintaining compatibility with agar-based media.

Dynamic Plate Layout Algorithm:

The `calculate_plate_layout()` method implements intelligent well assignment based on experimental parameters:

$$N_{colonies,total} = N_{constructs} \times N_{dilutions} \times N_{replicates} \tag{8}$$

$$N_{colonies,per\_dilution} = N_{constructs} \times N_{replicates} \tag{9}$$

The algorithm determines single vs. dual-plate requirements using capacity thresholds: single plate if $N_{colonies,per\_dilution} \leq 48$ and dual plates otherwise. For single-plate layouts, dilutions are assigned to plate halves (wells 1-48 and 49-96) with systematic buffer spacing.

Volume Optimization Methodology:

Through empirical validation using varied dispensing volumes (1-8μL), established 4μL as optimal colony dispensing volume. This volume enables proper colony separation for individual picking while maintaining compatibility with 96-well plate spacing (9mm center-to-center distance). Volumes exceeding 4μL resulted in colony overlap, while volumes below 3μL showed reduced plating efficiency.

Automated Workflow Integration:

The plating protocol coordinates multi-step processing: LB distribution to dilution wells, bacteria transfer with mixing (5 repetitions, 19μL volume), serial dilution between plates,

and final colony plating with optimized dispensing height (`well.top(-8)`). The implementation minimizes tip usage through strategic tip retention during multi-step transfers.

Sample preparation

Serial Dilution Mathematics:

The `PlateWithGradient` class implements precise concentration series calculation:

$$C_n = C_0/\text{dilution\_factor}^n \tag{10}$$

$$V_{required,sample} = V_{prefill} \times N_{steps} \times N_{replicates} + V_{initial} \times (1 - r_{mix}) \tag{11}$$

$$V_{required,inducer} = V_{initial} \times r_{mix} \tag{12}$$

where C_n is concentration at step n, $V_{prefill}$ is pre-filling volume per well, $V_{initial}$ is initial mixture volume, and r_{mix} is initial mix ratio.

Volume Safety Margin Calculation:

Required volumes include 20% safety margins with ceiling functions: $V_{stock} = \lceil V_{calculated} \times 1.2 \rceil$. The `_calculate_required_volumes()` method accounts for initial mixture preparation, well pre-filling, and transfer losses.

Gradient Creation Algorithm:

Serial dilution implementation employs systematic well-to-well transfers with consistent mixing (3 repetitions, 50% well volume). The algorithm pre-fills destination wells with diluent, creates initial high-concentration mixtures, and performs sequential transfers maintaining constant final volumes.

Calibration

The `Calibration` class has two child classes, `iGEM GFP OD` and `iGEM RGB`. `iGEM GFP OD` is used to prepare a plate for calibration of green fluorescence[1] and the cell count from OD.[?] `iGEM RGB OD` is used to prepare a plate for calibration of red, green and blue fluorescence[?] and cell count from OD.[?]

Calibration

iGEM Protocol Compliance:

Both `GFPODCalibration` and `RGBODCalibration` implement standardized protocols following iGEM InterLab study specifications (https://old.igem.org/wiki/images/a/a4/InterLab_2022_-_Calibration_Protocol_v2.pdf). The GFP/OD protocol utilizes fluorescein and silica nanoparticles across 4 replicates (2 per calibrant), while the RGB/OD protocol expands to 8 replicates covering fluorescein, sulforhodamine 101, cascade blue, and nanoparticles.

Buffer Management Strategy:

Calibration protocols support both individual tube and falcon tube configurations. For falcon tubes, the system maps single large-volume sources to multiple destination ranges, while individual tubes provide dedicated sources for each calibrant series. Buffer assignment follows layout-driven mapping with precise volume tracking.

Serial Dilution Implementation:

1:2 serial dilutions across 11 concentration points (including blanks) with final volume normalization to 200μL. The RGB protocol includes waste disposal steps using a dedicated `binit` container, preventing cross-contamination between calibrant series.

Utils

SmartPipette Class Implementation:

Addresses critical real-world automation challenges through intelligent aspiration positioning. The class integrates with Opentrons' liquid tracking API to monitor real-time volume changes and calculate safe aspiration heights.

Conical Tube Height Calculation:

The `get_conical_tube_aspiration_height()` method implements volume-based positioning:

$$h_{asp} = \max\left(\frac{V_{current}}{V_{max}} \times (d_{tube} - 10) - 10, 3\right) \tag{13}$$

where h_{asp} is aspiration height (mm), $V_{current}$ is current volume from API, V_{max} is maximum tube volume, d_{tube} is tube depth, 10mm accounts for tube threads, 10mm provides meniscus offset, and 3mm ensures minimum safe height.

Contamination Prevention Logic:

The `is_conical_tube()` method automatically detects conical labware through name pattern matching (`'conical' in well.parent.load_name.lower()`) or manual override. For conical tubes below 20% capacity, the system defaults to standard aspiration to prevent tip damage.

API Integration and Error Handling:

The implementation leverages `well.current_liquid_volume()` for real-time tracking with comprehensive exception handling. When API methods fail, the system provides safe fallback positioning (`return well`) and logs detailed error messages for debugging.

Liquid Transfer Wrapper:

The `liquid_transfer()` method encapsulates standard Opentrons operations with enhanced safety features including volume validation before transfer, intelligent aspiration location calculation, and comprehensive mixing support with volume-limited parameters.

Liquid Loading and Visualization System

Opentrons API Integration:

The refactored architecture implements comprehensive liquid tracking through the `protocol.define_liquid()` method with systematic color coding using a 24-color palette. Each reagent receives unique identifiers, volume specifications, and visual markers enabling real-time deck visualization.

Color Management System:

Implemented automated color assignment using modulo indexing: `color_index = len(tracking_dict) % len(colors)`. The 24-color palette provides distinct visual identification for complex protocols with multiple reagents, parts, and buffers.

Metadata Tracking Integration:

Each protocol maintains comprehensive dictionaries tracking reagent positions (`dict_of_parts_in_temp_mod_position`), assembly locations (`dict_of_parts_in_thermocycler`), and transformation products (`dna_list_for_transformation_protocol`). This metadata enables protocol chaining and downstream automation.

Visual Feedback Improvements:

The Opentrons App integration displays real-time deck setup information including reagent positions, required volumes, and liquid locations with color-coded wells. This approach eliminates previous Excel-based output methods, reducing setup errors and providing immediate visual feedback during protocol preparation.

Volume Tracking Accuracy:

The system maintains precise volume accounting through API integration, enabling intelligent liquid handling decisions such as mixing volume limitations, aspiration height optimization, and reagent depletion warnings.

Validation and Quality Assurance

Comprehensive Error Checking:

Each protocol class implements multi-level validation including parameter range checking, hardware capacity constraints, and logical consistency verification. Validation methods raise descriptive `ValueError` exceptions with specific guidance for protocol modification.

Protocol Validation Algorithms:

Assembly protocols validate part count limits, thermocycler capacity, and reagent module constraints. Transformation protocols verify tube requirements against module capacity. Plating protocols ensure colony count limits and validate plate capacity requirements.

Gene expression study

Discussion

In the future we would like to improve the connection with Design tools such as LOICA[?] and SynBioSuite.[?] We would also like to improve the connection to Learn tools by capturing more and better metadata in a semi-automated way. Although this process is difficult to fully automate as users will always be the ones who know what samples and reagents are provided, we think that creating better interfaces would better facilitate the process of metadata acquisition and standardization. We aim to generate a digital plate, ready to be connected to tools like Flapjack[2] and SynBioHub.[3] Compared to other tools such as

PyLabRobot[?] or the Opentrons API that help users to create protocols, PUDU has a set of protocols defined as classes where their arguments are required inputs from the user or small modifications, and PUDU captures metadata in a standardized format. Finally, we want to generalize our liquid handling control generating protocols in the Laboratory Open Protocol (LabOP) standard and expand the calibration scripts using absolute protein quantification.[?]

Methods

Software Architecture Refactoring

The PUDU platform underwent comprehensive refactoring implementing object-oriented design principles specifically tailored for laboratory automation challenges. Unlike conventional software applications, laboratory automation requires extensive parameterization to handle physical constraints including equipment variability, labware differences, and real-world liquid handling limitations.

Inheritance Hierarchy Design

The architecture implements multiple levels of inheritance with specialized functionality.
For assembly protocols:
`BaseAssembly` → `SBOLLoopAssembly`, `ManualLoopAssembly`, `Domestication`;
for calibration:
`BaseCalibration` → `GFPODCalibration`, `RGBODCalibration`;
for sample preparation:
`SamplePreparation` → `PlateSamples`, `PlateWithGradient`;
and for transformation:
`Transformation` → `HeatShockTransformation`.

Parameter Management System

Each protocol class exposes 15-32 configurable parameters organized into functional categories: hardware configuration (labware types, deck positions, module settings), liquid handling (volumes, rates, mixing parameters), protocol execution (replicates, starting positions, temperature profiles), and automation features (tip management, media capture, simulation modes). Default values are provided for all parameters, enabling immediate protocol execution while allowing customization for specific laboratory requirements.

References

(1) Beal, J.; Haddock-Angelli, T.; Baldwin, G.; Gershater, M.; Dwijayanti, A.; Storch, M.; De Mora, K.; Lizarazo, M.; Rettberg, R.; with the iGEM Interlab Study Contributors Quantification of bacterial fluorescence using independent calibrants. *PloS one* **2018**, *13*, e0199432.

(2) Yáñez Feliú, G.; Earle Gómez, B.; Codoceo Berrocal, V.; Muñoz Silva, M.; Nuñez, I. N.; Matute, T. F.; Arce Medina, A.; Vidal, G.; Vidal Céspedes, C.; Dahlin, J.; others Flapjack: Data management and analysis for genetic circuit characterization. *ACS Synthetic Biology* **2020**, *10*, 183–191.

(3) McLaughlin, J. A.; Myers, C. J.; Zundel, Z.; Mısırlı, G.; Zhang, M.; Ofiteru, I. D.; Goni-Moreno, A.; Wipat, A. SynBioHub: a standards-enabled design repository for synthetic biology. *ACS synthetic biology* **2018**, *7*, 682–688.

TOC Graphic