# ROBOT-TRIFLE: A ROBOTIC SYSTEM FOR AUTOMATED COMBINATORIAL GROWTH PHENOTYPING

BY

JEREMY KEMBALL

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Bioengineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Adviser:

Dr. Paul Jensen

## ABSTRACT

This project, robot-trifle, is an analysis pipeline for streamlining the automated construction of minimal chemically-defined media for microorganisms known to grow in culture. The first part of the pipeline analyzes known growth data to produce media elimination experiments that are planned and executed by the second and third sections. This software package aims to streamline metabolic profiling by automating the extensive bookkeeping and manual labor required to perform dozens of independent growth experiments. The metabolic profile of an organism can help inform metabolic reconstructions, community metabolism analysis, and genetic metabolism inference.

## ACKNOWLEDGEMENTS

This document would not be possible without the guidance of Paul Jensen, the encouragement of Tandy Warnow, and the unwavering support of Krista Smith.

**TABLE OF CONTENTS**

# CHAPTER 1: INTRODUCTION

The next generation sequencing revolution has made genomic data available in overwhelming amounts. Derivatives like deep sequencing, metagenomic sequencing, RNAseq, and transposon sequencing have only complicated the problem. Data are good, and more data are better, but validation is relatively thin on the ground. UniProt is a database specifically for tracking validated and corroborated genetic data, featuring half a million manually curated and experimentally verified proteins [1]. PDB contains other experimental verification data for about 122 thousand proteins [2]. By contrast, genetic information for 1.2 million proteins(just for primates) have been stored in GenBank [3]. Leaving out unassembled raw read datasets and unannotated raw sequence datasets for unfairly depressing the amount of corroboration for genetic annotations, protein-coding genes have experimental validation about 5% of the time.

There are some projects developing tools to 'mine' (really, to cross-reference) the experimental data of the pre-sequencing and pre-digital eras [4], but those data are 'fossil fuels'. Once depleted, the store of predigital science will not be replenished. Not to mention advances in genotyping, species discrimination, and culturing microbes make some of those results less than 100% reliable without independent corroboration, which would be the whole point of 'mining' these papers in the first place.

High-throughput automation of routine experiments is not only economically necessary but practically unavoidable. Human error rates for the simplest repetitive tasks hover around 1% under ideal conditions [5], and given the combinatorial explosions of experiments, false positives are bound to occur. For experiments done in triplicate, one experiment in a million will produce a unanimous incorrect result, and one in every three

thousand will produce a 2/3 majority incorrect result. The E. coli genome contains a shade more than four thousand genes [6], so on average about 120 genes will have at least one incorrect replicate. This analysis applies to the most routine possible experiment, as human error rates for more complex tasks are higher. More complex experiments involving biological variability bring their own sample size concerns. Robotic or machine experiment performance not only lowers the cost of an individual experiment but cuts the total number of experiments by reducing variability in individual experiments, and therefore increasing their reliability.

Liquid handling robots are a mature industry, with multiple competing suppliers. Most research groups nonetheless don't own one. The financial cost of the instrument is only part of the problem, although liquid handlers can be incredibly expensive. The epMotion 5070, Eppendorf's liquid handler, is quite small, quite specialized, and about 35 thousand dollars, depending on who you are. The individual tool heads are about 3 thousand dollars each [7]. If you open-source or design your own, machines are much cheaper, but require much more investment in support. If a lab owns an unusual instrument like a custom liquid handling robot, they commit to having one or two students who work with it regularly for the lifetime of the instrument, or instrument will go unused and the investment will be wasted.

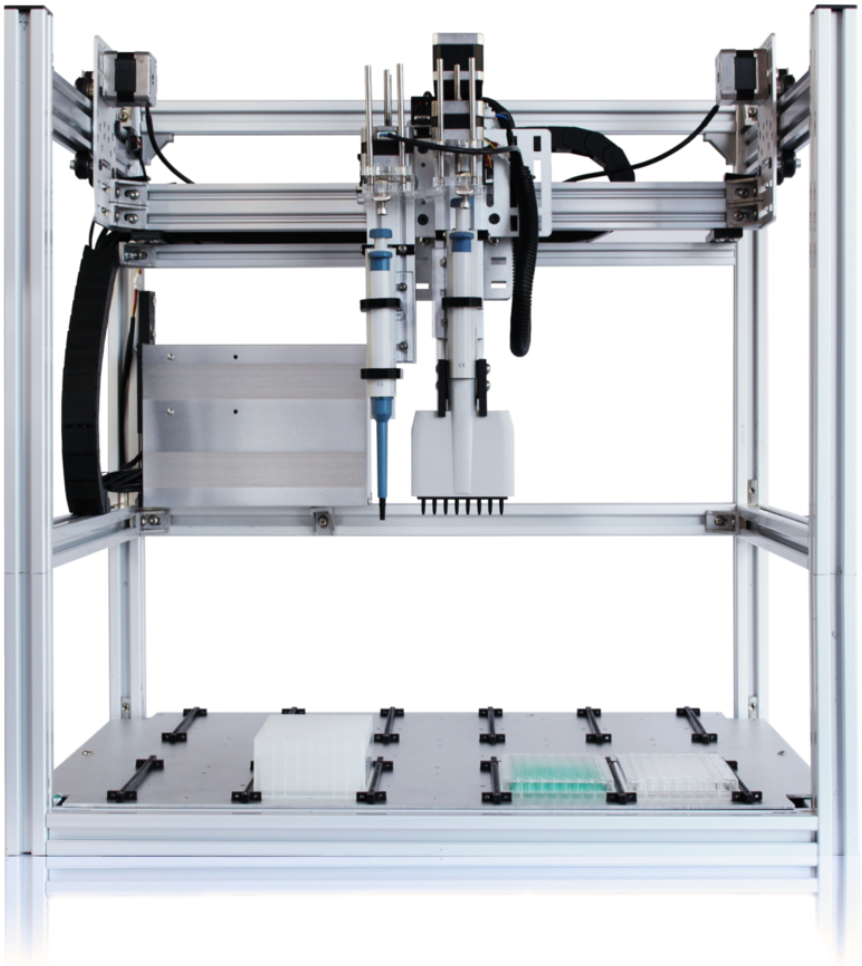
This project aims to lower the barrier to entry, so that liquid handling robots are no harder to use than a thermocycler. Using off-the shelf hardware and a declarative interface, minimal media can be calculated and growth phenotypes tabulated for modeling analysis. This will lower the investment in time and energy required to make use of an open-source liquid handler.

## CHAPTER 2: ARCHITECTURE

The robot itself used for this project is an open hardware autopipettor available from Opentrons, the OT-One S Hood (see figure 1). The hardware is open-source, and pre-assembled models are available from Opentrons. The experiments mentioned below are growth phenotype experiments, a particularly monotonous experiment series due mostly to their combinatorial nature. If a media condition(a list of compounds) is known to support growth of a microorganism, it's not necessarily true that all of those compounds are actually required. Moreover, which of those compounds are strictly essential reveals information about what metabolic processes the microorganism can perform. For example, most wild *Escherichia coli* strains can grow in M9 minimal media, which is composed of ammonia, salts, and glucose [8] [9]. Therefore these bacteria must be able to synthesize all amino acids present in their proteins from ammonia. Scientists have been performing this kind of experiment for decades [10], but the fundamental method has remained unchanged. Combinatorial growth phenotyping has not benefited from technological advancements, unlike PCR or gene sequencing.

The software is a three-layer system. The topmost layer(`branch_and_bound.py`) designs experiments abstractly, and provides a list of proposed experiments to the middle layer. The middle layer(`bartender.py`) accepts a layout file that has locations, amounts, and concentrations of compounds, as well as a destination plate specification. This layer accepts layouts and destination specifications and emits goal spreadsheets, which are specific amounts to move from specific wells. `bartender.py` is responsible for ensuring there are enough components to perform all the experiments.The bottommost layer, `stockings`, speaks to the robot through the Opentrons Python library, and

Figure 1: OT-One S Hood [11]



accepts instructions in spreadsheet form. This layer actually instructs the robot to move compounds back and forth.

The topmost layer, `branch_and_bound.py`, takes a standard media definition and a list of known growth conditions and known no-growth conditions and conducts a depth-first walk of the growth surface, keeping track of the unknown conditions it encounters and emitting them as a set of proposed experiments. 'branch and bound' is a misnomer: due to the inability to put a lower bound on the number of required components the branch and bound algorithm doesn't apply. However, the algorithm used is not quite as dumb as brute force search, exploiting the convex assumption of metabolic growth to prune out some branches. If a media removal is fatal( that is, a condition $S$ supports growth but the condition $S$ minus compound $c$ does not) no media conditions that are subsets of $S$ will support growth without $c$. Therefore, once you have determined that glucose removal(for example) abolishes growth in a given media, you don't need to check whether glucose removal abolishes growth in the absence of every other component. There are of course situations where this assumption is violated, but it is 'more true' towards the leaves of the search tree. If one component ($t$) is toxic in the absence of an antidote ($a$), when $a$ comes before $t$ alphabetically $a$ will be declared essential and show up in the list of media solutions, despite being nonessential in the absence of $t$.

The middle layer(`bartender.py`) takes as input a .yaml file specifying the available compounds and the kind of experiment that should be performed with them, focusing on simple metabolic growth experiments. The program calculates the appropriate amounts to transfer and verifies that the amount of compound available is sufficient, as well as laying out the experimental wells. This process is theoretically straightfor-

ward, converting the declarative language of concentrations of compounds in wells to a spreadsheet of volumes. It is the responsibility of the user to make sure the containers used have calibration data available. A more elegant solution to loading arbitrary containers and verifying their properties would require more sensors than the Opentrons One currently supports. After writing the `goals.csv` file, `stockings` can take over and actually transfer the compounds.

The bottommost layer(`stockings`) is simple. It requires a 'goals' file and a 'state' file. This layer actually performs all the transfers specified in the goals file. `stockings` always assumes you have given it well-formed goals and there is enough compound in each well to go around. Furthermore it assumes there's an entry in the calibration dictionary for each container. If the calibrated container is not found in the calibrations file the Opentrons API is accessing, `stockings` will exit with an exception. The alternative is just guessing at well locations and sizes, a recipe for damaging equipment.

`stockings` has functionality to pick up several destination well's worth of solution at once, saving on trips and therefore time. However, if only some of the solution needs to be dispensed, a hanging droplet can form on the tip. 'Blowing out' the pipette prevents this from happening if only one volume is being dispensed, but it requires emptying the tip completely. Human pipettors can touch the droplet to the side of the container to dislodge it if one forms. Performing a 'tip touch' with the autopipettor is possible, but it will sometimes contaminate the tip because the robot cannot see if a different droplet is present on the side of the container. To prevent droplet problems `stockings` can pick up several destination well's worth of compound at once, spacing each dose out with an air bubble inside the pipette tip. While dispensing, `stockings` dispenses the

compound solution, then the air bubble to prevent droplets from clinging to the tip and introducing error. This works fairly well, but is not the default behavior.

The state file is just a list of containers, each entry being a dictionary that defines its current volume in any yaml-compatible format. The goals data structure/spreadsheet is a big matrix with columns representing source wells and rows representing destination wells. The name of the container matters: a 'compounds' container can only be in one place. A 'compounds' container in slot A1 and a 'compounds' container in slot A2 will confuse the calibration file and only one will turn up. Additionally, there's no way to figure out which one you mean in the goals file. The reason why coordinates are given by container and not by slot is to allow different kinds of containers. A tube rack in the A2 slot has different calibration data from a 96 deep well plate. Attempting to access a container the wrong shape has potentially disastrous consequences, but the Opentrons library will actively complain if attempting to access a container that is uncalibrated and unsafe to access. The name of the container has to be unique, because the Opentrons API uses the container name to look up the container's properties, including deck location. The top-left cell must be 'wellname' or `stockings` will not know which of the columns is the name of the destination well, due to an implementation detail. Destinations are specified using 'containername'.'wellname'. Exotic containers featuring strange well names are supported if the Opentrons library can find the Placeable object by calling .wells('name'). `bartender.py` will not emit goal spreadsheets using strange well names, but `stockings` supports it anyway. There is a break in the pipeline between `bartender.py` and `stockings`, as there is between `branch_and_bound.py` and `bartender.py`, where human input is needed. At some point between `branch_and_bound.py`

and `bartender.py`, a human will have to make up stock solutions and update a layout file. It's unfriendly to require a layout just to reason from known growth conditions, and as a result the layout file isn't guaranteed to be present. The design loop containing `branch_and_bound.py` involves checking growth information, and the design loop containing `bartender.py` involves checking stockroom supplies. The two are so different that it makes sense to separate the two stages. There is no design loop containing `do_goals.py`. Once you have called `do_goals.py` you have to wait for inoculations to grow before doing any more reasoning.

It's worth noting if the run crashes `stockings` will update and save the volumes remaining in wells it knows about. It also does this if it doesn't crash. It will not modify the goals file to reflect how much it has successfully done. NB: This breaks one of the promises made to `stockings`: if called with a goals file and a state file, the state contains enough of all the components to complete the goals. However, once something has crashed `stockings` can no longer guarantee what, if anything, is in the goal wells. `stockings` could crash gracefully, update the state file, and check whether the goals are still possible. This has specifically been omitted as a design choice. By definition an unexpected error is unexpected, and the last thing an automated system should be doing is trying to fix an unexpected problem it is unequipped to diagnose or solve.

Where possible, each of the data structures passed back and forth by different parts of robot-trifle has an example in the body of the code, and when called without arguments writes these example structures to example .yaml files, as well as running a test using those example structures. A design problem common to dynamic languages without extremely strong typing systems is structure rot, where the data structure passed into

one function must be quite complex and there is no documentation about what it actually looks like, and the structure must be reverse-engineered from the function itself. The example structures are both documentation and development tool, keeping the format consistent and checking adherence to the actual format promised at the beginning of development.

They also serve a vital user interface role, allowing anyone who wants to define a media to just copy the example file and extend, supplement, or replace it without having to write a whole file from scratch and guess at the conventions and formatting. Unfortunately the .yaml files adhere rigorously to Postal's Law: "Be conservative in what you send, and liberal in what you accept.". .yaml files are therefore written with maximum type information, even though it's not necessary in this context. These automatically generated example files act like templates. I have tried to make them as complete as possible while remaining short. The data structures robot-trifle uses aren't particularly complex, and enforcing that they remain simple helps other users borrow parts of robot-trifle functionality for other projects without having to reverse-engineer whole systems.

## CHAPTER 3: BRANCH_AND_BOUND

It's often illustrative to walk through an entire process step-by-step to show how all of the parts are interlinked. The robot-trifle pipeline begins by creating a pair of files: `knowns.yaml` and `media.yaml`. These files have examples to show the formatting (`example_knowns.yaml` and `example_media.yaml` respectively). The example media is in units of moles. Once these files are defined, call `branch_and_bound.py`. If called without any arguments the script will print a usage hint and perform a test minimal media exploration.

When called with files, `branch_and_bound.py` quickly jumps to the actual behavior(line 107)

```
    knowns = yaml.load(kf)
with open(argv[2],'r') as mf:
    media = yaml.load(mf)
list_media = set(list(media.keys()))
proposals = []
known_oracle =
    partial(oracle,knowns=knowns,proposed=proposals)
```

This code reads the names of the input files (`argv[1]` is the first argument supplied to `branch_and_bound.py`, which should be the growth data file. `argv[2]` is the second argument, which should be the media definition file.) Because `branch_and_bound.py` doesn't take into account concentrations of components, only their presence or absence, the media definition has to be converted into a set of names. The list of new proposed experiments starts out empty, naturally.

```
    print("minimal
        ",sibling_sharing(list_media,[],known_oracle))
```

sibling_sharing is a function that takes a list of component names, a list of media components known to be essential, and a function that acts as a predictor of growth. It returns a list of media conditions where every remaining media compound is essential. It does most of the heavy lifting in this part of the analysis, but the setup can be a little hard to follow. This line defines a 'partial function' using the growth data and the list of proposals so far(this will be an empty list at the moment), using the existing function oracle.

```
  if media in knowns["no_growth"]:
    return False
  elif media in knowns["growth"]:
    return True
  else:
    proposed.append(media)
    return False
```

The oracle function checks a set of growth data and reports either that the condition is known to support growth, or known not to support growth. If growth is unknown, oracle saves this condition to a list of conditions called proposed and reports False. If it were to report growth(True) on this unknown condition, the algorithm would continue to inquire about further media modifications. Further removals might be a waste of time, depending on whether the organism actually grows on the unknown condition or not.

Returning up a level, the partial function known_oracle has been pre-loaded

with all the growth data we currently have, and an empty list ready to store all the unknown conditions asked of it.

```
    print("minimal
       ",sibling_sharing(list_media,[],known_oracle))
```

sibling_sharing is ready to be called, and is passed the current working media, a list of compounds known to be essential(none yet), and the oracle function just constructed. It will return a list of minimal media conditions, where no removal can be made without abolishing growth.

```
def sibling_sharing(media,dead_ingredients=[],oracle=lambda x:
   len(x)>0):
  #media is a set()
  solutions = []
  for ing in media:
    if ing in dead_ingredients:
      continue
    smallmedia = set([i for i in media if i is not ing])
    result = oracle(smallmedia)
    if not result:
      dead_ingredients.append(ing)
    else:
      # the full slice [:] copies the list.
      new_sol =
         sibling_sharing(smallmedia,dead_ingredients[:],oracle)
      for sol in new_sol:
```

```
            if sol not in solutions:
              solutions.append(sol)
    if len(solutions)==0:
      #all are fatal, this is minimal
      return [media]
    return solutions
```

This is the full definition of `sibling_sharing`. First, walk through all the ingredients in the working media we have. If any have been shown to be essential, skip them. If they haven't been skipped, construct a media missing that ingredient, and ask the oracle about it. If the oracle reports that growth isn't supported in the media with that ingredient deleted, add it to the list of essential components and move on.

If the oracle reports that growth *is* supported, recurse and ask `sibling_sharing` for a complete list of media conditions where growth is supported using only the remaining components. If any of those new solutions are duplicates, skip them. It's important to pass the recursive function a *copy* of the list of known essential components, so that the list of essential components at this level of the search tree is untouched.

If there are no solutions other than the current media condition, no further removal can be made without preventing growth, and the current media condition can be passed up the 'parent' call of `sibling_sharing`.

Duplicate solutions can happen under certain circumstances. If there are two unnecessary components(A and B), the search will check whether removing A from the media abolishes growth, and then recurse and check whether removing B in the absence of A abolishes growth. Later it will check the removal of B in the presence of A, and the

removal of A in the absence of B. This produces two solutions from different branches of the search tree that are nevertheless identical.

It's worth noting that this inefficiency is computational only, and only occurs when A and B are known to be nonessential and growth data supports all four media conditions support growth. If A is essential, the search will not check if deleting A in the absence of B abolishes growth. All real metabolisms have this kind of non-convex behavior, but by limiting the search space to compounds and media components with low toxicity at relatively low concentrations most non-convex behavior can be avoided. If arbitrarily complex metabolic interactions are modeled, finding all growth conditions can only be accomplished by exhaustively testing every combination.

```
print("proposed new experiments: ",proposals)
with open("new_proposed.yaml","w") as pf:
  yaml.dump([list(p) for p in proposals],pf)
```

The variable proposals now contains all the media conditions that are as small as possible given the available growth data, and these proposals are printed as well as saved to a file for later processing.

## CHAPTER 4: BARTENDER

Once `branch_and_bound.py` has completed, `new_proposals.yaml` will be populated with the next set of experiments that need to be conducted. `bartender.py` takes this experiment set (`new_proposals.yaml`), the original media definition (`media.yaml`), a small file containing information about the destination plate (`goal.yaml`), and a complete specification of what components are available and at what concentration. Units are in moles and microliters.

When called, `bartender.py` loads the files it's been passed, does a very tiny amount of formatting, and calls `proposals`. Not the best function name.

```
def proposals(proposed =[],media=example_media,goal=ex_goal):
  dest_wellnames = [goal["name"]+"."+offset_name(i) for i in
     range(0,goal["wells"])]
  sorted_wells = sorted(dest_wellnames)
  num_replicates = int(goal["wells"]/len(proposed))
  spec_dict = {dw:{} for dw in dest_wellnames}
  for i,prop in enumerate(proposed):
    for j,wn in enumerate(sorted_wells):
      if (j<num_replicates*i) or (j>= num_replicates*(i+1)):
        for cpd in prop:
          spec_dict[wn][cpd]=media[cpd]*goal["vol"]#in umols
  return spec_dict
```

`offset_name` is a small utility function defined on line 55, that just converts to the 'name' of a well('A1','A2', etc) from its numerical offset. Well 0 is 'A1' and well 95

is 'H12'. The opentrons api accepts either format, but for human reading purposes the wells are named with their coordinates. `proposals` converts from a list of proposed experiments to a specific list of wells, each one with a list of media components and how many micromoles of each are required. This data structure is stored in `spec_dict`, and later it will be referred to as a 'plan'.

```
    m,a = flatten(plan,layout)
```

This 'plan' is passed to `flatten`, a function that converts a dictionary of dictionaries to a simple spreadsheet structure. `flatten` is a long function, but the first 20 lines are mostly prep work.

```
def flatten(plan,layout):
  totals = defaultdict(int)
  locations = defaultdict(list)
  for container in layout.keys():
    for well,contents in layout[container].items():
      totals[contents[0]]+=contents[1]*contents[2]
      locations[contents[0]].append((
        container+"."+well,
        contents[1],
        contents[2]))
```

The first task is to take stock of exactly how much compound is available, and where each compound is stored. This is pretty straightforward enumeration of every well in every defined container, keeping a running total of every media component. In addition, the lookup table `locations` is populated with a list of where to find each component.

```
def get_from_locations(cpd,umoles,locations):
  for i,loc in enumerate(locations[cpd]):
    source,volume,molarity = loc
    if volume*molarity>= umoles:
      amt = umoles/molarity
      nt = (source,volume-amt,molarity)
      #i have to do it this way because tuples are immutable.
      locations[cpd][i] = nt
      return(source,amt)
  raise Exception("I won't pool the last little bits of this.
     I need more %s" %compound)
```

`get_from_locations` is a function declared locally, inside the scope of `flatten`, because it's useless without the running totals and lookup tables. Generally software parts should be interchangeable and re-usable, but in this case I'm using a function definition to draw a border between parts of code that know about molarity calculations and code that does not. In this case `get_from_locations` has to know about concentrations in order to keep track of how much component it has allocated already.

`get_from_locations` is not an optimal allocator. If two source wells containing 30 micromoles of compound are available, there are 60 micromoles in total. If 20 micromoles are requested three times, `get_from_locations` will raise an exception. This is a fail-fast safety valve to avoid situations where the last few drops of a component need to be pooled from multiple wells. The autopipette is particularly inaccurate in that kind of situation, for a number of reasons. The pipette tip performs poorly trying to

vacuum up droplets, mostly because the autopipette can't see or feel. The tip itself can jostle the plates by colliding with them, surface tension can influence precision, and accidentally incorporating air bubbles is much more likely. `get_from_locations` is not foolproof, but it also won't make an extra effort to do the wrong thing.

```
  matrix = [{"wellname":k} for k in plan.keys()]
  allsources=["wellname"]
  for m in matrix:
    ingredients = plan[m["wellname"]]
    for ing,umols in ingredients.items():
      totals[ing]-=umols
      if totals[ing]<0:
        raise Exception("Not enough %s to complete a transfer
            spreadsheet." % ing)
      (source,amt) = get_from_locations(ing,umols,locations)
      if source not in allsources:
        allsources.append(source)
      m[source]=amt
  return(matrix,allsources)
```

The remaining part of `flatten` does all of the remaining work. The built-in library `csv` accepts a list of dictionaries, so `flatten` creates one. In this spreadsheet, every source well for a media component is a column, and every destination well(the experimental ones) is a row. Each entry is the number of microliters that need to be transferred from that column to that row. As the list of dictionaries is being built, a complete list of every source well that's been used is maintained.

```
def write_to_file(matrix,allsources,file):
  with open(file,'w') as fakefile:


    cwriter = csv.DictWriter(fakefile,allsources,restval=0)
    cwriter.writeheader()
    for line in matrix:
      cwriter.writerow(line)
```

This function just leverages `csv` to export the list of dictionaries to a spreadsheet. This is why the top-left cell needs to be 'wellname', so that all the names of the destination wells are in the left-most column.

```
    with open('statefile.yaml','w') as sf:
      yaml.dump(convert_layout_to_state(layout),sf)
```

This little snippet is particularly useful. Having defined a layout file to allow component allocation, it didn't make sense to make the user write out a very similar(but not identical) state file for the next step. This automatically converts the layout file just used to make a goals spreadsheet into the state file. The state file is almost exactly like the layout file, but it is completely agnostic about components and concentrations.

## CHAPTER 5: STOCKINGS

The final step in the pipeline is actually queueing commands on the robot and executing them. The script for this is `do_goals.py`, and it requires a goals spreadsheet and a .yaml file with the current volumes of all the containers on the deck.

```
  with setup_robot() as robot:
    accomplish_goals(goals,state,robot)
    print("%s commands queued" % len(robot.commands()))
    robot.run(True)
```

The meat of `do_goals.py` is all here, and it's essentially a direct call to `stockings`.

```
@contextmanager
def setup_robot():
  robot = Robot()
  for port in list_of_ports:
    trycon(robot,port)
  if( robot.is_simulating()):
    print("no connection made")
  else:
    robot.home()
    robot._driver.speeds['z']=1200
  for (axis,pipette) in robot.get_instruments():
    pipette.load_persisted_data()
  try:
    yield robot
  finally:
```

```
    robot.disconnect()
```

This is a particularly elegant solution to the persistent robot connection problem. In order to queue commands or communicate with the robot at all, a connection must be opened. If that connection is not later closed, the robot will stay 'on', wasting power and heating up slowly. This setup function creates a context manager. When you enter the `with` block, the robot object is created. When the `with` block is exited, control returns to the setup function and the robot will be disconnected, even if the code inside the block crashes in the interim.

This block also loads pre-existing calibration data found in `calibrations/calibrations.js` and sets the z-motor speed a little slower to fix a motor resonance issue. If the robot stops moving on the z-axis and makes a horrible screeching noise instead, change this number.

```
def accomplish_goals(goals,deckstate,robot=Robot()):
  pip = robot.get_instruments()[0][1]
  #do stuff with deckstate to set current volumes of all the
     wells
  for container_name in deckstate.keys():
       container_instance =
          robot.deck.containers()[container_name]
       for wellname in deckstate[container_name].keys():
         container_instance.wells(wellname).vol =
            deckstate[container_name][wellname]
```

The work of actually executing the goals contained in the goal file provided to `do_goals.py` is accomplished inside the body of `accomplish_goals.py`. This function assumes

that pre-existing calibrations have been loaded successfully. As part of the code to get compounds from very large wells, the data structure representing the well itself is extended to include its current volume, and these first four lines just read the volume spelled out in the state file and update the data structure.

```
  try:
    for source in goals.keys():
      source_container, source_well = source.split('.')
      s_well =
         robot.containers()[source_container].wells(source_well)
      for destination, destination_amount in
         goals[source].items():
        #this needs some cleverness but it'll do for now
        dest_cont,dest_well = destination.split('.')
        d_well = robot.containers()[dest_cont].wells(dest_well)
        get_from_fat_well(float(destination_amount),s_well,pip)
        pip.dispense(d_well)
```

The goals are accomplished one component at a time, so that tips can be re-used for transporting the same component to different destination wells. Ideally a fresh tip would be used for each and every transfer, but laying out a ten-component experiment on a 96 well plate would use up almost a thousand tips. At a certain point the opentrons robot deck can't accommodate enough tip boxes to make that possible.

`get_from_fat_well` is a specialized version of the opentrons `aspirate` function, and in fact calls `aspirate` internally. When laying out this kind of media experiment,

some components are provided in large reservoirs. When dealing with deep wells, the default behavior of just drawing from the bottom of the well can result in significant displacement problems. This function guesstimates where the surface of the fluid is likely to be based on how big the well is and how much has been withdrawn from it already. Care is taken not to try and aspirate from below the bottom of the well, which would collide the tip with the plate.

# CHAPTER 6: DISCUSSION

This toolkit provides a complete package of computational automation resources to streamline metabolic profiling experiments. The scope of the problems addressed by this pipeline is fairly limited. This pipeline partially automates finding chemically defined minimal media when a chemically defined non-minimal media is known. In addition, robot-trifle has a limited ability to leverage pre-existing data about microorganism growth to accelerate the microbial metabolic profiling process.

Luckily, the existing literature is full of chemically defined non-minimal media conditions and poorly characterized microbial species. Isolating and characterizing precise metabolic profiles for these species will illuminate community interactions and help tease apart microscopic ecosystems. Without these metabolic profiles, the fundamental structure of microbiomes is opaque. Microbiome structure includes information on which members produce which nutrients for the survival of the whole.

For example, symbiotic communities in the mouth allow strict anaerobes like Streptococcus mutans to grow, thrive, and produce the acid that dissolves enamel. In the short term S. mutans cannot survive without the other community members, but in the long term S. mutans provides a shelter and reliable food source for other mouth microbes. The other necessary members of this community are as yet unknown. Many species have been found living in these communities, but which species are symbiotic and which are merely commensal, parasitic, or coincidental is an open question. Which species drive the formation of microbial communities is critical to understanding and predicting the response of biofilms to environmental conditions. It may be possible to attack pathogens by attacking their symbionts, or install protective or benign microbiomes

to pre-empt infection or colonization by pathogenic species.

Future work in this area should focus on expanding the capability of robot-trifle or other automated reasoning systems to draw conclusions from existing data, including previously known growth conditions, detailed information about closely related taxa, and predictions made by metabolic modeling software. Incorporating probabilistic data will require a much more sophisticated Bayesian approach, maximizing information gained with each experiment set rather than a simple methodical search.

Perhaps it is overly optimistic to think that one day laboratory workers of all kinds will be able to say "oh, I'll just put it on the robot" whenever confronting a particularly pipette-intensive experiment, and not have to spend any time learning anything about programming in order to do so. On the other hand, the barriers to such a future are primarily financial and a lack of mature software. The 3D printing explosion is lowering the costs, and this project and others like it will make using the new hardware much simpler.

## REFERENCES


[1] UniProt, "UniProtKB/TrEMBL 2017_11." http://www.uniprot.org/statistics/TrEMBL.

[2] RCSB PDB, "RCSB PDB - Holdings Report." https://www.rcsb.org/pdb/statistics/holdings.do.

[3] NCBI, "GenBank and WGS Statistics." https://www.ncbi.nlm.nih.gov/genbank/statistics/.

[4] L. Grivell, "Mining the bibliome: searching for a needle in a haystack? New computing tools are needed to effectively scan the growing amount of scientific literature for useful information.," *EMBO reports*, vol. 3, pp. 200–3, mar 2002.

[5] J. T. Grudin, "Error Patterns in Novice and Skilled Transcription Typing," in *Cognitive Aspects of Skilled Typewriting*, pp. 121–143, New York, NY: Springer New York, 1983.

[6] F. R. Blattner, G. Plunkett, C. A. Bloch, N. T. Perna, V. Burland, M. Riley, J. Collado-Vides, J. D. Glasner, C. K. Rode, G. F. Mayhew, J. Gregor, N. W. Davis, H. A. Kirkpatrick, M. A. Goeden, D. J. Rose, B. Mau, and Y. Shao, "The complete genome sequence of Escherichia coli K-12.," *Science (New York, N.Y.)*, vol. 277, pp. 1453–62, sep 1997.

[7] Eppendorf, "epMotion®5070 - Liquid Handling Workstations, Automated Pipetting - Eppendorf." https://online-shop.

eppendorf.com/OC-en/Automated-Pipetting-44509/
Liquid-Handling-Workstations-44510/ep5070-PF-68886.html.

[8] C. S. Harbor, "M9 minimal medium (standard)," *Cold Spring Harbor Protocols*, vol. 2010, p. pdb.rec12295, aug 2010.

[9] C. S. Harbor, "M9 Salts," *Cold Spring Harbor Protocols*, vol. 2006, p. pdb.rec614, jun 2006.

[10] B. D. Davis and E. S. Mingioli, "MUTANTS OF ESCHERICHIA COLI REQUIRING METHIONINE OR VITAMIN B1," vol. March, 1950.

[11] Opentrons, "Opentrons." https://opentrons.com/robots/ot-one-s-hood.

[12] A. C. R. Tanner, J. M. J. Mathney, R. L. Kent, N. I. Chalmers, C. V. Hughes, C. Y. Loo, N. Pradhan, E. Kanasi, J. Hwang, M. A. Dahlan, E. Papadopolou, and F. E. Dewhirst, "Cultivable anaerobic microbiota of severe early childhood caries.," *Journal of clinical microbiology*, vol. 49, pp. 1464–74, apr 2011.

[13] K. Yasuda, T. Hsu, C. A. Gallini, L. J. Mclver, E. Schwager, A. Shi, C. R. DuLong, R. N. Schwager, G. S. Abu-Ali, E. A. Franzosa, W. S. Garrett, C. Huttenhower, and X. C. Morgan, "Fluoride Depletes Acidogenic Taxa in Oral but Not Gut Microbial Communities in Mice.," *mSystems*, vol. 2, no. 4, 2017.

[14] J. M. Dreyfuss, J. D. Zucker, H. M. Hood, L. R. Ocasio, M. S. Sachs, and J. E. Galagan, "Reconstruction and Validation of a Genome-Scale Metabolic Model for

the Filamentous Fungus Neurospora crassa Using FARM," *PLoS Computational Biology*, vol. 9, p. e1003126, jul 2013.

[15] A. Kouzuma, S. Kato, and K. Watanabe, "Microbial interspecies interactions: recent findings in syntrophic consortia.," *Frontiers in microbiology*, vol. 6, p. 477, 2015.

[16] M. Costalonga and M. C. Herzberg, "The oral microbiome and the immunobiology of periodontal disease and caries.," *Immunology letters*, vol. 162, pp. 22–38, dec 2014.