A GENETIC PROGRAMMING TECHNIQUE FOR PROTEIN OPTIMIZATION DEMONSTRATED IN MRI REPORTER GENES

By

Alexander Robert Bricco

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Biomedical Engineering—Doctor of Philosophy

2024

ABSTRACT

Reporter genes are important tools for researchers studying molecular and cellular biology as they give location and measurable values to the expression level of a given gene, as reporter genes link the activation of a gene to a detectable phenomenon. Reporter genes for MRI, allow these functions to be done at arbitrary tissue depth and noninvasively. Chemical Exchange Saturation Transfer (CEST) based reporter genes have shown promise in acting as reliable reporters in MRI but the relatively low sensitivity to the method has decreased its utility in research situations. Initial attempts to optimize existing CEST reporter genes proved difficult due to a series of technical challenges. This led to the development of a new protein engineering machine learning tool, the Protein Optimization Engineering Tool (POET). Using a process where POET and experimentation were used to develop improved CEST reporter genes resulted in new peptides that produce nearly a fourfold increase in contrast over prior art. Additionally POET is used to generate a reporter gene that produces significant contrast at a farther downfield frequency than prior CEST reporter genes.

I dedicate this work to my wife, Emily, and my Lord, Jesus Christ, who when I lacked strength enough to continue, lent me some of theirs.

ACKNOWLEDGEMENTS

This work would not have been possible without the guidance and assistance of many others. I would like to thank my collaborator and friend Illiya Miralavy who wrote the first (and several subsequent) version of POET, Dr. Jory Schlossau who optimized and improved it, and their advisor Dr. Banzhaf for guiding its development.

I would also like to thank Dr. Michael McMahon at Johns Hopkins University for guiding me through learning CEST MRI and keeping the project moving forward when I was still learning. Further thanks go to Dr. Christian Farrar, and his lab at Harvard Medical School for providing the information on exchange rates that help complete this project's flow.

This work was made possible via funding from NIH/NINDS: R01-NS098231; R01-NS104306 NIH/NIBIB: R01-EB031008; R01-EB030565; R01-EB031936; P41-EB024495 and NSF 2027113.

Lastly, but mostly, I would like to thank my advisor, Dr. Assaf Gilad, who introduced me to the project, taught me countless things about science and served as an exemplary role model for what it means to be a professor and run a laboratory.

TABLE OF CONTENTS

INTRODUCTION

Many portions of this section are adapted from a currently published paper[1].

Magnetic Resonance Imaging (MRI)

Magnetic Resonance Imaging (MRI) is a field of biomedical imaging that uses powerful magnetic fields to align the magnetic moments of unbalanced nuclei. MRI is widely used clinically to inform the diagnosis of a wide variety of diseases[2].

In an MRI scanner a strong homogenous magnetic field (B_0) is applied. This field causes the spin of unbalanced nuclei to align the field and allows them to resonate. Not all nucei in within the field are aligned, with the number of aligned nuclei being associated with the strength of the B_0 field. In clinical practice scanners have a B_0 field with a strength of 0.5T-3T, while preclinical scanners typically achieve strengths 7T-15T.

Along with aligning their magnetic moment, the nuclei spin around the axis of the B_0 field in a motion called precession. The Lamor frequency is the rate of precession and is linearly related to the strength of the B_0 field. Within MRI, frequency is often expressed in the number of millionths of the Lamor frequency (ppm), so for a 7T preclinical MRI would have a Lamor frequency of ~300 MHz for a hydrogen nucleus and one “ppm” on that magnet would represent 300 Hz for a hydrogen nucleus.

These nuclei can be shifted out of alignment with the B_0 field by exposing them to a radio frequency (RF) pulse in a perpendicular magnetic field (B_1). At the cessation of the RF pulse the nucei are in a higher energy state, referred to as excited, and decay from this high energy state to realign with the B_0 field in a process called relaxation. As these nuclei relax they produce a measurable change in the magnetization which is detected often by the same coils used to

generate the B_0 field and is the signal measured in MRI[3].

Chemical Exchange Saturation Transfer (CEST)

By applying an RF pulse before the imaging RF pulse, a nucleus may be saturated. When an imaging pulse is applied to these saturated nuclei it results in the nuclei being aligned antiparallel to the B_0 field. As these saturated nuclei undergo relaxation, they produce a negative signal as they drop to the normal excited state in addition to the normal signal as they relax from excited to their normal state. These signals cancel each other out and as a result can be said to produce no signal.

Although the B_0 field is the major contributor to the magnetic field experienced by a given nucleus undergoing MRI there is also an effect from the surrounding chemistry. Each nuclei surrounding the nucleus being imaged has its own magnetic field which produces a change in the Lamor frequency for those nuclei called a chemical shift[4]. This allows these nuclei to be saturated or excited specifically without saturating or exciting other nuclei in the surrounding area. The change in magnetic field, and thus the change in Lamor frequency is constant for a given chemistry. This allows for the specific saturation of protons associated with a given functional group.

If a saturation pulse is sustained, and the nucleus is a in a molecule where it can be exchanged with the solvent (such as a hydrogen nucleus in a hydroxyl group)[5], this saturated proton may be exchanged with the environment. If this exchange is fast enough, the bulk solvent will begin to have enough signal loss to generate a measurable decrease in signal, despite the low concentration of the exchangeable nucleus's chemical solute. This mechanism of contrast is called chemical exchange saturation transfer (CEST)[6]. This process can be seen graphically in

figure 1.

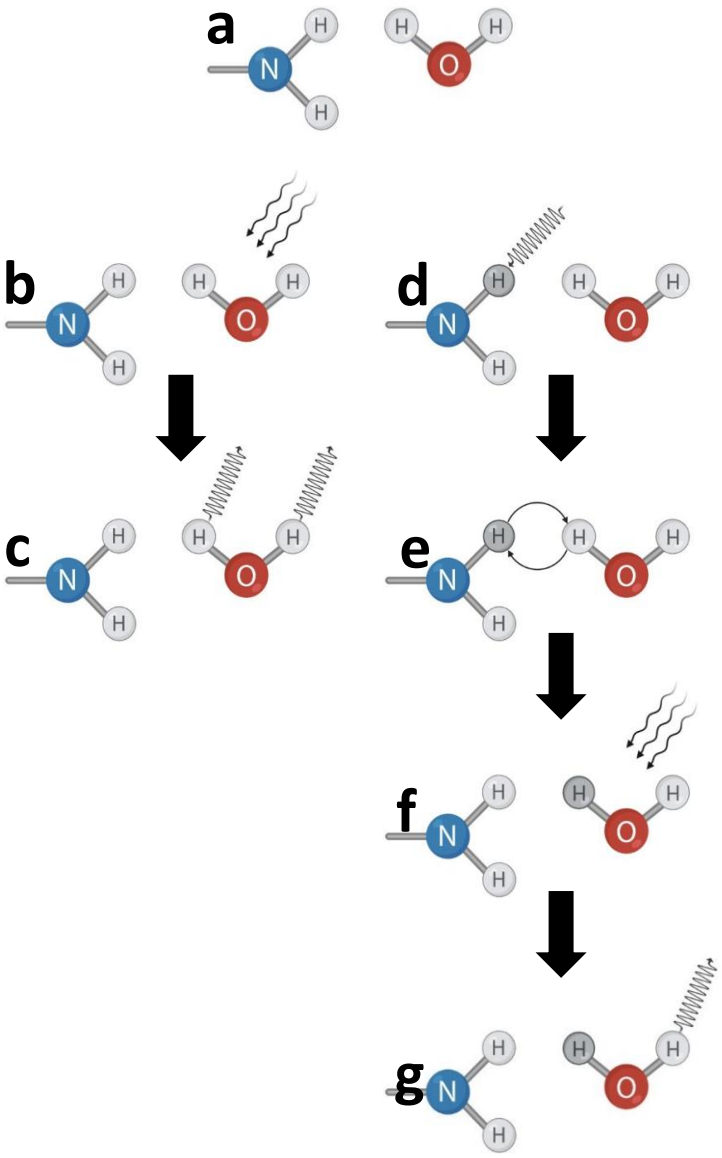


Figure 1: *Visualization of CEST MRI. An organic amide sits in solution with water (**a**). In a normal T1 weighted MRI scan a RF pulse is applied to the hydrogens in the water (**b**) and this results in signal (**c**). In a CEST scan, first a saturation pulse is applied, which saturates a proton on the amide (**d**). The saturated proton is then exchanged with the protons in the surrounding water (**e**). When the RF pulse is applied to produce an image (**f**), the saturated proton doesn't produce a signal, resulting in reduced signal (**g**).*

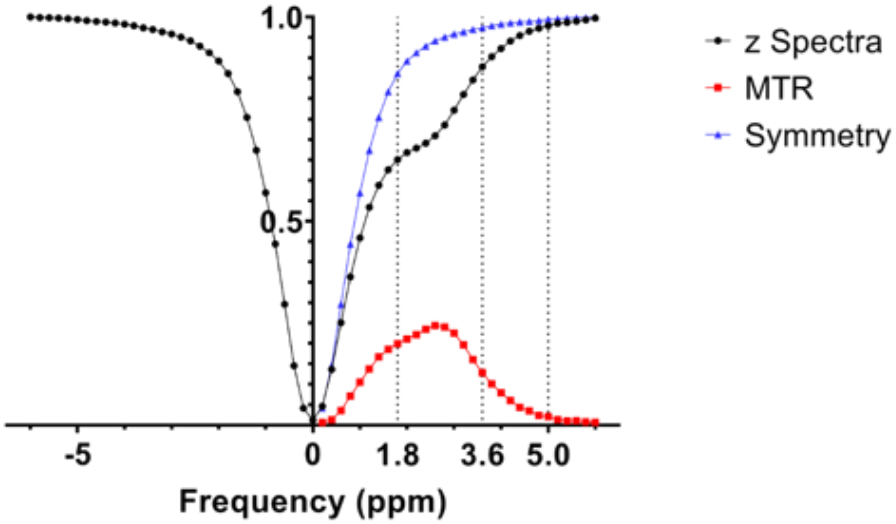
In the most ordinary example, a hydrogen nucleus (frequently simply called a proton) is exchanged with the water from an exchangeable organic group such as a hydroxyl or amide[7]. In CEST MRI 0 ppm is the water saturation frequency, and the subject is typically exposed to a series of saturation pulses at different frequencies symmetrical around the water saturation frequency, which is called a z-spectra. The effect of CEST after a saturation pulse is visible as a decrease in signal in the area where the exchangeable protons are present. This localized decrease in signal is called CEST contrast. CEST contrast is quantified by comparing the signal after a given saturation pulse with the signal produced at the frequency multiplied by -1. The difference between them is then generated divided by the signal at a frequency far off field from the water saturation frequency. The asymmetry due to the transfer of saturated particle results in the magnetization transfer ratio (MTR_{asym}) as shown in equation 1 below[8], where S^{ω} is the signal at a given frequency, $S^{-\omega}$ is the signal at the inverse of that frequency and S0 is the signal without any saturation. This can be seen visually in **figure 2**.

$$Eq.1 \qquad MTR_{asym} = \frac{S^{-\omega} - S^{\omega}}{S_0}$$

In addition to the MTR_{asym} it is important in experiments via correcting for mass and concentration, as larger molecules, can generate more contrast than smaller ones by virtue of having more exchangeable protons to saturate and higher concentrations of a CEST contrast agent produces more contrast. This correction is done by multiplying the MTR_{asym} by the concentration of protons in the solvent to the concentration of the CEST compound. This results in the proton transfer efficiency (PTE) of a given peptide see equation 2 below[8], where $[H_2O]$ is the molar concentration of H_2O in the sample and [peptide] is the molar concentration of the pepide. In this dissertation all PTE values are divided by 1000, for ease of communication.

$$Eq.2 \qquad PTE = MTR_{asym} * \frac{2 * [H_2O]}{[peptide]}$$

a



b

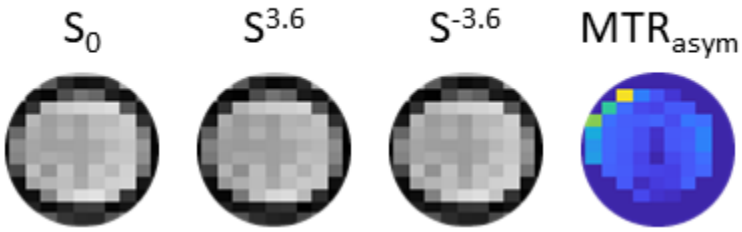


Figure 2: *Visualization of MTR calculation. (**a**) In black is the Z-spectra acquired from the highest contrast protein from the 5th generation of 3.6 development (see below). If the water was instead pure, it would be symmetrical across the 0 ppm point which can be seen in blue. The difference between the two is used to calculate the MTR_{asym} seen in red. (**b**) the three input calculations (S_0, S^{ω}, $S^{-\omega}$) for a frequency of 3.6 ppm offset from the water saturation frequency. S_0 was acquired without a saturation pulse. $S^{3.6}$ was acquired after a saturation pulse at 3.6 ppm offset from water. $S^{-3.6}$ was acquired after a saturation pulse at -3.6 ppm offset from water. The MTR_{asym} figure is the result of performing the calculation from equation 1 for the data on each pixel in the images.*

CEST contrast is generated naturally by any compound that exchanges imageable atoms with the surrounding solvent, including naturally occurring biomolecules. Carbohydrates produce contrast from their hydroxl groups, which has led to studies this hydroxl CEST signal as an indicator of metabolism and function in the brain[9] and in mouse livers[10]. Proteins produce contrast from the amine groups present in the backbone of their amino acid chain and are also able to produce contrast at other frequencies due to the presence of exchangeable protons in their functional groups. The saturation frequency used to generate CEST contrast from some functional groups common in biochemistry can be seen in **Table 1** below.

Table 1: Saturation frequency of common functional groups

Functional Group	Chemistry	Saturation Frequency(ppm)
Hydroxyl	R-OH	0.8
Primary Amine	R-NH_2	1.8
Secondary Amine	R-NH-R	3.6

Although functional groups produce contras at a consistent frequency, different compounds produce different amounts of MTR_{asym}. The primary influences are the number of exchangeable protons, the rate at which each of those protons exchange, and the accessibility of those exchangeable protons to the solvent[11,12].

Reporter Genes

In the study of genetics and molecular biology it can be difficult to tell cells apart from one another or to tell cells where a gene is active appart from cells where a gene is inactive. To overcome these challenges researchers employ reporter genes. A reporter gene is a gene, that when expressed, produces a measurable change that can be used to tell that the gene is currently being expressed and the level of expression. Typically this change is a result of a

protein being generated that has a detectable presence.

The first reporter genes were for optical imaging[13-15]. When these genes are expressed the proteins produced can generate light from chemical energy (bioluminescence), emit light at one frequency after stimulation with light from another frequency (fluorescence), or create a change in the color of a solution.

In MRI the initial efforts at reporter genes developed relied on generating T1 or T2 based contrast by binding metals within cells[16-20]. More recently there has been a demand for reporter genes that don't require an added substrate for them to work.

MRI reporter genes using CEST have shown great success[21], despite their relatively low sensitivity compared to T1 or T2 based reporter genes. The mechanism allows the generation of contrast without the addition of metals, and since the contrast requires a presaturation pulse, researchers can get a clear structural image without the effects of contrast.

The first reporter gene designed for CEST was the Lysine Rich Protein (LRP)[22,23] which true to its name was primarily a large series of lysines with other residues interspersed to improve expression and was designed from the observation of high contrast from poly-l-lysine (PLL), a synthetic lysine polymer, in prior experiments[24]. Later CEST reporter genes included human protamine (hPRM1)[25], and an adaption of the thymidine kinase system already used in positron emission tomography (PET)[26], both of which primarily produce contrast as a result of amine exchange at 3.6 ppm. There has also been work done to generate CEST contrast at other frequencies such as 0.8 ppm and 1.8 ppm[8], as when expressed alongside 3.6 ppm reporters they allow multiple different contrast agents to be detected in parallel for "multicolor" imaging. LRP remains the most used CEST reporter gene and has been used in the study of systems as

complex as. LRP remains the most used CEST reporter gene and has achieved a PTE of roughly 12.5 thousand. This leads to detection in the nM-mM range, allowing it to be used in the study of cardiac gene therapy[27] and oncolytic virotherapy[28], but leaves the analysis of smaller systems such as neuronal activation out of reach.

Protein Engineering

Proteins have been very useful for industrial, medical, and environmental purposes. However, having been developed in nature, few of these proteins are optimal for usage in the purposes that we have taken them for. Protein engineering is based on either generating a new protein to fulfill a task that there has yet to be a natural protein for, or to optimize a protein for a given task.

The most common mechanism for protein optimization is via directed evolution[29]. From a starting point of a natural protein, and a test capable to measuring that protein's fitness in the required task. After a base line is developed mutations are generated by a variety of means in the genetic code for the protein. Each of these mutant variants are then measured for their fitness. The most fit protein then becomes the new base line, new mutations are introduced into it. This process continues until the protein has hit the desired level of function or the experimenter chooses to stop. Although straight forward and easy to perform, directed evolution has difficulties where the process can be locked in a plateu or local maximum and be unable to find its way to the global maximum[30,31].

There have been many attempts to use computational methods to aid in protein engineering to help bypass the time limitations, and lab requirements of directed evolution. One notable example is Rosetta[32] which computationally determines the shape a protein will take allowing

researchers to attach pieces of other proteins to a common backbone[33], which makes it very easy for the design of a protein when the protein engineer knows what overall shape they are trying to achieve.

Since the ideal conformation of an engineered protein is sometimes unknown, machine learning tools have been developed to aid in protein engineering. A notable example is Protein Optimization with Optimal Learning (POOL) which makes use of Bayseyan classification[34]. There are other machine learning models[35] such as AlphaFold which uses deep learning to model protein structure[36], UniREP which uses a generalized understanding of protein structure to provide a generalized model[37] and more recently language models that predict interactions by sequence without developing a structure[38]. Existing machine learning methods of designing and optimizing proteins currently rely on large amounts of training data, and long proteins which provides increased context for understanding a given protein. Otherwise the training data that enables optimization tools like POOL to work relies on proteins that share a high degree of similarity existing within the training data.

Genetic Programming

Evolutionary Computation is a field in computer science, studying algorithms inspired by biological evolution. *Genetic Programming (GP)[39,40]* is among powerful evolutionary computation techniques that evolves solutions to difficult structural design tasks as a general problem solver. GP has been used in protein engineering before to predict key protein motifs[41], to evolve energy functions for evaluating protein structures[42], and predict protein-protein interactions related to disease[43]. This earlier work demonstrates the capability of GP to model features in the protein search domain, and in particular its ability to extract features relevant

for a prediction task. This is a central capability in biological applications where often high-dimensional inhomogeneous datasets are used as input to predict output values. In addition to creating predictive models, the underlying mechanisms of GP allows it to come up with novel models, often on first sight surprising or even counter-intuitive to the user[44]. Over the last decades, GP has proven to produce human-competitive solutions to many problems[45].

Significance

My work has pursued the development of new CEST reporter genes, as well as demonstrating the utility of a tool for generalized protein engineering. The reporter genes offer a wealth of new options for scientists to study the molecular biology and genetics of living creatures non-invasively through use of MRI. As the sensitivity of such reporters increase, smaller events such as the activity of neurons in the brain, could be measured using such reporter genes.

POET can help scientists develop proteins for a variety of purposes. Currently it has been used to engineer reporter genes or targeting proteins for extracellular vesicles (EVs)[46]. The approach of POET can be used to help develop any important small proteins, with such option as silk monomers[47], or pain killers[46].

METHODS

Directed Evolution

For directed evolution experiments I used an error prone polymerase kit as a means of introducing mutations into the gene of interest. The DNA target was replicated via a polymerase chain reaction (PCR) using the error prone polymerase. The resulting sequences were cloned using a TOPO TA Cloning Kit (ThermoFisher, K4600-01). pCR II-TOPO recombinant plasmids were transformed into One Shot TOP10 Chemically Competent E. coli (ThermoFisher, C404010). Bacterial colonies were grown on ampicillin-rich agar plates overnight at 37 °C. Colony PCR was performed using Quick-Load Taq 2X master mix (New England Biolabs, M0271L). Successful colonies were grown in 100 μg/mL ampicillin-rich lysogeny broth overnight at 37 °C. DNA was extracted using PureLink Quick Plasmid Miniprep Kit (Thermofisher, K4510-02) and then sent to MSU genomics core for DNA sequencing.

MRI

Scanners

MRI experiments were done on three different MRI machines. One of them was a 7T Bruker horizontal bore preclinical MRI, at the Biomedical and Physical Sciences Building (BPS) at MSU. The next was a 7T Bruker horizontal bore preclinical MRI at the Institute for Quantitative Health Science and Engineering (IQ). The last one was a 11.7T Bruker horizontal bore preclinical MRI that was used by our collaborator at Johns Hopkins University (JHU).

Pulse Program

CEST spectra, also known as Z-spectra were captured in scans that spanned -7 to 7 ppm offset from water frequency in units of 0.2 ppm. The saturation pulses have a saturation power of 4.7

μT and lasted for 4 seconds. To do B_0 correction a WASSR scan[48] was also performed spanning from -1.5 to 1.5 in units of 0.1 ppm. The saturation pulses for the WASSR scans had a saturation power of 1.25 μT and lasted for 2 seconds. Both the WASSR and CEST scans used a modified RARE scan with a RARE factor of 16 and a resolution of 48 X 48 covering a field of view of 40 mm x 40 mm, with a slice thickness of 3 mm. Both WASSR and CEST scans had a TR of 10s, and a TE of 4.74 ms.

The resulting scans results in a series of images, all covering the same geometric area that contains the entire phantom. Each image is taken after the application of a saturation pulse at a different frequency allowing for comparisons between the intensities of each voxel across the range of saturation frequencies.

To account for changes in the B_0 field over time a new WASSR scan was done before each CEST scan. Four to five Z-spectra were acquired in each experiment.

Phantom and Phantom Preparation

MRI experiments were done in a custom designed 3D printed phantom with 12 wells within a central body which contains water to prevent the magnetic field from distorting at the air water interface. Each well is designed to hold 250 μL of liquid. The circular distribution of the wells ensures that each well is experiencing a similar field strength. The shape of the phantom can be seen in **figure 3** below.

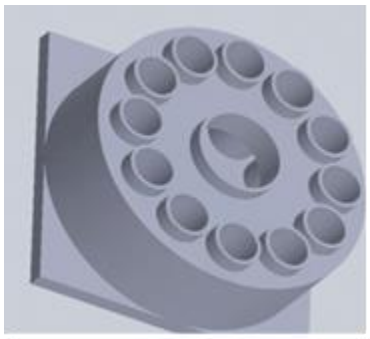


Figure 3: *Imaging Phantom. A 3D image from the software used to design the phantom.*

Proteins were ordered from Genscript (Piscataway NJ). All samples were dissolved to a concentration of 5 mg/mL in PBS. After dissolution, pH was correct for each sample's to 7.4 via titration with 0.1M HCl & 0.1 M NaOH and measuring the pH with pH paper.

In a typical experiment five experimental proteins would be used. In addition to these, there would be wells of PBS, a negative control; salmon protamine (sPRM), a positive control; and a peptide of twelve consecutive lysines (K12), which is used as a comparison to LRP, since LRP represents the contrast generated by currently used CEST reporter genes.

Data Processing

CEST Processing

The data from all MRI scans were processed using an in-house MATLAB script based on prior literature[49,50]. First the ROIs are isolated automatically using a greedy region growing algorithm, which finds the highest intensity voxel not already in a mask. The program then checks the voxels adjacent to all the voxels in the mask, and if they aren't in another mask and produce at least 55% of the intensity of the initial voxel of the mask they are added to the mask. The mask grows until it can't acquire additional voxels, and then a new voxel is selected as above. Masks including areas that aren't part of one of the phantom's wells are excluded if they aren't

circular, determined by their ratio of diameter to area, or of an unusual size. All masks are visually confirmed before use in image processing.

B_0 correction is done using the data from the WASSR scan[48] done immediately prior to the scan that is being corrected. In this scan a spline interpolation of the Z-spectra is used to determine the true central point of the water saturation peak. Later in the process this B_0 map is used to correct the Z-spectra for each voxel via a spline interpolation.

From a single CEST experiment the contrast values are determined by averaging all voxels within the same ROI. All repetitions of the same scan are averaged together to determine the final MTR_{asym} value for a given well, and all the voxels in a well are averaged together to achieve a single value for each peptide at each frequency. The PTE values are determined by normalizing the values according to water, adjusting them to account for molar concentrations and then correcting them the literature value of K12 of 12.5*1000[8] to retain consistency across the different scanners used by us and our collaborators. For ease of communication and entry into POET (see below), these PTE values are divided by 1000.

T-test mapping

To the generate t-test maps used in this dissertation, I used an in-house MATLAB script based on prior literature[49,50]. The intensity of each voxel across five experiments at 3.6 ppm was compared to the intensity of the same voxel at -3.6 ppm using a two tailed unpaired t-test. The resulting p-value was assigned to the same 2d location of the voxel to produce the map.

Protein Optimization Engineering Tool (POET)

POET is a genetic program that was developed in a collaboration between the Banzhaf and Gilad labs, for the purpose of improving the function of peptides. POET uses genetic

programming as its means of learning.

As input data POET takes in a table of a series of peptides as amino acid sequences using single letter amino acid codes and a numeric measure of fitness of the protein. These values are loaded into POET in the form of a comma separated value (CSV) file, the first 10 lines of the training data can be seen **Table 2** below. Within this work, the training data of POET uses the PTE values for the appropriate frequency (3.6 ppm for project 2, and 5.0 ppm for project 1) as the fitness value for each peptide, but the concept is deliberately vague, and can be any quality of a peptide that can be numerically quantified and the experimenter wishes to optimize. This open definition of fitness in POET allows it to be used in a variety of applications. PTE was chosen as the measurement for fitness to allow the use of existing data from literature in training POET.

Table 2: Example POET input data

Sequence	Fitness
KKKKKKKKKKK	12.5
KSKSKSKSKSKS	17
KHKHKHKHKHKH	12.7
KGKGKGKGKGKG	10.8
KSSKSSKSSKSS	13.2
KGGKGGKGGKGG	11.8
KSSSKSSSKSSS	13
KGGGKGGGKGGG	12.1
RRRRRRRRRRR	22
RSRSRSRSRSRS	12.8

POET generates models called Protein Optimization Evolved Models (POEMs), which predict protein fitness as a function of a peptide's sequence. Each POEM consists of a series of rules.

Each rule is composed of a motif and a weight. A motif is a series of amino acids with a length between 1 and 10. A weight is a number, which may be positive if it benefits fitness or negative if it impairs fitness. Each model assesses peptide sequences by applying its rules: the predicted fitness for each peptide is the sum of the weights for the motifs it contains.

POEMs are generated in groups of 100 called populations. Each time POET is run it begins with a population where the rules in each POEM are random, while the later populations are generated from a process of tournament selection, crossover, and mutation.

Each POEM within the population is randomly assigned to a group of five called a tournament. The POEMs within a tournament are evaluated by their ability to accurately predict fitness of peptides in the training data. The two most accurate POEMs in each tournament produce four new POEMs for a new population, where each POEM's rules are randomly inherited from each of the two tournament winners. An additional POEM is produced by applying "mutations" to the best POEM in the tournament.

Mutations in POET are modifications to the rules of a POEM in a manner analogous to mutational changes to DNA in biological evolution. The types of mutation can be separated into two groups: POEM mutations and rule mutations. When a POEM is being mutated, first POET randomly determines for each POEM mutation whether it occurs, then each rule of the POEM, has a chance for each of the rule mutations to occur to it. The different types of mutation performed in POET can be seen in **Table 3**.

Table 3: Mutations in POET

Mutation Type	Type	Chance	Description
Add Rule	POEM	20%	A randomly generated new rule is added to the POEM
Remove Rule	POEM	20%	A randomly selected existing rule is removed from the POEM
Adjust Weight	Rule	20%	The rule's weight is increased or decreased by a percentage of its prior weight.
Add Letter	Rule	10%	The rule's motif has an additional letter added to it unless the rule already has the maximum number of letters.
Remove Letter	Rule	10%	The rule's motif has a letter removed from it, unless the rule has one of fewer letters.

This cycle of generating rules, comparing POEMs to produce a new population of models. If a rule is useful for predicting the training data, it will improve the accuracy of the POEMs containing it. As these rules mutate, beneficial mutations will increase the accuracy of the model, leading to their presence in later populations, while detrimental mutations decrease the POEM's accuracy and thus make them less likely to be present in later populations. This results in the accrual of useful rules that increase the POEM's as the learning process continues, despite all the rules being generated randomly, and being mutated randomly.

When running POET on MSU's High Power Computing Center (HPCC), POET's process of

learning was run multiple times in parallel, typically 100. Each of these runs produced their own series of POEMs from the initial steps and evolved in different ways.

After POEMs are developed, POET can be used to predict the fitness of a peptide from its sequence. First POET finds the most accurate model at predicting the training data and then that model is used to predict the fitness of the peptide. Additionally before the fitness values of a peptide are calculated, POET applies a simple amino acid model of hydrophobicity[51] to predict if the amino acids are soluble.

The learning process of POET can be visualized in **figure 4**.

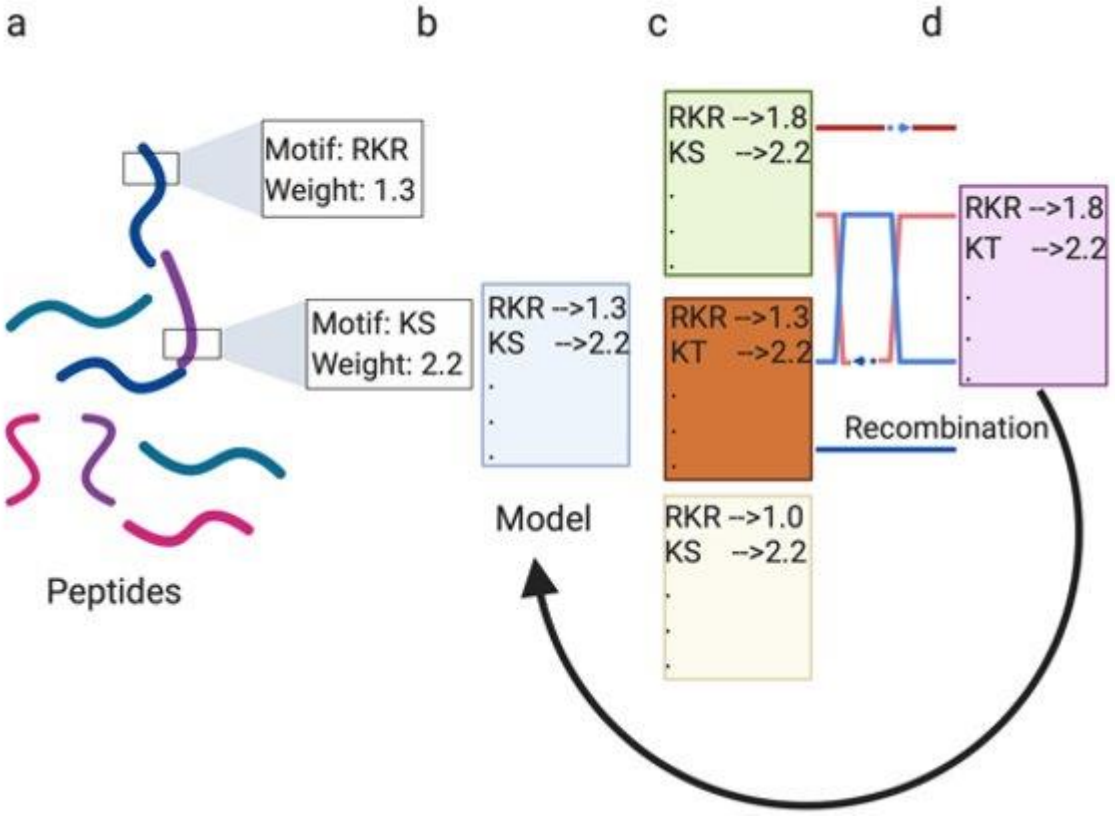


Figure 4: *Visual depiction of POET. Motifs, such as arginine-lysine-arginine (RKR) or lysine-serine (KS), are identified from the peptides in the training data provided to POET (**a**). Weights are assigned to these models to create POEMs (**b**). A population of POEMs undergo mutation and*

Figure 4 (cont'd)

*recombination (**c**) to produce the next generation of POEMs (**d**), which themselves are used to generate later generations of models using further cycles of recombination and mutation.*

When using POET to produce a list of peptides to examine, first it generates a random list of peptides, predicts the fitness for each peptide on the list, sorts the list and returns the top results. The number of randomly generated peptides, length of the peptides generated, and the number of top peptides that are returned are variables that can be altered in POET's config file, for ease of use in different applications. My experiments used 10,000, 12, and 10 respectively for these inputs.

For more information on POET please consult an article published by my collaborator on it[52].

POET Design Cycle

The cycle of using POET would begin with predictions of new peptides, which would then be synthesized commercially by Genscript (Piscataway, NJ). These peptides then had their contrast measured as described above. The new data from these peptides was then added to the overall training data. Then POET is started again from the beginning, to develop a new series of POEMs using this newer, larger, body of training data. This design cycle can be visualized in **figure 5** below.

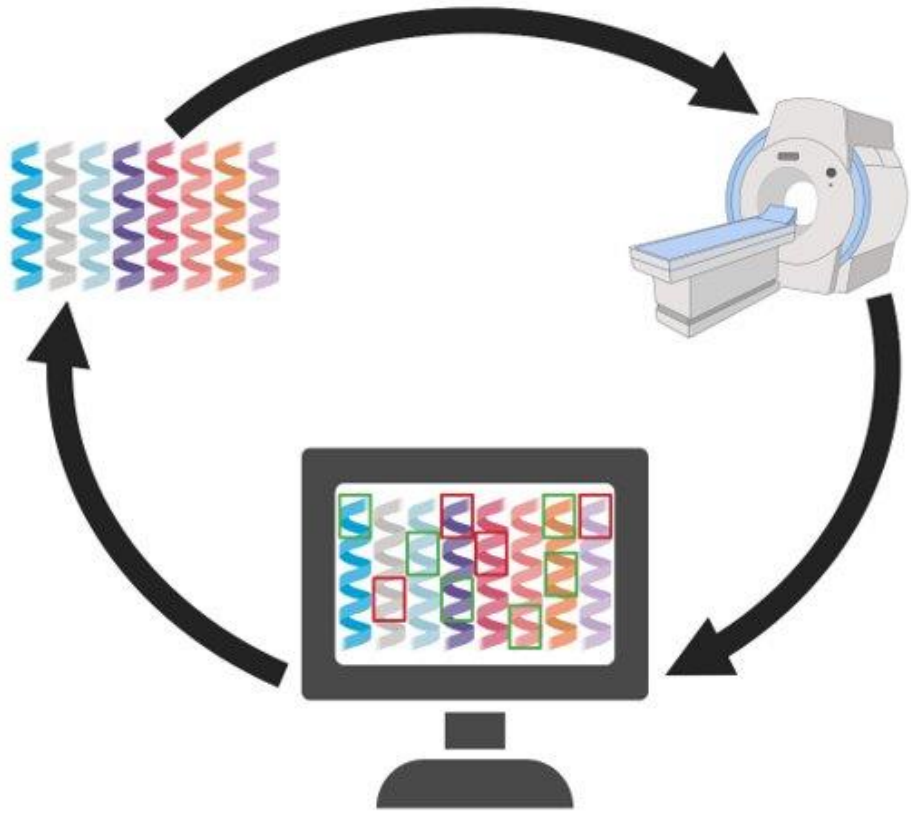


Figure 5: *Design Cycle using POET. POET identifies motifs from a preexisting dataset (bottom). This leads to identifying proteins for testing (upper left). The proteins are tested, here using an MRI (upper right) and the data is added to POET to begin the cycle anew (bottom).*

PROJECT 1: IMPROVING CEST PEPTIDE SENSITIVITY THROUGH DIRECTED EVOLUTION

Introduction

Prior work established that sPRM, an alternate histone produced in sperm cells to tightly bind DNA, was able to function as a good CEST contrast agent, since the many positively charged arginine residues used to bind positively charged DNA provided many highly accessible amide protons. Use of sPRM as a transgenic reporter gene, however, presents the long-term issue of immune system rejection common to using a non-human gene within a human body. In human sperm there is a homolog to sPRM, human protamine 1 (hPRM1)[53]. Experiments in testing the CEST contrast produced by hPRM1 demonstrated substantial contrast, but less than that of sPRM[25,54] and its localization in sperm cells should prevent it from interfering with the background signal in humans. We hypothesized that if we were to apply optimization to hPRM1 we would be able to increase its contrast without significantly increasing its immunogenicity.

Results

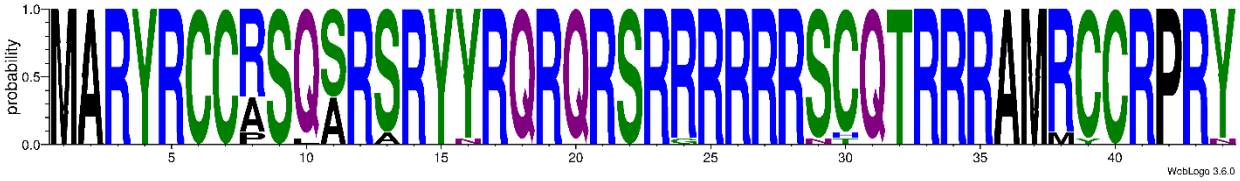


Figure 6: *Logo of Human Protamine Variants. The height of each letter demonstrates the relative occurrence of a given amino acid in the mutant protein using single letter amino acid codes. The wild type for hPRM1 is MARYRCCRSQSRSRYYRQRQSRRRRRRSCQTRRRAMRCCRPRY This figure was made using WebLogo 3.6.0.*

The experiment was repeated three times producing three plates and 65 colonies where the sequence was analyzed. Across all of the studied peptides sequences there were 23 different

mutants. Of these there were 13 different mutations to the overall protein sequence. A logo of the mutants generated can be seen in **figure 6.**

Discussion

Of the 13 mutations to the protein sequence, only 4 of them occurred to one of the sequences arginine residues, despite the arginine residues making up 20 of the 44 amino acids in the overall sequence. This can be understood as a result of arginine codon bias. Arginine has more codons than any other amino acid. Any change to the third nucleic acid results in a silent mutation, while an alteration to the second amino acid has two chances to result in a silent mutation and one chance to mutate into a stop codon.

Conclusion

Due to the slow progress of these directed evolution experiments, the project was ended before the contrast of the mutants could be evaluated or the immunogenicity of the mutants could be measured. Whether an improved version of hPRM1 could be produced via directed evolution remains unclear. Furthermore, whether any of hPRM1 mutants have immunogenicity that would prevent them from clinical application is unclear. Although this project did not produce the reporter gene that it set out to produce, it did generate insights into the development of CEST reporter genes using directed evolution and provide insights on what can be done to overcome these limitations.

Mutagenesis seems to introduce changes very slowly into hPRM1. The density of arginine residues and the likelihood of an arginine codon mutating into another arginine codon present a high likelihood that any mutations to the DNA sequence would result in a silent mutation. Further the process of evaluating the contrast from these variants produces many challenges

such as purification, and standardization of concentration, which could obscure any effects of a change in sequence if the change is small, such as the mutation of only a single amino acid into a similar amino acid in a chain of 44.

This would suggest that if further developments are to be made in the engineering of a CEST reporter gene a means of development they must be done using a method other than directed evolution. Optimally it would scan allow a greater variety of different protein biochemistries to be assayed.

PROJECT 2: IMPROVING CEST CONTRAST AT 3.6 PPM VIA MACHINE LEARNING

Many portions of this section are adapted from a currently published paper[1].

Introduction

The results of directed evolution experiments encountered a variety of problems. The mutation count was low, the silent mutation rate was high and normalizing for protein in each sample presented technical difficulties. This led us to pursue a different means of developing the CEST reporter genes using machine learning. This provided issues in that in prior literature there was only one paper which provided a database of peptides, but it only had 31 peptides each being 12 amino acids long. Current machine learning methods of protein engineering require far larger dataset, typically thousands of proteins, and rely on longer sequences, typically over 100 amino acids, to provide context.

This led to the collaboration with the Banzhaf lab and along with that the development of POET. The first experiments using POET would focus on the development of a CEST producing peptide (CESTide) that produced contrast at 3.6 ppm, which is the amine resonance frequency. This was taken as a focus due to the relatively large amount of prior data available in literature and the lab's prior focus on reporter genes at this frequency, such as LRP, while there aren't published results that would allow a training library at other frequencies, such as 1.8 ppm or .8 ppm. Despite 1.8 and .8 ppm peaks being larger than the 3.6 ppm peaks in many cases, they are far closer to the water saturation frequency, which produces issues with contrast to noise ratio (CNR) especially on scanners of clinical strength. This would demonstrate POET's ability to understand a relatively well understood question.

Generational Development

The first generation resulted in only one soluble peptide which generated less contrast than K12. The issue of solubility led to integrating a solubility estimation into POET's predictor (see methods). The Z-spectra and MTR values for generation 1 can be seen in **figure 7**.

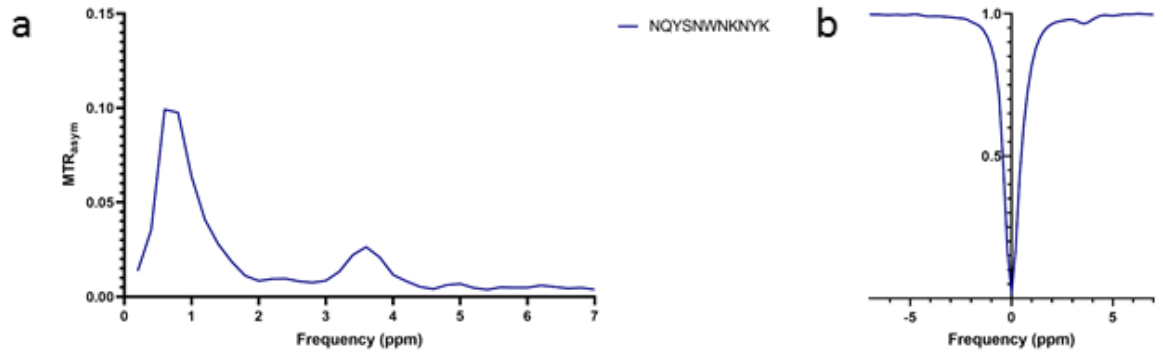


Figure 7: *3.6 ppm Generation 1. Shows the MTR* ***(a)*** *and Z-spectra* ***(b)*** *acquired from the only soluble peptide in the 1st generation of 3.6 ppm POET design. The results are from one experiment.*

The second generation saw an increase in contrast and universal solubility. The best peptide in the generation produced more contrast than K12. Further this peptide, NSSNHSNNMPCQ, had a neutral charge (pI 7.32), which was of interest since prior attempts to engineer a CEST reporter gene focused on incorporating as many positively charged residues as possible, leading them to have a strong positive charge[55]. The Z-spectra and MTR values for generation 2 can be seen in **figure 8**.

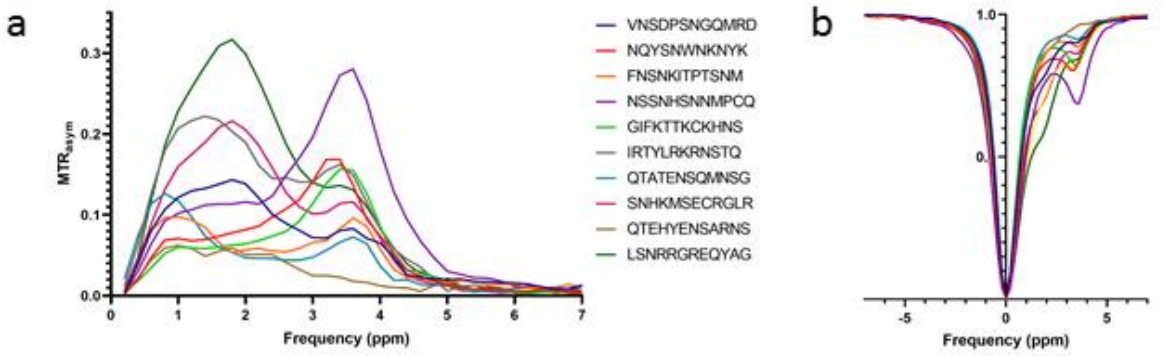


Figure 8: *3.6 ppm Generation 2. Shows the MTR* ***(a)*** *and Z-spectra* ***(b)*** *acquired from the peptides in the 2nd generation of 3.6 ppm POET design. Data is taken from a single experiment. Error bars show standard deviation.*

Generation 3 was developed unusually, with half of the peptides produced using a model that was developed to predict generation 2, but was overlooked in that generation due to not being the most accurate at the time. The model was in the final population for predicting generation 2, but didn't have the lowest error when determining which POEM would be used with the predictor. A post-hoc analysis determined that when incorporating the data gathered from the generation 2 peptides the model's error was lower than a model trained from scratch with both generations of data. The other five were predicted from a model developed as normal. The peptides from generation 2 model produced significantly (p=0.01) more contrast than those with an entirely new model. Similar post-hoc analysis wasn't done on later generations in an attempt to maintain a consistent experimental framework across generations. Later generations wouldn't use this same split model approach to maintain a consistent experimental framework across generations. The Z-spectra and MTR values for generation 3 can be seen in

figure 9.

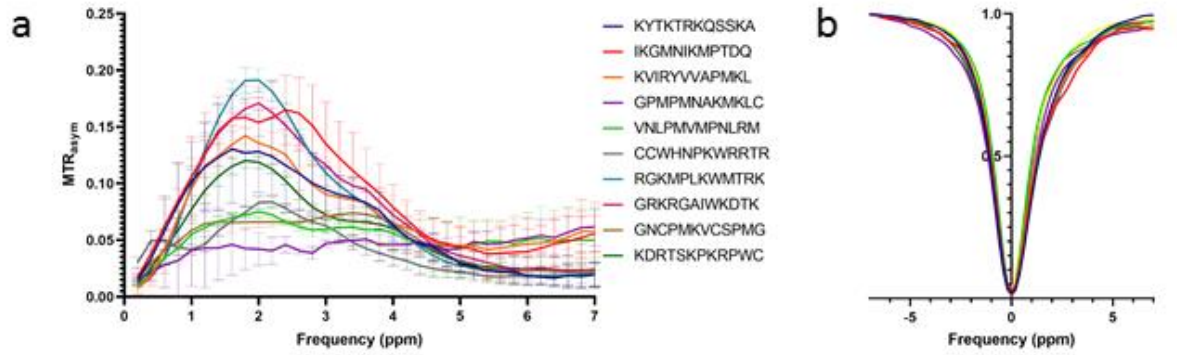


Figure 9: *3.6 ppm Generation 3. Shows the MTR* ***(a)*** *and Z-spectra* ***(b)*** *acquired from the peptides in the 3rd generation of 3.6 ppm POET design. Points are the average from two experiments for the first five peptides or five experiment for the last five peptides. Error bars show standard deviation.*

The 4th generation was the first generation to see a decrease in maximal contrast to 22.30, but the average of the generation did see increase to 14.49. The Z-spectra and MTR values for generation 4 can be seen in **figure 10**.

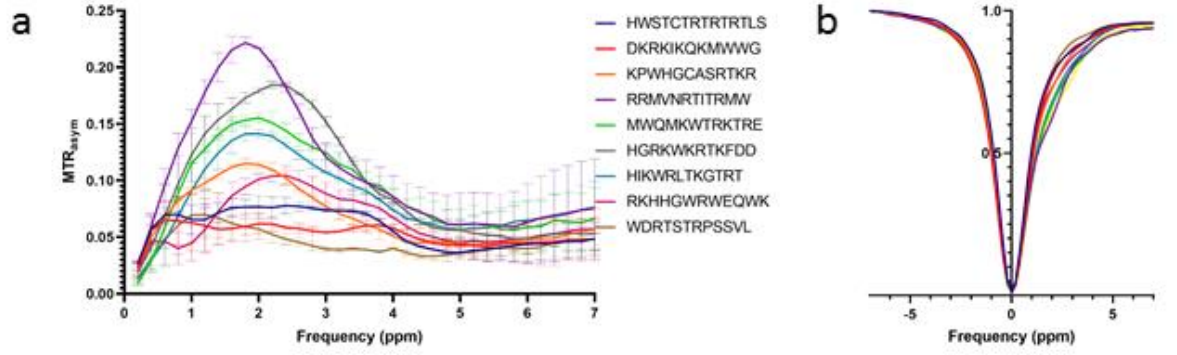


Figure 10: *3.6 ppm Generation 4. Shows the MTR* ***(a)*** *and Z-spectra* ***(b)*** *acquired from the peptides in the 4th generation of 3.6 ppm POET design. All values are from the average of two experiments. Error bars show standard deviation.*

In the 5th generation, two new peptides were discovered that produced more contrast than any prior generation: WFGLQRHLKKKD (40.58) and LELKLGKRPMGW (43.64) and the average contrast increased to 23.14. The Z-spectra and MTR values for generation 5 can be seen in **figure 11**.

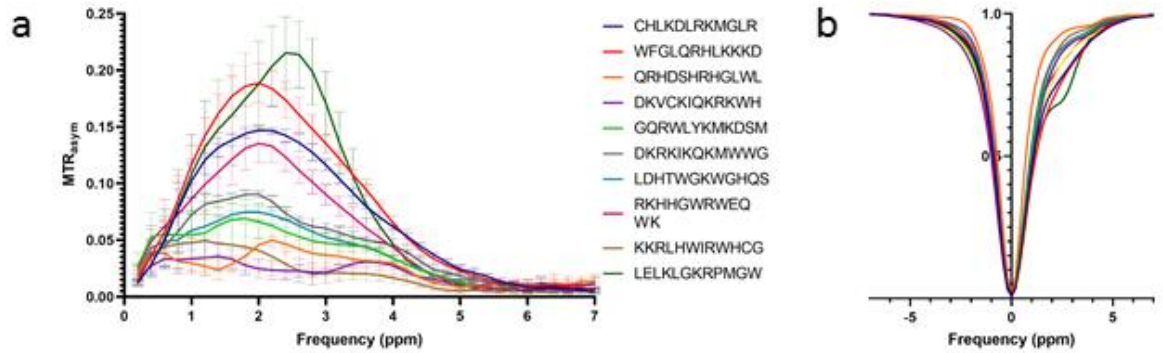


Figure 11: *3.6 ppm Generation 5. Shows the MTR* ***(a)*** *and Z-spectra* ***(b)*** *acquired from the peptides in the 5th generation of 3.6 ppm POET design. Points are taken from the average of four experiments. Error bars show standard deviation.*

During the production of the 6th generation of peptide optimization the peptides were 10 amino acids long instead of the 12 that is normal, due to an experimenter error in using POET's predictor function. The maximum contrast produce was 31.40, which is less than the top CESTides of generation 5. The average contrast also decreased to 14.30. The Z-spectra and MTR

values for generation 6 can be seen in **figure 12**.

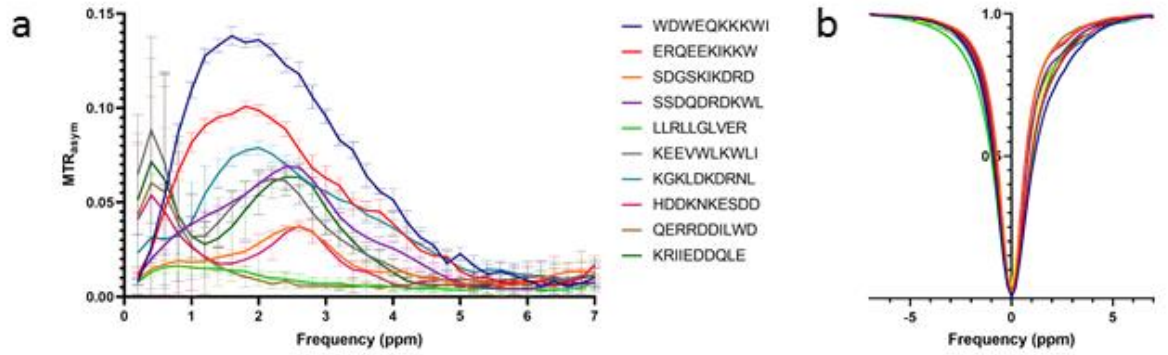


Figure 12: *3.6 ppm Generation 6. Shows the MTR* ***(a)*** *and Z-spectra* ***(b)*** *acquired from the peptides in the 6th generation of 3.6 ppm POET design. Points are taken from the average of five experiments for the first five peptides or four experiments for the last five peptides. Error bars show standard deviation.*

The 7th generation of peptide optimization saw the two highest contrast peptides generated by POET: LWSDIKMKLKKT (PTE of 49.4) and KMGKLIGIPVLK (PTE of 47.8). This generation also achieved an average PTE of 21.67, the second highest after generation 5. The Z-spectra and MTR values for generation 7 can be seen in **figure 13**.

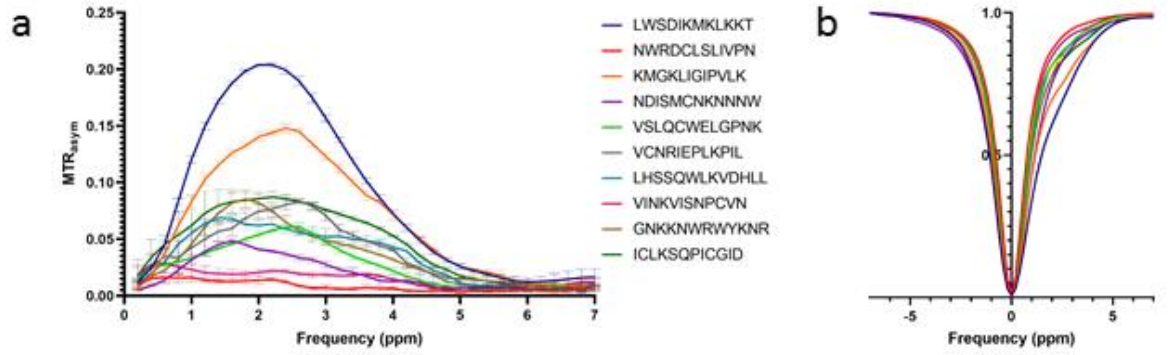


Figure 13: *3.6 ppm Generation 7. Shows the MTR* ***(a)*** *and Z-spectra* ***(b)*** *acquired from the peptides in the 7th generation of 3.6 ppm POET design. Points are taken from the average of five*

Figure 13 (cont'd)

experiments. Error bars show standard deviation.

The 8th generation of CESTides saw a decrease in both peak and average to 30.59 and 18.54 respectively. The Z-spectra and MTR values for generation 8 can be seen in **figure 14.**

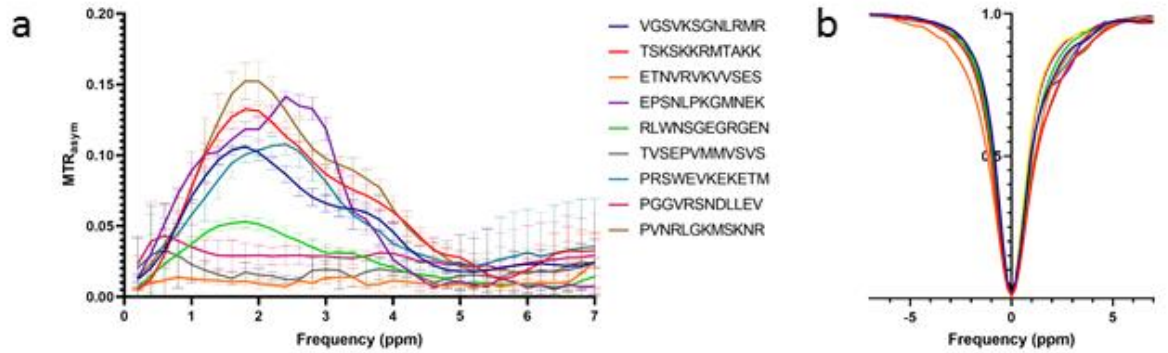


Figure 14: *3.6 ppm Generation 8. Shows the MTR* ***(a)*** *and Z-spectra* ***(b)*** *acquired from the peptides in the 8th generation of 3.6 ppm POET design. Points are taken from the average of four experiments for the first five peptides or five experiments for the last five peptides. Error bars show standard deviation.*

In the 9th generation of CESTides the maximum contrast increased relative to the 8th generation to 30.64, but the contrast average contrast decreased to 12.25. The Z-spectra and MTR values for generation 9 can be seen in **figure 15**.

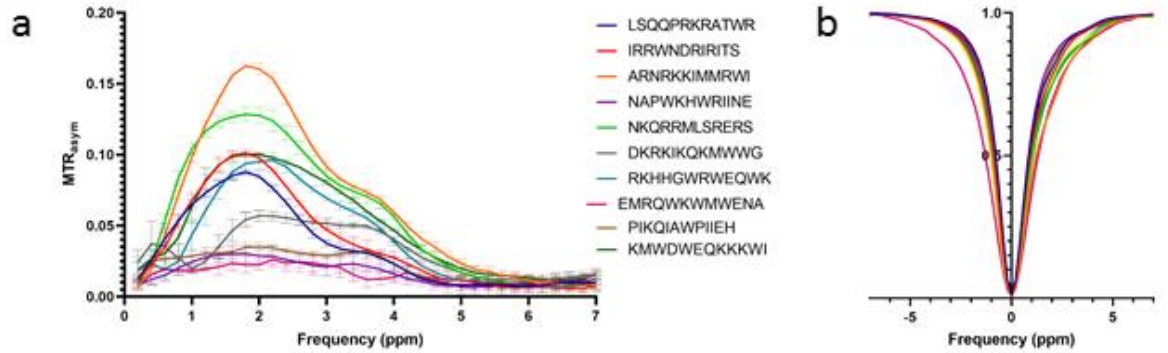


Figure 15: *3.6 ppm Generation 9. Shows the MTR* ***(a)*** *and Z-spectra* ***(b)*** *acquired from the*

Figure 15 (cont'd)

peptides in the 9th generation of 3.6 ppm POET design. Points are taken from the average of five experiments. Error bars show standard deviation.

In the 10th and final generation of 3.6 ppm peptide evolution there was an increase in the peak contrast and average contrast to generation 9, with values of 29.03 and 17.64 respectively. The 10th generation can be seen in **figure 16** and the contrast generated in each generation can be seen in **figure 17**.

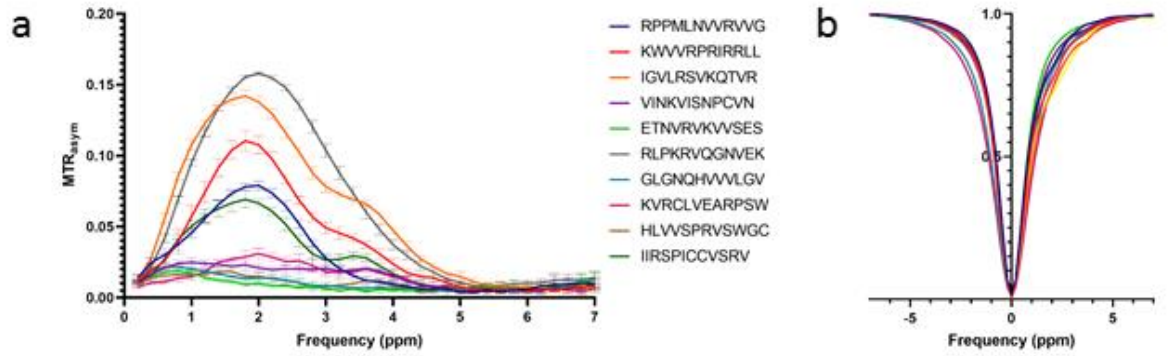


Figure 16: *3.6 ppm Generation 10. Shows the MTR* ***(a)*** *and Z-spectra* ***(b)*** *acquired from the peptides in the 10th generation of 3.6 ppm POET design. Points are taken from the average of five experiments for the first five peptides or three experiments for the last five peptides. Error bars show standard deviation.*

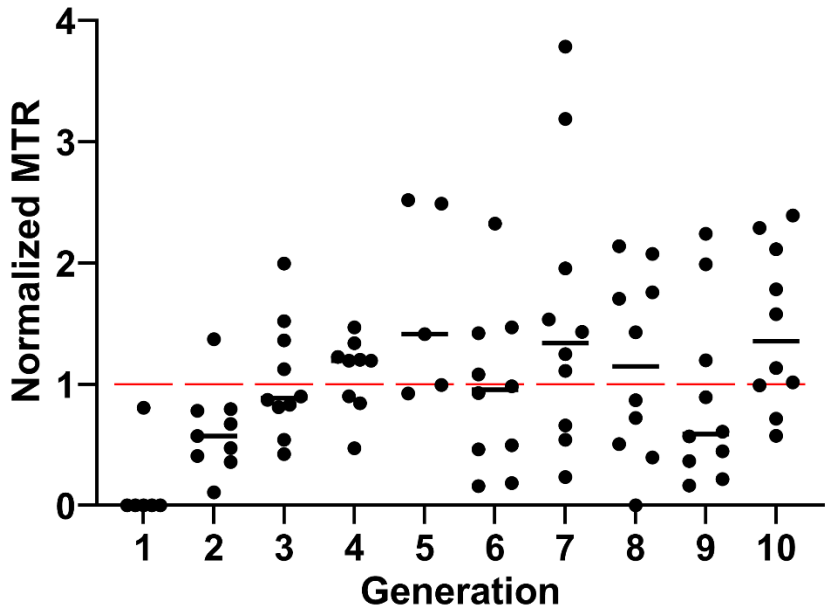


Figure 17: *Contrast by generation for 3.6 ppm peptides. Each point represents a single peptide from the generation. The red dashed line is the contrast generated by K12, used as a comparison to K12. Black lines represent the median of the generation. Contrast values are normalized to K12.*

Sensitivity Experiments

To examine whether the increases in exchange rate and concentration were able to lead to higher sensitivity, I examined differing concentrations of K12 and KYTKTRKQSSKA (the highest contrast peptide from generation 3, chosen for its high exchange rate). T-test maps were generated from this experiment as described in methods. The comparison between the two samples can be seen in **figure 18**. KYTK produced more contrast and had more voxels determined to significant by a t-test at every concentration.

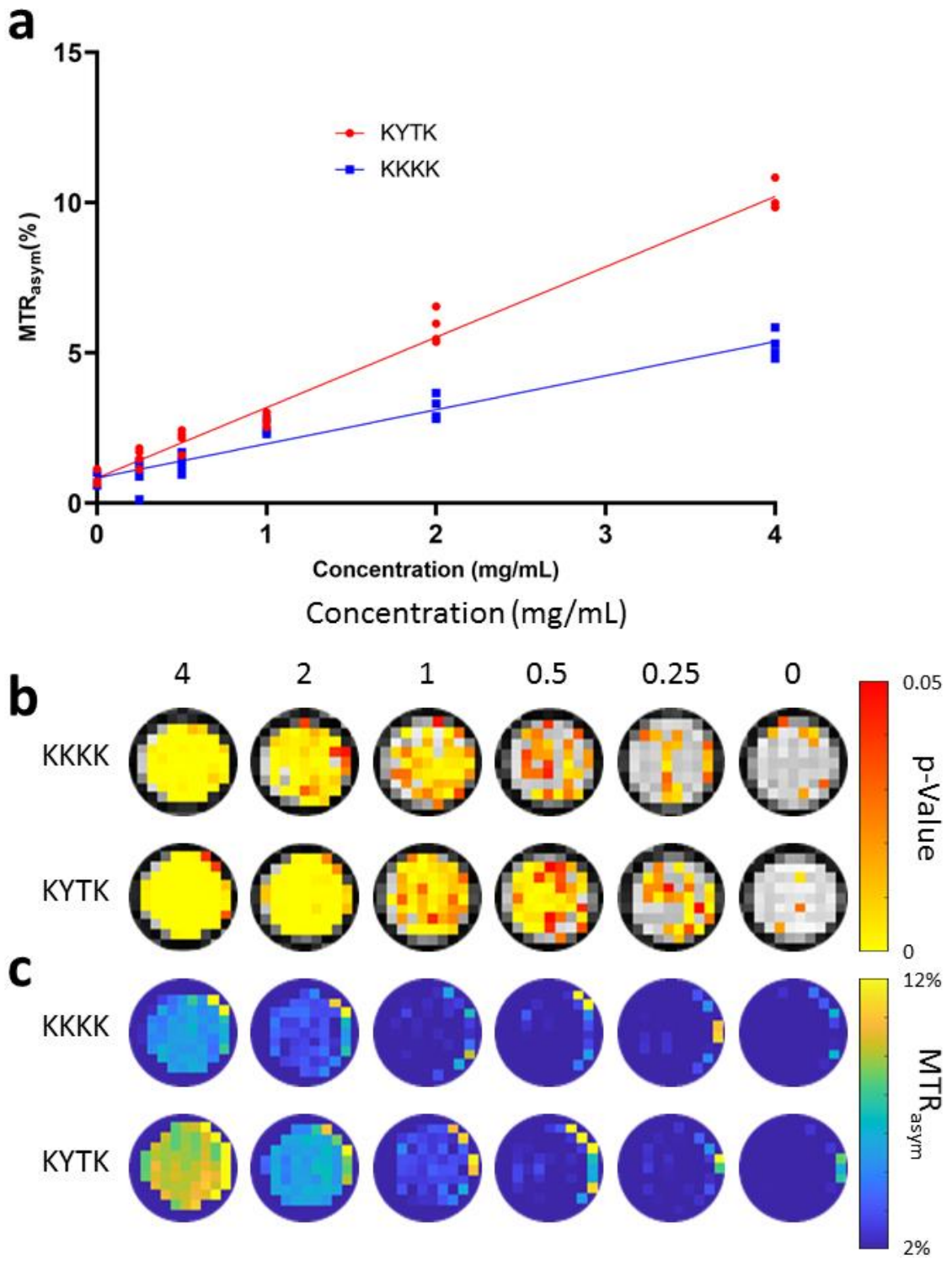


Figure 18: *Sensitivity increase from exchange rate increase. Wells of KYTK and K12 graphed by concentration and contrast* ***(a)****, compared by the results of a student's t-test* ***(b)****, and the MTR values for each of the wells* ***(c)****.*

Interesting Discoveries

The CESTides found using POET demonstrated several unique qualities as a result of not being developed in a manner of iterative changes to a single base peptide. This is most clearly evident in the unusual combinations of amino acids seen within the peptides. Earlier attempts to create CEST reporter genes were highly homogenous, relying on large numbers of lysine and arginine residues. Although lysine is the most commonly used amino acid in the CESTides studied, no amino acid holds a majority of the residues like is seen in hPRM[25] or LRP[22].The variety of amino acids used by the CESTides generated with POET can be seen in **figure 19**.

These trends also form trends in the learning throughout the generations of POET design and optimization as seen in **figure 20**. The most notable trend following lysine and arginine which form major parts of currently studied CEST reporter genes such as LRP and hPRM1. Initially lysine and arginine make up a small portion of the generations. Identifying the importance of these residues in generating contrast, POET increases the amount of lysine and arginine in its predictions up through generation five. The concentration decreases in generations six and seven, where the highest peak contrast is achieved. Generations eight and nine see the lysine and arginine content stabilize, before increasing in generation 10. This perhaps follows POET learning how to generate contrast via lysine and arginine content before reaching a fitness plateau and attempting to find additional means of increasing contrast, such as the surges in hydrophobic residues such as valine in generation nine and leucine in generation seven.

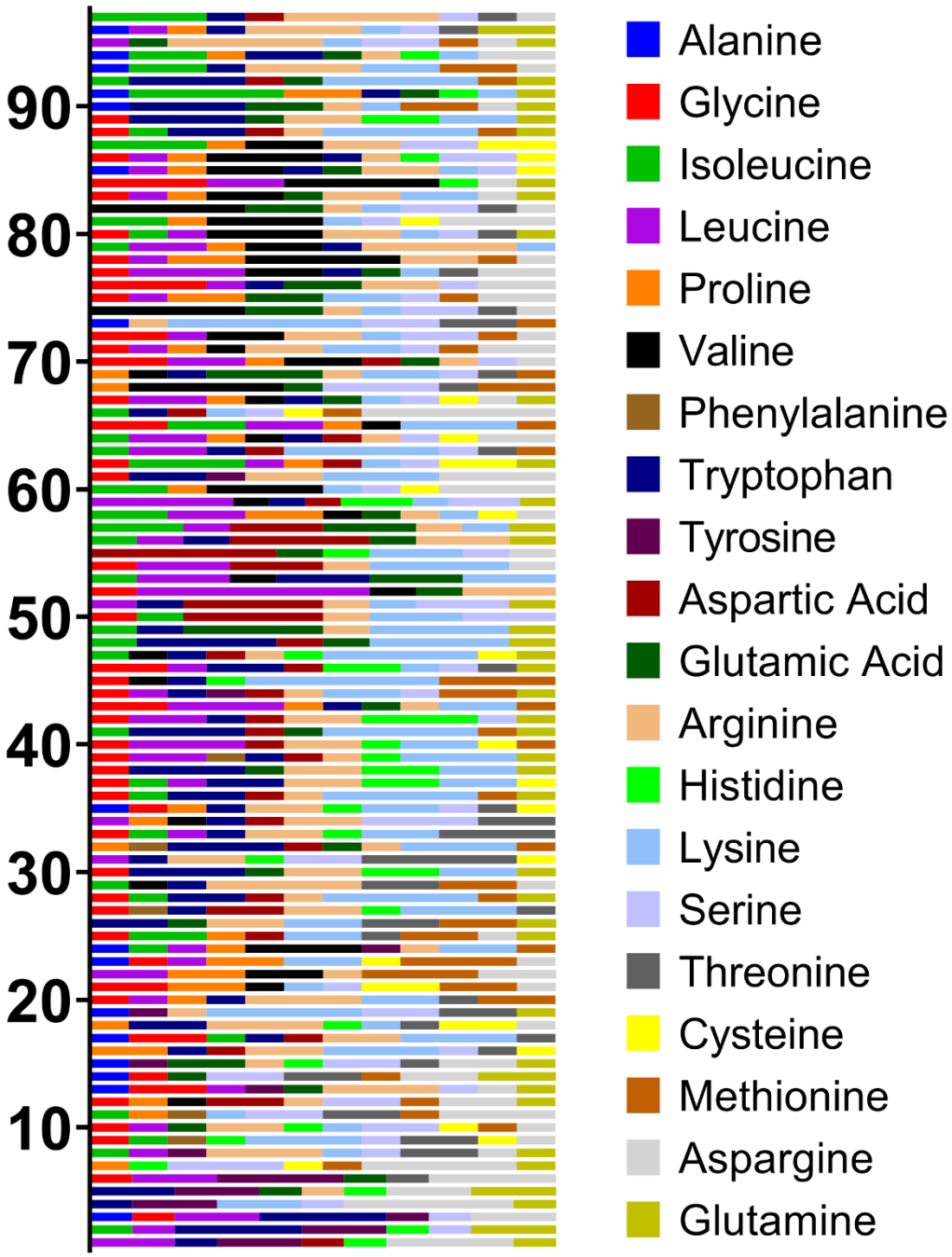


Figure 19: *Diversity of Amino acids within CESTides generated with POET. Every peptide generated is shown in the order of its testing.*

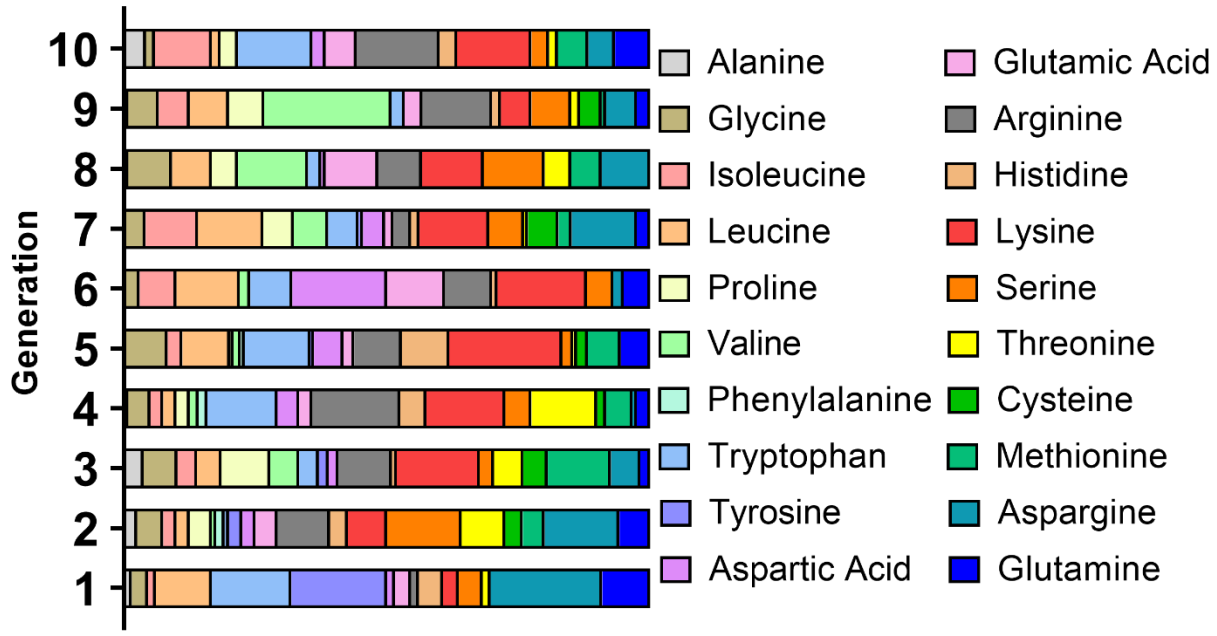


Figure 20: *Generational Changes in Amino Acid Usage.*

Another area of interest is the way these changing chemistries have impacted chemical properties, most notably the protein's charge. In prior literature peptides were engineered to generate more contrast by introducing additional positively charged residues, such as lysine[55]. The relationship between the charge of the CESTide and the contrast it generated can be clearly seen in **figure 21**. Although the highest contrast producing peptides are still positively charged, peptides generated by POET demonstrated higher contrast than any peptide in the training data with neutral and negatively charged peptides.

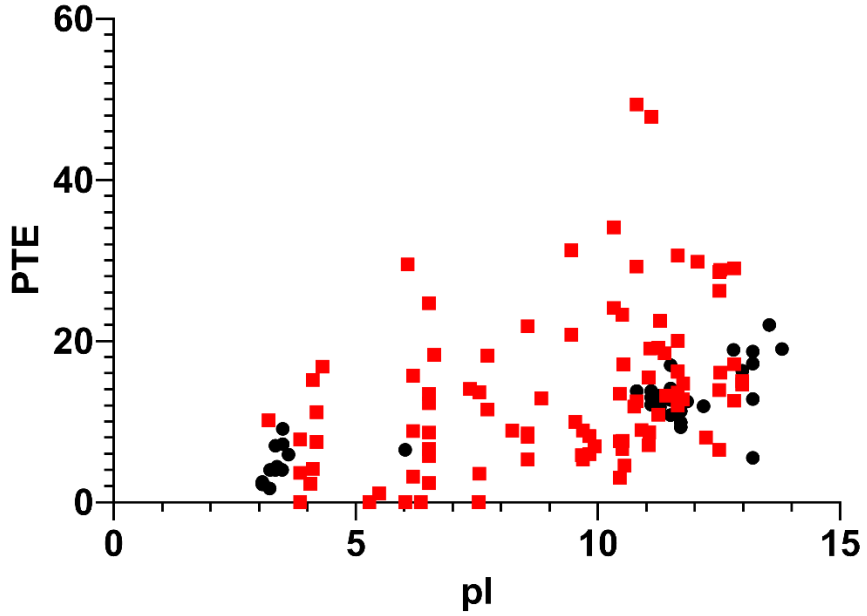


Figure 21: *Relationship between charge and contrast. Black circles are the peptides from literature used as training data. Red squares are the peptides generated using POET.*

Analysis of POET Predictions

We sought to examine the differences between the peptides generated by POET to determine if POET was converging toward a solution. This was calculated via the nearest neighbor distance from peptides in the same generation using Grantham distance, which takes into consideration differences between the size, charge, and hydrophobicity of different amino acids[56]. The basic assumption is that amino acids that are similar in chemical composition, polarity and molecular volume are more likely to change throughout evolution as they are less disruptive to protein function. To determine whether POET was learning and converging on a solution, we compared the Grantham distance between the peptides discovered with POET with peptides that were generated randomly. We first examined the intergenerational nearest neighbor distance by comparing finding the shortest Grantham distance within each peptide's generation and all

prior generations. As the Grantham distance decreased with an increase in the number of generations, this implies that learning took place since it shows that the predictions of POET are more similar than would be generated by randomness and are decreasing in distance faster. Next, we examined the intragenerational nearest neighbor distance by comparing each peptide to all peptides in the same generation to determine the most similar peptide. We find that the distance stays lower than the random simulation, implying that there is a form of selection occurring since the distance is lower than that of random peptides. . As POET optimizes, it would be expected to see the diversity of new generations decrease as POET begins to single out the area of the global maximum in the fitness landscape. The distance in the experiments are however, not decreasing by generation which suggests that POET is not converging on a solutionn . The nearest neighbor analysis over the CESTides generated by POET can be seen in **figure 22**.

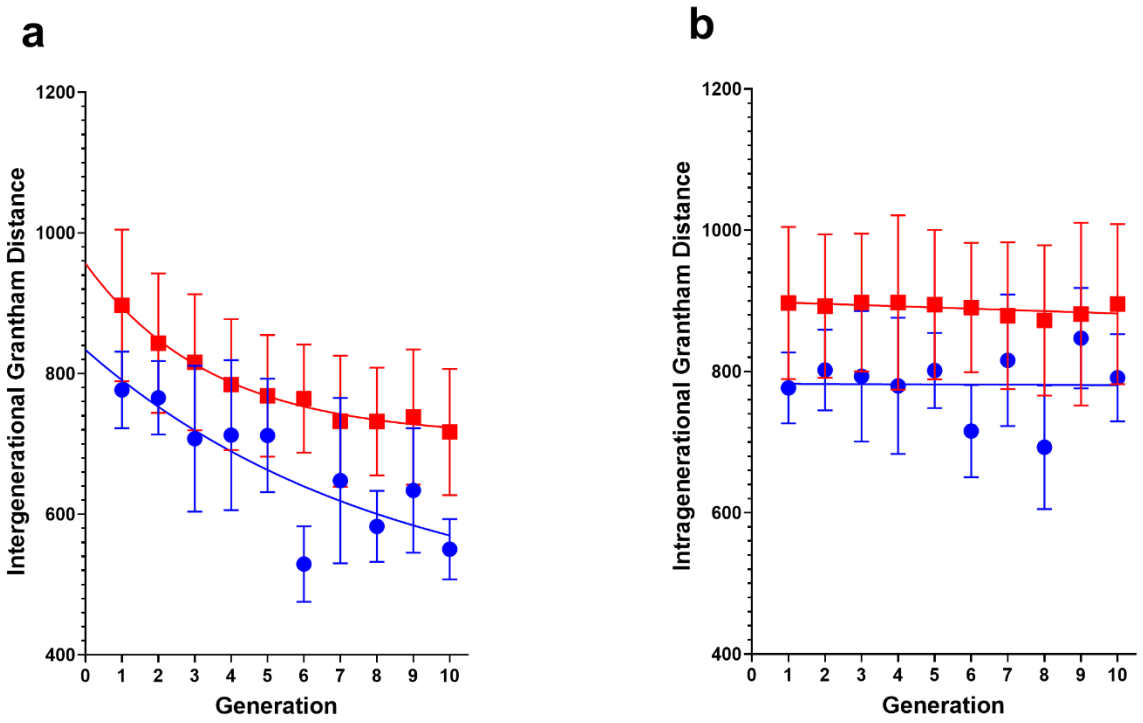


Figure 22: *Learning with POET. The intergenerational* ***(a)*** *and intragenerational* ***(b)*** *nearest neighbor distance from all the CESTides produced by POET. The data from the actual POET data set is shown in blue circles, while the data from a random simulation is shown as red squares. Brackets show the 95% CI for the data.*

Conclusions

POET demonstrated the ability to discover several new peptides with higher contrast than those in prior literature, with the peak seeing roughly a fourfold increase over K12. This peptide can be seen compared to K12 in **figure 23**. With work done through collaborators indicating that the increased contrast generated is a result of an increased exchange rate[1].

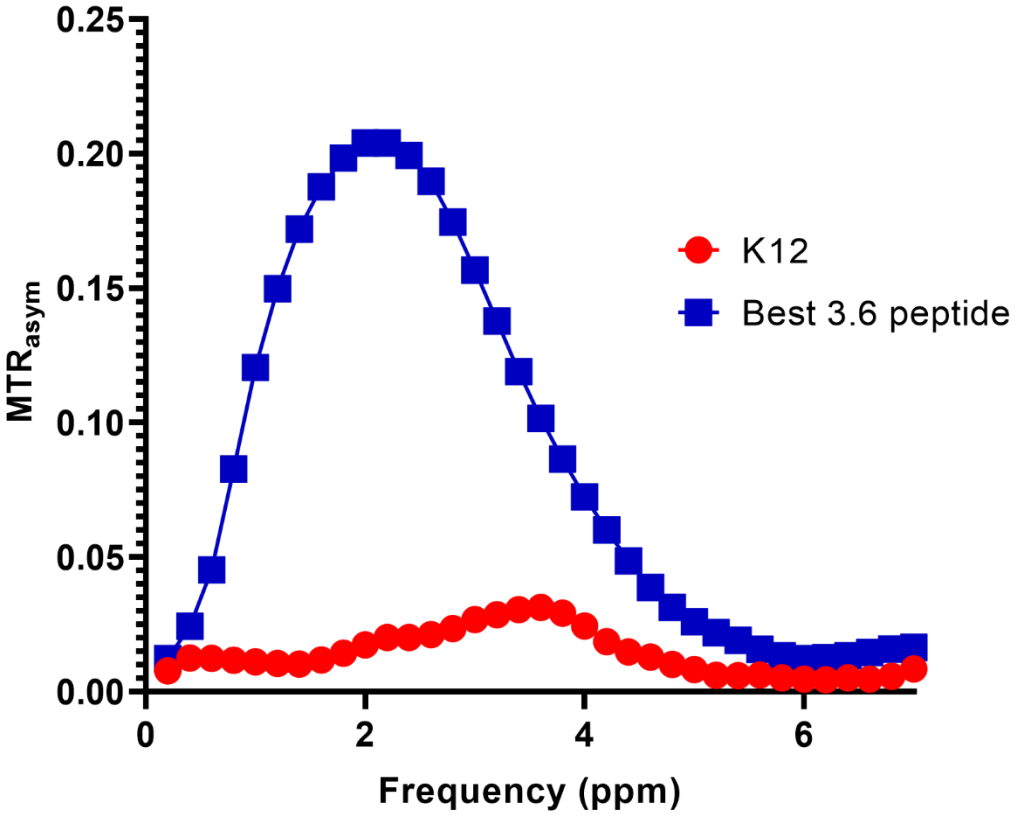


Figure 23: *Top 3.6 ppm peptide vs K12. The top peptide produced by the 3.6 ppm POET experiments is shown beside the K12 MTR. Data is the average of five experiments.*

This suggests that POET is a good tool for engineering CEST reporter genes and possibly peptides in general. Although not universal, the increased contrast appears to be generated by an increased rate of proton exchange. Most of the peptides that were discovered in the search have chemistries that mean that they would likely to never have been found by standard methods, such as directed evolution, within a feasible time scale.

There is an unknown loss of improvement after the 7th generation, which could have many possible sources. One possibility is that of a fitness plateau as POET isn't able to generate a better peptide with the same mechanism.

Another possibility is in a known bug of POET that wasn't discovered until after the final experiment was run. The bug results in the decrease in overall learning done by POET by ending POET's learning process prematurely. This problem scaled with the increase in time to evaluate a population because of the increasing volume of learning data.

PROJECT 3: IMPROVING CEST CONTRAST AT 5.0 PPM VIA MACHINE LEARNING

Introduction

Generating contrast at 3.6 ppm is very common for proteins and has been the approach taken by using hPRM1, LRP and scGFP, as well as prior experimentation using POET. This is because due to the ease of incorporating exchangeable amine groups into proteins using a lysine, histidine, or arginine residue. This ease however works in reverse and these same residues are used frequently in naturally occurring proteins, the sum of which make up a large portion of the contents of a cell. This results in a high background signal at 3.6 ppm (see **figure 24**). There are measures of CEST contrast at other frequencies than 3.6, notably 5.0 which is generated by thymidine and other biomolecules[6]. It is believed that it is related exchangeable protons bound to ring structures such as found in histidine or tryptophan, but no protein CEST agents have been developed that generate contrast at that frequency. If a protein CEST agent were developed that generated contrast at 5 ppm, it would be easier to distinguish from the background in a biological system. This would result in increased specificity and sensitivity.

Since POET doesn't understand the mechanism of contrast, but instead determines relationships between the amino acid sequence and the output, we sought to use POET to generate a new class of reporters at 5 ppm.

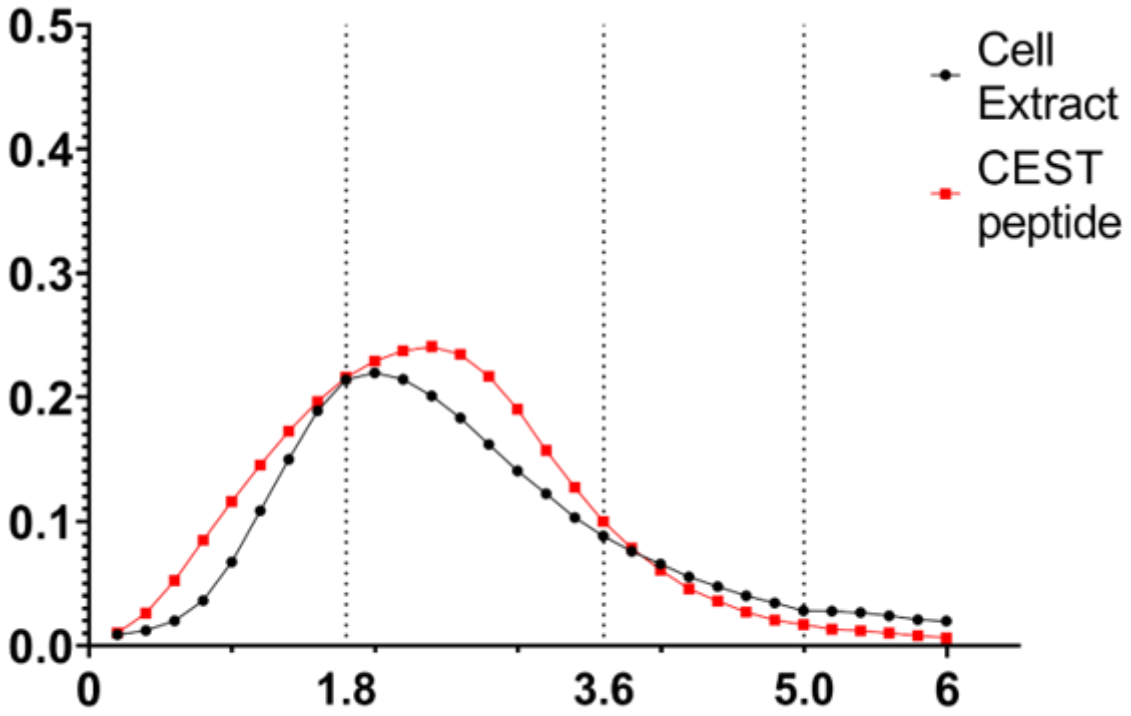


Figure 24: *CEST Background. In black circles is the MTR_{asym} of cell lysate from E. coli. In red squares is the best protein from generation 5 of the 3.6ppm POET experiments.*

Generational Development

A full z spectrum running from -7 to 7 ppm was acquired for prior work in the generation of 3.6 ppm reporters (see above). The contrast values at 5 ppm were used as the initial training data (generation 0), containing all the data from generations 1-5 (all peptides with a gen less than 5 and a series value of 3.6 in appendix 1). For purposes of statistical analysis, the training data was used as an understanding of the contrast produced by a random soluble peptide. The overall contrast generated by the peptides in each generation can be seen in **figure 25**.

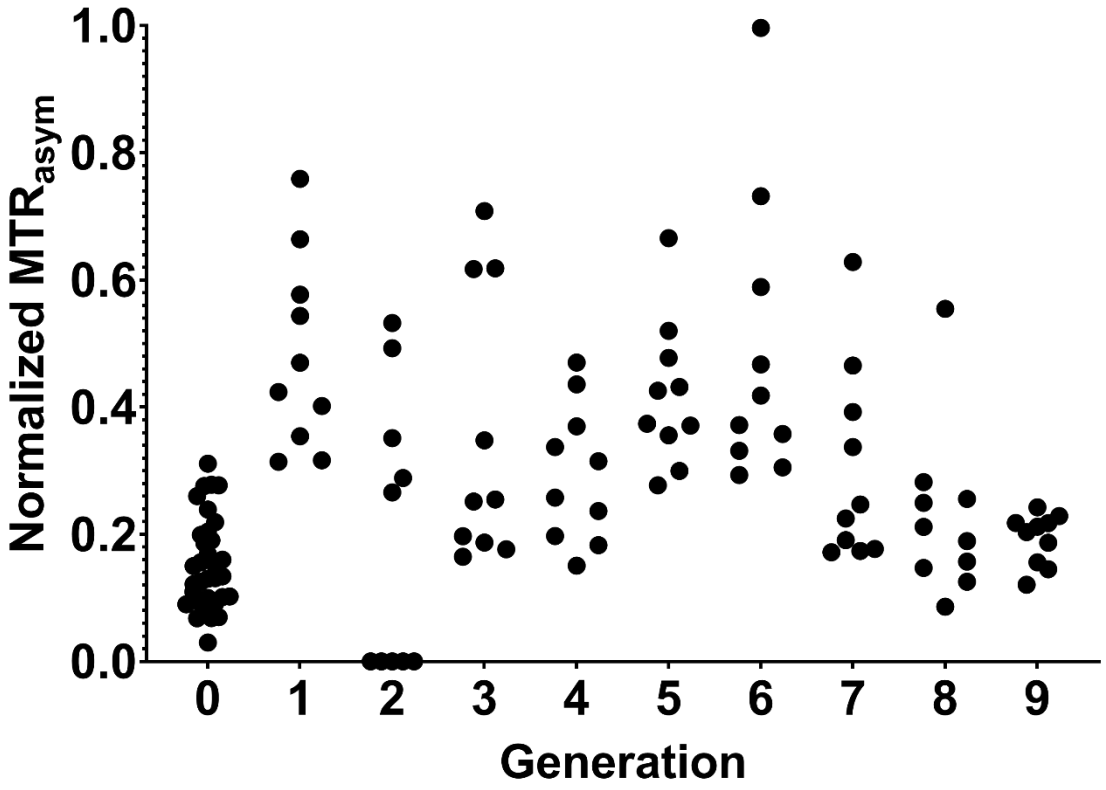


Figure 25: *5ppm contrast in each generation. Each peptide tested from the POET development of a 5ppm contrast agent is included. MTR values are normalized to the 3.6 peak of K12. Generation 0 represents the learning data.*

The first generation saw significant ($p= 1.2*10^{-15}$) improvement over the training data, with the best peptide having a PTE of 10.1 and the average being 6.9. None of the peptides show clear signs of generating contrast from an exchangeable proton at 5 ppm and instead most of the signal at 5 ppm seems to be part of a gradual decrease of the 3.6 ppm peak. The MTR_{asym} and Z-spectra for the 1st generation can be seen in **figure 26**.

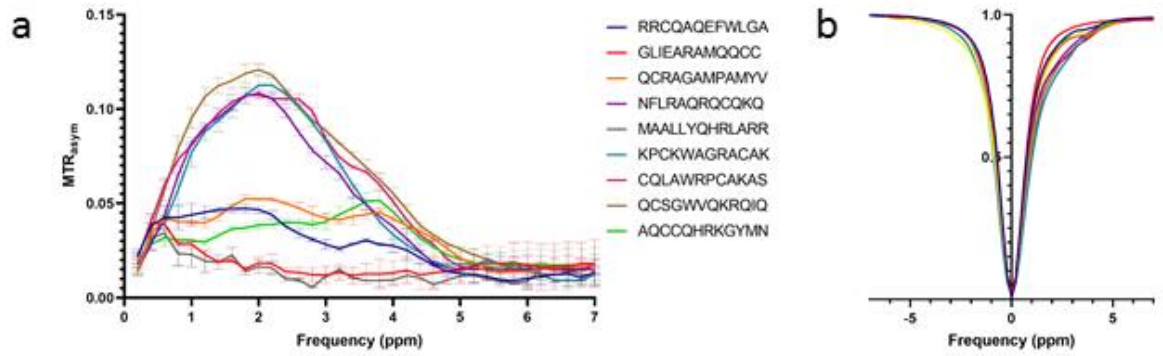


Figure 26: *5 ppm Generation 1. Shows the MTR* ***(a)*** *and Z-spectra* ***(b)*** *acquired from the peptides in the 1st generation of 5 ppm POET design. Points are the average from three experiments for the first six peptides or two experiment for the last 4 peptides. Error bars show standard deviation.*

The second generation had difficulties with solubility with five of the peptides for that generation being insoluble and going unmeasured. The generation has a lower average than generation 1 as well, with an average of 5.6 among the soluble peptides, but the peptides performed significantly better than the training data (p=0.005). The MTR_{asym} and Z-spectra for the 2nd generation can be seen in **figure 27**.

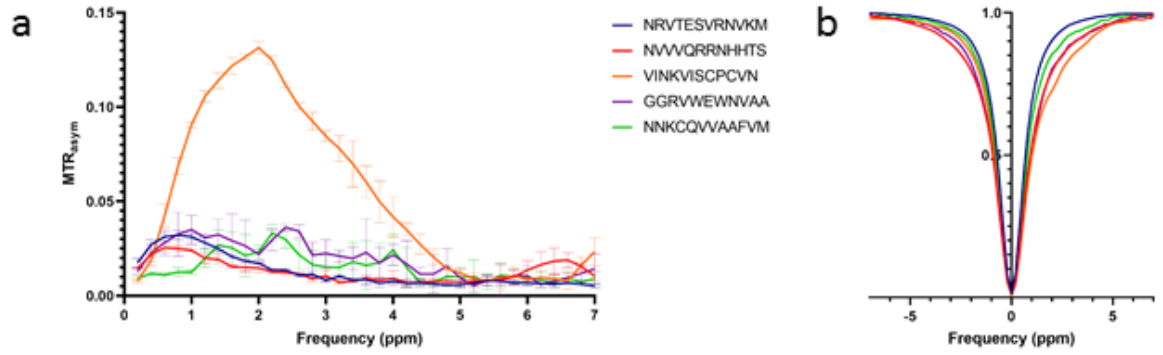


Figure 27: *5 ppm Generation 2. Shows the MTR* ***(a)*** *and Z-spectra* ***(b)*** *acquired from the peptides in the 2nd generation of 5 ppm POET design. Points are the average from three experiments.*

Figure 27 (cont'd)

Error bars show standard deviation.

In the third generation like the second generation saw a decrease in both maximum (10.77) and average contrast (5.32). The results were still significantly (p=0.008) better than the training data. The MTR_{asym} and Z-spectra for the 3rd generation can be seen in **figure 28**.

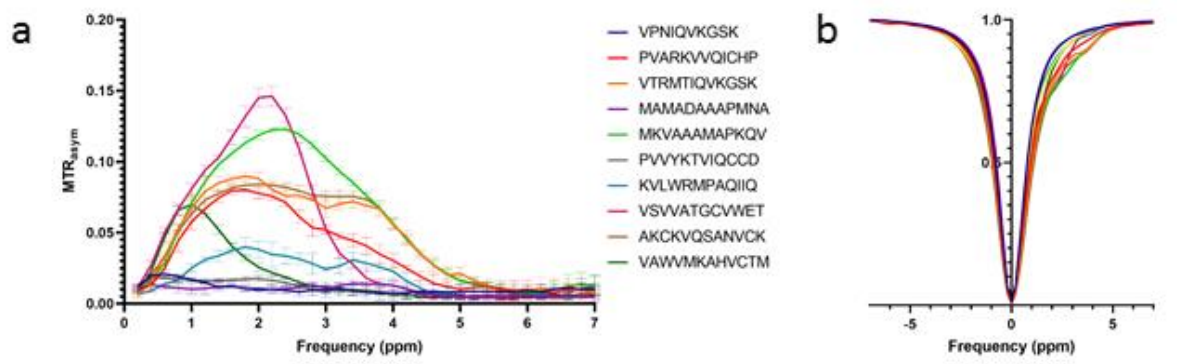


Figure 28: *5 ppm Generation 3. Shows the MTR* ***(a)*** *and Z-spectra* ***(b)*** *acquired from the peptides in the 3rd generation of 5 ppm POET design. Points are the average from four experiments for the first five peptides or five experiments for the last five. Error bars show standard deviation.*

In the fourth generation the maximum contrast decreased to a PTE of 7.1 and the average decreased to 4.38. The results were still significantly (p=0.008) better than the training data. The MTR_{asym} and Z-spectra for the 3rd generation can be seen in **figure 29**.

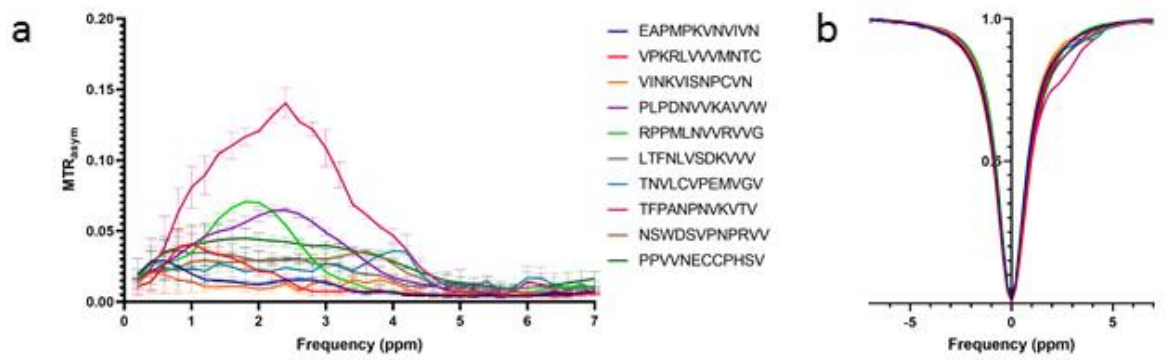


Figure 29: *5 ppm Generation 4. Shows the MTR* ***(a)*** *and Z-spectra* ***(b)*** *acquired from the peptides in the 4th generation of 5 ppm POET design. Points are the average from five experiments for the first five peptides or three for the last five. Error bars show standard deviation.*

In the fifth generation maximum and average contrast increased to 9.73 which is higher than the fourth generation but remained lower than the first generation. The MTR_{asym} and Z-spectra for the 5th generation can be seen in **figure 30**.

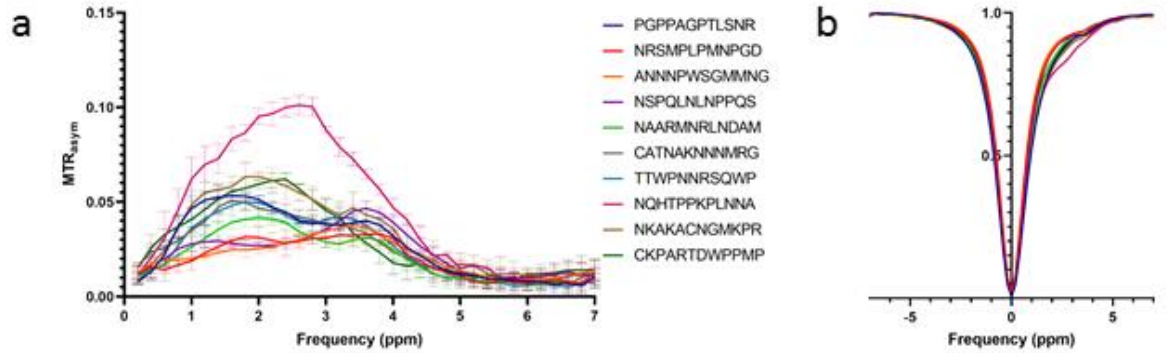


Figure 30*: 5 ppm Generation 5. Shows the MTR* ***(a)*** *and Z-spectra* ***(b)*** *acquired from the peptides in the 5th generation of 5 ppm POET design. Points are the average from five experiments. Error bars show standard deviation.*

The sixth generation saw the peak of the 5 ppm experiments. Its top CESTide, KVNFNKAVSNLK, produced the highest 5 ppm PTE of 14.23 and it appears to form a distinct peak, despite an

apparent contribution from the amide exchangeable at 3.6 ppm. The average contrast for this generation was also the highest at a PTE of 7.01. The MTR_{asym} and Z-spectra for the 6th generation can be seen in **figure 31**.

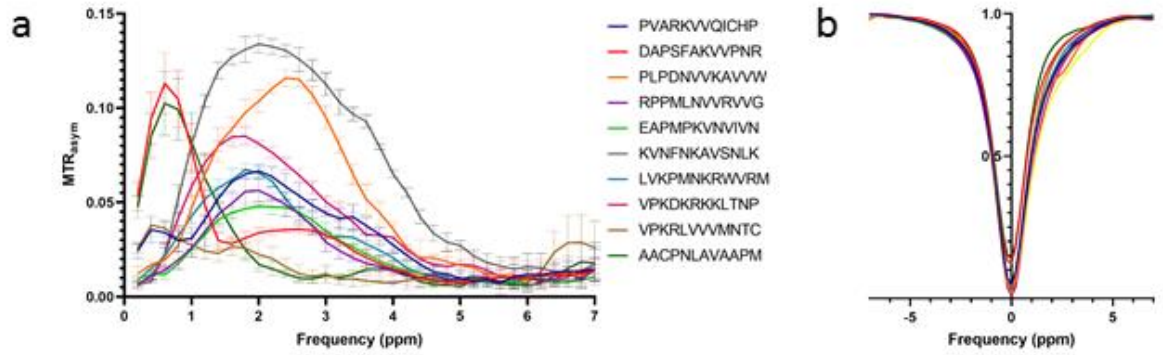


Figure 31: *5 ppm Generation 6. Shows the MTR* ***(a)*** *and Z-spectra* ***(b)*** *acquired from the peptides in the 6th generation of 5 ppm POET design. Points are the average from five experiments. Error bars show standard deviation.*

In the seventh generation the maximum contrast decreased to 8.01 and the average contrast decreased to 3.82. The MTR_{asym} and Z-spectra for the 7th generation can be seen in **figure 32**.

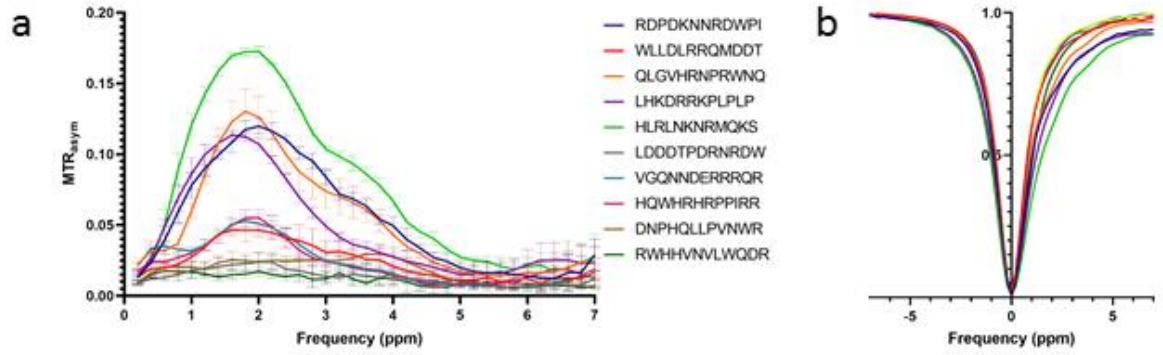


Figure 32: *5 ppm Generation 7. Shows the MTR* ***(a)*** *and Z-spectra* ***(b)*** *acquired from the peptides in the 7th generation of 5 ppm POET design. Points are the average from five experiments. Error bars show standard deviation.*

The eighth generation continued the pattern of decrease with a maximum contrast value of 7.26 and an average contrast value of 3.14. This generation wasn't significantly different than the initial training data (p=0.062). The MTR_{asym} and Z-spectra for the 8th generation can be seen in **figure 33**.

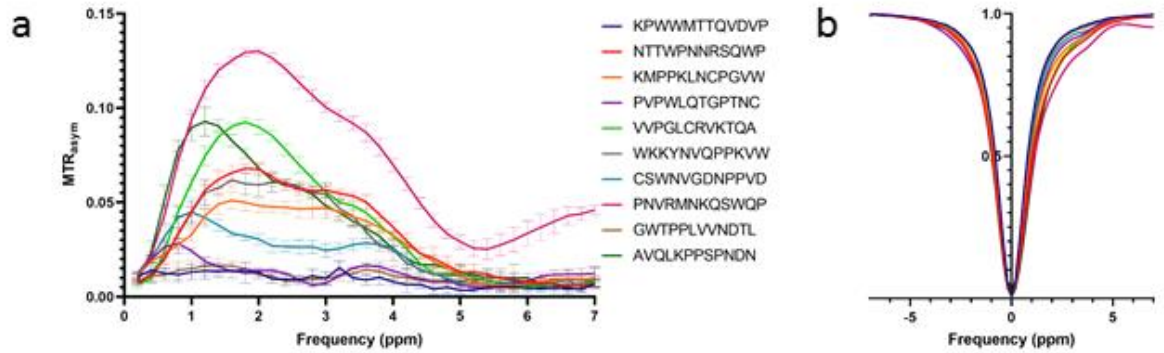


Figure 33: *5 ppm Generation 8. Shows the MTR* ***(a)*** *and Z-spectra* ***(b)*** *acquired from the peptides in the 8th generation of 5 ppm POET design. Points are the average from five experiments for the first five peptides or four for the last five. Error bars show standard deviation.*

In the ninth and final generation performed the worst out of all the generations, performing significantly (p=0.017) worse than the training data. The generation's maximum is the lowest generational maximum at 3.67 and the average was also the lowest with a mean contrast of 2.81. The MTR_{asym} and Z-spectra for the 9th generation can be seen in **figure 34**.

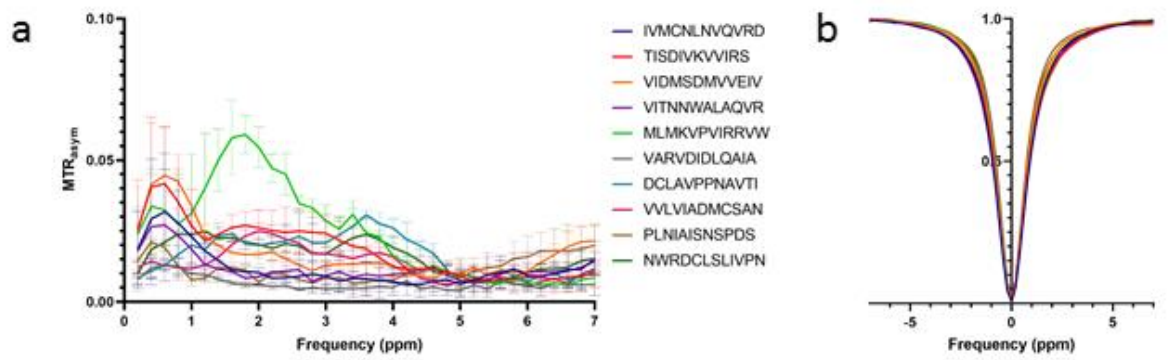


Figure 34: *5 ppm Generation 9. Shows the MTR* ***(a)*** *and Z-spectra* ***(b)*** *acquired from the peptides in the 9th generation of 5 ppm POET design. Points are the average from four experiments for the first five peptides or three for the last five. Error bars show standard deviation.*

Conclusions

Using POET, I was able to develop a single peptide (KVNFNKAVSNLK) that produces a greater amount of contrast at 5 ppm than K12 produces at 3.6 ppm which was over three times the highest amount of 5 ppm contrast in any of the data used to train the initial POEMs for the 5 ppm experiments. This peptide should prove a useful tool as a basis for reporter genes and could become the baseline for later optimization work via other methods such as directed evolution. The contrast produced by this top peptide compared to the contrast of K12 can be seen in **figure 35**.

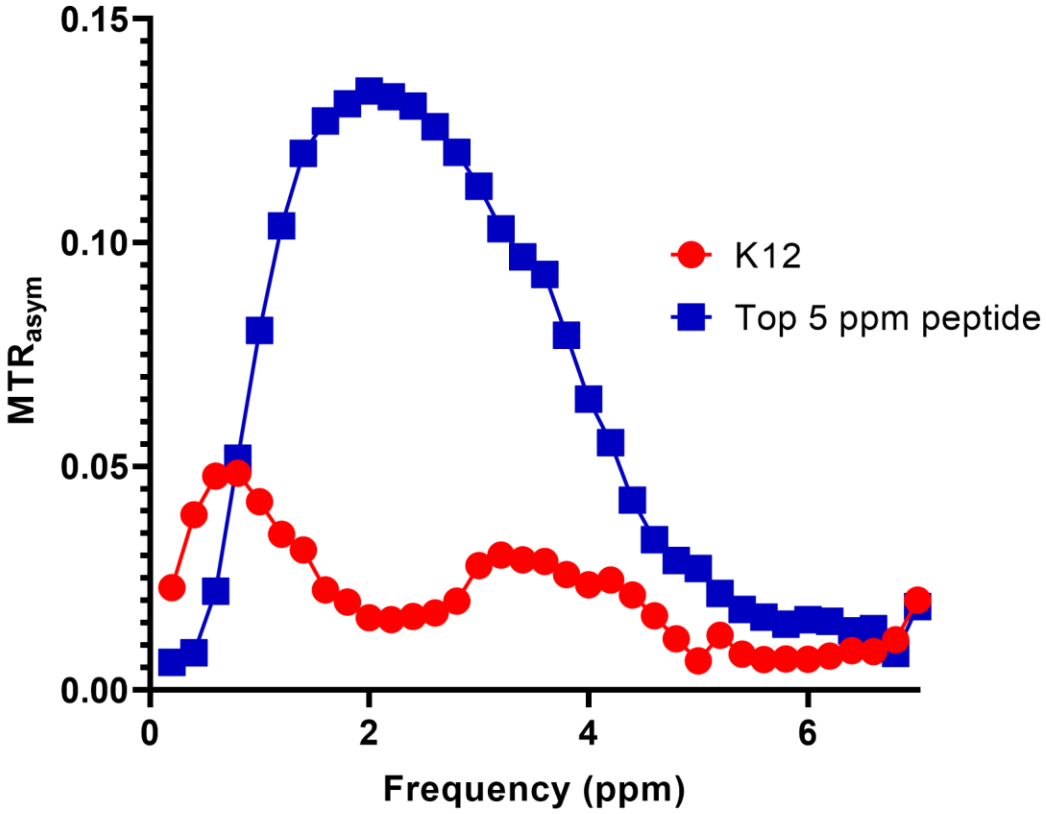


Figure 35: *Top 5ppm peptide vs K12. The top peptide produced by the 5ppm POET experiments is shown beside the K12 MTR. Data is the average of five experiments.*

POET was unable to continue developing better 5 ppm peptides after the 6th generation, and later generations saw increasingly low contrast produced by the peptides being developed, instead of the maximum contrast increasing as the amount of training data increased. The reason for this decrease is unclear, but many factors are possible contributors.

One possible cause is the initial dataset was composed of peptides designed to generate contrast at 3.6 ppm. In such a dataset the majority of 5 ppm contrast is a result of decay from the peak at 3.6 ppm instead of contrast by a proton exchanging at 5 ppm. This bias might have caused POET's learning to focus on increasing 5 ppm contrast in the relatively inefficient

manner of increasing the 3.6 peak. This could be adjusted instead of the PTE at 5 ppm as the fitness value instead calculate the fitness as in equation 3 below.

$$Eq.\ 3 \quad Fitness = {}^{PTE_{5\ ppm}}\Big/_{PTE_{3.6\ ppm}}$$

The fitness being calculated in this way would maintain a selective pressure to produce peptide that generate high contrast at 5 ppm, but make peptides that have a high contribution to their 5 ppm contrast from the 3.6 peak have low fitness, and thus be less likely to be suggested. Another problem was one in the design of POET itself discovered after the experiments were performed, where a bug in POET caused the learning process to end prematurely as the length of learning time increased from additional training data. This bug has been addressed in the current version of POET, but wasn't addressed when this research took place.

CONCLUSION

Functionality of POET

Using POET I have been able to engineer CESTides that produce nearly four times as much contrast as what was priorly used as the state of the art. I was additionally able to develop a 5ppm reporter that produces contrast at 5ppm at a similar scale to what was produced by PLL at 3.6 ppm. This suggests that POET is a useful tool for protein engineers and synthetic biologists, for the development of peptides that can't be screened in a quick or efficient way. The dramatic differences between the peptides discovered and peptides that were used as a starting point would imply that directed evolution would not have developed them within an efficient amount of time.

As the dataset of POET grows its ability to learn seems to decrease after crossing a certain point which I have not had the ability to experimentally determine, but colleagues focusing on improving POET did find several improvements and fix several bugs that allows POET to develop POEMs with lower error faster than they did before. Whether this weakness in learning persists into the current version of POET is a point for further research.

Impact

POET has shown to be an effective means of designing new reporter genes for MRI. This demonstration and the early results from the development of uPar peptides suggest that POET will be useful for researchers seeking to optimize peptides by their function in ways that explore a wide area of the search space.

The 3.6 series peptides show to be unique in chemistry and produce contrast that exceeds that of prior peptides. If developed into fully functional reporter genes, these peptides show

likelihood of becoming the backbone of later efforts for generating MRI reporter genes using CEST.

Future Research

Over the course of me writing this dissertation, POET has continued to develop with the objective of making it more efficient, and able to solve different kinds of problems. There is substantial room for improvement still as POET's ability to learn grows faster and more complete.

Since POET is seeking a simple correlation between peptide function and sequence, it is currently being used to optimize proteins used in guiding extracellular vesicles. These peptides would be used to help the vesicles deliver drugs to the area they are most needed.

The peptides themselves are not at a finished point. To make the jump from protein contrast agent to reporter gene, the peptides need to be assembled into complete proteins and expressed in cells. Work on this has already begun in the Gilad Lab[57].

Several of the proteins discovered by using POET demonstrate properties that are highly unexpected, especially the reporters that have neutral or negative charge. Closer examination to these peptides could yield discoveries in protein biochemistry on what factors influence the exchange rate of amine protons.

The peptides developed with this project are not yet full reporter genes. To reach the point of being full reporter genes the peptides would need to be combined into a longer protein, which would allow more efficient expression *in vivo*. Further to ensure that there are no immunogenicity problems with the fully developed reporter gene, it should go through testing with software to predict immunogenicity, and also undergo testing in humanized small animals

prior to it being useful in clinical work.

REFERENCES

1 Bricco, A. R. *et al.* Protein Optimization Evolving Tool (POET) based on genetic programming. *bioRxiv*, 2022.2003. 2005.483103 (2022).

2 Smith-Bindman, R. *et al.* Trends in Use of Medical Imaging in US Health Care Systems and in Ontario, Canada, 2000-2016. *Jama* **322**, 843-856, doi:10.1001/jama.2019.11456 (2019).

3 Plewes, D. B. & Kucharczyk, W. Physics of MRI: a primer. *Journal of magnetic resonance imaging* **35**, 1038-1054 (2012).

4 Brateman, L. Chemical shift imaging: a review. *American Journal of Roentgenology* **146**, 971-980 (1986).

5 Meyerhoff, D. J., Rooney, W. D., Tokumitsu, T. & Weiner, M. W. Evidence of multiple ethanol pools in the brain: an in vivo proton magnetization transfer study. *Alcoholism: Clinical and Experimental Research* **20**, 1283-1288 (1996).

6 Ward, K., Aletras, A. & Balaban, R. S. A new class of contrast agents for MRI based on proton chemical exchange dependent saturation transfer (CEST). *Journal of magnetic resonance* **143**, 79-87 (2000).

7 Van Zijl, P. C. & Yadav, N. N. Chemical exchange saturation transfer (CEST): what is in a name and what isn't? *Magnetic resonance in medicine* **65**, 927-948 (2011).

8 McMahon, M. T. *et al.* New “multicolor” polypeptide diamagnetic chemical exchange saturation transfer (DIACEST) contrast agents for MRI. *Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine* **60**, 803-812 (2008).

9 Nasrallah, F. A., Pagès, G., Kuchel, P. W., Golay, X. & Chuang, K.-H. Imaging brain deoxyglucose uptake and metabolism by glucoCEST MRI. *Journal of Cerebral Blood Flow & Metabolism* **33**, 1270-1278 (2013).

10 Van Zijl, P. C., Jones, C. K., Ren, J., Malloy, C. R. & Sherry, A. D. MRI detection of glycogen in vivo by using chemical exchange saturation transfer imaging (glycoCEST). *Proceedings of the National Academy of Sciences* **104**, 4359-4364 (2007).

11 Woessner, D. E., Zhang, S., Merritt, M. E. & Sherry, A. D. Numerical solution of the Bloch equations provides insights into the optimum design of PARACEST agents for MRI. *Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine* **53**, 790-799 (2005).

12 Zhang, S., Merritt, M., Woessner, D. E., Lenkinski, R. E. & Sherry, A. D. PARACEST agents: modulating MRI contrast via water proton exchange. *Accounts of chemical research* **36**,

783-790 (2003).

13 Casadaban, M. J., Chou, J. & Cohen, S. N. In vitro gene fusions that join an enzymatically active beta-galactosidase segment to amino-terminal fragments of exogenous proteins: Escherichia coli plasmid vectors for the detection and cloning of translational initiation signals. *Journal of bacteriology* **143**, 971-980 (1980).

14 Chalfie, M., Tu, Y., Euskirchen, G., Ward, W. W. & Prasher, D. C. Green fluorescent protein as a marker for gene expression. *Science* **263**, 802-805 (1994).

15 Brasier, A., Tate, J. & Habener, J. Optimized use of the firefly luciferase assay as a reporter gene in mammalian cell lines. *Biotechniques* **7**, 1116-1122 (1989).

16 Louie, A. Y. *et al.* In vivo visualization of gene expression using magnetic resonance imaging. *Nature biotechnology* **18**, 321-325 (2000).

17 Weissleder, R. *et al.* In vivo magnetic resonance imaging of transgene expression. *Nature medicine* **6**, 351-354 (2000).

18 Cohen, B. *et al.* MRI detection of transcriptional regulation of gene expression in transgenic mice. *Nature medicine* **13**, 498-503 (2007).

19 Genove, G., DeMarco, U., Xu, H., Goins, W. F. & Ahrens, E. T. A new transgene reporter for in vivo magnetic resonance imaging. *Nature medicine* **11**, 450-454 (2005).

20 Kodibagkar, V. D., Yu, J., Liu, L., Hetherington, H. P. & Mason, R. P. Imaging β-galactosidase activity using 19F chemical shift imaging of LacZ gene-reporter molecule 2-fluoro-4-nitrophenol-β-D-galactopyranoside. *Magnetic resonance imaging* **24**, 959-962 (2006).

21 Gilad, A. A., Bar-Shir, A., Bricco, A. R., Mohanta, Z. & McMahon, M. T. Protein and peptide engineering for chemical exchange saturation transfer imaging in the age of synthetic biology. *NMR in Biomedicine*, e4712 (2022).

22 Gilad, A. A. *et al.* Artificial reporter gene providing MRI contrast based on proton exchange. *Nature biotechnology* **25**, 217-219 (2007).

23 Perlman, O. *et al.* Redesigned reporter gene for improved proton exchange-based molecular MRI contrast. *Scientific reports* **10**, 20664 (2020).

24 Goffeney, N., Bulte, J. W., Duyn, J., Bryant, L. H. & Van Zijl, P. C. Sensitive NMR detection of cationic-polymer-based gene delivery systems using saturation transfer via proton exchange. *Journal of the American Chemical Society* **123**, 8628-8629 (2001).

25 Bar-Shir, A. *et al.* Human protamine-1 as an MRI reporter gene based on chemical exchange. *ACS chemical biology* **9**, 134-138 (2014).

26 Bar-Shir, A. *et al.* Transforming thymidine into a magnetic resonance imaging probe for monitoring gene expression. *Journal of the American Chemical Society* **135**, 1617-1624 (2013).

27 Meier, S. *et al.* Non-invasive detection of adeno-associated viral gene transfer using a genetically encoded CEST-MRI reporter gene in the murine heart. *Scientific reports* **8**, 4638 (2018).

28 Farrar, C. T. *et al.* Establishing the lysine-rich protein CEST reporter gene as a CEST MR imaging detector for oncolytic virotherapy. *Radiology* **275**, 746-754 (2015).

29 Romero, P. A. & Arnold, F. H. Exploring protein fitness landscapes by directed evolution. *Nature reviews Molecular cell biology* **10**, 866-876 (2009).

30 Goldsmith, M. & Tawfik, D. S. Enzyme engineering: reaching the maximal catalytic efficiency peak. *Current opinion in structural biology* **47**, 140-150 (2017).

31 Meyer, J. R. *et al.* Repeatability and contingency in the evolution of a key innovation in phage lambda. *Science* **335**, 428-432 (2012).

32 Rohl, C. A., Strauss, C. E., Misura, K. M. & Baker, D. in *Methods in enzymology* Vol. 383 66-93 (Elsevier, 2004).

33 Loshbaugh, A. L. & Kortemme, T. Comparison of Rosetta flexible-backbone computational protein design methods on binding interactions. *Proteins: Structure, Function, and Bioinformatics* **88**, 206-226 (2020).

34 Tallorin, L. *et al.* Discovering de novo peptide substrates for enzymes using machine learning. *Nature Communications* **9**, 5253, doi:10.1038/s41467-018-07717-6 (2018).

35 Xu, Y. *et al.* Deep Dive into Machine Learning Models for Protein Engineering. *Journal of Chemical Information and Modeling* **60**, 2773-2790, doi:10.1021/acs.jcim.0c00073 (2020).

36 David, A., Islam, S., Tankhilevich, E. & Sternberg, M. J. The AlphaFold database of protein structures: a biologist's guide. *Journal of molecular biology* **434**, 167336 (2022).

37 Alley, E. C., Khimulya, G., Biswas, S., AlQuraishi, M. & Church, G. M. Unified rational protein engineering with sequence-based deep representation learning. *Nature Methods* **16**, 1315-1322, doi:10.1038/s41592-019-0598-1 (2019).

38 Dong, T. N., Brogden, G., Gerold, G. & Khosla, M. A multitask transfer learning framework for the prediction of virus-human protein–protein interactions. *BMC Bioinformatics* **22**, 572, doi:10.1186/s12859-021-04484-y (2021).

39 Koza, J. R. Genetic programming as a means for programming computers by natural

selection. *Statistics and computing* **4**, 87-112 (1994).

40 Banzhaf, W., Nordin, P., Keller, R. E. & Francone, F. D. *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications*. (Morgan Kaufmann Publishers Inc., 1998).

41 Koza, J. R. & Andre, D. in *Evolutionary Computation: Theory and Applications* 171-197 (World Scientific, 1999).

42 Widera, P., Garibaldi, J. M. & Krasnogor, N. GP challenge: evolving energy function for protein structure prediction. *Genetic Programming and Evolvable Machines* **11**, 61-88 (2010).

43 Vyas, R. *et al.* Application of genetic programming (GP) formalism for building disease predictive models from protein-protein interactions (PPI) data. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* **15**, 27-37 (2016).

44 Lehman, J. *et al.* The surprising creativity of digital evolution: A collection of anecdotes from the evolutionary computation and artificial life research communities. *arXiv preprint arXiv:1803.03453* (2018).

45 Koza, J. R. Human-competitive results produced by genetic programming. *Genetic programming and evolvable machines* **11**, 251-284 (2010).

46 Mironidou-Tzouveleki, M. & Dokos, C. Nature's drug store attacks again: the cone snail's pain relief toxin. *Aristotle University Medical Journal* **34**, 23-30 (2007).

47 Vendrely, C. & Scheibel, T. Biotechnological production of spider-silk proteins enables new applications. *Macromolecular bioscience* **7**, 401-409 (2007).

48 Kim, M., Gillen, J., Landman, B. A., Zhou, J. & Van Zijl, P. C. Water saturation shift referencing (WASSR) for chemical exchange saturation transfer (CEST) experiments. *Magnetic Resonance in Medicine: An Official Journal of the International Society for Magnetic Resonance in Medicine* **61**, 1441-1450 (2009).

49 Gilad, A. A. *et al.* Functional and molecular mapping of uncoupling between vascular permeability and loss of vascular maturation in ovarian carcinoma xenografts: the role of stroma cells in tumor angiogenesis. *International journal of cancer* **117**, 202-211 (2005).

50 Gilead, A., Meir, G. & Neeman, M. The role of angiogenesis, vascular maturation, regression and stroma infiltration in dormancy and growth of implanted MLS ovarian carcinoma spheroids. *International journal of cancer* **108**, 524-531 (2004).

51 Wimley, W. C. & White, S. H. Experimentally determined hydrophobicity scale for proteins at membrane interfaces. *Nature Structural Biology* **3**, 842-848,

doi:10.1038/nsb1096-842 (1996).

52 Miralavy, I., Bricco, A., Gilad, A. & Banzhaf, W. Using Genetic Programming to Predict and Optimize Protein Function. *arXiv preprint arXiv:2202.04039* (2022).

53 McKAY, D. J., RENAUX, B. S. & DIXON, G. H. Human sperm protamines: Amino-acid sequences of two forms of protamine P2. *European Journal of Biochemistry* **156**, 5-8 (1986).

54 Oskolkov, N. *et al.* Biophysical characterization of human protamine-1 as a responsive CEST MR contrast agent. *ACS Macro Letters* **4**, 34-38 (2015).

55 Bar-Shir, A., Liang, Y., Chan, K. W., Gilad, A. A. & Bulte, J. W. Supercharged green fluorescent proteins as bimodal reporter genes for CEST MRI and optical imaging. *Chemical Communications* **51**, 4869-4871 (2015).

56 Grantham, R. Amino Acid Difference Formula to Help Explain Protein Evolution. *Science* **185**, 862-864 (1974).

57 Fillion, A. J. *et al.* Development of a Synthetic Biosensor for Chemical Exchange MRI Utilizing In Silico Optimized Peptides. *bioRxiv*, 2023.2003. 2008.531737 (2023).

APPENDIX A: CESTIDE PROPERTIES

Table 4: CESTide Properties

Sequence	Series	Gen	Mass	Charge	pI	MTR @3.6	MTR @5	3.6/K12	5.0/K12	3.6 PTE	5.0 PTE
NQYSNWNKNYK	3.6	1	1458.54	2	9.93	2.64%	0.68%	0.3997	0.1036	5.3300	1.3821
NSSNHSNNMPCQ	3.6	2	1332.39	0.2	7.36	28.03%	4.69%	1.3671	0.2290	19.9573	3.3431
IRTYLRKRNSTQ	3.6	2	1535.76	4	12.23	16.22%	1.99%	0.7912	0.0973	10.0214	1.2319
GIFKTTKCKHNS	3.6	2	1363.59	3.2	10.51	15.64%	2.35%	0.7630	0.1144	10.8835	1.6319
SNHKMSECRGLR	3.6	2	1417.62	2.2	9.82	11.62%	1.47%	0.5670	0.0718	7.7795	0.9858
FNSNKITPTSNM	3.6	2	1353.51	1	9.69	9.61%	1.85%	0.4688	0.0903	6.7376	1.2979
VNSDPSNGQMRD	3.6	2	1319.36	-1	4.11	8.31%	2.09%	0.4054	0.1021	5.9771	1.5051
LSNRRGREQYAG	3.6	2	1406.51	2	11.05	13.64%	2.34%	0.6653	0.1140	9.2007	1.5767
QTATENSQMNSG	3.6	2	1267.29	-1	3.85	7.27%	1.43%	0.3545	0.0697	5.4404	1.0699
QTEHYENSARNS	3.6	2	1435.42	-0.8	5.48	2.31%	2.18%	0.1127	0.1062	1.5273	1.4396
KDRTSKPKRPWC	3.6	3	1501.76	3.9	11.06	6.68%	3.47%	1.6561	0.8601	21.4496	11.1401
GRKRGAIWKDTK	3.6	3	1415.65	4	11.75	9.80%	3.99%	2.4314	0.9899	33.4077	13.6019
CCWHNPKWRRTR	3.6	3	1642.92	4.2	11.38	4.81%	2.31%	1.1928	0.5717	14.1218	6.7689
KYTKTRKQSSKA	3.6	3	1425.64	5	11.28	8.76%	3.72%	1.5194	0.6450	20.7299	8.7997
RGKMPLRWMTRK	3.6	3	1559.96	5	12.81	8.94%	3.06%	2.2182	0.7589	27.6587	9.4622
GNCPMKVCSPMG	3.6	3	1223.52	0.9	8.23	7.40%	3.63%	1.8346	0.9003	29.1656	14.3128
VNLPMVMPNLRM	3.6	3	1414.81	1	10.55	6.15%	4.63%	1.0656	0.8024	14.6495	11.0312
GPMPMNAKMKLC	3.6	3	1320.72	1.9	9.67	5.09%	4.89%	0.8821	0.8470	12.9911	12.4750
KVIRYVVAPMKL	3.6	3	1416.82	3	10.9	8.55%	4.54%	1.4825	0.7865	20.3534	10.7975
IKGMNIKMPTDQ	3.6	3	1375.66	1	9.53	11.14%	4.64%	1.9309	0.8049	27.3020	11.3816
MWQMKWTRKTRE	3.6	4	1681.00	3	11.65	10.97%	6.22%	1.7114	0.9701	19.8029	11.2255
HGRKWKRTKFDD	3.6	4	1573.76	3.2	11.05	11.56%	5.64%	1.8044	0.8802	22.3019	10.8788
DKRKIKQKMWWG	3.6	4	1603.94	4	11.25	6.05%	4.33%	0.9433	0.6756	11.4397	8.1935
RRMVNRTITRMW	3.6	4	1619.97	4	12.98	10.39%	6.17%	1.6217	0.9625	19.4717	11.5567
RKHHGWRWEQWK	3.6	4	1733.94	3.5	11.65	8.28%	4.67%	1.2924	0.7282	14.4985	8.1692
HWSTCTRTRTLS	3.6	4	1448.61	2.2	10.53	0.00%	0.00%	0.0000	0.0000	0.0000	0.0000
WWWKPKREDFMK	3.6	4	1737.04	2	10.5	7.31%	3.85%	1.1410	0.6001	15.3203	8.0583

Table 4 (cont'd)

Sequence	Series	Gen	Mass	Charge	pI	MTR @3.6	MTR @5	3.6/K12	5.0/K12	3.6 PTE	5.0 PTE
HIKWRLTKGTRT	3.6	4	1496.77	4.2	12.53	9.54%	5.79%	1.4892	0.9039	19.3531	11.7464
WDRTSTRPSSVL	3.6	4	1404.53	1	10.45	3.98%	3.70%	0.6208	0.5769	8.5967	7.9896
KPWHGCASRTKR	3.6	4	1426.66	4.2	11.65	6.64%	4.53%	1.0365	0.7064	14.1323	9.6317
DKRKIKQKMWWG	3.6	5	1603.94	4	11.35	5.27%	2.13%	1.5834	0.640964	19.20307	7.773029
KKRLHWIRWHCG	3.6	5	1619.95	4.4	11.65	2.03%	0.60%	0.6114	0.180985	7.341661	2.173127
RKHHGWRWEQWK	3.6	5	1733.94	3.5	11.65	7.05%	1.47%	2.1189	0.441435	23.76972	4.951965
WFGLQRHLKKKD	3.6	5	1555.83	3.2	11.08	10.80%	2.78%	3.2462	0.834161	40.58454	10.42875
CHLKDLRKMGLR	3.6	5	1469.83	10.78	3.2	9.20%	3.04%	2.7647	0.912535	36.58821	12.07611
QRHDSHRHGLWL	3.6	5	1541.68	1.7	10.45	3.33%	1.06%	1.0017	0.317516	12.63926	4.006044
LELKLGKRPMGW	3.6	5	1427.76	2	10.79	10.66%	1.60%	3.2035	0.481419	43.64305	6.558619
GQRWLYKMKDSM	3.6	5	1542.83	2	10.75	4.53%	1.66%	1.3625	0.499469	17.17881	6.297021
LDHTWGKWGHQS	3.6	5	1451.55	0.5	7.72	4.68%	1.66%	1.4077	0.49875	18.86464	6.683378
DKVCKIQKRKWH	3.6	5	1568.90	4.2	10.8	3.11%	1.28%	0.9342	0.384813	11.58307	4.770885
WDWEQKKKWI	3.6	6	1446.66	1	9.45	7.77%	2.28%	2.3350	0.6839	31.3955	9.1958
ERQEEKIKKW	3.6	6	1373.56	1	9.45	4.90%	2.12%	1.4733	0.6368	20.8631	9.0175
SDGSKIKDRD	3.6	6	1120.18	0	6.51	1.66%	0.85%	0.4987	0.2569	8.6591	4.4611
SSDQDRDKWL	3.6	6	1249.29	-1	4.31	3.61%	1.00%	1.0840	0.3018	16.8780	4.6992
LLRLLGLVER	3.6	6	1181.48	1	10.45	0.62%	0.40%	0.1851	0.1209	3.0482	1.9900
KEEVWLKWLI	3.6	6	1343.62	0	6.51	2.62%	0.97%	0.8278	0.3058	11.9835	4.4267
KGKLDKDRNL	3.6	6	1186.37	2	10.5	4.93%	1.99%	1.5579	0.6296	25.5433	10.3234
HDDKNKESDD	3.6	6	1202.15	-2.8	4.19	1.45%	0.90%	0.4578	0.2858	7.4068	4.6245
QERRDDILWD	3.6	6	1345.42	-2	4.06	0.59%	0.69%	0.1869	0.2194	2.7023	3.1713
KRIIEDDQLE	3.6	6	1258.38	-2	4.11	2.95%	0.61%	0.9340	0.1934	14.4364	2.9897
VCNRIEPLKPIL	3.6	7	1394.73	0.9	8.55	5.36%	1.23%	1.5409	0.3541	21.4892	4.9386
LHSSQWLKVDHLL	3.6	7	1575.82	0.5	7.72	5.00%	1.85%	1.4376	0.5306	17.7453	6.5493
VINKVISNPCVN	3.6	7	1299.55	0.9	8.55	1.89%	0.72%	0.5440	0.2065	8.1418	3.0914
GNKKNWRWYKNR	3.6	7	1649.86	5	11.76	4.36%	1.64%	1.2538	0.4702	14.7815	5.5431
ICLKSQPICGID	3.6	7	1289.57	-0.1	6.07	6.83%	2.30%	1.9645	0.6600	29.6311	9.9543
LWSDIKMKLKKT	3.6	7	1490.86	3	10.8	11.90%	3.13%	3.7870	0.9970	49.4092	13.0076

Table 4 (cont'd)

Sequence	Series	Gen	Mass	Charge	pI	MTR @3.6	MTR @5	3.6/K12	5.0/K12	3.6 PTE	5.0 PTE
LWSDIKMKLKKT	3.6	7	1490.86	3	10.8	11.90%	3.13%	3.7870	0.9970	49.4092	13.0076
NWRDCLSLIVPN	3.6	7	1429.65	-0.1	6.18	0.73%	0.46%	0.2337	0.1456	3.1803	1.9811
KMGKLIGIPVLK	3.6	7	1296.71	3	11.1	10.03%	3.18%	3.1918	1.0104	47.8780	15.1564
NDISMCNKNNNW	3.6	7	1452.58	-0.1	6.18	2.07%	0.69%	0.6580	0.2180	8.8105	2.9197
VSLQCWELGPNK	3.6	7	1373.58	-0.1	6.18	3.48%	0.86%	1.1072	0.2751	15.6792	3.8963
TVSEPVMMVSVS	3.6	8	1265.50	-1	3.85	1.97%	1.55%	0.4845	0.3811	7.4473	5.8584
PRSWEVKEKETM	3.6	8	1519.73	0	6.62	5.79%	2.42%	1.4242	0.5959	18.2290	7.6270
PGGVRSNDLLEV	3.6	8	1255.38	-1	4.19	3.11%	2.26%	0.7658	0.5569	11.8653	8.6293
PVNRLGKMSKNR	3.6	8	1399.67	4	12.53	8.95%	2.68%	2.2014	0.6583	30.5927	9.1487
VGSVKSGNLRMR	3.6	8	1303.54	3	12.51	6.22%	1.90%	1.7579	0.5361	26.2309	7.9991
TSKSKKRMTAKK	3.6	8	1393.71	6	12.06	7.55%	2.94%	2.1357	0.8311	29.8071	11.5992
ETNVRVKVVSES	3.6	8	1346.49	0	6.51	1.40%	0.92%	0.3946	0.2602	5.7005	3.7595
EPSNLPKGMNEK	3.6	8	1343.51	0	6.51	6.03%	1.10%	1.7061	0.3107	24.7003	4.4982
RLWNSGEGRGEN	3.6	8	1374.42	0	6.51	3.06%	1.25%	0.8660	0.3547	12.2564	5.0203
RPPMLNVVRVVG	3.6	9	1336.66	2	12.5	1.55%	0.64%	0.4524	0.1865	6.5834	2.7145
KWVVRPRIRRLL	3.6	9	1592.00	5	12.98	4.19%	1.07%	1.2228	0.3120	14.9403	3.8123
IGVLRSVKQTVR	3.6	9	1355.64	3	12.51	6.91%	1.68%	2.0159	0.4893	28.9254	7.0210
VINKVISNPCVN	3.6	9	1299.55	0.9	8.55	2.04%	0.64%	0.5951	0.1873	8.9079	2.8029
ETNVRVKVVSES	3.6	9	1346.49	0	6.51	0.61%	0.63%	0.1792	0.1841	2.5889	2.6591
RLPKRVQGNVEK	3.6	9	1423.67	3	11.65	7.37%	1.32%	2.2428	0.4009	30.6422	5.4776
GLGNQHVVVLGV	3.6	9	1191.39	0.2	7.55	0.71%	0.50%	0.2155	0.1521	3.5190	2.4829
KVRCLVEARPSW	3.6	9	1443.72	1.9	9.82	2.00%	0.78%	0.6087	0.2373	8.2005	3.1976
HLVVSPRVSWGC	3.6	9	1339.57	1.2	8.55	1.20%	0.76%	0.3656	0.2306	5.3092	3.3481
IIRSPICCVSRV	3.6	9	1345.68	1.9	8.83	2.93%	0.72%	0.8920	0.2199	12.8927	3.1790
DKRKIKQKMWWG	3.6	10	1603.94	4	11.25	5.03%	2.02%	1.5780	0.6349	19.1361	7.6999
RKHHGWRWEQWK	3.6	10	1733.94	3.5	11.65	5.98%	1.37%	1.8767	0.4311	21.0528	4.8359
EMRQWKWMWENA	3.6	10	1694.94	0	6.51	1.53%	0.94%	0.4802	0.2945	5.5107	3.3791
PIKQIAWPIIEH	3.6	10	1444.73	0.2	7.55	3.19%	1.20%	1.0018	0.3757	13.4878	5.0584
KMWDWEQKKKWI	3.6	10	1706.03	2	10.33	6.78%	2.35%	2.1270	0.7384	24.2503	8.4193

Table 4 (cont'd)

Sequence	Series	Gen	Mass	Charge	pI	MTR @3.6	MTR @5	3.6/K12	5.0/K12	3.6 PTE	5.0 PTE
ARNRKKIMMRWI	3.6	10	1603.02	5	12.81	7.75%	2.83%	2.3923	0.8744	29.0288	10.6099
NAPWKHWRIINE	3.6	10	1563.77	1.2	9.69	2.32%	0.78%	0.7157	0.2423	8.9019	3.0134
NKQRRMLSRERS	3.6	10	1560.79	4	12.51	7.41%	2.15%	2.2884	0.6651	28.5194	8.2886
LSQQPRKRATWR	3.6	10	1526.75	4	12.81	3.21%	0.89%	0.9899	0.2755	12.6112	3.5103
IRRWNDRIRITS	3.6	10	1585.82	3	12.5	3.67%	1.17%	1.1343	0.3612	13.9124	4.4307
RRCQAQEFWLGA	5	1	1464.66	0.9	8.55	3.05%	1.40%	0.7700	0.3542	10.2256	4.7036
GLIEARAMQQCC	5	1	1322.58	-0.1	6.23	1.27%	1.68%	0.3218	0.4237	4.7330	6.2316
QCRAGAMPAMYV	5	1	1297.58	0.9	8.53	4.53%	2.63%	1.1436	0.6639	17.1431	9.9518
NFLRAQRQCQKQ	5	1	1519.74	2.9	11.48	5.54%	1.59%	1.4000	0.4017	17.9181	5.1412
MAMADAAAPMNA	5	1	1164.38	-1	3.75	1.48%	1.24%	0.3734	0.3140	6.2378	5.2457
AQCCQHRKGYMN	5	1	1164.38	2.2	3.75	5.15%	2.28%	1.2996	0.5770	21.7106	9.6385
MAALLYQHRLARR	5	1	1598.92	3.2	12.21	1.04%	1.20%	0.2748	0.3165	3.3429	3.8498
KPCKWAGRACAK	5	1	1318.62	3.9	10.51	5.98%	1.78%	1.5767	0.4698	23.2581	6.9305
CQLAWRPCAKAS	5	1	1333.59	1.9	8.83	6.92%	2.06%	1.8239	0.5433	26.6024	7.9241
QCSGWVQKRQIQ	5	1	1460.67	1.9	9.84	7.56%	2.88%	1.9930	0.7590	26.5398	10.1078
NRVTESVRNVKM	5	2	1432.66	2	11.48	0.93%	0.78%	0.316758	0.265939	4.300599	3.61064
NVVVQRRNHHTS	5	2	1446.58	2.5	12.5	6.99%	1.44%	2.386539	0.492561	32.09008	6.623116
VINKVISCPCVN	5	2	1288.59	0.9	8.23	2.29%	1.56%	0.782529	0.532269	11.81219	8.034538
GGRVWEWNVAA	5	2	1244.36	0	6.34	1.80%	1.03%	0.613375	0.35141	9.587929	5.49304
NNKCQVVAAFVM	5	2	1323.59	0.9	8.55	1.80%	1.03%	2.1930	1.2564	32.2278	18.4637
VPNIQVKGSK	5	3	1069.26	2	10.8	0.95%	0.87%	0.2746	0.2514	4.9961	4.5740
PVARKVVQICHP	5	3	1346.65	2.2	9.48	4.44%	1.20%	1.2888	0.3476	18.6148	5.0211
VTRMTIQVKGSK	5	3	1347.63	3	11.82	7.20%	2.12%	2.0923	0.6170	30.1993	8.9059
MAMADAAAPMNA	5	3	1164.38	-1	3.75	1.44%	0.57%	0.4173	0.1646	6.9711	2.7492
MKVAAAMAPKQV	5	3	1244.58	2	10.8	8.52%	2.13%	2.4757	0.6181	38.6914	9.6601
PVVYKTVIQCCD	5	3	1367.64	-0.1	6.07	0.83%	0.54%	0.2867	0.1869	4.0781	2.6585
KVLWRMPAQIIQ	5	3	1482.84	2	11.66	3.07%	0.74%	1.0557	0.2549	13.8476	3.3434
VSVVATGCVWET	5	3	1250.43	-1.1	3.85	2.77%	0.57%	0.9526	0.1969	14.8188	3.0629
AKCKVQSANVCK	5	3	1278.55	2.9	9.67	7.54%	2.06%	2.5923	0.7082	39.4381	10.7740

Table 4 (cont'd)

Sequence	Series	Gen	Mass	Charge	pI	MTR @3.6	MTR @5	3.6/K12	5.0/K12	3.6 PTE	5.0 PTE
VAWVMKAHVCTM	5	3	1375.73	1.2	8.56	0.95%	0.51%	0.3273	0.1765	4.6271	2.4957
EAPMPKVNVIVN	5	4	1310.57	0	6.34	0.74%	0.41%	0.274349	0.150771	4.071816	2.237699
VPKRLVVVMNTC	5	4	1358.72	1.9	9.84	0.83%	0.53%	0.308921	0.197219	4.422446	2.823342
VINKVISNPCVN	5	4	1299.55	0.9	8.55	1.56%	0.64%	0.578747	0.236603	8.662444	3.541376
PLPDNVVKAVVW	5	4	1336.58	0	6.23	3.57%	0.85%	1.320008	0.314956	19.20996	4.58353
RPPMLNVVRVVG	5	4	1336.66	2	12.5	1.35%	0.49%	0.498041	0.182992	7.247509	2.662906
LTFNLVSDKVVV	5	4	1333.58	0	6.23	2.11%	1.06%	0.6720	0.3371	9.8021	4.9172
TNVLCVPEMVGV	5	4	1260.53	-1.1	3.85	3.30%	1.37%	1.0539	0.4358	16.2628	6.7245
TFPANPNVKVTV	5	4	1286.48	1	9.69	6.87%	1.16%	2.1917	0.3698	33.1381	5.5918
NSWDSVPNPRVV	5	4	1369.49	0	6.23	3.52%	0.81%	1.1243	0.2576	15.9681	3.6585
PPVVNECCPHSV	5	4	1280.48	-0.8	5.35	3.59%	1.47%	1.1448	0.4701	17.3907	7.1409
PGPPAGPTLSNR	5	5	1163.29	1	10.55	4.01%	1.37%	1.0934	0.3739	18.2825	6.2517
NRSMPLPMNPGD	5	5	1328.52	0	6.23	3.31%	1.30%	0.9028	0.3556	13.2179	5.2058
ANNNPWSGMMNG	5	5	1292.41	0	6.02	3.77%	1.56%	1.0282	0.4258	15.4752	6.4084
NSPQLNLNPPQS	5	5	1308.40	0	6.02	4.66%	1.75%	1.2728	0.4775	18.9221	7.0986
NAARMNRLNDAM	5	5	1376.57	1	10.45	3.11%	1.01%	0.8490	0.2769	11.9968	3.9130
CATNAKNNNMRG	5	5	1293.44	1.9	9.84	3.61%	1.23%	1.0891	0.3710	16.3789	5.5788
TTWPNNRSQWP	5	5	1386.48	1	10.55	4.11%	0.99%	1.2405	0.2997	17.4028	4.2046
NQHTPPKPLNNA	5	5	1330.46	1.2	9.69	7.27%	2.20%	2.1977	0.6658	32.1296	9.7337
NKAKACNGMKPR	5	5	1317.59	3.9	11.08	4.70%	1.72%	1.4193	0.5199	20.9532	7.6752
CKPARTDWPPMP	5	5	1398.66	0.9	8.55	3.38%	1.43%	1.0205	0.4317	14.1924	6.0038
PVARKVVQICHP	5	6	1346.65	2.2	9.84	4.24%	1.19%	1.6667	0.4670	24.0737	6.7461
DAPSFAKVVPNR	5	6	1300.47	1	9.71	2.45%	0.78%	0.9633	0.3053	14.4078	4.5663
PLPDNVVKAVVW	5	6	1336.58	0	6.23	6.51%	1.86%	2.5585	0.7317	37.2337	10.6487
RPPMLNVVRVVG	5	6	1336.66	2	12.5	2.06%	1.06%	0.8099	0.4181	11.7854	6.0843
EAPMPKVNVIVN	5	6	1310.57	0	6.34	2.63%	0.95%	1.0341	0.3719	15.3478	5.5189
KVNFNKAVSNLK	5	6	1361.60	3	11.1	9.68%	2.90%	3.3259	0.9962	47.5116	14.2310
LVKPMNKRWVRM	5	6	1557.98	4	12.53	3.15%	1.04%	1.0837	0.3578	13.5299	4.4677
VPKDKRKKLTNP	5	6	1423.71	4	11.25	3.98%	1.71%	1.3678	0.5888	18.6873	8.0437

Table 4 (cont'd)

Sequence	Series	Gen	Mass	Charge	pI	MTR @3.6	MTR @5	3.6/K12	5.0/K12	3.6 PTE	5.0 PTE
VPKRLVVVMNTC	5	6	1358.72	1.9	9.84	0.95%	0.96%	0.3274	0.3314	4.6876	4.7443
AACPNLAVAAPM	5	6	1128.37	-0.1	6.02	1.47%	0.85%	0.5062	0.2931	8.7267	5.0518
RDPDKNNRDWPI	5	7	1525.63	0	6.51	6.85%	2.43%	1.3125	0.4653	16.7336	5.9324
WLLDLRRQMDDT	5	7	1561.77	-1	4.31	2.91%	1.17%	0.5568	0.2248	6.9350	2.7995
QLGVHRNPRWNQ	5	7	1504.66	2.2	12.5	6.60%	2.05%	1.2639	0.3924	16.3390	5.0730
LHKDRRKPLPLP	5	7	1469.78	3.2	11.65	3.61%	1.76%	0.6919	0.3373	9.1566	4.4643
HLRLNKNRMQKS	5	7	1524.80	4.2	12.53	9.33%	3.28%	1.7866	0.6282	22.7905	8.0131
LDDDTPDRNRDW	5	7	1517.52	-3	3.85	1.51%	0.96%	0.2999	0.1908	3.8435	2.4451
VGQNNDERRRQR	5	7	1527.61	2	12.02	2.16%	0.89%	0.4303	0.1770	5.4785	2.2541
HQWHRHRPPIRR	5	7	1675.91	4.7	12.98	2.00%	0.87%	0.3976	0.1738	4.6152	2.0166
DNPHQLLPVNWR	5	7	1488.66	0.2	7.55	2.95%	1.24%	0.5860	0.2467	7.6571	3.2236
RWHHVNVLWQDR	5	7	1645.83	1.5	10.45	1.25%	0.86%	0.2484	0.1718	2.9358	2.0302
KPWWMTTQVDVP	5	8	1487.73	0	6.23	1.03%	0.51%	0.1746	0.0861	2.2833	1.1255
NTTWPNNRSQWP	5	8	1500.58	1	10.55	5.35%	1.66%	0.9058	0.2820	11.7415	3.6550
KMPPKLNCPGVW	5	8	1369.70	1.9	9.67	4.24%	1.51%	0.7186	0.2556	10.2048	3.6293
PVPWLQTGPTNC	5	8	1312.50	-0.1	6.02	1.66%	0.87%	0.2810	0.1470	4.1649	2.1789
VVPGLCRVKTQA	5	8	1270.55	1.9	9.84	4.84%	1.25%	0.8197	0.2118	12.5488	3.2418
WKKYNVQPPKVW	5	8	1572.86	3	10.64	4.28%	1.14%	0.7118	0.1893	8.8021	2.3413
CSWNVGDNPPVD	5	8	1302.38	-2.1	3.49	2.85%	0.95%	0.4739	0.1571	7.0779	2.3468
PNVRMNKQSWQP	5	8	1484.69	2	11.66	9.19%	3.34%	1.5277	0.5545	20.0150	7.2645
GWTPPLVVNDTL	5	8	1311.49	-1	3.75	1.43%	0.75%	0.2383	0.1254	3.5338	1.8603
AVQLKPPSPNDN	5	8	1279.40	0	6.23	4.01%	1.50%	0.6662	0.2495	10.1289	3.7936
IVMCNLNVQVRD	5	9	1403.68	-0.1	6.18	0.82%	0.77%	0.1658	0.1558	2.2970	2.1585
TISDIVKVVIRS	5	9	1329.59	1	9.71	1.97%	1.05%	0.3957	0.2121	5.7882	3.1025
VIDMSDMVVEIV	5	9	1349.62	-3	3.38	1.38%	1.20%	0.2773	0.2424	3.9963	3.4932
VITNNWALAQVR	5	9	1384.59	1	10.55	0.98%	0.93%	0.1964	0.1867	2.7587	2.6231
MLMKVPVIRRVW	5	9	1527.99	3	12.51	3.08%	1.08%	0.6206	0.2178	7.9005	2.7725
VARVDIDLQAIA	5	9	1283.48	-1	4.11	0.53%	0.60%	0.1070	0.1206	1.6214	1.8275
DCLAVPPNAVTI	5	9	1212.42	-1.1	3.75	3.07%	1.14%	0.6178	0.2287	9.9122	3.6693

Table 4 (cont'd)

Sequence	Series	Gen	Mass	Charge	pI	MTR @3.6	MTR @5	3.6/K12	5.0/K12	3.6 PTE	5.0 PTE
VVLVIADMCSAN	5	9	1234.49	-1.1	3.75	1.74%	0.72%	0.3494	0.1450	5.5049	2.2843
PLNIAISNSPDS	5	9	1227.33	-1	3.75	0.77%	1.08%	0.1558	0.2176	2.4694	3.4490
NWRDCLSLIVPN	5	9	1429.65	-0.1	6.18	2.37%	1.01%	0.4776	0.2033	6.4986	2.7662

Notes on data:

The PTE values are calculated through the contrast after it's normalized contrast to K12. Where K12's PTE is 12.5 *1000. This is maintained consistent with our collaborators and the literature values that formed the initial training data for the 3.6 experiments. Data is provided for every peptide that I can find the experiment where it was measured. When possible I use the experiment or experiments that were used to develop the learning data for POET. The first two generations were thus measured by my collaborator Dr. Michael McMahon at Johns Hopkins University, on a 9T scanner instead of the 7T scanners that I used to perform my experiments. Thus contrast values are relatively high in the first two generations, although the contrast relative to K12 is low. PTE values are all divided by 1000, to maintain consistency with the original literature values.

APPENDIX B: POET CODE

POET was written for use in python 3.6.4. This is the version of POET that I used in my experiments detailed in the main body of this dissertation, presented should a reader wish to reproduce my work or deeply examine the code to answer a question. This version of the code is no longer current and contains several known bugs. For an updated version of POET please contract me and I can direct you to an appropriate GitHub.

POET

```
# POET: Protein Optimization Enhancing Tool
# Authors:
#                       Iliya "eLeMeNOhPi" Alavy - Department of Computer Science and Engineering - Michigan State University
#                   Alexander Bricco - Department of Bioengineering -  Michigan State University
#                       All Rights Reserved @ Michigan State University

import argparse
import pandas as pd
import pop as Population
import settings
import optimizer as Optimizer
import predictor as P
import individual as I
import math
import fitness as F
import os
import random as R
from subprocess import call
import archivist as Archivist

def main():
        print ("\n\n############################################################\n")
        print ("POET V3.0 \n")
        print ("############################################################\n")

        print ("Configuring the application...\n")

   # Argument descriptions
```

```
	parser = argparse.ArgumentParser(description='Finds a model to predict fitness value of any given protein sequence. Fitness can be manually defined to any protein characteristic but our main goal is to predict CEST ability of proteins')

	parser.add_argument('-learn', default=settings.default_learn, help='Path to the learn data (format: csv, default: ' + settings.default_learn + ')')
	parser.add_argument('-translation', default=settings.TT, help='Path to the translation table (format: csv, default: ' + settings.TT + ')')
	parser.add_argument('-pop', help='Path to the initial population files. If not specified, this application uses random initial population as default (format: csv)')
	parser.add_argument('-model', help='Path to a generated model to determine a given protein\'s fitness. You will need to use -seq after this command')
	parser.add_argument('-seq', help='A sequence to be tested using an already specified model. This command only runs if it\'s used jointly with the -model command')
	parser.add_argument('-predict', help='Number of potential protein sequences you want the program to predict. Must be jointly used with -seqsize and -iter')
	parser.add_argument('-seqsize', help='Size of the protein sequences for prediction')
	parser.add_argument('-iter', help='Number of iterations to predict/find potential proteins')
	parser.add_argument('-hpcc', help="Number of replications you need to queue on hpcc. Uses the default config file")
	parser.add_argument('-o', help="Output file name")
	parser.add_argument('-r', default=settings.runs, help='Number of GP iterations to find a model.')
	parser.add_argument('-seed', help='The random seed')
	parser.add_argument('-f', help="Gets path to a model as it's input and returns the fitness of it") #ToDo:: Code this part.
	parser.add_argument('-c', nargs='*', help="Compares the fitness of all given models")
	parser.add_argument('-al', nargs='*', help="Computes the average length of all given models")
	parser.add_argument('-archive', nargs='*', help='Setups the default output directories if necessary and archives existing files/results')

	args = parser.parse_args()

	# Read the tables
	settings.learn_df = pd.read_csv(args.learn)
	settings.TT = pd.read_csv(args.translation)

	arch = Archivist.Archivist()

	if args.archive != None:
		arch.setup(True)
		print("Archiving completed Successfully!")
```

```
		exit(1)

	# Setting up the random seed
	if args.seed != None:
		settings.seed = int(args.seed)
	R.seed(settings.seed)

	# total number of runs
	if args.r != None:
		settings.runs = int(args.r)

	# output file name
	if args.o != None:
		settings.output_file_name = args.o

	# Should be after initializing the output file name ^^^
	arch.setup()

	# run on hpcc
	if args.hpcc != None:
		hpcc(int(args.hpcc), 13, 10000, 300)
		exit(1)

	# get the fitness of a given model
	if args.f != None:
		modelFitness(args.f)

	# compare a bunch of models
	if args.c != None:
		compareModels(args.c)

	# calculate the average length of the models
	if args.al != None:
		averageLength(args.al)

	# Make a dictionary out of the translation table
	for i, row in settings.TT.iterrows():
		settings.dic[row[0]] = row[1]

	# Translate the data using the translation table in case we are not using a numeric
translation table
	#if not is_numeric():
		#for i, row in settings.learn_df.iterrows():
		#	translatedSeq = ""
```

```
                        #origSeq = row[0]
                        #for j in range(len(origSeq)):
                        #       try:
                                #       translatedSeq += settings.dic[origSeq[j]]
                        #       except IndexError:
                        #               print("Could not find all the required data in the
Translation Table. Exiting...")
                        #               exit(0)
                        #settings.learn_df.iloc[i, 0] = translatedSeq

        # ToDo:: I never checked the prediction part
        if args.predict != None:
                if args.seqsize == None or args.iter == None:
                        raise("Prediction mode needs -seqsize and -iter in order to work")
                else:
                        predictor = P.Predictor(int(args.predict), int(args.seqsize)+1, int(args.iter))
                        predictor.predict()
                        return
        elif args.pop == None and args.seq == None and args.model == None:
                pop = Population.Population()
                opt = Optimizer.Optimizer(pop)
                opt.optimize()
        elif args.seq != None and args.model != None:
                proteinSeq = args.seq
                modelPath = args.model
        else:
                raise("Invalid arguements.")
                pass
        return

def is_numeric():
        codes = settings.TT['code']
        for i in range(codes.size):
                try:
                        float(codes[i])
                except ValueError:
                        return False
        return True

def modelFitness(path):
        model = I.Individual()
        model.makeFromFile(path)
        f = F.Fitness()
        fitness = f.measureTotal(model)
```

```
        print("Pro-Predictor: Fitness (RMSE) of {}: {}".format(path, fitness))
        exit(1)

def compareModels(paths):
        model = I.Individual()
        avg = 0
        best = 100000
        bestModel = ''
        for path in paths:
                model.makeFromFile(path)
                f = F.Fitness()
                fitness = f.measureTotal(model)
                avg += fitness
                print("Pro-Predictor: Fitness (RMSE) of {}: {}".format(path, fitness))
                if fitness < best:
                        bestModel = path
                        best = fitness
        avg /= len(paths)
        print("Pro-Predictor: Best model: {} with RMSE: {} Average RMSE: {}".format(bestModel,
best, avg))
        exit(1)

def averageLength(paths):
        model = I.Individual()
        for index, path in enumerate(paths):
                model.makeFromFile(path)
                f = F.Fitness()
                fitness = f.measureTotal(model)
                sumL = 0
                counter = 0
                for rule in model.rules:
                        if rule.status:
                                try:
                                        if math.isnan(rule.pattern):
                                                continue
                                except:
                                        pass
                                sumL += len(rule.pattern)
                                counter += 1
                if counter == 0:
                        print("Pro-Predictor: {}: Old Model. Old Models are Not
Supported".format(path))
                        continue
                avg = sumL/counter
```

```
                print("Pro-Predictor: {}: Fitness (RMSE): {} Average Rule Length: {}".format(path,
fitness, avg))
        exit(1)

def hpcc(reps, hours, runs, startingSeed):
        for i in range(reps):
                filename = "subs/{}.sb".format(i)
                file = open(filename, "w")
                file.write("#!/bin/bash --login\n")
                file.write("\n########## SBATCH Lines for Resource Request ##########\n\n")
                file.write("#SBATCH --time={}:02:00           # limit of wall clock time - how long
the job will run (same as -t)\n".format(hours))
                file.write("#SBATCH --nodes=1-5              # number of different nodes - could be
an exact number or a range of nodes (same as -N)\n")
                file.write("#SBATCH --ntasks=5               # number of tasks - how many tasks
(nodes) that you require (same as -n)\n")
                file.write("#SBATCH --cpus-per-task=2        # number of CPUs (or cores) per task
(same as -c)\n")
                file.write("#SBATCH --mem-per-cpu=2G         # memory required per allocated
CPU (or core) - amount of memory (in bytes)\n")
                file.write("#SBATCH --job-name POET_rep_{}     # you can give your job a name
for easier identification (same as -J)\n".format(i))
                file.write("\n########## Command Lines to Run ##########\n\n")
                file.write("module pandas\n")
                # file.write("module load GCC/6.4.0-2.28 OpenMPI  ### load necessary modules,
e.g\n")
                file.write("cd ~/POET\n")
                file.write("srun -n 5 python poet.py -r {} -o {} -seed {}\n".format(runs, i,
i+startingSeed))
                file.write("cd batch\n")
                file.write("scontrol show job $SLURM_JOB_ID     ###
```

Optimizer

```
# Authors: Iliya "eLeMeNOhPi" Alavy - Department of Engineering - Michigan State University
#                   Alexander Bricco - Department of Bioengineering -  Michigan State University

import settings
import pop as Population
import fitness as F
import pandas as pd
import archivist as Archivist
import copy
import random as R
import individual as I
```

```
import rule as Rule
import time

class Optimizer:
	def __init__(self, population):
		self.TT = settings.TT
		self.codes = self.TT['code']
		self.P = population
		self.runs = settings.runs
		self.tournamentSize = settings.tournament_size
		self.logInterval = settings.pop_log_interval
		self.crossRate = settings.cross_rate
		self.ruleSize = settings.rule_size
		self.ruleCount = settings.maximum_rule_count
		self.minWeight = settings.rule_weight_min
		self.maxWeight = settings.rule_weight_max
		self.output_file_name = settings.output_file_name
		self.mAR = settings.mut_add_rule
		self.mRR = settings.mut_remove_rule
		self.mCW = settings.mut_change_weight
		self.mATP = settings.mut_add_to_pattern
		self.mRFP = settings.mut_remove_from_pattern
		self.mCWmin = 0
		self.mCWmax = 1
		pass

	def optimize(self):
		# We need an instance of the Fitness and Archivist classes for later usage
		fitness = F.Fitness()
		arch = Archivist.Archivist()

		# ToDo::
		# raise("Add dynamic mutation rates to escape from premature
convergence.\nAdd Remove duplicate rules to make space for the new rules without changing
the fitness")

		# growthLog = 0
		# mutRates = [self.mAR, self.mRR, self.mCW, self.mATP, self.mRFP,
self.mCWmin, self.mCWmax]

		# For all the generations
		for i in range(self.runs):
			# To log the time
			start_time = int(round(time.time() * 1000))
```

```
                    # Measure fitness
                    avgFitness = 0.0
                    avgRuleCount = 0.0
                    avgUsedRulesCount = 0.0

                    # To calculate the best fitness
                    # Temporarily set the first member as the best member
                    self.P.pop[0].fitness = fitness.measureTotal(self.P.pop[0])
                    bestIndividual = self.P.pop[0]

                    for j in self.P.pop:
                            j.fitness = fitness.measureTotal(j)
                            avgFitness += j.fitness
                            avgRuleCount += len(j.rules)
                            avgUsedRulesCount += j.usedRulesCount
                            if j.fitness < bestIndividual.fitness:
                                    bestIndividual = j

                    # To calculate the average fitness and the average
                    avgFitness = round(avgFitness / float(len(self.P.pop)), 3)
                    avgRuleCount = round(avgRuleCount / float(len(self.P.pop)), 0)
                    avgUsedRulesCount = round(avgUsedRulesCount / float(len(self.P.pop)),
0)
                    bestFitness = round(bestIndividual.fitness, 3)
                    bestRuleCount = len(bestIndividual.rules)
                    bestUsedRulesCount = bestIndividual.usedRulesCount

                    # Log the outcome before doing the changes to the population /
generating a new population
                    log_string = "{}: -b {} -rc {} -urc {} ||| -a {} -arc {} -aurc {}".format(i,
bestFitness, bestRuleCount, bestUsedRulesCount, avgFitness, avgRuleCount,
avgUsedRulesCount)

                    # Print the evolutionary log
                    print(log_string)

                    # Dynamic Mutation
                    # If the fitness is not growing every 25 generations, increase the rates
                    # if i%25 == 0 and bestIndividual.fitness == growthLog:
                    #       print("Mutation rate increasing")
                    #       self.mAR += 0.05
                    #       self.mRR += 0.1
                    #       self.mCW += 0.2
```

```
                    #       self.mATP += 0.1
                    #       self.mRFP += 0.1
                    #       self.mCWmin = 0
                    #       self.mCWmax = 0.1
                    #       if self.mCW >= 0.6:
                    #               self.mAR = mutRates[0]
                    #               self.mRR = mutRates[1]
                    #               self.mCW = mutRates[2]
                    #               self.mATP = mutRates[3]
                    #               self.mRFP = mutRates[4]
                    # elif i%25 == 0 and growthLog != bestIndividual.fitness:
                    #       print("Mutation rate reset")
                    #       self.mAR = mutRates[0]
                    #       self.mRR = mutRates[1]
                    #       self.mCW = mutRates[2]
                    #       self.mATP = mutRates[3]
                    #       self.mRFP = mutRates[4]
                    #       self.mCWmin = mutRates[5]
                    #       self.mCWmax = mutRates[6]
                    #       growthLog = bestIndividual.fitness


                    # Log the evolution
                    arch.saveEvo(log_string)

                    # Create a copy of the population
                    newPop = copy.deepcopy(self.P)
                    newPop.pop.clear()
```

Note: This is the code for storing the population with an interval of generations. I temporarily disabled this feature to see if it needs any further developments. Right now I'm only concerned about the core.

```
                    # # Check if we need to store the data
                    # if i % self.logInterval == 0 and self.logInterval > 0:
                    #       index = 0
                    #       for individual in self.P.pop:
                    #               data = []
                    #               for rule in individual.rules:
                    #                       data.append([rule.pattern, rule.weight,
rule.status])
                    #               # Store the data
                    #               df =
pd.DataFrame(data,columns=['pattern','weight','used'])
```

```
                        #                   arc.saveCSV(df, self.outputPath + "/" + str(i),
str(100+index))
                        #                   index += 1

                        # Elitism
                        newPop.pop.append(bestIndividual)

                        # Save the best model
                        data = []
                        for rule in bestIndividual.rules:
                                data.append([rule.pattern, rule.weight, rule.status])
                        df = pd.DataFrame(data,columns=['pattern','weight','status'])
                        arch.saveModel(df)

                        # Select Parents (Tournament Selection) and Crossover
                        for k in range(len(self.P.pop)-1):
                                tournament = []
                                offspring = I.Individual()

                                for j in range(self.tournamentSize):
                                        # Randomly append as much individuals to the
tournament as we need
                                        tournament.append(self.P.pop[R.randint(0,
len(self.P.pop)-1)])

                                tournament = self.bubbleSortTournament(tournament)

                                # We got two best parents
                                parentA = copy.deepcopy(tournament[0])
                                parentB = copy.deepcopy(tournament[1])

                                # Do the crossover magic - Cluster crossover
                                # Efficiency thing. Find the greater rule length
                                lenA = len(parentA.rules)
                                lenB = len(parentB.rules)
                                maxLen = lenA
                                if lenA < lenB:
                                        maxLen = lenB

                                # we keep track of the rules we want to add to the offspring
                                rules = []
                                for j in range(maxLen):
                                        if j < lenA:
                                                ruleA = parentA.rules[j]
```

```
                                        if ruleA.status == 1:
                                                rules.append(ruleA)
                                        elif R.random() < self.crossRate: # we give unused
rules some chance to get selected
                                                rules.append(ruleA)

                                if j < lenB:
                                        ruleB = parentB.rules[j]
                                        if ruleB.status == 1:
                                                rules.append(ruleB)
                                        elif R.random() < self.crossRate: # we give unused
rules some chance to get selected
                                                rules.append(ruleB)



                        offspring.rules = rules
                        offspring.bubbleSort()

                        # Resize the offspring so it doesn't exceed the maximum allowed
count
                        while len(offspring.rules) > self.ruleCount:
                                countGreens = 0
                                for index in range(len(offspring.rules)-1, -1, -1):
                                        if(countGreens >= index):
                                                del(offspring.rules[index])
                                                break
                                        else:
                                                if(offspring.rules[index].status == 0):
                                                        del(offspring.rules[index])
                                                        break
                                                else:
                                                        countGreens += 1

                                newPop.pop.append(offspring)

                self.P.pop = copy.deepcopy(newPop.pop)

                # Mutations

                # We keep a copy of the elite
                elite = copy.deepcopy(self.P.pop[0])

                for indv in self.P.pop:
```

```
                # On Model
                if R.random() <= self.mAR:
                        # add rule
                        self.mut_add_rule(indv)

                if R.random() <= self.mRR:
                        # remove rule
                        self.mut_remove_rule(indv)

                # On Rule
                for rule in indv.rules:
                        if R.random() <= self.mCW:
                                # change weight
                                self.mut_change_weight(rule)
                        if R.random() <= self.mATP:
                                # add to pattern
                                self.mut_add_to_pattern(rule)
                        if R.random() <= self.mRFP:
                                # remove from pattern
                                self.mut_remove_from_pattern(rule)
                                if rule.pattern == "":
                                        indv.rules.remove(rule)
                        indv.bubbleSort()

        zeroFitness = fitness.measureTotal(self.P.pop[0])

        # Check if elite got worse
        if elite.fitness < zeroFitness:
                self.P.pop[0] = elite

        # self.P.pop[0].print()

        # for indv in self.P.pop:
        #       # print(len(indv.rules))
        #       self.removeExtra(indv)
        #       # print("after: {}".format(len(indv.rules)))

        # self.P.pop[0].print()

        # To Log the time
        end_time = int(round(time.time() * 1000)) - start_time
        # print(end_time)
pass
```

```
        def bubbleSortTournament(self, t):
            n = len(t)
            # Traverse through all array elements
            for i in range(n):
                # Last i elements are already in place
                for j in range(0, n-i-1):
                    # traverse the array from 0 to n-i-1
                    # Swap if the element found is greater
                    # than the next element
                    if t[j].fitness > t[j+1].fitness:
                     t[j], t[j+1] = t[j+1], t[j]
            return t

        # Add a random rule mutation
        def mut_add_rule(self, individual):
                if len(individual.rules) >= self.ruleCount:
                        return
                pattern = ""
                weight = round(R.uniform(self.minWeight, self.maxWeight), 2)
                # Add these many rules
                for i in range(R.randint(1, self.ruleSize)):
                        # Rule size is calculated randomly, and now we need to select a random
combination of codes with a specified size
                        code = self.codes[R.randint(0, self.codes.size - 1)]
                        randomchar = code[R.randint(0, (len(code) - 1))]
                        pattern += randomchar
                rule = Rule.Rule(pattern, weight,0)
                individual.rules.append(rule)

        # Remove rule mutation
        def mut_remove_rule(self, individual):
                # print("mrr")
                fitness = F.Fitness()
                if len(individual.rules) == 0:
                        return
                tempRand = R.randint(0, len(individual.rules)-1)
                del(individual.rules[tempRand])

        # Add to weight mutation
        def mut_change_weight(self, rule):
                optRand = R.randint(0, 1)
                weightRand = round(R.uniform(self.mCWmin, self.mCWmax), 2)
                if optRand == 0:
                        # Addition
```

```
                rule.weight += weightRand
        elif optRand == 1:
                # Substraction
                rule.weight -= weightRand
        pass

# Alter patterns mutation (add letter)
def mut_add_to_pattern(self, rule):
        if len(rule.pattern) >= self.ruleSize:
                return
        pattern = rule.pattern
        code = self.codes[R.randint(0, self.codes.size - 1)]
        randomchar = code[R.randint(0, (len(code) - 1))]
        if len(pattern) == 0:
                pattern = randomchar
        else:
                insPos = R.randint(0, len(pattern))
                pattern = pattern[0:insPos] + randomchar + pattern[insPos:(len(pattern))]
        rule.pattern = pattern

# Alter patterns mutation (remove letter)
def mut_remove_from_pattern(self, rule):
        if len(rule.pattern) == 0:
                return
        if len(rule.pattern) == 1:
                rule.pattern = ""
                return
        pattern = rule.pattern
        insPos = R.randint(0, len(pattern)-1)
        pattern = pattern[0:insPos] + pattern[insPos+1:(len(pattern))]
        rule.pattern = pattern

def removeExtra(self, indv):
        # removeList = []
        for j, rule in enumerate(indv.rules):
                for k in range(j+1, len(indv.rules)):
                        if indv.rules[k].pattern == indv.rules[j].pattern:
                                # removeList.append(indv.rules[j])
                                if indv.rules[j].status == 0:
                                        indv.rules.remove(indv.rules[j])
                                        j -= 1
                                elif indv.rules[k].status == 0:
                                        indv.rules.remove(indv.rules[k])
                                        k -= 1
```

```
                                    break
            # for i in removeList:
            #         indv.rules.remove(i)
```

Settings

```
# Authors: Iliya "eLeMeNOhPi" Alavy - Department of Engineering - Michigan State University
#                   Alexander Bricco - Department of Bioengineering -  Michigan State University

import configparser

import random as R

global rule_size, TT, default_learn, default_unseen, population_size, alphabet_size,
maximum_rules_count, rule_value_min, rule_value_max

config = configparser.ConfigParser()
config.read('config.ini')

# Parameter Initialization

# LEARN
default_learn = config['LEARN']['learn_data']
default_unseen = config['LEARN']['unseen_data']

# PREDICTOR
prediction_model = config['PREDICTOR']['prediction_model']

# TRANSLATION
TT = "data/translation/" + config['TRANSLATION']['translation_table']
dic = {}
alphabet_size = int(config['TRANSLATION']['alphabet_size'])

# GENETIC PROGRAMMING
population_size = int(config['GP']['population_size'])
rule_size = int(config['GP']['maximum_rule_size'])
maximum_rule_count = int(config['GP']['maximum_rule_count'])
rule_weight_min = float(config['GP']['rule_weight_min'])
rule_weight_max = float(config['GP']['rule_weight_max'])
runs = int(config['GP']['runs'])
pattern_mode = int(config['GP']['pattern_mode'])
tournament_size = int(config['GP']['tournament_size'])
cross_rate = float(config['GP']['crossover_unused_selection_chance'])

# output_name
```

```
pop_log_interval = int(config['OUTPUT']['pop_log_interval'])
output_file_name = config['OUTPUT']['output_name']

# GENERAL
seed = int(config['GENERAL']['seed'])
debug = int(config['GENERAL']['debug'])

# MUTATION
mut_add_rule = float(config['MUTATION']['mut_add_rule'])
mut_remove_rule = float(config['MUTATION']['mut_remove_rule'])
mut_change_weight = float(config['MUTATION']['mut_change_weight'])
mut_add_to_pattern = float(config['MUTATION']['mut_add_to_pattern'])
mut_remove_from_pattern = float(config['MUTATION']['mut_remove_from_pattern'])
```

Pop

```
# Authors: Iliya "eLeMeNOhPi" Alavy - Department of Engineering - Michigan State University
#                   Alexander Bricco - Department of Bioengineering -  Michigan State University

import individual as Individual
import settings

class Population:
        # constructor for random initialization
        def __init__(self):
                self.size = settings.population_size
                self.TT = settings.TT
                self.pop = []

                print ("Initializing a population with size of " +str(self.size) + "...\n")

                if self.is_numeric():
                        print ("Translation Table supports the NUMERIC mode - Generating
formula...\n")
                        self.populate_formulas()
                else:
                        print ("Translation Table supports the PATTERN mode - Generating
rules...\n")
                        self.populate_rules()

        # Randomly initializes the population with rules
        def populate_rules(self):
                for i in range(self.size):
                        indv = Individual.Individual()
                        indv.init_pattern()
```

```
                self.pop.append(indv)

        # Randomly initializes the population with formulas
        def populate_formulas(self):
                raise("This feature is not coded yet.")
                for i in range(self.size):
                        indv = Individual.Individual()
                        indv.init_formula()
                        self.pop.append(indv)

        # Uses the input files to create the population
        def populate_preset(self, population):
                raise ValueError('Uncharted terretories... Exiting')
                pass

        # Returns True if all the codes are numeric, otherwise returns False
        def is_numeric(self):
                codes = self.TT['code']
                for i in range(codes.size):
                        try:
                                float(codes[i])
                        except ValueError:
                                return False
                return True
```

Individual

```
# Authors: Iliya "eLeMeNOhPi" Alavy - Department of Engineering - Michigan State University
#                  Alexander Bricco - Department of Bioengineering -  Michigan State University
import rule as Rule
import settings
import random as R
import pandas as pd

class Individual:
        # Constructor
        def __init__(self):
                self.TT = settings.TT
                self.aSize = settings.alphabet_size
                self.rules = []
                self.usedRules = {}
                self.usedRulesCount = 0
                self.ruleSize = settings.rule_size
                self.maxRuleCount = settings.maximum_rule_count
                self.minWeight = settings.rule_weight_min
```

```
            self.maxWeight = settings.rule_weight_max
            self.fitness = 0
            self.validate = 0
            self.extra = {}

        def makeFromFile(self, file):
            # print ("Creating an Individual from file...")
            self.rules = []
            self.usedRules = {}
            self.fitness = 0
            tempIndv = pd.read_csv(file)
            tmpPatterns = tempIndv['pattern']
            tmpWeights = tempIndv['weight']
            try:
                tmpStatus = tempIndv['status']
            except:
                print("Pro-Predictor: model {} does not have status column, putting 0 for
all.".format(file))
                tmpStatus = [0]*len(tmpPatterns)
            # print("size " + str(tmpPatterns.size))
            for i in range(0, len(tmpPatterns) - 1):
                # print (str(i) + " " + str(tmpPatterns[i]) + " " + str(tmpWeights[i]))
                rule = Rule.Rule(tmpPatterns[i],tmpWeights[i], tmpStatus[i])
                self.rules.append(rule)
            pass

        def init_pattern(self):
            codes = self.TT['code']
            for i in range(R.randint(1, int(self.maxRuleCount/3))):
                pattern = ""
                weight = round(R.uniform(self.minWeight, self.maxWeight), 2)
                # Add these many rules
                for j in range(R.randint(1, self.ruleSize)):
                    # Rule size is calculated randomly, and now we need to select a
random combination of codes with a specified size
                    code = codes[R.randint(0, codes.size - 1)]
                    randomchar = code[R.randint(0, (len(code) - 1))]
                    pattern += randomchar
                rule = Rule.Rule(pattern, weight, 0)
                self.rules.append(rule)

            # for i in self.rules:
            #     print(str(i.pattern) + " => " + str(i.weight))
```

```
                # print ("\n\n")
                self.bubbleSort()

                # for i in self.rules:
                #         print(str(i.pattern) + " => " + str(i.weight))


        def init_formula(self):
                raise Exception ("uncharted territories")
                pass

        def bubbleSort(self):
           n = len(self.rules)
           # Traverse through all array elements
           for i in range(n):
              # Last i elements are already in place
              for j in range(0, n-i-1):
                 # traverse the array from 0 to n-i-1
                 # Swap if the element found is greater
                 # than the next element
                 if len(self.rules[j].pattern) < len(self.rules[j+1].pattern):
                  self.rules[j], self.rules[j+1] = self.rules[j+1], self.rules[j]

        def print(self):
                for kh, rule in enumerate(self.rules):
                        print("{}- {} - {} - {}".format(kh, rule.pattern, rule.weight, rule.status))
```

Fitness

```
# Authors: Iliya "eLeMeNOhPi" Alavy - Department of Engineering - Michigan State University
#                   Alexander Bricco - Department of Bioengineering -  Michigan State University

import random as R
import settings
import individual as I
import math

class Fitness:
        def __init__(self):
                self.learn = settings.learn_df
                self.mode = settings.pattern_mode # 0 is the summation mode and 1 is the
multiplication mode
                self.k = 0 # for 10-fold cross validation implementation
```

```
                pass

        def measureTotal(self, individual):
                # Re-initalize the values to 0
                fitness = 0.0 # this is the model's fitness
                sum = 0.0
                individual.usedRulesCount = 0

                # Make sure that every rule status is 0
                for rule in individual.rules:
                        rule.status = 0

                # Iterating through the training set
                for i, row in self.learn.iterrows():
                        # Pretty much initialization
                        sequence = row[0]
                        actualFitness = row[1]
                        # This is the starting position of the sequence that we're looking at each
time.
                        pos = 0
                        measuredFitness = 0.0 # Fitness of individual i

                        # Iterate through the sequence
                        while pos < len(sequence):
                                # Check every rule
                                for rule in individual.rules:
                                        # Find the length of the rule
                                        try:
                                                length = len(rule.pattern)
                                        except:
                                                # happens in the case of empty rules.
                                                continue

                                        # Continue if the rule length is more than the unchecked
sequence length
                                        if length < (len(sequence) - pos) or length == 0:
                                                continue

                                        reverse_pattern = rule.pattern[::-1]
                                        # Normal or Reverse
                                        if ((rule.pattern == sequence[pos : (pos + length)]) or
reverse_pattern == sequence[pos : (pos + length)]):
                                                # rule is found. Update its status and the
usedRulesCount to avoid further computation
```

```
                                        if rule.status == 0:
                                                rule.status = 1
                                                individual.usedRulesCount += 1

                                        # Check the multiplication/summation mode. We
always use the summation mode tho.
                                        if self.mode == 0:
                                                measuredFitness += rule.weight
                                        elif self.mode == 1:
                                                measuredFitness *= rule.weight

                                        # If the rule is found, we don't wanna check any
other "shorter" rules on the very same position. We wanna update the positioning and look for
other rules. If no rule was found, the position updates anyway after the end of this loop so the
difference is only the break command which happens only if the rule is found. This defenitely
takes more processing power and considers overlapping rules but this is the way to go to
consider all the possible cases. Also, note that this raises the issue of exponential slow-down
when the number of rules in a table increase. Therefore, I assume we need some sort of bloat
removal when the speed of the tool has decreased drastically.
                                        break

                                # Update the position
                                pos += 1

                        # Calculate the error after the estimation
                        error = (measuredFitness - actualFitness)**2
                        fitness += error

                MSE = fitness / int(self.learn.size)
                RMSE = math.sqrt(MSE)
                return RMSE

        def measure(self, individual):
                raise("update this function according to measureTotal")
                if self.k == 10:
                        self.k = 0
                fitness = 0.0
                sum = 0.0
                for i, row in self.learn.iterrows():
                        if i >= int(self.learn.size/10) * self.k and i <
int(self.learn.size/10)*(self.k+1):
                                continue
                        sequence = row[0]
                        actualFitness = row[1]
```

```
                pos = 0
                measuredFitness = 0.0 # Fitness of individual i
                if self.mode == 1:
                        measuredFitness = 1.0

                # Iterate through the sequence
                while pos < len(sequence):
                        # Check every rule
                        found = False
                        for rule in individual.rules:
                                length = len(rule.pattern)
                                reverse_pattern = rule.pattern[::-1]
                                # Normal or Reverse
                                if (pos + length < len(sequence) and (rule.pattern ==
sequence[pos : (pos + length)]) or reverse_pattern == sequence[pos : (pos + length)]):
                                        # rule is found
                                        individual.usedRules[rule.pattern] = True
                                        # print("pos is at " + str(pos) + " sequence is: " +
sequence + " found: " + sequence[pos : (pos + length)] + " weight: " + str(rule.weight))
                                        if self.mode == 0:
                                                measuredFitness += rule.weight
                                        elif self.mode == 1:
                                                measuredFitness *= rule.weight
                                        pos = pos + length
                                        found = True
                                        break
                        if not found:
                                pos += 1
                        if found and length == 0:
                                pos += 1
                        # print("pos: "+str(pos)+" len: "+str(len(sequence)))
                error = (measuredFitness - actualFitness)**2
                #ToDo:: Checkout limiting measuredFitness to possitive values

                fitness += error
                # print ("actualFitness: " + str(actualFitness) +" measuredFitness:
"+str(measuredFitness)+" error: "+str(error)+" accuracy: "+ str(accuracy))
                # print("Training:: {}- Actual: {}, Estimate: {} Squared Error: {}".format(i,
actualFitness, measuredFitness, error))
        MSE = fitness / int(self.learn.size - int(self.learn.size/10))
        RMSE = math.sqrt(MSE)
        # print("ENDDDD:: MSE: {}, RMSE: {}".format(MSE, RMSE))
        return RMSE
```

```
        def validate(self, individual):
                raise("update this function according to measureTotal")
                if self.k == 10:
                        self.k = 0
                fitness = 0.0
                sum = 0.0
                for i, row in self.learn.iterrows():
                        if not (i >= int(self.learn.size/10) * self.k and i <
int(self.learn.size/10)*(self.k+1)):
                                continue
                        sequence = row[0]
                        actualFitness = row[1]
                        pos = 0
                        measuredFitness = 0.0 # Fitness of individual i
                        if self.mode == 1:
                                measuredFitness = 1.0

                        # Iterate through the sequence
                        while pos < len(sequence):
                                # Check every rule
                                found = False
                                for rule in individual.rules:
                                        length = len(rule.pattern)
                                        reverse_pattern = rule.pattern[::-1]
                                        # Normal or Reverse
                                        if (pos + length < len(sequence) and (rule.pattern ==
sequence[pos : (pos + length)]) or reverse_pattern == sequence[pos : (pos + length)]):
                                                # rule is found
                                                individual.usedRules[rule.pattern] = True
                                                # print("pos is at " + str(pos) + " sequence is: " +
sequence + " found: " + sequence[pos : (pos + length)] + " weight: " + str(rule.weight))
                                                if self.mode == 0:
                                                        measuredFitness += rule.weight
                                                elif self.mode == 1:
                                                        measuredFitness *= rule.weight
                                                pos = pos + length
                                                found = True
                                                break
                                if not found:
                                        pos += 1
                                if found and length == 0:
                                        pos += 1
                                # print("pos: "+str(pos)+" len: "+str(len(sequence)))
                        error = (measuredFitness - actualFitness)**2
```

```
                        fitness += error
                        # print ("actualFitness: " + str(actualFitness) +" measuredFitness:
"+str(measuredFitness)+" error: "+str(error)+" accuracy: "+ str(accuracy))
                        # print("Validation:: {}- Actual: {}, Estimate: {} Squared Error: {}".format(i,
actualFitness, measuredFitness, error))
                MSE = fitness / int(self.learn.size / 10)
                # print(self.learn.size / 10)
                RMSE = math.sqrt(MSE)
                # print("ENDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD:: MSE: {}, RMSE:
{}".format(MSE, RMSE))
                return RMSE

        def measure_unseen(self, individual):
                raise("update this function according to measureTotal")
                if self.unseen.size <= 0:
                        return 0
                fitness = 0.0
                sum = 0.0
                for i, row in self.unseen.iterrows():
                        sequence = row[0]
                        actualFitness = row[1]
                        pos = 0
                        measuredFitness = 0.0 # Fitness of individual i
                        if self.mode == 1:
                                measuredFitness = 1.0

                        # Iterate through the sequence
                        while pos < len(sequence):
                                # Check every rule
                                found = False
                                for rule in individual.rules:
                                        length = len(rule.pattern)
                                        # Not tested #ToDo::
                                        reverse_pattern = rule.pattern[::-1]
                                        if (pos + length < len(sequence) and (rule.pattern ==
sequence[pos : (pos + length)]) or reverse_pattern == sequence[pos : (pos + length)]):
                                                # print("pos is at " + str(pos) + " sequence is: " +
sequence + " found: " + sequence[pos : (pos + length)] + " weight: " + str(rule.weight))
                                                if self.mode == 0:
                                                        measuredFitness += rule.weight
                                                elif self.mode == 1:
                                                        measuredFitness *= rule.weight
                                                pos = pos + length
                                                found = True
```

```
                                        break
                                if not found:
                                        pos += 1
                                if found and length == 0:
                                        pos += 1
                        error = (measuredFitness - actualFitness)**2
                        fitness += error
                        # print ("actualFitness: " + str(actualFitness) +" measuredFitness:
"+str(measuredFitness)+" error: "+str(error)+" accuracy: "+ str(accuracy))
                MSE = fitness / self.learn.size
                RMSE = math.sqrt(MSE)
                return RMSE

        def predict(self, sequence, individual):
                fitness = 0.0
                pos = 0

                # Iterate through the sequence
                while pos < len(sequence):
                        # Check every rule
                        found = False
                        for i, rule in enumerate(individual.rules):
                                if rule.status == 0:
                                        continue
                                try:
                                        length = len(rule.pattern)
                                except:
                                        continue
                                reverse_pattern = rule.pattern[::-1]
                                if (pos + length < len(sequence) and (rule.pattern == sequence[pos
: (pos + length)]) or reverse_pattern == sequence[pos : (pos + length)]):
                                        # print("pos is at " + str(pos) + " sequence is: " + sequence
+ " found: " + sequence[pos : (pos + length)] + " weight: " + str(rule.weight))
                                        if self.mode == 0:
                                                fitness += rule.weight
                                        elif self.mode == 1:
                                                fitness *= rule.weight
                                        pos = pos + length
                                        found = True
                                        break
                        if not found:
                                pos += 1
                        if found and length == 0:
                                pos += 1
```

```
                return fitness
```

Rule

```
# Authors: Iliya "eLeMeNOhPi" Alavy - Department of Engineering - Michigan State University
#                  Alexander Bricco - Department of Bioengineering -  Michigan State University

class Rule:
        def __init__(self, pattern, weight, status):
                self.pattern = pattern
                self.weight = weight
                self.status = status
```

Predictor

```
# Authors: Iliya "eLeMeNOhPi" Alavy - Department of Engineering - Michigan State University
#                  Alexander Bricco - Department of Bioengineering -  Michigan State University
import settings
import fitness as F
import random as R
import individual as I

class Sequence:
        def __init__(self, pattern, fitness):
                self.pattern = pattern
                self.fitness = fitness
                pass

# all I know is that iter is a lie... #ToDo::
class Predictor:

        def __init__(self, count, size, iter):
                self.count = count
                self.size = size
                self.iter = iter
                self.pop = []
                self.output = []
                self.popSize = 100
                self.codes = settings.TT['key']
                self.hydroDic = {
                        "I":-0.31,
                        "L":-0.56,
                        "F":-1.13,
                        "V":0.07,
                        "M":-0.23,
```

```
                "P":0.45,
                "W":-1.85,
                "J":0.17,
                "T":0.14,
                "E":2.02,
                "Q":0.58,
                "C":-0.24,
                "Y":-0.94,
                "A":0.17,
                "S":0.13,
                "N":0.42,
                "D":1.23,
                "R":0.81,
                "G":0.01,
                "H":0.96,
                "K":0.99
        }
        pass

def predict(self):
        self.populate()
        self.sort()
        for i in range(self.count):
                print(self.pop[i].pattern+" -> "+str(self.pop[i].fitness))
        pass

def populate(self):
        print ("Populating the sequence poolP...\n")
        # Read the best model and turn it to an individual
        individual = I.Individual()
        individual.makeFromFile(settings.prediction_model)

        objF = F.Fitness()
        for i in range(self.popSize):
                pattern = ""
                if self.size <= 0:
                        self.size = R.randint(8, 12)
                for j in range(self.size - 1):
                        rand = R.randint(0, len(self.codes) - 1)
                        pattern += self.codes[rand]
                if self.hydrophobic(pattern):
                        tempF = 0
                else:
                        tempF = objF.predict(pattern, individual)
```

```
                    sequence = Sequence(pattern, tempF)
                    self.pop.append(sequence)

        def hydrophobic(self, pattern):
                temp = 0
                for char in pattern:
                        temp += self.hydroDic[char]
                if temp >= 0:
                        return False
                else:
                        return True

        def sort(self):
                print ("Sorting the sequences...\n")
                n = len(self.pop)
                # Traverse through all array elements
                for i in range(n):
                        # Last i elements are already in place
                        for j in range(0, n-i-1):
                                # traverse the array from 0 to n-i-1
                                # Swap if the element found is greater
                                # than the next element
                                if self.pop[j].fitness < self.pop[j+1].fitness:
                                        self.pop[j], self.pop[j+1] = self.pop[j+1], self.pop[j]

# Restrict the amino acid search
# empirical research
```