DEVELOPING AN EFFICIENT PRECISION EDITING METHODOLOGY FOR *LEISHMANIA MEXICANA* TO INVESTIGATE THE KINETOCHORE COMPLEX.

Charlotte Hughes

Master of Science

University of York

Biology

September 2024

1 PREFACE

1.1 Abstract

Leishmania mexicana is a parasitic protozoan, and one of the causative agents of cutaneous leishmaniasis – a skin infection causing large lesions. *Leishmania spp.* have some unusual biological features, due to their early evolutionary split from other eukaryotes. One is their unique kinetochore complex – the protein complex responsible for binding the chromosomes to microtubules during mitosis. To evaluate the role of specific phosphorylation sites on essential kinetochore proteins, a selection-free precision editing strategy using the CRISPR-Cas9 system was investigated in promastigotes. Genomic DNA was targeted with 120 nt single-stranded oligonucleotide repair DNA to generate 10 unique amino acid substitutions to create phosphosite mutants from kinetochore proteins KKT1, KKT2, KKT4 and KKT7 but was only successful in 2.0% of clones. Comparatively, using 160 bp double-stranded repair DNA targeting 6 phosphosites between KKT2, KKT4 and KKT7 generated phosphodeficient, phosphomimetic and synonymous mutants at each target site tested. Across 18 unique transfections, PCR screening detected integration of the repair template in 24.6% of clones screened. Surprisingly, following Sanger sequencing, it was found that 29.2% of clones screened were in fact edited. Mutant clones were predominantly homozygous (21.7% of clones), including at least one clone per transfection. Kinetochore phosphosite mutant clones were assessed for growth changes and cell cycle dysregulation, but no apparent phenotypes were detected. Lastly, to pave the way for higher-throughput precision editing using this method, a Python script was developed to replicate the design process used to create the 160 bp repair templates. The script uses a FASTA file, codon usage table and a simple Excel spreadsheet configuration file to design the desired repair template with a single nonsynonymous mutation, and additional synonymous mutations for screening purposes. It also generates a corresponding synonymous-mutation only repair template, as well as screening primers and primers to produce the repair templates for a ready-to-go approach.

1.2 Table of Contents

1.3 List of Tables

1.4 List of Figures

1.5 List of Accompanying Material

Filename	Description
Hughes_202006829_repair_template_input_excel.xlsx	Template Excel spreadsheet configuration file for instructing the Python script.
Hughes_202006829_main.py	Python file which when executed reads the repair_template_input_excel.xlsx file and designs repair templates and primers.
Hughes_202006829_reading_input_file.py	A Python file which is needed for main.py to execute.
Hughes_202006829_codon_dictionaries.py	A Python file which is needed for main.py to execute.
Hughes_202006829_codon_dataframes.py	A Python file which is needed for main.py to execute.

Hughes_202006829_formatting_functions.py	A Python file which is needed for main.py to execute.
Hughes_202006829_stitching_functions.py	A Python file which is needed for main.py to execute.
Hughes_202006829_validator.py	A Python file which is needed for main.py to execute.
Hughes_202006829_primer_functions.py	A Python file which is needed for main.py to execute.
Hughes_202006829_repair_template_input_excel_batch.xlsx	Template Excel spreadsheet configuration file for instructing the batch version of the Python script.
Hughes_202006829_main_batch.py	A modified version of main.py Python file which when executed, reads repair_template_input_excel_batch.xlsx and generates several sets of repair templates and primers.
Hughes_202006829_repair_template_input_excel_multi_mutant.xlsx	Template Excel spreadsheet configuration file for instructing the multi-mutant version of the Python script.
Hughes_202006829_main_multi_mutant.py	A modified version of main.py Python file which when executed, reads repair_template_input_excel_multi_mutant.xlsx to generate a pair of repair templates with up to 5 nonsynonymous target mutations.

Please note that due to the naming requirements for submission, these files will not run the Python script as intended unless the prefix (Hughes_202006829_) is removed from all Python (.py) and Excel (.xlsx) file names before use. Additionally, all files must be saved in the same folder/directory to run as intended.

1.6 Acknowledgements

Thank you to Juliana Carnielli and Vincent Geoghegan for their support with designing repair templates, choosing targets and general advice on this project. Thank you to Nicola Baker for the idea to create a computer programme to scale up this approach. Thank you to Sandy Macdonald for providing the Python training and Robert Callender for additional Python help. Thank you to all the colleagues at York for their general help and advice, especially those who provided genes to test the Python script on. Thank you to my supervisors Jeremy Mottram and Joana Correia Faria, and TAP panel member Pegine Walrad for their support and enthusiasm throughout the entire project. A huge thank you to the University of York for signing the Technician commitment to continue to upskill technicians like me, and to the department of biology for supporting me to do so.

1.7 Author's Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for a degree or other qualification at this University or elsewhere. All sources are acknowledged as references.

2 CHAPTER ONE - INTRODUCTION

2.1 Leishmaniasis Disease and the Leishmania Lifecycle

Leishmania mexicana is a protozoan parasite and one of the causative agents for cutaneous leishmaniasis (CL) (Burza, Croft and Boelaert, 2018). CL causes lesions on the skin, which in some instances can be self-healing, but many often leave disfiguring scars. *Leishmania spp.* are transmitted by female sandflies, which bite mammals and feed on their blood. When a sandfly feeds on infected blood, it can propagate the infection to the next animal it feeds on, spreading the infection (Burza, Croft and Boelaert, 2018). As such, *Leishmania spp.* require a complex digenetic lifecycle to survive inside the different hosts, experiencing different temperatures, pHs and nutrient availabilities.

Whilst carried by the sandfly, the parasites differentiate into the replicative promastigote form, as they live inside the midgut where nutrient availability is good. Once the number of cells has expanded, the promastigotes prepare for reinfection of a mammalian host, and differentiate into metacyclic promastigotes. In doing so, they move up into the stomodeal valve of the sandfly and block it with a gel plug (Sacks, 1989; Rogers, Chance and Bates, 2002). The gel plug aids the parasite by affecting the way the sandfly feeds to increase its chances of infecting the next host. During feeding, the metacyclic promastigotes are injected into the host, where they interact with macrophages and are phagocytosed. For many infectious organisms, this would mean death, but *Leishmania spp.* have evolved to live in this environment. Once inside the macrophage, the parasite differentiates into the amastigote form, which are adapted to the acidic pH of the phagosome and the reduced nutrient availability (McConville and Naderer, 2011). They also change morphology, taking on a more round cell body shape, with a reduced flagellum that does not protrude from the cell body (Burza, Croft and Boelaert, 2018). However, amastigotes do not just survive inside the phagosome, they are able to replicate there, leading to the macrophage bursting, releasing the amastigotes. From there, amastigotes can re-infect macrophages, until either a sandfly takes up the blood meal from this host or the host clears the infection. If a sandfly takes up the infected blood meal, the amastigotes differentiate back into promastigotes, starting the cycle again.

There is a need to investigate the biology of *Leishmania* species, as there are around 1 million new cases of leishmaniasis worldwide each year (Burza, Croft and Boelaert, 2018). Current treatments are largely chemotherapeutic, toxic, and can often be ineffective (Madusanka, Silva and Karunaweera, 2022). Understanding the unique biology of these parasites can help to find new drug targets and treatment approaches to reduce the burden of this disease. Additionally, since *L. mexicana* causes one of the least severe forms of leishmaniasis and grows well *in vitro*, *L. mexicana* has become a model for understanding the biology of *Leishmania spp.*

2.2 Gene Editing and CRIPSR-Cas9

One way to investigate the biology of *Leishmania spp.* is through gene editing. Gene editing can be used to elucidate the function of specific genes and the proteins encoded by them. Gene editing in *Leishmania* began using homologous recombination-based strategies from donor DNA containing large homologous sequences (Cruz and Beverley, 1990). Whilst this strategy was generally effective, the discovery of the bacterial CRISPR-Cas9 system has allowed gene editing to become quicker, easier and more efficient than before.

In the CRISPR-Cas9 system, Cas9 is an endonuclease that can make double-stranded DNA (dsDNA) breaks at a specific sequence of DNA, as directed by a single-guide RNA (sgRNA) (Gasiunas et al., 2012). In bacteria, CRISPR (Clustered, regularly interspaced, short palindromic repeats) are a stored library of reference sequences from viral invaders. When a reference sequence is transcribed into an sgRNA, it directs the Cas9 endonuclease to cleave the sequence, hence removing viral sequences present in its genome and avoiding damage from viral replication. The cleavage of the DNA from Cas9 always takes place 3 nucleotides away from a short motif called the Protospacer Adjacent Motif (PAM), which, in the most commonly used *S. thermophilus* Cas9, is an NGG motif (Gasiunas et al., 2012). By providing an sgRNA made *in vitro*, it is possible to "hijack" the Cas9 endonuclease activity to make a dsDNA break in any known DNA sequence ending with an NGG. Whilst Cas9 has very high specificity, some mismatches within the sequence can enable the dsDNA break to proceed, which can lead to off-target breaks elsewhere in the genome (Hsu et al., 2013). That being said, the precision and specificity of the CRISPR-Cas9 system is so versatile that it is indispensable in modern molecular biological research.

The CRISPR-Cas9 system was first tested in *Leishmania* in 2015 by Sollelis et al. This first approach used episomal expression of Cas9, with transfection of linearised plasmid containing sgRNA under a U6 RNA polymerase III promoter, and an antibiotic selection marker flanked with two ~1 kb homology regions to replace the target locus. This was able to successfully generate null mutants in a single round of transfection, which prior to Cas9 would have required two rounds to remove both alleles (Sollelis et al., 2015). Whilst this approach was effective, generation of repair templates with such large homology regions is cumbersome. To attempt to tackle this issue, Beneke et al. (2017) investigated whether smaller homology regions would remain effective. They showed that homology regions of just 30 bp were equally as efficient when Cas9-directed breaks were made. Additionally, Beneke et al. (2017) used Cas9 integrated into the genome, and a T7 RNA polymerase (T7 RNAP). Stable integration of Cas9 helped reduce some of the concerns over variable Cas9 expression that Sollelis et al. (2015) experienced with episomal expression. Changing the sgRNA promoter to a T7 promoter also simplified the process. Using a T7 RNAP allowed transfection of DNA constructs containing a T7 promoter, which are then transcribed into the sgRNA *in cellulo*. In this cell line, the T7 RNAP gene was also integrated in the genome. From this, Beneke et al. developed a toolkit to make deleting and tagging genes easier than before (Beneke et al., 2017). This toolkit provides template sequences for either gene deletion or gene tagging which are contained on plasmids, and can be amplified with primers containing a 20 nt annealing sequence and 30 nt homology arm sequences. These repair templates contain an antibiotic resistance gene, to act as a positive selection marker for cells that have been edited, and to remove untransfected cells from the population. This method was demonstrated through the knock-out of flagellum genes in *L. mexicana, L. major and T. brucei* (Beneke et al., 2017). Because of the simplicity and versatility of such an approach, large-scale projects have used this system to generate mutants at scale. One example of this was the deletion and tagging of every kinase in the genome by Baker et al. (2021). However, this toolkit is limited to mutations at a whole gene scale due to the inclusion of the antibiotic resistance marker, which can only be incorporated at either end of, or in place of a gene. In order to generate mutations at a sub-gene scale (e.g. single nucleotide changes), an alternate approach is required: precision editing.

Presently, to make precision mutations that target a single amino acid of a protein, constructs have to be created for each gene - cloning the gene of interest into a plasmid and then editing it *in vitro*, before replacing the endogenous gene with the mutant version (Figure 1A). This method can be effective and has been used before, such as by Nerusheva and Akiyoshi in 2016 to generate mutants of interest in trypanosomes to investigate how KKT2 localises to the kinetochore. Similarly, Saldivia et al. (2020) generated a mutant version of CLK1/KKT10 on a plasmid, which was then inserted into the tubulin locus. Subsequently, RNA interference (RNAi) was used to prevent expression of the WT allele (Figure 1C). However, this process of cloning, editing and reintegrating is time consuming and laborious for what may be only a handful of nucleotide changes on one gene. Hence, it is not scalable to evaluate larger numbers of sites of interest, such as in a library screen. It also still relies on incorporation of a positive selection marker, which is a relatively large-scale change, which may not be suitable for all applications.

Smaller constructs have been used as repair templates for CRISPR-directed mutants in a range of kinetoplastids, typically in the form of oligonucleotide repair templates (Figure 1B). Zhang and Matlashewski (2015) used single-stranded oligonucleotide repair templates with 25 nt homology arms to modify the miltefosine transporter gene, to incorporate premature stop codons into *L. donovani* (Pal and Dam, 2022). Rico et al. (2018) also used oligonucleotide repair templates with 50 nt homology arms to modify the aquaglyceroporin gene in *T. brucei*. Also in *T. brucei*, 68 nt oligonucleotides have been used as repair templates to generate enzymatic mutations to the CPSF3 gene, mutating only 8 nucleotides in total (Wall et al., 2018). Medeiros et al. (2017) used oligonucleotides as repair templates to introduce premature stop codons in fluorescent reporter genes in *T. cruzi*, using recombinantly produced Cas9 ribonucleoprotein complexes rather than endogenous Cas9 expression (Lander and Chiurillo, 2019). Interestingly, small oligonucleotide-derived repair templates have also been effective at generating precision edited mutants without the use of CRISPR-Cas9 system to generate drug resistant cell lines in *T. brucei* (Altmann et al., 2022). Outside of kinetoplastids but within the realm of parasitology, similar protocols have been used to modify *Plasmodium falciparum* using 200 nt oligonucleotide repair templates (Crawford et al., 2017) and either 50 bp and 125 bp double-stranded oligonucleotide or 125 bp PCR generated repair templates in *Trichomonas vaginalis* (Janssen et al., 2018).

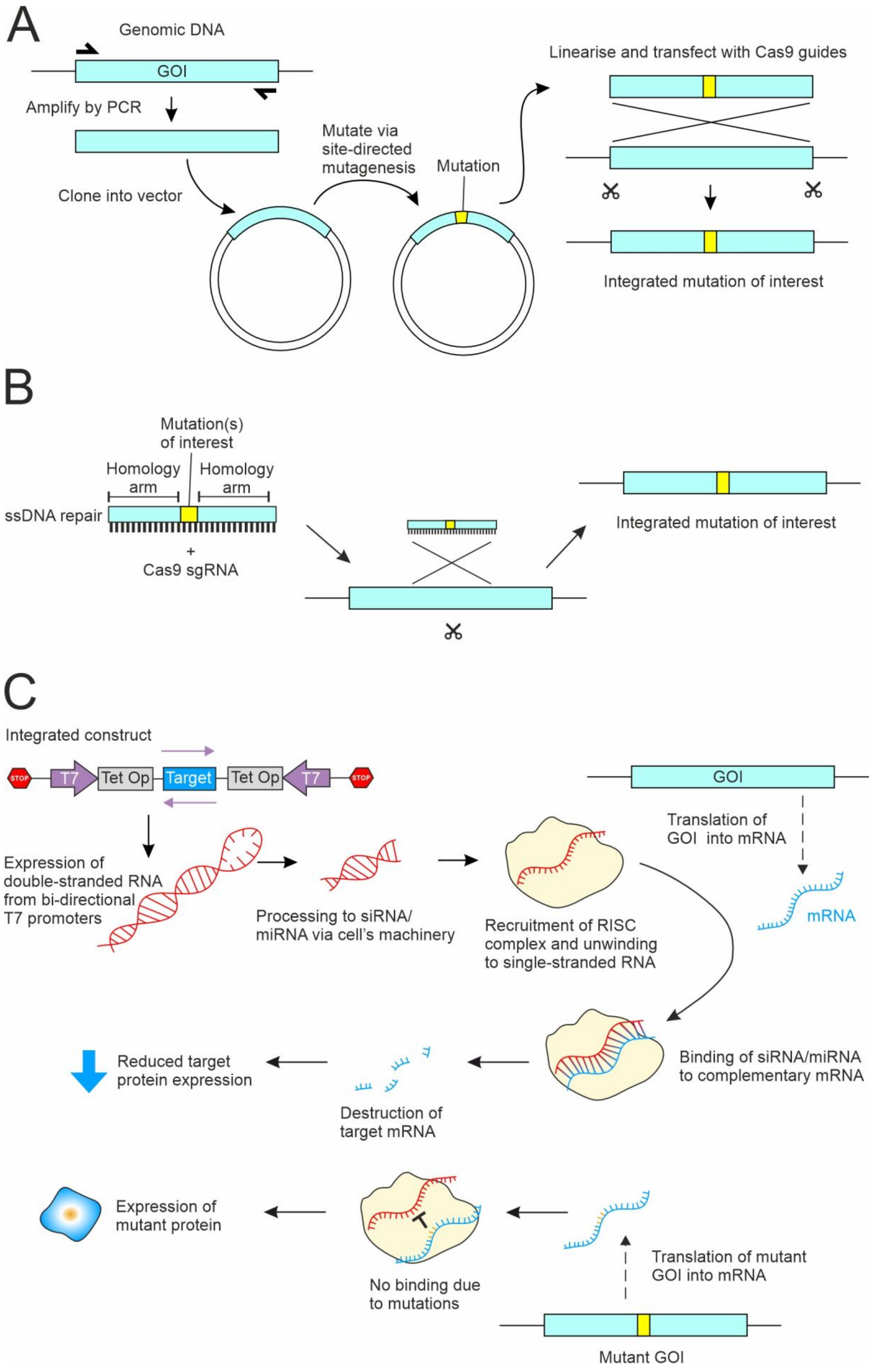

A
Genomic DNA
GOI
Amplify by PCR
Clone into vector
Mutate via
site-directed
mutagenesis
Mutation
Linearise and transfect with Cas9 guides
Integrated mutation of interest
B
Mutation(s)
of interest
Homology
arm
Homology
arm
ssDNA repair
+
Cas9 sgRNA
Integrated mutation of interest
C
Integrated construct
STOP
T7
Tet Op
Target
Tet Op
T7
STOP
GOI
Translation of
GOI into mRNA
mRNA
Expression of
double-stranded RNA
from bi-directional
T7 promoters
Processing to siRNA/
miRNA via cell's machinery
Recruitment of RISC
complex and unwinding
to single-stranded RNA
Binding of siRNA/miRNA
to complementary mRNA
Reduced target
protein expression
Destruction of
target mRNA
Expression of
mutant protein
No binding due
to mutations
Translation of mutant
GOI into mRNA
Mutant GOI

Figure 1. Current site-directed mutagenesis technologies for kinetoplastids. A) Mutagenesis through whole gene replacement of a gene of interest (GOI). The GOI is amplified from genomic DNA and cloned into a vector. *In vitro* mutagenesis approaches are used to generate the mutation of interest in the vector. The mutated vector is linearised (either by PCR amplification or restriction digest) to retrieve the mutated gene, and is transfected into a CRISPR-Cas9 cell line with sgRNAs targeting either end of the gene of interest. Following the double-stranded DNA (dsDNA) break by Cas9, homology-directed repair (HDR) leads to integration of the repair template. In some instances, a positive-selection marker may also be included in the repair template to select for mutant cells. B) Oligonucleotide/single-stranded DNA (ssDNA) precision editing approaches used by other groups in the literature (Zhang and Matlashewski, 2015; Medeiros et al., 2017; Rico et al., 2018; Wall et al., 2018; Pal and Dam, 2022). A repair template is designed containing homology arms and the mutation(s) of interest. This is synthesized as an oligonucleotide and transfected into CRISPR-Cas9 competent cells with one or two sgRNAs targeting the region adjacent to the mutation of interest. The dsDNA break leads to HDR and integration of the repair template, though efficiency of this approach is typically low due to no selection for transfected cells. C) RNA-interference (RNAi) for expression of mutant GOI (not possible in most *Leishmania* species, but present in *Trypanosoma brucei*). A cell line is generated containing the construct indicated on the left to enable expression of a double-stranded RNA (dsRNA) corresponding to a GOI. This construct uses bi-directional T7 promoters to generate a self-complementary RNA sequence and is controlled by a tetracycline (Tet) inducible operon to enable controlled induction of expression. This dsRNA is generated from a region of sequence corresponding to the mRNA of the GOI. The cell's internal machinery processes the dsRNA to small-interfering or micro-RNA (siRNA or miRNA). This leads to recruitment of the RISC complex which enables recognition of mRNA from the GOI and eventually degradation of this mRNA, generating a knock-down effect on gene expression. If a mutant GOI is present in the cell line that has altered sequence sufficient to prevent binding of the WT-specific siRNA/miRNA, then expression of the mutant copy can take place whilst the WT GOI is knocked-down. This approach can allow for mutant gene expression of essential genes without requiring removal of the WT copy of the GOI.

One thing that remains apparent is that there is large variation in the design of small, often oligonucleotide-derived, repair templates between different groups. Whilst adjustments may be necessary between species, given that the protocols used as part of the CRISPR-Cas9 editing toolkit are transferable between the TriTryps (*Leishmania, T. brucei* and *T. cruzi*) (Beneke et al., 2017), it stands to reason that the same repair template designs will work similarly across these species. This would suggest that development of an optimised method for one of these species could have benefits for them all.

2.3 The Kinetochore Complex

2.3.1 VERTEBRATES AND HIGHER EUKARYOTES

The kinetochore is a protein complex responsible for connecting microtubules to DNA during mitotic (and meiotic) division (Musacchio and Desai, 2017). This complex is composed of two major parts – the inner and outer kinetochore. The inner kinetochore is the direct linker to the genomic DNA, whilst the outer kinetochore links the inner kinetochore to the microtubule spindle.

In vertebrates, the inner kinetochore contains an unusual centromere-specific histone H3 variant CENP-A (CENtromere Protein A) and 16 other proteins which make up the constitutive centromere associated network (CCAN) – see Figure 2A for schematic (Sridhar and Fukagawa, 2022). Whilst CENP-A is not present in all species, in those that retain it, it forms the basis for the kinetochore, replacing typical histone H3 at the centromere. CENP-A then recruits CENP-C and CENP-N (both part of the CCAN) to form a centromeric nucleosome (Sridhar and Fukagawa, 2022). The other members of the CCAN interact with each other in discrete sub-complexes, and help tightly trap the DNA in order to transmit the forces of the mitotic spindle.

In species lacking CENP-A, CENP-T provides an alternate DNA-binding starting point for the kinetochore. Interestingly, CENP-T is also present in many species containing CENP-A genes, providing an alternate, often favoured, method of DNA attachment (Sridhar and Fukagawa, 2022). CENP-T proteins are less well conserved than CENP-A proteins between eukaryotes. When CENP-T binds DNA, it forms its own complex, made up of sub-complexes typically containing CENP-T and -W, and CENP-S and –X (Sridhar and Fukagawa, 2022).

Similarly to CENP-A, they form a nucleosome-like structure on the chromatin, but preferentially bind to linker DNA rather than nucleosome-bound DNA. When CENP-A is also present, the CENP-T complex forms between two CENP-A nucleosomes, as is the case in humans. CENP-T also interacts with the outer kinetochore, via its long unstructured N-terminal region, and is regulated by phosphorylation by CDK1.

The inner kinetochore is linked to the mitotic spindle via the outer kinetochore. The outer kinetochore is composed of a 10-member protein network called the KMN network. The KMN network is subsequently composed of several sub-complexes which give it its name – Knl1C, Mis12C and Ndc80C (Sridhar and Fukagawa, 2022). The Ndc80C complex forms the primary microtubule binding site, and is helped to localise to the kinetochore through members of the Mis12C complex, which can be disrupted by phosphorylation from Aurora B kinase. As well as interacting with the Ndc80C complex, the Mis12C complex also facilitates interactions with the Knl1C complex. Knl1C complex in turn facilitates further protein-protein interactions, which allows it to make contact with proteins involved in regulation of the kinetochore, error correction, and activation and silencing of the Spindle Assembly Checkpoint (SAC) (Sridhar and Fukagawa, 2022). The outer kinetochore also has several other accessory proteins which form other complexes, namely the Dam1 complex and the Ska complex, which are found variably across species.

As already alluded to, the full complement of these proteins are not present in all eukaryotic species. Overall, the inner kinetochore has shown a wider diversity in components than the outer kinetochore (Sridhar and Fukagawa, 2022), although examples exist of systems with a wide range of absent inner and outer kinetochore components.

2.3.2 KINETOPLASTIDS

Many of the components of the kinetochore are conserved across numerous eukaryotic species, but kinetoplastids are an unusual exception, in that their kinetochore proteins lack homology to almost all of the canonical components (Akiyoshi and Gull, 2014). To date, 25 unique proteins have been identified in the trypanosomatid inner kinetochore (Akiyoshi and Gull, 2014; Nerusheva and Akiyoshi, 2016; Nerusheva, Ludzia and Akiyoshi, 2019;

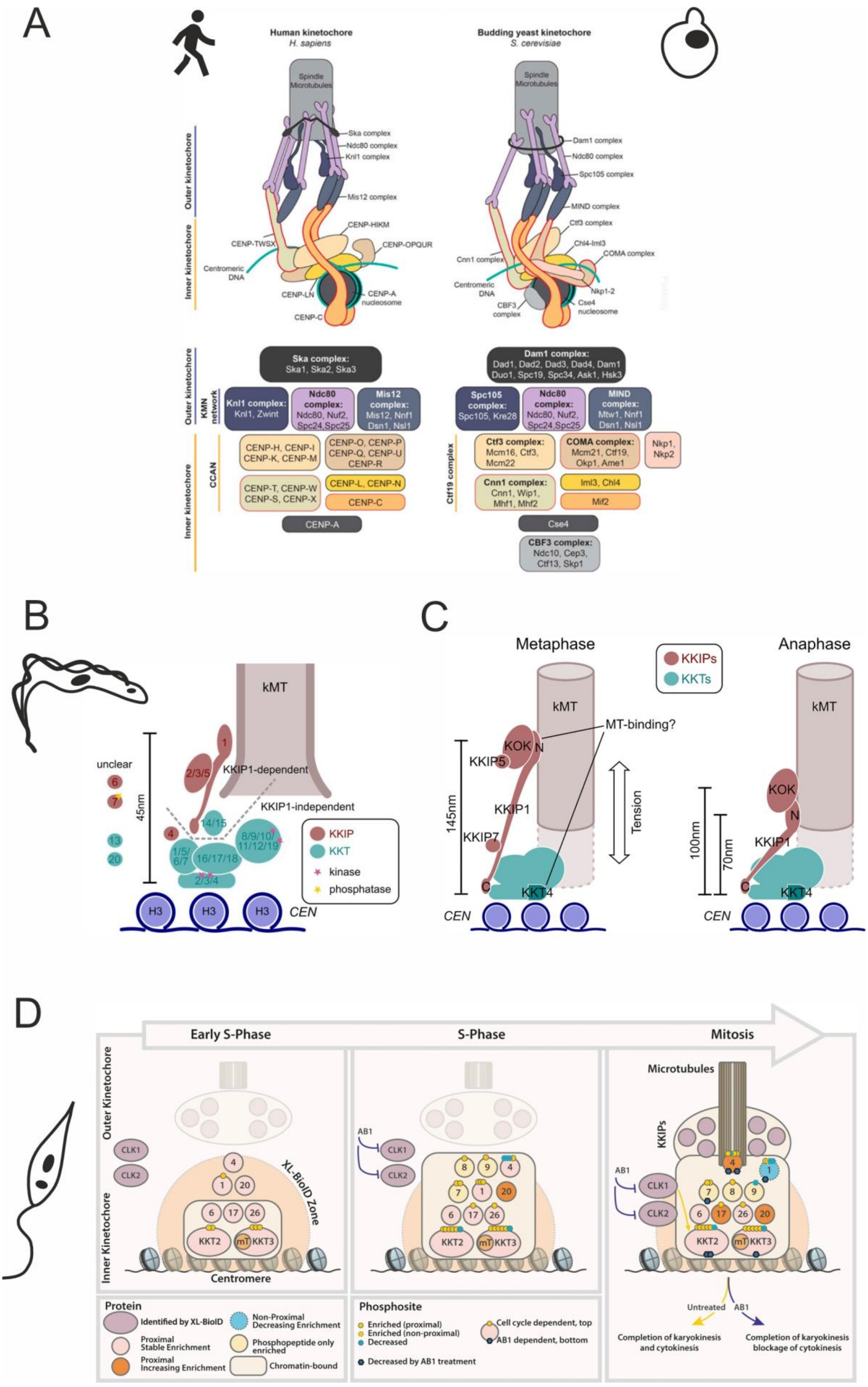
A
Human kinetochore
H. sapiens
Budding yeast kinetochore
S. cerevisiae
Spindle Microtubules
Ska complex
Ndc80 complex
Knl1 complex
Mis12 complex
CENP-HIKM
CENP-TWSX
CENP-OPQUR
Centromeric DNA
CENP-LN
CENP-A nucleosome
CENP-C
Dam1 complex
Spc105 complex
MIND complex
Ctf3 complex
Chl4-Iml3
COMA complex
Cnn1 complex
Nkp1-2
CBF3 complex
Cse4 nucleosome
Outer kinetochore
Inner kinetochore
KMN network
CCAN
Ctf19 complex
Ska complex: Ska1, Ska2, Ska3
Knl1 complex: Knl1, Zwint
Ndc80 complex: Ndc80, Nuf2, Spc24,Spc25
Mis12 complex: Mis12, Nnf1 Dsn1, Nsl1
CENP-H, CENP-I CENP-K, CENP-M
CENP-O, CENP-P CENP-Q, CENP-U CENP-R
CENP-T, CENP-W CENP-S, CENP-X
CENP-L, CENP-N
CENP-C
CENP-A
Dam1 complex: Dad1, Dad2, Dad3, Dad4, Dam1 Duo1, Spc19, Spc34, Ask1, Hsk3
Spc105 complex: Spc105, Kre28
Ndc80 complex: Ndc80, Nuf2, Spc24,Spc25
MIND complex: Mtw1, Nnf1 Dsn1, Nsl1
Ctf3 complex: Mcm16, Ctf3, Mcm22
COMA complex: Mcm21, Ctf19, Okp1, Ame1
Nkp1, Nkp2
Cnn1 complex: Cnn1, Wip1, Mhf1, Mhf2
Iml3, Chl4
Mif2
Cse4
CBF3 complex: Ndc10, Cep3, Ctf13, Skp1
B
kMT
unclear
6
7
13
20
45nm
1
2/3/5
KKIP1-dependent
KKIP1-independent
14/15
4
1/5/6/7
16/17/18
8/9/10/11/12/19
2/3/4
KKIP
KKT
kinase
phosphatase
H3
H3
H3
CEN
C
Metaphase
Anaphase
KKIPs
KKTs
kMT
MT-binding?
KOK
N
KKIP5
KKIP1
KKIP7
C
KKT4
145nm
Tension
CEN
kMT
KOK
N
KKIP1
C
KKT4
100nm
70nm
CEN
D
Early S-Phase
S-Phase
Mitosis
Outer Kinetochore
Inner Kinetochore
CLK1
CLK2
XL-BioID Zone
4
1
20
6
17
26
KKT2
mT
KKT3
Centromere
AB1
CLK1
CLK2
8
9
4
7
1
20
6
17
26
KKT2
mT
KKT3
Microtubules
KKIPs
AB1
CLK1
CLK2
4
1
7
8
9
6
17
26
20
KKT2
mT
KKT3
Untreated
AB1
Completion of karyokinesis and cytokinesis
Completion of karyokinesis blockage of cytokinesis
Protein
Identified by XL-BioID
Non-Proximal Decreasing Enrichment
Proximal Stable Enrichment
Phosphopeptide only enriched
Proximal Increasing Enrichment
Chromatin-bound
Phosphosite
Enriched (proximal)
Enriched (non-proximal)
Decreased
Cell cycle dependent, top
AB1 dependent, bottom
Decreased by AB1 treatment

Figure 2. Kinetochore complex schematic diagrams from eukaryotic organisms. A) Human (left) and budding yeast (S. cerevisiae, right) kinetochores, adapted from Sridhar and Fukagawa (2022). Homologous complexes between humans and yeast have been indicated in the same colours and kinetochore homologs have been shown in the corresponding positions. B and C) Current understanding of the *Trypanosoma brucei* kinetochore adapted from B: D'Archivio and Wickstead (2017), and C: Brusini et al. (2021). Both studies used pull downs of various kinetochore components and RNAi depletion to develop this model. kMT – kinetochore microtubule. In C, the KOK (kinetoplastid outer kinetochore) complex contains KKIP2-4, 6, 8-12. N and C indicate the positions of the respective termini of KKIP1. D) Current understanding of the *Leishmania mexicana* kinetochore adapted from Geoghegan et al. (2022). Data based on proximity of proteins and phospho-proteins relative to KKT3 (inner kinetochore).

Geoghegan et al., 2022). These proteins have been systematically named *K*inetoplastid *K*ine*t*ochore proteins (KKT) 1-26 (excluding KKT21 due to renaming). As well as these components, there are also 12 KKT-interacting proteins (KKIPs), identified in *Trypanosoma brucei*, which make up the outer kinetochore – see Figure 2B for schematic (D'Archivio and Wickstead, 2017; Brusini et al., 2021). Only KKIP1, which has been identified to be a highly divergent Ndc80/Nuf2 homologue (D'Archivio and Wickstead, 2017), and KKT14 and KKT15 which have been identified as divergent Bub1 and Bub3 proteins (Ballmer et al., 2024), have homology to canonical kinetochore components. None of the other KKT or KKIP proteins share sequence similarity nor known structural similarity, with canonical kinetochore proteins, and are not found outside kinetoplastids. However, within kinetoplastids, there is high conservation with the KKT proteins, and some conservation of KKIPs (Akiyoshi and Gull, 2014; Brusini et al., 2021). Whilst some of the functions of specific KKTs and KKIPs are beginning to be understood, many of these proteins are still of unknown function with no known protein domains.

Of the KKT proteins that have had more detailed investigation, it is understood that KKT4 has microtubule-binding properties, but is unusually found in the inner kinetochore (Llauró et al., 2018). Additionally, the inner kinetochore contains four protein kinases (KKT2, KKT3, KKT10 (CLK1) and KKT19 (CLK2)), of which KKT2 and KKT3 are known to have centromere localisation domains. KKT2 and KKT3 are thought to make up the foundation of the kinetochore by binding to the DNA using their divergent POLO box domains, allowing other kinetochore proteins to localise to them (Nerusheva and Akiyoshi, 2016; Marcianò et al., 2021; Ishii et al., 2022). KKT10/CLK1 is known to phosphorylate KKT2, but little is known about the substrates of KKT2's and KKT3's kinase domains (Saldivia et al., 2021). KKT10 and KKT19 were identified first as being cdc2-like kinases (CLKs) in *T. brucei* (Altmann et al., 2013), and subsequently as members of the kinetochore (Akiyoshi and Gull, 2014). KKT10/CLK1 has been shown to be important for kinetochore formation, causing KKT2 to improperly localise on KKT10/CLK1 inhibition, as well as regulation to kinetochore assembly (Saldivia et al., 2020, 2021). Recently, KKT14 and KKT15 have been identified as divergent Bub1 and Bub3 proteins, which are involved in the spindle checkpoint of other organisms, and are needed for accurate chromosome segregation in *T. brucei* (Ballmer et al., 2024). KKIP1 has been shown to provide a linker between the inner and outer kinetochores (Brusini et al., 2021) (Figure 2B and C). As previously mentioned KKIP1 is a highly divergent Ncd80/Nuf2 homologue, which occupies a similar niche, bridging the inner and outer kinetochore (Brusini et al., 2021). KKIP2-4, 6, and 8-12 form a stable complex that is part of the outer kinetochore in *T. brucei*, and have been found to interact with many proteins thought to be involved in RNA-processing (Nerusheva, Ludzia and Akiyoshi, 2019; Brusini et al., 2021). However, it should be noted that no homologs have been identified for KKIP3, 4, 6, 9, 11 or 12 in *L. mexicana*, so whether the outer kinetochore has the same structure in *Leishmania* is not known (Brusini et al., 2021).

Having a core role in cell division, many components of the kinetochore have been identified as essential in *Leishmania* promastigotes. Of these, KKT2 and KKT3 are essential, as well as one of either KKT10 or KKT19 being necessary for survival (but not both) (Baker et al., 2021). Additionally, KKT7, KKT9, KKT11 and KKT12 are also necessary for consistent chromosomal segregation in trypanosomes, with growth defects quickly developing following RNAi knockdown (Akiyoshi and Gull, 2014). Similar fitness defects have been seen following RNAi knockdown of all KKT proteins in trypanosomes (Horn, 2022).

Given the presence of protein kinases playing a core role in the kinetochore, the role of phosphorylation has been investigated in the *L. mexicana* kinetochore throughout the cell cycle by Geoghegan et al. (2022) – see Figure 2D for schematic. Phosphorylation was shown to be a dynamic process in the cell cycle, with a peak in phosphorylated proteins during S-phase, in many cases, independent of protein levels. In particular, they identified several peptides which changed phosphorylation state disproportionately to their protein levels throughout the cell cycle. These peptides included phosphorylation sites S493 and S530 on KKT2; and T120-S144, S300, T318-S328 and T421-T430 on KKT4. Specific phosphorylation sites were unable to be derived from all phospho-peptides due to the presence of several serine and threonine residues within some peptides. KKT7 S304 also showed a strong decrease in phosphorylation following AB1 treatment (which blocks KKT10/CLK1 and KKT19/CLK2 mediated phosphorylation). This suggested that phosphorylation plays a key role in the control of the kinetochore complex during the cell cycle. However, to date, very little is known about the effect of individual phosphorylation events on kinetochore function, and which kinases are responsible.

As the kinetochore of trypanosomatids such as *L. mexicana* is so unique, they pose interesting questions on both the unique evolutionary biology of these organisms, and their propensity to be drug targets. As such, this project aims to investigate the kinetochore complex of *L. mexicana* through precision editing with the CRISPR-Cas9 system.

This project has several aims. The first aim is to investigate the biology of the kinetochore in *L. mexicana* using mutants generated with this precision editing methodology, initially starting with phosphorylation sites identified in the kinetochore by Geoghegan et al. (2022). The second aim is to investigate improvements to the efficiency of this methodology. The final aim is to investigate ways to scale-up this precision editing through computer-aided design.

3 CHAPTER TWO - METHODOLOGIES

3.1 Cell Culture

T7Cas9 *Leishmania mexicana* promastigotes (Beneke et al., 2017) were grown in HOMEM media with 10% Fetal Bovine Serum (FBS) and 1% penicillin-streptomycin (henceforth called 10% FBS HOMEM). T7Cas9 cells were also kept under continual selection with 50 µg/ml hygromycin and 75 µg/ml nourseothricin at 25°C in non-vented TC coated flasks. Cells were passaged 1 in 1000 weekly until passage 20, when cells were replaced with lower passage cells from cryo-storage.

3.2 Single-stranded Oligonucleotide Repair Template Design

To attempt to generate mutations of interest, single-stranded DNA (ssDNA) repair templates were designed using strategies adapted from unpublished work by Juliana Carnielli and similar approaches in the literature (Zhang and Matlashewski, 2015; Medeiros et al., 2017; Rico et al., 2018; Wall et al., 2018; Pal and Dam, 2022). Genomic sequences for genes of interest were retrieved from TriTrypDB.org from the *Leishmania mexicana* MHOM/GT/2001/U1103 genome. Target site was identified, and ~60 nt either side was selected to create a region of a total of 120 nt (Figure 3). Genomic sequences for this region were used for sgRNA design on EuPaGDT (http://grna.ctegd.uga.edu/). The highest ranking two guides in as close proximity to the target site as possible were chosen – with one making a break before, and the other after the target site. The first 30 nt and final 30 nt of the 120 nt region were kept as the native sequence (homology arms). Sequences corresponding to the protospacer motifs and PAM sequences were recoded using an alternate codon with the highest frequency of usage from *Leishmania infantum* (from https://www.kazusa.or.jp/codon/) - see Appendix 7.2.4.1 for a copy of the table. The *L. infantum* dataset was used as a reference rather than *L. mexicana* because the *L. mexicana* dataset was calculated from only 93 CDS sequences. The total *L. mexicana* genome contains 9,169 genes (Fiebig, Kelly and Gluenz, 2015), and as such this data set only represents about 1% of the genome which was deemed unlikely to be representative. The *Leishmania* species with the highest coverage of the genome was *L. infantum*, covering 8,139 CDS sequences out of their total 8,241 predicted protein coding genes (Rogers et al., 2011), so this dataset was used instead. The codon sequence for the target mutation was

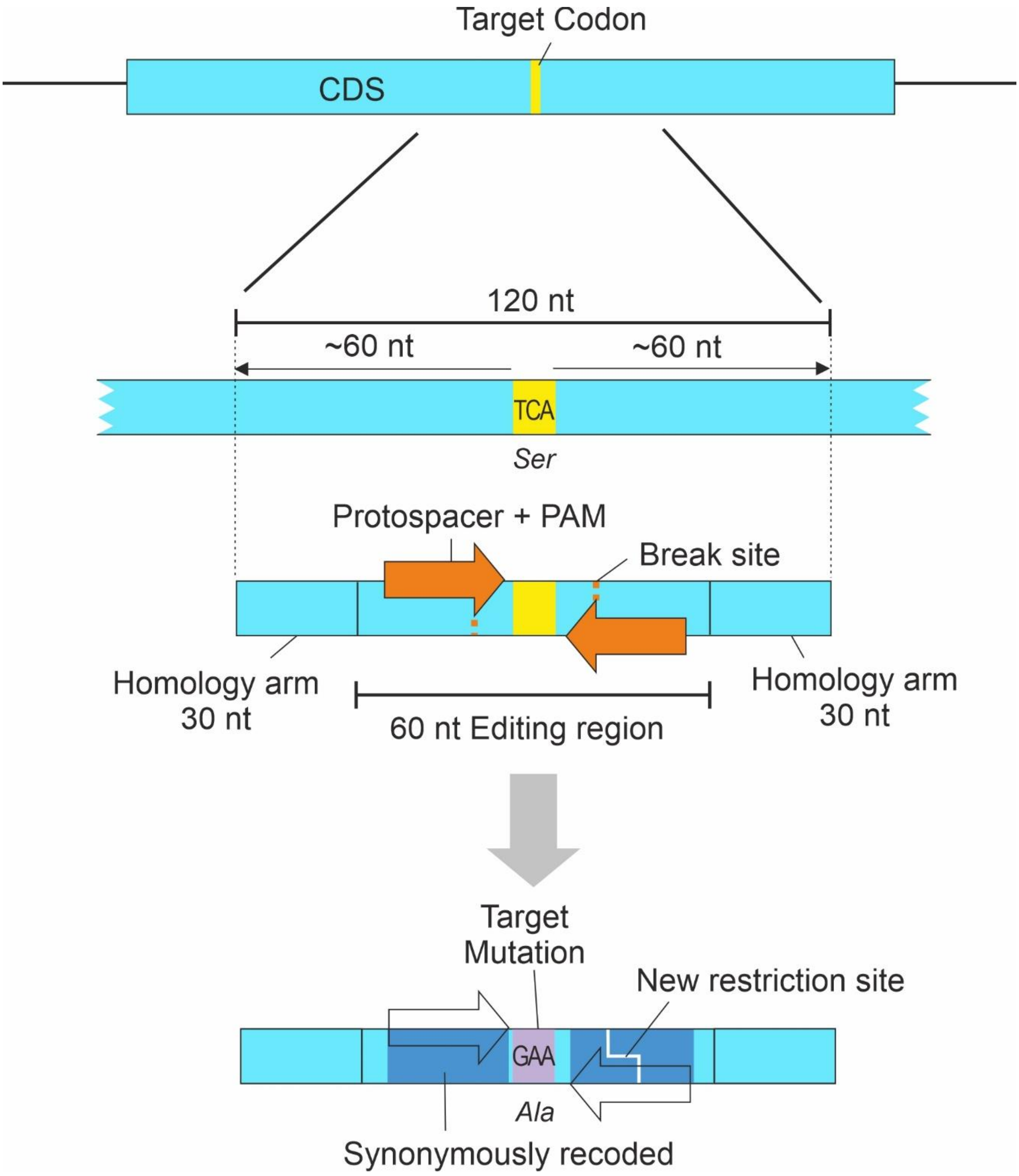


Figure 3. Single-stranded repair template design process as described in Methods 3.2. First, the target codon (yellow) was identified in the genomic sequence. Next, a region of approximately 60 nt either side of the target codon, to a final length of exactly 120 nt was selected. sgRNA guides (orange arrows) were designed in the centre of this sequence (editing region), such that one break site was either side of the target (orange dashed line). The 30 nt at each end of this region were left as the native sequence to allow homologous

recombination (homology arms). Lastly, the sgRNA protospacer and PAM sequences were recoded (dark blue), and the target mutation was incorporated (purple). Note that sgRNA protospacer sequences could be on either strand, but synonymous recoding of the plus strand was still employed when sgRNA protospacer sequences were on the minus strand, but in the complementary positions. The synonymous recoding also either added a new restriction site or removed an existing one for screening purposes (white line). Not to scale.

taken as the highest frequency usage codon for the desired amino acid. The exception to this recoding strategy was when generating a change in the restriction digestion pattern, where alternate codons were chosen to either add or remove a restriction site to the sequence. This design process is also shown in Figure 3.

3.3 sgRNA Production

Method as per Beneke et al. (2017). Briefly, protospacer sequences were incorporated into the following template primer, in place of the N's: 5'-gaaattaatacgactcactataggNNNNNNNNNNNNNNNNNNNNgttttagagctagaaatagc (Merck) – see Appendix 7.2.5.1 for primer sequences. This primer contains a T7 promoter sequence and an annealing region to bind to the primer OL6137 (G00 from Beneke et al., 2017). An annealing and amplification reaction was completed with 100 µM target specific sgRNA primer and OL6137. Annealing and amplification took place with Q5 polymerase (NEB), as per manufacturer's instructions and with the following cycling conditions: 98°C for 30 seconds (1 cycle); 98°C for 10 seconds, 60°C for 30 seconds and 72°C for 15 seconds (35 cycles); 72°C for 10 minutes (1 cycle). Resulting reactions were examined on an agarose gel to check for expected products, and stored at -20°C between production and use in downstream applications. The final construct contains a T7 RNA polymerase promoter sequence, the protospacer sequence and the CRISPR RNA backbone in DNA form, which is transcribed endogenously into RNA by T7 polymerase.

3.4 DNA Preparation for Transfection

sgRNA PCR products were purified using a PCR purification kit (Qiagen) as per the manufacturer's instructions, except eluting in 10 µl of sterile distilled water. Oligonucleotides for repair templates (Merck) were ordered dry, and resuspended at 2 µg/µl. 5 µl of each of the sgRNA purified PCR product (approximately 2.5 µg) and the repair template (10 µg) were combined.

3.5 Transfection and Cloning

T7Cas9 promastigote cells were grown until mid-log phase. 5 x 10^6 cells were pelleted at 1000 x *g* for 10 minutes, washed once in Phosphate Buffered Saline (PBS), and pelleted again. Cells were resuspended in 100 µl P3 Primary Cell Nucleofector® Solution (Lonza) and 10 µl of sgRNA-repair DNA mix. Cells were electroporated using a Lonza 4D Nucleofector® Unit using programme FI115, and promptly transferred to pre-warmed HOMEM media containing 20% FBS and 1% penicillin-streptomycin (henceforth called 20% FBS HOMEM), but without addition of other antibiotics. Cells were recovered overnight at 25°C. The following morning, cells were counted, and plated out into 96-well plates at a density of 0.5 cells/well, in 20% FBS HOMEM. Clones were left to grow in the 96-well plates at 25°C for an additional 2 weeks. Clones were then chosen at random and passaged into 12-well plates of 10% FBS HOMEM for subsequent growth.

3.6 Single-Stranded Screening

3.6.1 SCREENING OF CLONES

Genotyping of selected clones was completed through a restriction digest strategy. Stationary phase cells were pelleted at 1000 x *g* for 10 minutes, and washed once in PBS. Pellets were frozen dry at -20°C. After thawing, genomic DNA was extracted using Rapid Extract PCR Kit (PCR Biosystems), as per manufacturer's instructions, except skipping the addition of water and final centrifugation step. DNA was stored at -20°C between uses.

2 µl of DNA was used for a screening PCR with VeriFi polymerase mix (PCR Biosystems) on DNA collected from transfected clones and a T7Cas9 parental cell line as a WT control. This PCR spanned the entire region where the repair template was expected to integrate, as

well as some of the surrounding genomic sequence. See Appendix 7.2.6 for details of specific primers and cycling conditions. PCR products were confirmed on an agarose gel. Successful PCR products were purified using a PCR Purification Kit (Qiagen) as per manufacturer's instructions. Purified PCR products were quantified using a nanodrop and adjusted to the same concentration using the elution buffer.

Due to the inclusion of a restriction site change in the repair template, the genotype could be determined by digesting the previous PCR. To do so, 500 ng of purified PCR product from each clone and the parental T7Cas9 cell line (WT) was digested with the corresponding enzyme listed in Appendix 7.2.6. The reaction was incubated for 1 hour at the appropriate temperature, and then frozen at -20°C to halt the reaction. Undigested input DNA and digested DNA were run out on agarose gels to determine genotype. Undigested DNA from the parental cells and clones indicating a mutant genotype were sent for Sanger Sequencing (Eurofins) with the primers indicated in Appendix 7.2.5.3.

3.7 Single-Stranded Pooled Experiment

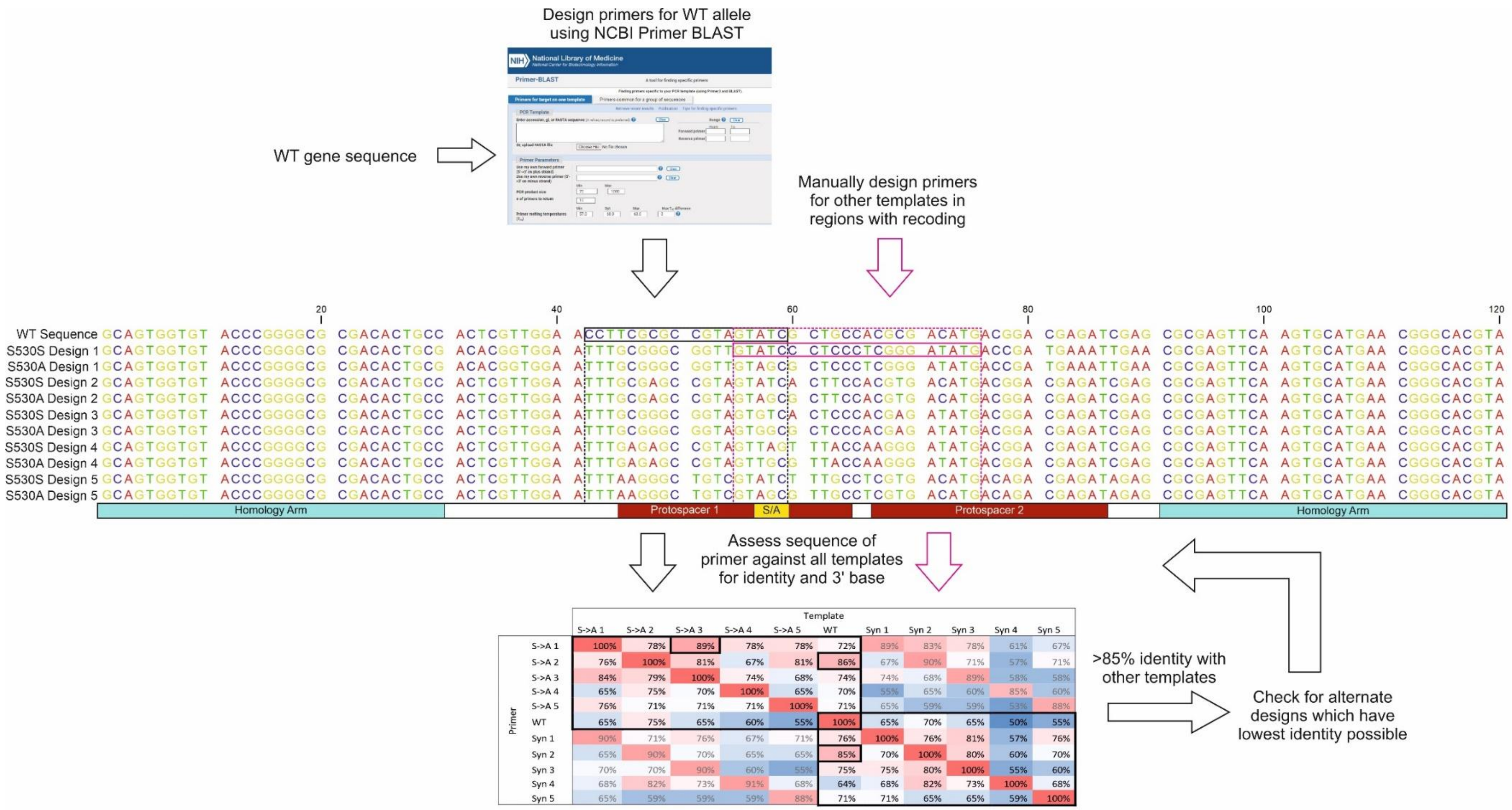
3.7.1 DESIGN OF REPAIR TEMPLATES

In order to assess the effect of the possible silent mutations, a small library of repair templates were designed to assess integration of each design, targeting either KKT2 S493 or KKT2 S530. Repair templates were designed mostly as before, except with changes in strategy for the synonymous recoding. Five recoding strategies were used to generate 5 unique repair templates for each target site. Each design had a different subset of possible synonymous mutations to choose from, which restricted both starting genomic sequences that could be mutated and what they could be mutated to. Strategies were devised based on the different levels of efficacy of mutations in single-stranded repair templates used in this report, as well as other data from the lab (Hannah Jones and Juliana Carnielli, unpublished data). In short, mutations were categorised based on the number of known instances of integration of each possible silent mutation across all precision editing attempts within these datasets. Subsequently, each design was constrained to use only mutations of a similar level of demonstrated integration (e.g. mutations which were only found in successful repair templates or mutations which had only been selected in failed transfections). For more details on the criteria to choose acceptable mutations for each

Table 1. Alignment of recoded regions of pooled repair templates. Black text indicates WT sequence, orange text indicates synonymously recoded sequence, yellow highlight indicates a non-synonymous mutation from serine to alanine.

Name																			
KKT2 S493 Translation	A	P	R	T	S	R	S	V	R	R	S	V	S/A	L	T	E	Q	E	R
KKT2 S493 WT region	GCC	CCT	CGC	ACG	TCT	CGA	TCA	GTG	CGT	CGT	AGC	GTC	AGC	TTA	ACG	GAG	CAG	GAG	CGG
KKT2 S493A design 1	GCG	CCA	CGG	ACC	TCA	CGT	TCT	GTC	CGT	CGT	AGC	GTA	GCG	CTA	ACC	GAA	CAA	GAA	CGT
KKT2 S493S design 1	GCG	CCA	CGG	ACC	TCA	CGT	TCT	GTC	CGT	CGT	AGC	GTA	TCG	CTA	ACC	GAA	CAA	GAA	CGT
KKT2 S493A design 2	GCG	CCT	CGA	ACG	TCT	CGA	TCA	GTG	CGT	CGT	AGC	GTA	GCG	CTC	ACG	GAG	CAG	GAG	CGT
KKT2 S493S design 2	GCG	CCT	CGA	ACG	TCT	CGA	TCA	GTG	CGT	CGT	AGC	GTA	TCG	CTC	ACG	GAG	CAG	GAG	CGT
KKT2 S493A design 3	GCG	CCA	CGC	ACG	TCT	CGA	TCA	GTG	CGT	CGT	TCG	GTA	GCG	CTA	ACG	GAG	CAA	GAA	CGT
KKT2 S493S design 3	GCG	CCA	CGC	ACG	TCT	CGA	TCA	GTG	CGT	CGT	TCG	GTA	AGT	CTA	ACG	GAG	CAA	GAA	CGT
KKT2 S493A design 4	GCG	CCT	AGG	ACG	AGT	CGA	AGC	GTG	AGG	CGT	AGC	GTA	GCG	CTC	ACG	GAG	CAA	GAG	AGA
KKT2 S493S design 4	GCG	CCT	AGG	ACG	AGT	CGA	AGC	GTG	AGG	CGT	AGC	GTA	TCC	CTC	ACG	GAG	CAA	GAG	AGA
KKT2 S493A design 5	GCA	CCC	CGT	ACA	TCT	CGA	TCA	GTG	CGT	CGT	TCC	GTA	GCG	CTC	ACA	GAG	CAG	GAG	AGG
KKT2 S493S design 5	GCA	CCC	CGT	ACA	TCT	CGA	TCA	GTG	CGT	CGT	TCC	GTA	TCT	CTC	ACA	GAG	CAG	GAG	AGG

Name																					
KKT2 S530 1 guide Translation	A	T	R	W	N	L	R	A	V	V	S/A	L	P	R	D	M	T	D	E	I	E
KKT2 S530 WT region	GCC	ACT	CGT	TGG	AAC	CTT	CGC	GCC	GTA	GTA	TCG	CTG	CCA	CGC	GAC	ATG	ACG	GAC	GAG	ATC	GAG
KKT2 S530A design 1	GCG	ACA	CGG	TGG	AAT	TTG	CGG	GCG	GTT	GTA	GCG	CTC	CCT	CGG	GAT	ATG	ACC	GAT	GAA	ATT	GAA
KKT2 S530S design 1	GCG	ACA	CGG	TGG	AAT	TTG	CGG	GCG	GTT	GTA	TCC	CTC	CCT	CGG	GAT	ATG	ACC	GAT	GAA	ATT	GAA
KKT2 S530A design 2	GCC	ACT	CGT	TGG	AAT	TTG	CGA	GCC	GTA	GTA	GCG	CTT	CCA	CGT	GAC	ATG	ACG	GAC	GAG	ATC	GAG
KKT2 S530S design 2	GCC	ACT	CGT	TGG	AAT	TTG	CGA	GCC	GTA	GTA	TCA	CTT	CCA	CGT	GAC	ATG	ACG	GAC	GAG	ATC	GAG
KKT2 S530A design 3	GCC	ACT	CGT	TGG	AAT	TTG	CGG	GCG	GTA	GTG	GCG	CTC	CCA	CGA	GAT	ATG	ACG	GAC	GAG	ATC	GAG
KKT2 S530S design 3	GCC	ACT	CGT	TGG	AAT	TTG	CGG	GCG	GTA	GTG	TCA	CTC	CCA	CGA	GAT	ATG	ACG	GAC	GAG	ATC	GAG
KKT2 S530A design 4	GCC	ACT	CGT	TGG	AAT	TTG	AGA	GCC	GTA	GTT	GCG	TTA	CCA	AGG	GAT	ATG	ACG	GAC	GAG	ATC	GAG
KKT2 S530S design 4	GCC	ACT	CGT	TGG	AAT	TTG	AGA	GCC	GTA	GTT	AGT	TTA	CCA	AGG	GAT	ATG	ACG	GAC	GAG	ATC	GAG
KKT2 S530A design 5	GCC	ACT	CGT	TGG	AAT	TTA	AGG	GCT	GTC	GTA	GCG	TTG	CCT	CGT	GAC	ATG	ACA	GAC	GAG	ATA	GAG
KKT2 S530S design 5	GCC	ACT	CGT	TGG	AAT	TTA	AGG	GCT	GTC	GTA	TCT	TTG	CCT	CGT	GAC	ATG	ACA	GAC	GAG	ATA	GAG


Design primers for WT allele using NCBI Primer BLAST
WT gene sequence
Manually design primers for other templates in regions with recoding
Assess sequence of primer against all templates for identity and 3' base
>85% identity with other templates
Check for alternate designs which have lowest identity possible
NIH National Library of Medicine
National Center for Biotechnology Information
Primer-BLAST
A tool for finding specific primers
Finding primers specific to your PCR template (using Primer3 and BLAST).
Primers for target on one template
Primers common for a group of sequences
Retrieve recent results
Publication
Tips for finding specific primers
PCR Template
Enter accession, gi, or FASTA sequence
Clear
Range
Clear
From
To
Forward primer
Reverse primer
Or, upload FASTA file
Choose File
No file chosen
Primer Parameters
Use my own forward primer (5'->3' on plus strand)
Use my own reverse primer (5'->3' on minus strand)
Clear
Min
Max
PCR product size
70
1000
# of primers to return
10
Min
Opt
Max
Max Tm difference
Primer melting temperatures (Tm)
57.0
60.0
63.0
3
20
40
60
80
100
120
Homology Arm
Protospacer 1
S/A
Protospacer 2
Homology Arm

```
WT Sequence     GCAGTGGTGT ACCCGGGGCG CGACACTGCC ACTCGTTGGA ACCTTCGCGC CGTAGTATCG CTGCCACGCG ACATGACGGA CGAGATCGAG CGCGAGTTCA AGTGCATGAA CGGGCACGTA
S530S Design 1  GCAGTGGTGT ACCCGGGGCG CGACACTGCG ACACGGTGGA ATTTGCGGGC GGTTGTATCC CTCCCTCGGG ATATGACCGA TGAAATTGAA CGCGAGTTCA AGTGCATGAA CGGGCACGTA
S530A Design 1  GCAGTGGTGT ACCCGGGGCG CGACACTGCG ACACGGTGGA ATTTGCGGGC GGTTGTAGCG CTCCCTCGGG ATATGACCGA TGAAATTGAA CGCGAGTTCA AGTGCATGAA CGGGCACGTA
S530S Design 2  GCAGTGGTGT ACCCGGGGCG CGACACTGCC ACTCGTTGGA ATTTGCGAGC CGTAGTATCA CTTCCACGTG ACATGACGGA CGAGATCGAG CGCGAGTTCA AGTGCATGAA CGGGCACGTA
S530A Design 2  GCAGTGGTGT ACCCGGGGCG CGACACTGCC ACTCGTTGGA ATTTGCGAGC CGTAGTAGCG CTTCCACGTG ACATGACGGA CGAGATCGAG CGCGAGTTCA AGTGCATGAA CGGGCACGTA
S530S Design 3  GCAGTGGTGT ACCCGGGGCG CGACACTGCC ACTCGTTGGA ATTTGCGGGC GGTAGTGTCA CTCCCACGAG ATATGACGGA CGAGATCGAG CGCGAGTTCA AGTGCATGAA CGGGCACGTA
S530A Design 3  GCAGTGGTGT ACCCGGGGCG CGACACTGCC ACTCGTTGGA ATTTGCGGGC GGTAGTGGCG CTCCCACGAG ATATGACGGA CGAGATCGAG CGCGAGTTCA AGTGCATGAA CGGGCACGTA
S530S Design 4  GCAGTGGTGT ACCCGGGGCG CGACACTGCC ACTCGTTGGA ATTTGAGAGC CGTAGTTAGT TTACCAAGGG ATATGACGGA CGAGATCGAG CGCGAGTTCA AGTGCATGAA CGGGCACGTA
S530A Design 4  GCAGTGGTGT ACCCGGGGCG CGACACTGCC ACTCGTTGGA ATTTGAGAGC CGTAGTTGCG TTACCAAGGG ATATGACGGA CGAGATCGAG CGCGAGTTCA AGTGCATGAA CGGGCACGTA
S530S Design 5  GCAGTGGTGT ACCCGGGGCG CGACACTGCC ACTCGTTGGA ATTTAAGGGC TGTCGTATCT TTGCCTCGTG ACATGACAGA CGAGATAGAG CGCGAGTTCA AGTGCATGAA CGGGCACGTA
S530A Design 5  GCAGTGGTGT ACCCGGGGCG CGACACTGCC ACTCGTTGGA ATTTAAGGGC TGTCGTAGCG TTGCCTCGTG ACATGACAGA CGAGATAGAG CGCGAGTTCA AGTGCATGAA CGGGCACGTA
```

	Template										
Primer	S->A 1	S->A 2	S->A 3	S->A 4	S->A 5	WT	Syn 1	Syn 2	Syn 3	Syn 4	Syn 5
S->A 1	100%	78%	89%	78%	78%	72%	89%	83%	78%	61%	67%
S->A 2	76%	100%	81%	67%	81%	86%	67%	90%	71%	57%	71%
S->A 3	84%	79%	100%	74%	68%	74%	74%	68%	89%	58%	58%
S->A 4	65%	75%	70%	100%	65%	70%	55%	65%	60%	85%	60%
S->A 5	76%	71%	71%	71%	100%	71%	65%	59%	59%	53%	88%
WT	65%	75%	65%	60%	55%	100%	65%	70%	65%	50%	55%
Syn 1	90%	71%	76%	67%	71%	76%	100%	76%	81%	57%	76%
Syn 2	65%	90%	70%	65%	65%	85%	70%	100%	80%	60%	70%
Syn 3	70%	70%	90%	60%	55%	75%	75%	80%	100%	55%	60%
Syn 4	68%	82%	73%	91%	68%	64%	68%	82%	73%	100%	68%
Syn 5	65%	59%	59%	59%	88%	71%	71%	65%	65%	59%	100%

Figure 4. Example pooled repair template transfection screening primer design process. In order to specifically detect each of the five repair templates in each pooled transfection, an iterative primer design process was used to design screening primers for each repair template. The process aimed to identify primers which were least likely to cross-react with other repair templates. An initial primer set were designed on the WT gene sequence, with one primer outside the repair template region (not shown), and another falling across a region of recoding in the repair templates (black outline). Primers binding to each other template were designed manually (one example shown in pink outline). Primers were designed to have reasonable annealing temperatures to pair with the external primer and ideally a unique 3' base. Once a potential primer sequence was determined, the sequence was compared to the other templates in the same pool (i.e. non-synonymous or synonymous templates, and WT sequence) at the corresponding location (dotted outlines). Final primers used have the lowest identity possible with other templates, preferably less than 85%. Primers that do not have less than 85% identity with another template are highlighted with dark outlines in the comparison table.

design, see Results 4.2, with the acceptable codon changes lists in Appendix 7.2.8. Synonymous repair templates were the same as the serine to alanine designs, except choosing the serine codon from the appropriate list for that design. An alignment of the recoded regions of these designs is shown in Table 1.

3.7.2 DNA PREPARATION, TRANSFECTION AND GENOMIC DNA EXTRACTION

sgRNA guides were made and prepared as before. Instead of using one repair template, five different repair templates were mixed in equal proportions (2 µg/µl stock), and 5 µl of the mixed repair templates were used for each transfection. These repair templates and the combinations of the pools they were mixed in are listed in Appendix 7.2.4.3.1, with sequences of each design in Appendix 7.2.4.3. Transfection was otherwise as before, except cells were not cloned out and were left as populations.

After one week of recovery, half of the culture was pelleted and washed as before. DNA was extracted using a genomic DNA extraction kit (ENZA) as per the manufacturer's instructions. DNA was also collected from the parental T7Cas9 cell line in the same way.

3.7.3 *SCREENING PCR FOR REPAIR TEMPLATE INTEGRATION FROM POOLED* TRANSFECTIONS

A PCR strategy was used to detect the correct integration of each repair template in the pooled transfection. This strategy involved the use of one shared primer outside of the repair template region, and one primer inside the repair template in a region unique to that repair template. The screening primer design process to detect integration of specific repair templates is shown in Figure 4. Briefly, primers were designed manually to ensure the binding region was as unique to the specific design as possible. Ideally, each primer had <85% identity against other repair templates in the same transfection, and where possible with a unique 3' base, to reduce the chances of amplification against other repair templates. Screening primer sequences and cycling conditions using Q5 polymerase (NEB) can be found in Appendix 7.2.7. To clarify, one mastermix was set up per transfection containing the genomic DNA to ensure that any absence of a band was not due to the absence of template DNA. Resulting PCR products were assessed on an agarose gel.

3.8 Repair Template Design – Double-Stranded

3.8.1 DESIGN

To assess whether double-stranded DNA was more efficient at generating precision mutants, repair template designs targeting KKT2 S493A, KKT2 S530A (1 guide design), and KKT2 S923A were reused. The only change to the design was that the homology arms were extended outwards from the mutagenized site by 20 bp to a final length of 160 bp repair template. Additionally, repair templates were designed to generate KKT2 S25 mutants to either alanine, glutamic acid or synonymous serines. The repair template was designed as per the pooled design 2 recoding methodology (see Methods 3.7.1 and Appendix 7.2.4.4 for details), except extending the length (as with the other targets) and reducing the emphasis on the inclusion of a restriction site (as PCR-based screening was to be used). For new targets, sgRNAs were preferentially chosen to leave a gap between each protospacer sequence, and recoding focussed on incorporating more A's and T's, to aid screening primer design. For targets attempted with single-stranded repair templates, the same sgRNAs and recoding were used, just with the extension to the homology arms of the repair template and modifying repair templates to include glutamic acid and synonymous mutations as well. Designs and full sequences can be found in Appendices 7.2.4.4 and 7.2.5.5.

3.8.2 PRODUCTION OF REPAIR TEMPLATES

For double stranded repair templates, the design was split into roughly half, with an overlapping region of 18-20 bp (Figure 8). Oligonucleotides (Merck) for each half were annealed and amplified as in the method described to produce sgRNA (Methods 3.3), except adjusting the annealing temperature to each reaction. Resulting reactions were run out on an agarose gel to check for correct product formation. Products were purified with a PCR purification kit (Qiagen) as per the manufacturer's instructions, except eluted in 10 µl of water.

3.8.3 DNA PREPARATION, TRANSFECTION AND GENOMIC DNA EXTRACTION

sgRNA was prepared as before – primer sequences can be found in Appendix 7.2.5.1. Repair templates and sgRNA guides were cleaned up using a PCR Purification Kit (Qiagen), as per the manufacturer's instructions, except eluting in 10 µl of sterile distilled water. 5 µl of this was used for each transfection as before (Methods 3.5). This was approximately 5 µg of repair template per transfection, reduced from 10 µg used in the single-stranded DNA (ssDNA) transfections, as well as doubling the quantity of sgRNA compared to previous ssDNA transfections (approximately 5 µg sgRNA DNA). Transfected cells were recovered in 20% FBS HOMEM containing 10 µM 6-biopterin (Merck) both before and during cloning, as 6-biopterin has been shown to improve growth of *Leishmania* (Trager, 1969). Clones were expanded following 1 week of recovery. On expansion, all cells were transferred to 10% FBS HOMEM without 6-biopterin. Cell pellets were collected from populations and clones as before.

3.8.4 SCREENING PCRS

5 µl of the extracted genomic DNA was used for each screening PCR with VeriFi polymerase mix (PCR Biosystems) – see Appendix 7.2.5.6 for specific primer and cycling details. WT and mutant PCRs were set up independently, so an absence of a band in either reaction was considered to be a failure, but technical error was not controlled for due to the large quantities of reactions. PCR products were run out on agarose gels to determine genotype.

3.8.5 SEQUENCING

Clones indicating a positive result in the mutant PCR reaction were taken forward for Sanger sequencing. 5 µl of genomic DNA was used for an additional PCR that covered the whole repair template with Q5 polymerase (NEB) – see Appendix 7.2.5.7 for primers. PCR products were checked on an agarose gel, and then purified with a PCR purification kit (Qiagen) as per manufacturer's instructions. PCR products were sent for Sanger Sequencing (Genewiz) with the same primers used for amplification.

3.9 Alamar Blue Growth Assay

Where possible, two homozygous clones of each kinetochore phosphosite mutation were selected at random. Where two were not available, either an additional clone with a different genotype was chosen, or only one clone was used. When a non-homozygous clone was used due to a lack of homozygous clones, where possible, it was chosen to have the target mutation on both alleles, and with as much of the repair template integrated into both alleles as possible. I.e. a "complex" mutant homozygous for the target was favoured over fully heterozygous mutants due to concerns of replacement of the target mutation with the WT allele. But a heterozygous mutant was used if no other mutants were available. Cultures were grown to mid-log phase in 10% FBS HOMEM. Dilutions of each culture were prepared to 2500 cells/ml in the same media and were seeded onto 96-well plates in triplicate such that 500 cells were seeded per well. A medium only control was also included, and empty wells were filled to the same volume with PBS. 96-well plates were prepared in duplicate (one for use as a day 0 plate, one for use as a day 5 plate). The day 0 plate was immediately supplemented with 40 µl 0.0125% (w/v) resazurin (Alamar blue) in PBS into each well (except those containing PBS) and left to incubate in the dark at 25°C for 4-6 hours. The day 5 plate, was returned to the incubator for 5 days, then supplemented with Alamar blue in the same way. After incubation with the Alamar blue, the fluorescence at emissions of 590 nm was measured with a BMG Labtech CLARIOstar® microplate reader. The readings of the wells containing cells were corrected to the media-only wells (blank). The mean of the triplicate wells was normalised to the parental T7Cas9 control to calculate the percentage growth.

3.10 Flow Cytometry

As the mutations generated all targeted the kinetochore complex, it was expected that these mutations would lead to a cell cycle progression phenotype such as an accumulation in one phase of the cell cycle. To assess this, the quantity of DNA content per cell was assessed through propidium-iodide flow cytometry. The same clones as per the Alamar Blue assay were grown in 10% FBS HOMEM media, and 1 x 10^7 mid-log phase cells were pelleted at 1000 x g for 10 minutes. Cells were washed once in PBS with 5 mM EDTA (PBS-EDTA), and the pellet was resuspended in PBS with PBS-EDTA. Cold methanol was added slowly to a final concentration of 70% (v/v) and were left at 4°C to fix overnight. After fixation, samples were diluted to 36.8% methanol (v/v) by adding PBS-EDTA and cells were pelleted as before. The pellet was washed once in PBS-EDTA and was resuspended in PBS-EDTA with 10 µg/ml propidium iodide and 10 µg/ml RNaseA. Samples were incubated in the dark at 4°C overnight, gently resuspended and transferred to a 96-well plate, splitting the sample between three wells per cell line. Samples were analysed on a CytoFLEX LX355, gating for parasite cells, followed by single cells (singles). Each well was set to record 20,000 events in singles, measuring the propidium iodide, as well as forward and side scatter. FCS Express 7 was used to analyse the results. The gating used to collect the data was replicated for analysis, and the number of cells was plotted against the propidium iodide intensity. The proportion of cells under each peak was assessed using the built-in DNA content analysis (Multicycle) to fit 1 cycle using model 5. The percentages of cells in each cell cycle stage (G1, S and G2/M) were collated for the triplicate wells, which was then averaged. The replicates were the averaged and plotted, with the exception of KKT2 S530E clone 21 and KKT2 S25E clone 11 - see Results section 4.6 for further details.

3.11 Statistical Analyses

Statistical analyses were performed using GraphPad Prism version 8.3.0. For the Alamar blue growth assay, a one-way ANOVA test with Dunnett's multiple comparisons was performed, comparing the means of each cell line with the parental T7Cas9.

For the cell cycle flow cytometry, a 2-way ANOVA test with Dunnett's multiple comparisons test was performed, set to compare the mean of each cell cycle stage for each cell line against the corresponding mean of the T7Cas9 parental cell cycle stage. Even though the percentages are linked (i.e. if G1 is higher, S + G2/M must be lower), each cell cycle stage was assessed independently to simplify the analysis.

4 CHAPTER THREE - RESULTS

4.1 Single-Stranded Repair Templates

To investigate the impact of phosphorylation on kinetochore proteins in *Leishmania mexicana*, a CRISPR-Cas9 precision editing strategy was used to attempt to ablate phosphosites of interest on kinetochore proteins. These sites were chosen based on data from Geoghegan et al. (2022), which indicated importance in the cell cycle. In order to generate the chosen mutations, the workflow shown in Figure 5 was used. Briefly, a repair template was designed and synthesised as a 120 nt oligonucleotide. This method was adapted from unpublished work by Juliana Carnielli, which had used a similar approach previously, to investigate the kinase domain of KKT2 by mutating the gate-keeper residue of the ATP-binding domain. Similar use of single-stranded oligonucleotide repair templates has also shown to be effective in kinetoplastids in the literature - Figure 1B (Zhang and Matlashewski, 2015; Medeiros et al., 2017; Rico et al., 2018; Wall et al., 2018; Pal and Dam, 2022).

Following transfections, up to 40 clones were screened using a restriction digest strategy to look for the presence of mutant alleles (Figure 5, see Appendix 7.2.6 for further details). If present, the clone was sequenced with Sanger sequencing. In addition to nonsynonymous mutations to ablate the phosphosite, a non-phosphosite positive control targeting KKT2 M146G (the gate-keeper residue of the kinase domain), and synonymous mutations for KKT7 S304 (KKT7 S304S) and KKT4 S300 (KKT4 S300S) were tested as additional positive controls. A summary of the results can be found in Table 2.

Of the 16 different transfections, only 3 showed incorporation of the mutation – one of which was the positive control (KKT2 M146G) and had been produced previously (Juliana Carnielli, unpublished work). Other than KKT2 M146G, integration of the repair template was detected in 3 clones in KKT2 S493A and 2 clones in KKT4 S300A (Figure 6). Homozygous integration of the repair template was detected in all three transfections, but heterozygous integration was only detected in KKT2 M146G (Table 2).

In addition to these clones, one clone from each of the KKT2 S493A and KKT4 S300A transfections indicated an unexpected genotype (KKT2 S493A Clone 37 and KKT4 S300A Clone 16, Figure 6A and B). Sanger sequencing revealed that these clones incorporated the

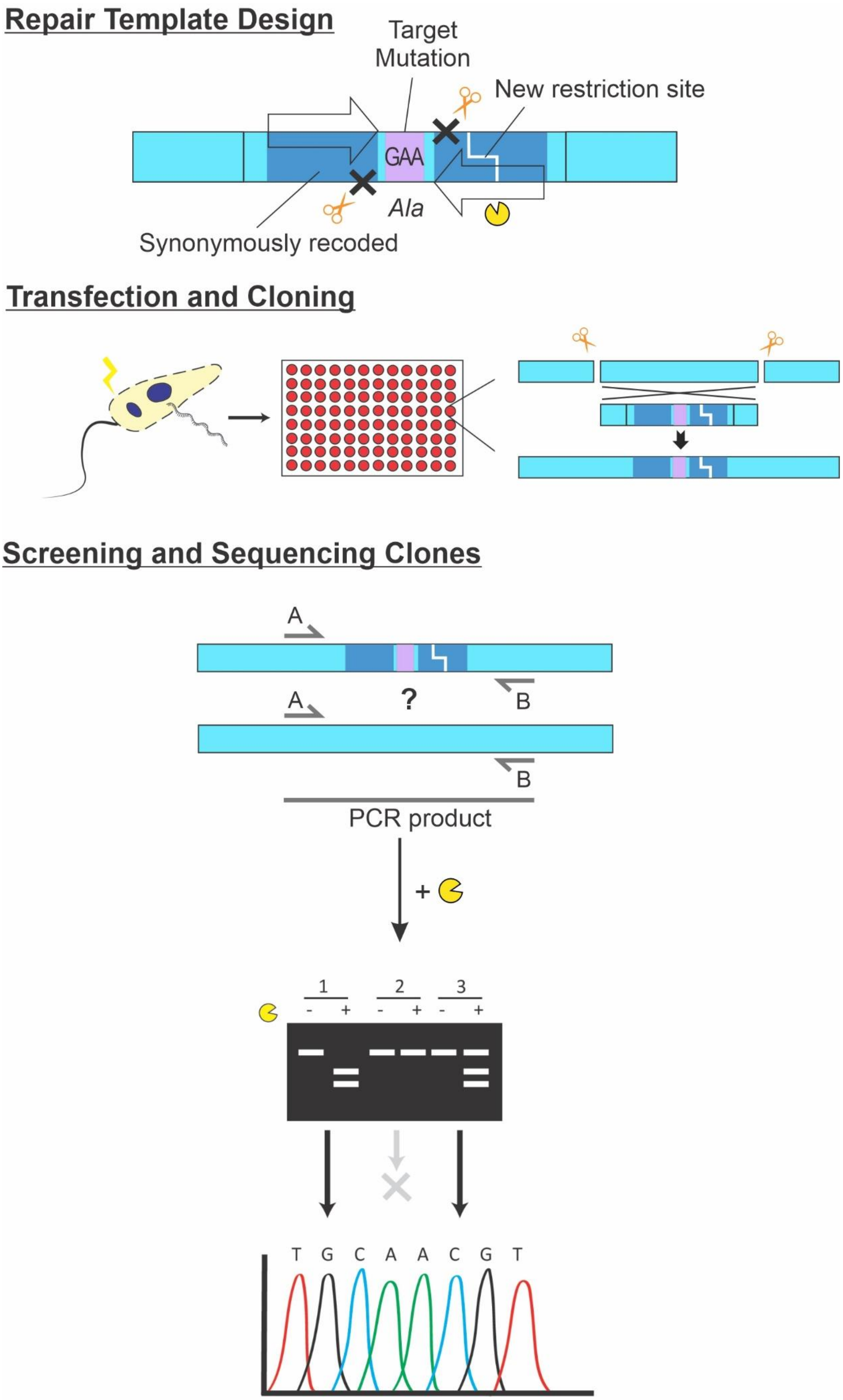

Repair Template Design
Target Mutation
New restriction site
GAA
Ala
Synonymously recoded
Transfection and Cloning
Screening and Sequencing Clones
A
A
?
B
B
PCR product
+
1
2
3
- +
- +
- +
T G C A A C G T

Figure 5. Schematic of the single-stranded DNA precision editing workflow. Repair template is designed to: mutate the target site codon, remove Cas9 break sites and change (add or remove) a restriction site. The repair template was synthesised as a 120 nt single-stranded oligonucleotide (adapted from Juliana Carnielli, unpublished work). Cells were transfected with the repair template and sgRNA corresponding to the Cas9 break sites which were removed in the repair template. Following transfection, cells were cloned and allowed to recover to integrate the repair template. To screen clones, a PCR was completed with primers A and B to cover the entire repair template region. The PCR product was then digested with the appropriate restriction enzyme (as per the modified site in the design) to assess the genotype of the clone. Clones indicating a homozygous or heterozygous incorporation of the repair template were confirmed with Sanger sequencing.

repair template differently on each allele. Each of these repair templates had two regions of synonymous mutations (corresponding to the two protospacer targeting sequences) with a gap in between, such that the synonymous mutations were effectively separated in two (Figure 6A and B). In these clones, sequencing revealed that one of the regions of recoding was integrated in a homozygous manner, whilst the other showed a heterozygous incorporation. This suggested that recombination occurred at different places on each allele. It is likely that the short break in synonymous mutations (11 bp on KKT2 S493A, 21 bp on KKT4 S300A) was used for recombination instead of the intended homology arms. Luckily, the target codons were part of the region which was mutated on both alleles in each of these clones. As these clones had a mix of homozygosity and heterozygosity, their genotype has been designated as “complex”.

Surprisingly, no mutant clones were detected when using synonymous mutation only repair templates (Table 2). These repair templates were intended to act as a positive control, given that the coding sequence was unaltered. More strikingly, from the five synonymous repair templates tested, one of the corresponding non-synonymous mutations was successfully generated (KKT4 S300A) - Figure 6. Both KKT4 S300A and KKT4 S300S transfections were completed in parallel, suggesting that the lack of detected KKT4

Table 2. Genotyping results for transfections using single-stranded oligonucleotide repair template, following detection by restriction digest and Sanger sequencing.

Mutation	No. of Clones Screened	Homozygotes	Heterozygotes	Complex Genotypes	Percentage of Mutants (all genotypes)
KKT1 S1449A	7	0	0	0	-
KKT2 M146G	40	2	1	0	7.5%
KKT2 S493A	41	2	0	1	7.3%
KKT2 S505A	39	0	0	0	-
KKT2 S505S	40	0	0	0	-
KKT2 S505A+S506A	23	0	0	0	-
KKT2 S505S+S506S	20	0	0	0	-
KKT2 S530A	21	0	0	0	-
KKT2 S530A 1 guide	24	0	0	0	-
KKT2 S530S 1 guide	16	0	0	0	-
KKT2 S923A	21	0	0	0	-
KKT4 S300A	33	1	0	1	6.0%
KKT4 S300S	40	0	0	0	-
KKT4 S422A	15	0	0	0	-
KKT7 S304A	20	0	0	0	-
KKT7 S304S	10	0	0	0	-
TOTAL	402	5	1	2	2.0%

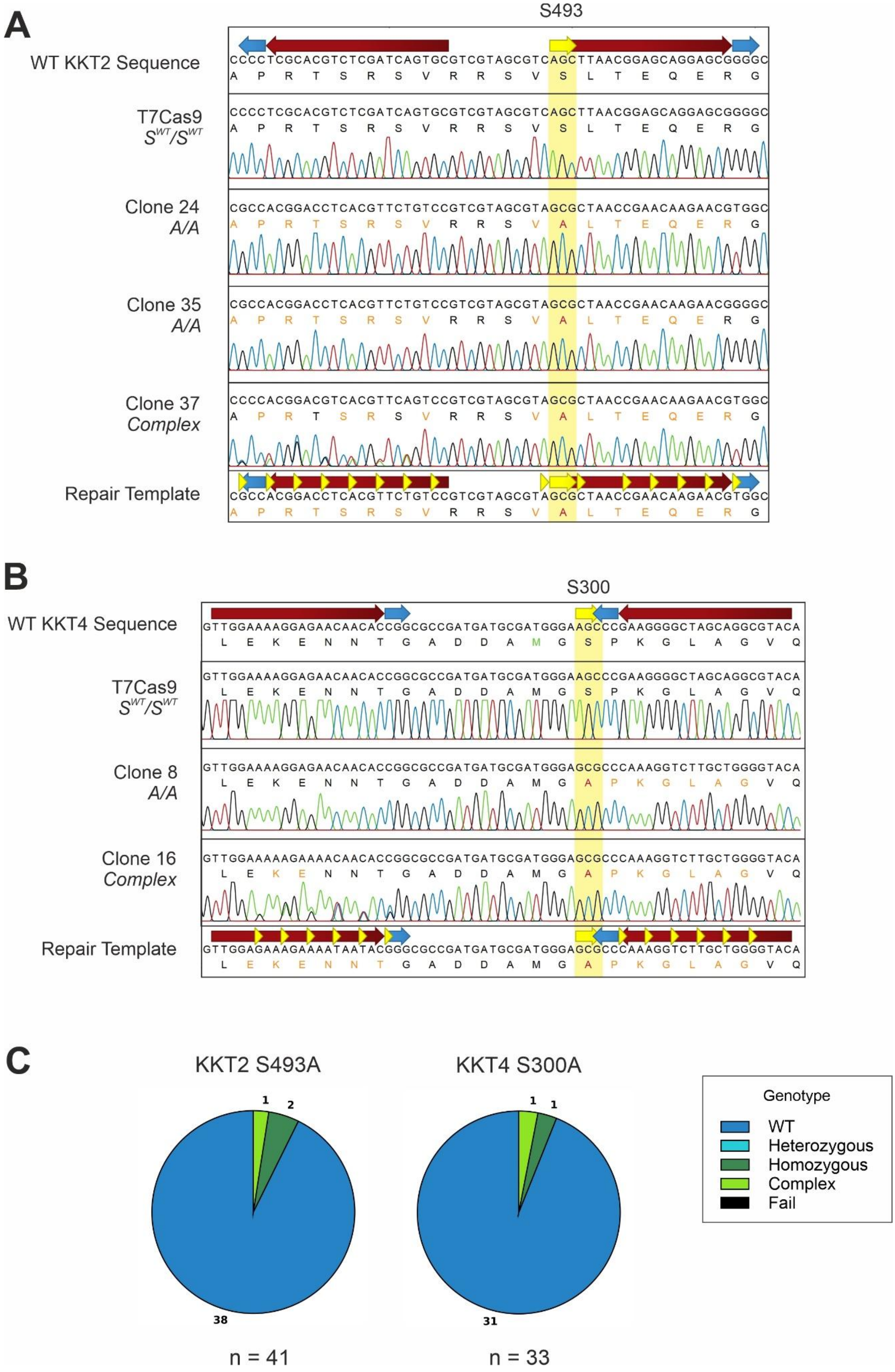

A
S493
WT KKT2 Sequence
CCCCTCGCACGTCTCGATCAGTGCGTCGTAGCGTCAGCTTAACGGAGCAGGAGCGGGGC
A P R T S R S V R R S V S L T E Q E R G
T7Cas9
S^WT/S^WT
CCCCTCGCACGTCTCGATCAGTGCGTCGTAGCGTCAGCTTAACGGAGCAGGAGCGGGGC
A P R T S R S V R R S V S L T E Q E R G
Clone 24
A/A
CGCCACGGACCTCACGTTCTGTCCGTCGTAGCGTAGCGCTAACCGAACAAGAACGTGGC
A P R T S R S V R R S V A L T E Q E R G
Clone 35
A/A
CGCCACGGACCTCACGTTCTGTCCGTCGTAGCGTAGCGCTAACCGAACAAGAACGGGGC
A P R T S R S V R R S V A L T E Q E R G
Clone 37
Complex
CCCCACGGACGTCACGTTCAGTCCGTCGTAGCGTAGCGCTAACCGAACAAGAACGTGGC
A P R T S R S V R R S V A L T E Q E R G
Repair Template
CGCCACGGACCTCACGTTCTGTCCGTCGTAGCGTAGCGCTAACCGAACAAGAACGTGGC
A P R T S R S V R R S V A L T E Q E R G
B
S300
WT KKT4 Sequence
GTTGGAAAAGGAGAACAACACCGGCGCCGATGATGCGATGGGAAGCCCGAAGGGGCTAGCAGGCGTACA
L E K E N N T G A D D A M G S P K G L A G V Q
T7Cas9
S^WT/S^WT
GTTGGAAAAGGAGAACAACACCGGCGCCGATGATGCGATGGGAAGCCCGAAGGGGCTAGCAGGCGTACA
L E K E N N T G A D D A M G S P K G L A G V Q
Clone 8
A/A
GTTGGAAAAGGAGAACAACACCGGCGCCGATGATGCGATGGGAGCGCCCAAAGGTCTTGCTGGGGTACA
L E K E N N T G A D D A M G A P K G L A G V Q
Clone 16
Complex
GTTGGAAAAAGAAAACAACACCGGCGCCGATGATGCGATGGGAGCGCCCAAAGGTCTTGCTGGGGTACA
L E K E N N T G A D D A M G A P K G L A G V Q
Repair Template
GTTGGAGAAAGAAAATAATACGGGCGCCGATGATGCGATGGGAGCGCCCAAAGGTCTTGCTGGGGTACA
L E K E N N T G A D D A M G A P K G L A G V Q
C
KKT2 S493A
1
2
38
n = 41
KKT4 S300A
1
1
31
n = 33
Genotype
WT
Heterozygous
Homozygous
Complex
Fail

Figure 6. Single-stranded oligonucleotide repair template precision editing results. Sequencing results for mutant clones from KKT2 S493A transfection (A), and KKT4 S300A transfection (B). Genotype, represented by the single-letter amino acid code that the target codon translates to is indicated below the cell line name, with superscript "WT" indicating that the codon shares the same DNA sequence as the reference sequence for identical encoded amino acids (WT DNA sequences - KKT2 S493: AGC; KKT4 S300A: AGC). Red arrows indicate protospacer sequences in WT sequences and equivalent position in repair templates; blue arrows indicate PAM sites in WT sequences and equivalent position in repair templates; yellow arrows indicate target site in WT sequences and mutated sites in repair templates. The translation is shown below each DNA sequence, with black text indicating the same protein and DNA sequence as the reference sequence, orange text indicating the same protein sequence but a different DNA sequence to the reference, and red indicating a difference in the protein sequence and hence DNA sequence. Sequencing results are cropped to show only the synonymously recoded region. (C) Summary genotyped results from KKT2 S493A and KKT4 S300A transfections. Total number of clones screened is represented by the n value, with the number of clones represented by each slice adjacent to the slice.

S300S mutant clones was unlikely to be caused by a technical failure. In addition, the repair template for KKT4 S300A had all the same synonymous recoding as KKT4 S300S, demonstrating that the other mutations were tolerated. This led to the conclusion that the efficiency of this transfection was likely very poor.

4.2 Pooled Repair Templates

To investigate whether the design of the oligonucleotide repair template was the cause of the low efficiency, five different oligonucleotide repair templates were designed targeting KKT2 S493 (positive control) and KKT2 S530, to generate both serine to alanine mutants, and serine to serine synonymous mutants. Each design used different criteria for selecting synonymous mutations to incorporate, based on analysis of oligonucleotide repair templates which had shown to work previously (analysis not shown - in addition to data in this report, data also came from Juliana Carnielli and Hannah Jones, unpublished work). From the previous data, all possible synonymous mutations were categorised as either: (A) the specific codon change had worked every time it was tried; (B) the specific codon change had appeared in both transfections which did generate mutants, and also in others that did not successfully generate mutants; (C) the specific codon change had not worked any time it was tried; or (D) the specific codon change was untested. These lists formed the basis of each design criteria. Design 1 utilised the previously generated repair templates to have as a comparator. Design 2 used recoding which met the criteria of (A). However, this list was very short, and so design 2 also had access to list (B) to ensure sufficient coverage of different target sequences to be able to remove PAM sites. Design 3 exclusively used codon changes from list (B). Design 4 primarily used list (B), except that for the three amino acids encoded by 6 triplet codes (serine, arginine and leucine), mutations were only chosen from codons that had altered the first base. Lastly, design five combined lists (C) and (D). As none of these lists contained the full complement of genomic codons, not all codons within the editing region could be mutated in every design. As such, this created varied spacing between synonymous mutations in the different designs (Table 1). Full lists can be found in Appendix 7.2.8, with full repair template sequences available in Appendix 7.2.4.3.

Each of the five oligonucleotide repair templates for each of the four mutations (KKT2 S493A, KKT2 S493S, KKT2 S530A and KKT2 S530S) were mixed in equal proportion respectively, and this mix was transfected into cells with sgRNA common to that target site (Figure 7A). Whilst there was the possibility that any given cell could take up more than one oligonucleotide repair template, given the low efficiency previously seen, it was thought that this was unlikely. Additionally, as the aim of this experiment was to assess the effect of the repair template design on integration, cells were not cloned but left as a population.

DNA was collected from the population of cells after one week of recovery, to minimise loss of cells with a lethal phenotype. This DNA was assessed for the presence of WT alleles, and for each of the five different oligonucleotide repair templates mixed in each transfection using specific primers for each potential allele i.e. WT allele and each oligonucleotide repair template design (Figure 7B). Across all four transfections using this method, strikingly, integration of 18 of the 20 unique oligonucleotide repair template designs were detected by PCR (Figure 7C and D). One of the two not detected (KKT2 S530A design 2) was unclear whether it was present due to the primers also amplifying WT DNA. The other (KKT2 S530S design 5) was the only reaction that did not produce a clear PCR product with either WT or transfected KKT2 S530S population DNA. Given that the equivalent nonsynonymous design for design 5 was detected, it is unlikely that it was something inherent about the design that caused the integration of this repair template to be undetected. But the exact reason design 5 of the S530S transfection was not detected was not explored further.

It should be noted that whilst screening primers were carefully designed to distinguish each repair template by requiring the primer to have less than 85% identity with another design than its target, due to the small region to choose from, some screening primers shared the 3' base with another oligonucleotide repair template other than its target (Figure 4). Hypothetically, this could allow amplification to occur in the absence of its intended target repair template, but in the presence of another sharing the same 3' base. As such, it is not entirely possible to rule out that for example, the primer that recognises design 2 of a transfection was amplifying from DNA which had integrated design 3. However, it is clear that all of the KKT2 S493A/S and KKT2 S530S primers did not amplify when tested with WT DNA, and that all except one of the KKT2 S530A primers also did not react with WT DNA. As such, it is safe to conclude that mutant cells from one or more of the designs were present in the population, suggesting that the recoding strategy used in the design of the oligonucleotide repair template did not appear to have a bias with regards to integration.

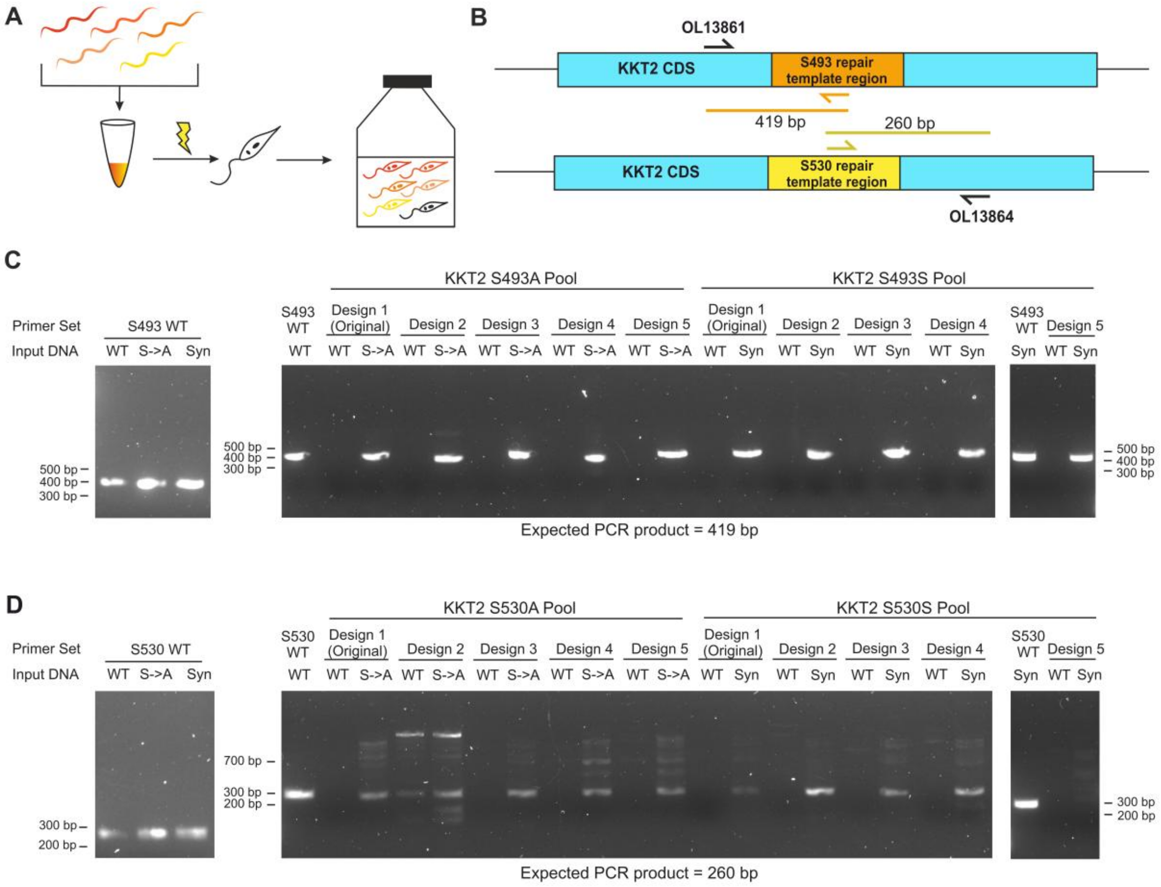


Figure 7. Single-stranded pooled repair template results. (A) Workflow of the transfection. Five repair templates targeting the same codon were combined in equal proportion and transfected into T7Cas9 cells. Cells were grown as a population following transfection. (B) Screening PCRs schematic. Orange primer corresponds to a design specific primer for either S493A or S493S. Yellow primer corresponds to a design specific primer for either S530A or S530S. Primer sequences can be found in Appendix 7.2.5.4. (C and D) Agarose gels of genotyping PCRs as described in B for S493A and synonymous pools (C), or S530A and synonymous pools (D). WT input DNA indicates parental T7Cas9 DNA, S->A indicates input DNA was from the serine to alanine mutant pool for that target site, and Syn indicates input DNA from the synonymous mutant pool for that target site.

4.3 KKT2 Synonymous Mutations Using Double-Stranded Repair Templates

From the pooled repair template experiment (Figure 7), it was apparent that nonsynonymous mutations were possible in KKT2, but were not generated at high enough efficiency to be detected at a clonal level with the current methodology (Figure 6). To separate the effect of the methodology from the potential impact of a nonsynonymous mutation, a new methodology using double-stranded DNA repair templates was tested using repair templates that would only produce synonymous mutations. If successful, the method would then be tested on nonsynonymous mutations.

As such, double-stranded repair templates for synonymous equivalent designs of KKT2 S493S, S530S and S923S were created, as well as a repair template for KKT2 S25S using the same strategy as before. Whilst the recoding was generally the same as the single-stranded repair templates, the homology arms were increased to 50 bp for a final repair template length of 160 bp. The repair templates were produced using a PCR reaction, by annealing and extending two primers, as shown in Figure 8.

12 clones from each transfection were screened by PCR for integration of the repair template (see Appendix 7.2.2.2 for all agarose gel images). Of these clones, several of them indicated the presence of a mutant allele (Table 3). Positive clones by PCR were sent for Sanger sequencing – example sequencing results can be found in Figure 9. Homozygous mutants were confirmed in all four transfections (Figure 9 and Table 3).

As previously described, the repair templates used in this experiment shared the same synonymous recoding designs as some oligonucleotide repair templates tested previously, with the exception of the codon corresponding to the target serine. Sequencing revealed that in some mutated KKT2 S923S clones, the cells did not take up the necessary recoding to generate the restriction site change, but did integrate other parts of the repair template (Figure 9D – clone 11). In this design, a single base change at the 5’ end of the repair

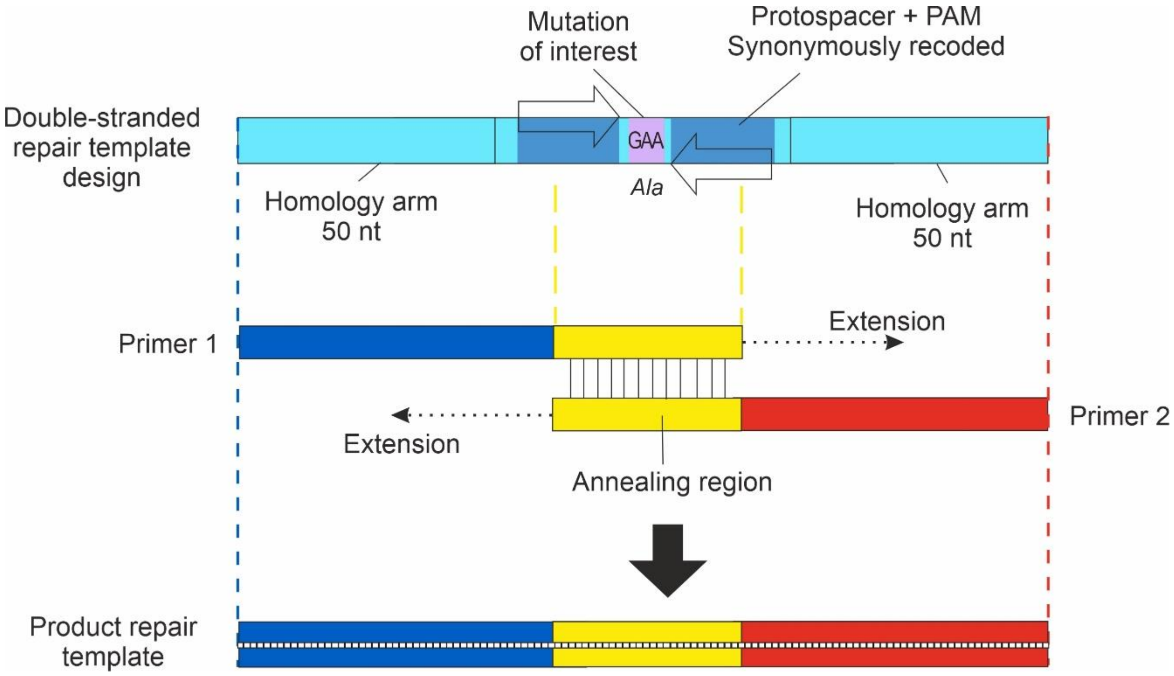


Figure 8. Production of double-stranded repair template using oligonucleotide primers. Each design was split into a forward and reverse primer that encompassed about half of the total repair template, with an overlapping annealing region (yellow). Primers were annealed together in a PCR reaction and extended to complete the entire repair template, and the product was checked by gel electrophoresis. For sequences, see Appendix 7.2.4.4 and 7.2.5.5. Diagram not to scale.

Table 3. Genotyping results of synonymous mutant clones from transfections using double-stranded repair templates when screening 12 clones.

Transfection	Homozygous Mutants	Heterozygous Mutants
KKT2 S25S	5 (4)	2 (0)
KKT2 S493S	0* (2)	3* (0)
KKT2 S530S	1 (1)	0 (0)
KKT2 S923S	1 (1)	3 (2)
TOTAL	7 (8)	8 (2)

Numbers inside brackets indicate results confirmed by sequencing.

*Genotyping PCR was unclear.

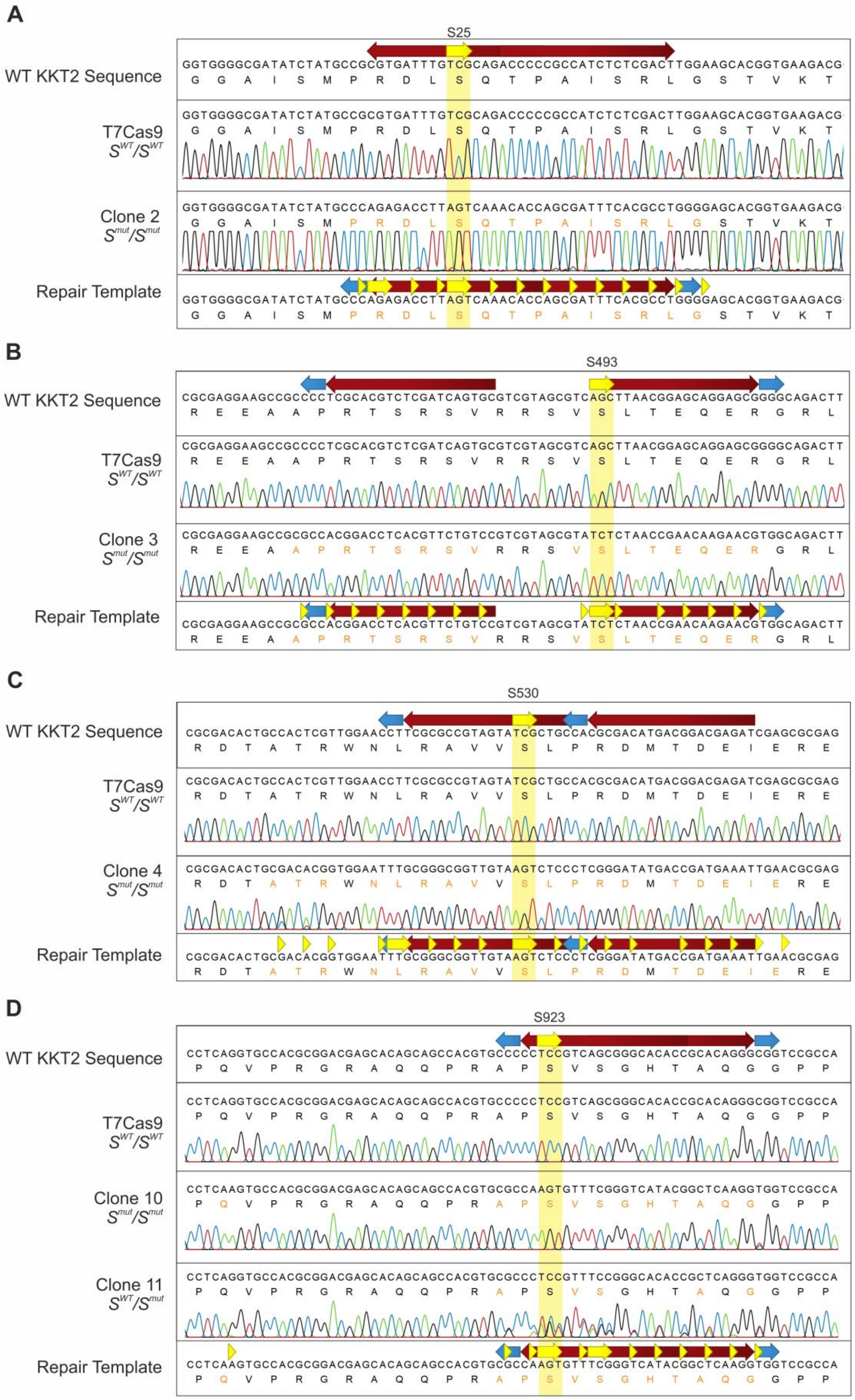

A
S25
WT KKT2 Sequence
GGTGGGGCGATATCTATGCCGCGTGATTTGTCGCAGACCCCCGCCATCTCTCGACTTGGAAGCACGGTGAAGACG
G G A I S M P R D L S Q T P A I S R L G S T V K T
T7Cas9
S^{WT}/S^{WT}
GGTGGGGCGATATCTATGCCGCGTGATTTGTCGCAGACCCCCGCCATCTCTCGACTTGGAAGCACGGTGAAGACG
G G A I S M P R D L S Q T P A I S R L G S T V K T
Clone 2
S^{mut}/S^{mut}
GGTGGGGCGATATCTATGCCCAGAGACCTTAGTCAAACACCAGCGATTTCACGCCTGGGGAGCACGGTGAAGACG
G G A I S M P R D L S Q T P A I S R L G S T V K T
Repair Template
GGTGGGGCGATATCTATGCCCAGAGACCTTAGTCAAACACCAGCGATTTCACGCCTGGGGAGCACGGTGAAGACG
G G A I S M P R D L S Q T P A I S R L G S T V K T
B
S493
WT KKT2 Sequence
CGCGAGGAAGCCGCCCCTCGCACGTCTCGATCAGTGCGTCGTAGCGTCAGCTTAACGGAGCAGGAGCGGGGCAGACTT
R E E A A P R T S R S V R R S V S L T E Q E R G R L
T7Cas9
S^{WT}/S^{WT}
CGCGAGGAAGCCGCCCCTCGCACGTCTCGATCAGTGCGTCGTAGCGTCAGCTTAACGGAGCAGGAGCGGGGCAGACTT
R E E A A P R T S R S V R R S V S L T E Q E R G R L
Clone 3
S^{mut}/S^{mut}
CGCGAGGAAGCCGCGCCACGGACCTCACGTTCTGTCCGTCGTAGCGTATCTCTAACCGAACAAGAACGTGGCAGACTT
R E E A A P R T S R S V R R S V S L T E Q E R G R L
Repair Template
CGCGAGGAAGCCGCGCCACGGACCTCACGTTCTGTCCGTCGTAGCGTATCTCTAACCGAACAAGAACGTGGCAGACTT
R E E A A P R T S R S V R R S V S L T E Q E R G R L
C
S530
WT KKT2 Sequence
CGCGACACTGCCACTCGTTGGAACCTTCGCGCCGTAGTATCGCTGCCACGCGACATGACGGACGAGATCGAGCGCGAG
R D T A T R W N L R A V V S L P R D M T D E I E R E
T7Cas9
S^{WT}/S^{WT}
CGCGACACTGCCACTCGTTGGAACCTTCGCGCCGTAGTATCGCTGCCACGCGACATGACGGACGAGATCGAGCGCGAG
R D T A T R W N L R A V V S L P R D M T D E I E R E
Clone 4
S^{mut}/S^{mut}
CGCGACACTGCGACACGGTGGAATTTGCGGGCGGTTGTAAGTCTCCCTCGGGATATGACCGATGAAATTGAACGCGAG
R D T A T R W N L R A V V S L P R D M T D E I E R E
Repair Template
CGCGACACTGCGACACGGTGGAATTTGCGGGCGGTTGTAAGTCTCCCTCGGGATATGACCGATGAAATTGAACGCGAG
R D T A T R W N L R A V V S L P R D M T D E I E R E
D
S923
WT KKT2 Sequence
CCTCAGGTGCCACGCGGACGAGCACAGCAGCCACGTGCCCCCTCCGTCAGCGGGCACACCGCACAGGGCGGTCCGCCA
P Q V P R G R A Q Q P R A P S V S G H T A Q G G P P
T7Cas9
S^{WT}/S^{WT}
CCTCAGGTGCCACGCGGACGAGCACAGCAGCCACGTGCCCCCTCCGTCAGCGGGCACACCGCACAGGGCGGTCCGCCA
P Q V P R G R A Q Q P R A P S V S G H T A Q G G P P
Clone 10
S^{mut}/S^{mut}
CCTCAAGTGCCACGCGGACGAGCACAGCAGCCACGTGCGCCAAGTGTTTCGGGTCATACGGCTCAAGGTGGTCCGCCA
P Q V P R G R A Q Q P R A P S V S G H T A Q G G P P
Clone 11
S^{WT}/S^{mut}
CCTCAGGTGCCACGCGGACGAGCACAGCAGCCACGTGCGCCCTCCGTTTCCGGGCACACCGCTCAGGGTGGTCCGCCA
P Q V P R G R A Q Q P R A P S V S G H T A Q G G P P
Repair Template
CCTCAAGTGCCACGCGGACGAGCACAGCAGCCACGTGCGCCAAGTGTTTCGGGTCATACGGCTCAAGGTGGTCCGCCA
P Q V P R G R A Q Q P R A P S V S G H T A Q G G P P

Figure 9. Example sequencing results for mutant clones from transfections using double-stranded repair templates. (A) Example homozygous mutant clone from KKT2 S25S transfection. (B) Example homozygous mutant clone from KKT2 S493S transfection. (C) Example homozygous mutant clone from KKT2 S530S transfection. (D) Example homozygous and heterozygous mutant clones from KKT2 S923S transfection. In all panels, genotype, represented by the single-letter amino acid code encoded, is indicated below the cell line name. Superscript "WT" indicates the codon sequence for that residue is the same as the WT reference sequence, and superscript "mut" indicates the synonymously mutated sequence (KKT2 S493: WT – AGC, mutant – TCT; KKT2 S530: WT – TCG, mutant – AGT; KKT2 S25: WT – TCG, mutant – AGT; KKT2 S923: WT – TCC, mutant – AGT). Red arrows indicate protospacer sequences in WT sequences and equivalent position in repair templates; blue arrows indicate PAM sites in WT sequences and equivalent position in repair templates; yellow arrows indicate target site in WT sequences and mutated sites in repair templates. The translation is shown below each DNA sequence, with black text indicating the same protein and DNA sequence as the reference sequence, orange text indicating the same protein sequence but a different DNA sequence to the reference, and red indicating a difference in the protein sequence and hence DNA sequence.

template removes the restriction site. However, this change is isolated from the other mutations. In fact, there is another ~30 bp of homologous DNA between this base change and the remaining recoded sequence, so it is plausible that the cells used this region for recombination instead of the intended homology arms upstream of this base. This was not surprising, following previous results which showed recombination could occur with shorter stretches of homologous sequence (11 bp - Figure 6A and B). However, this result suggests that screening strategies should be designed to recognise larger regions of continuous sequence recoding, as single isolated base changes are not consistently integrated, and could lead to false negative results. It is plausible, that this was happening in the single-stranded oligonucleotide repair template transfections targeting KKT2 S923A and thus mutant clones were misidentified as WT because alternate homologous recombination had occurred that did not remove the expected restriction site.

4.4 Kinetochore Phosphosite Mutations Using Double-Stranded DNA Repair Templates

Phosphosites on kinetochore proteins (KKT2: S25, S493, S530 and S923; KKT4 S422; and KKT7 S304) were targeted for mutation to either alanine, glutamic acid or a synonymous alternate serine codon using 160 bp dsDNA repair templates. Following transfection, initially 12 clones were screened for integration of the repair template by PCR, followed by a further 12 clones if none were detected in the first batch (see Appendix 7.2.2 for agarose gel images). If PCR screening indicated a potential integration event of the repair template, a second PCR which amplified over the entire template region was completed, and this PCR product was sent for Sanger sequencing. Due to the transfection being selection-free, it was expected that clones could either be homozygous mutants (i.e. the repair template integrated successfully on both alleles), heterozygous mutants (i.e. the repair template only integrated on one allele, leaving the other with the WT/native sequence), or homozygous WT (i.e. the repair template failed to integrate on either allele leaving both alleles with the native sequence). Following the ssDNA transfections, it was also possible to find “complex” mutants but it was unclear how likely this would be.

Of the 18 different transfections, at least one homozygous mutant clone was identified in each transfection following Sanger sequencing, except for KKT2 S923S. This result in of itself suggests that this methodology has a vast improvement in efficiency in comparison to the other methods investigated in this project. One mutant clone within 12 suggests a minimum efficiency of 8.3%, compared to the less than 2% efficiency when using single-stranded DNA (Figure 6).

From these 18 transfections, 29.2% of clones screened showed integration of the repair template (Table 4). Overall, 21.7% of clones screened were homozygous mutants, with the remaining 7.5% of mutant clones being either heterozygous or complex mutants. These percentages were calculated using the repeat KKT2 S923S transfection where no mutant clones could be detected. As previous data suggested that is possible to generate this mutation, using the data from the previous transfection (Table 3) for this site instead increases the editing efficiency to 30.4% - 22.1% homozygous, 6.3% heterozygous and 2.1% complex. Either way, the overall editing efficiency was around 30%, with just over 20% of

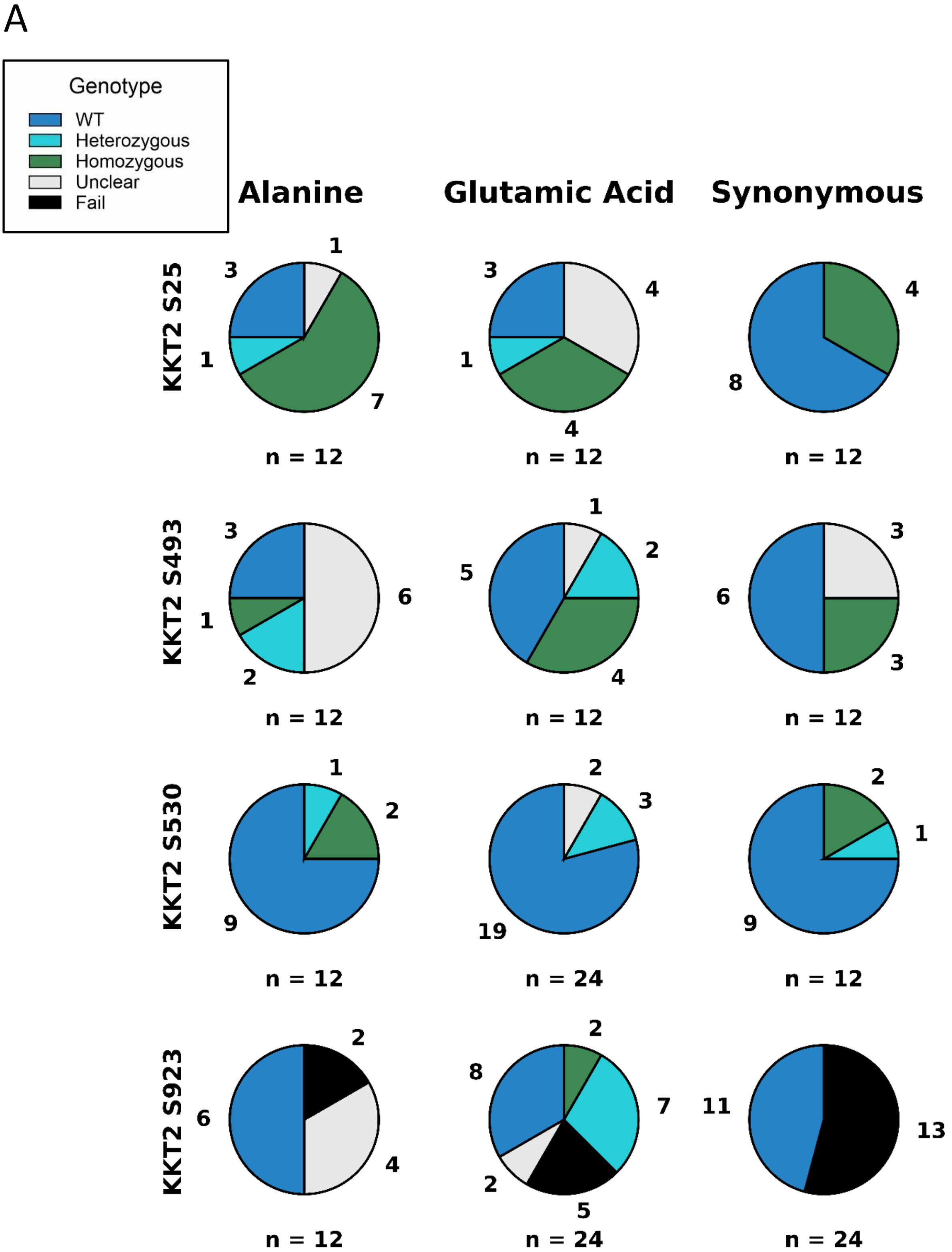
Table 4. Kinetochore phosphosite mutant genotypes summary as detected by PCR screen and Sanger sequencing. Percentages are of the total number of clones screened.

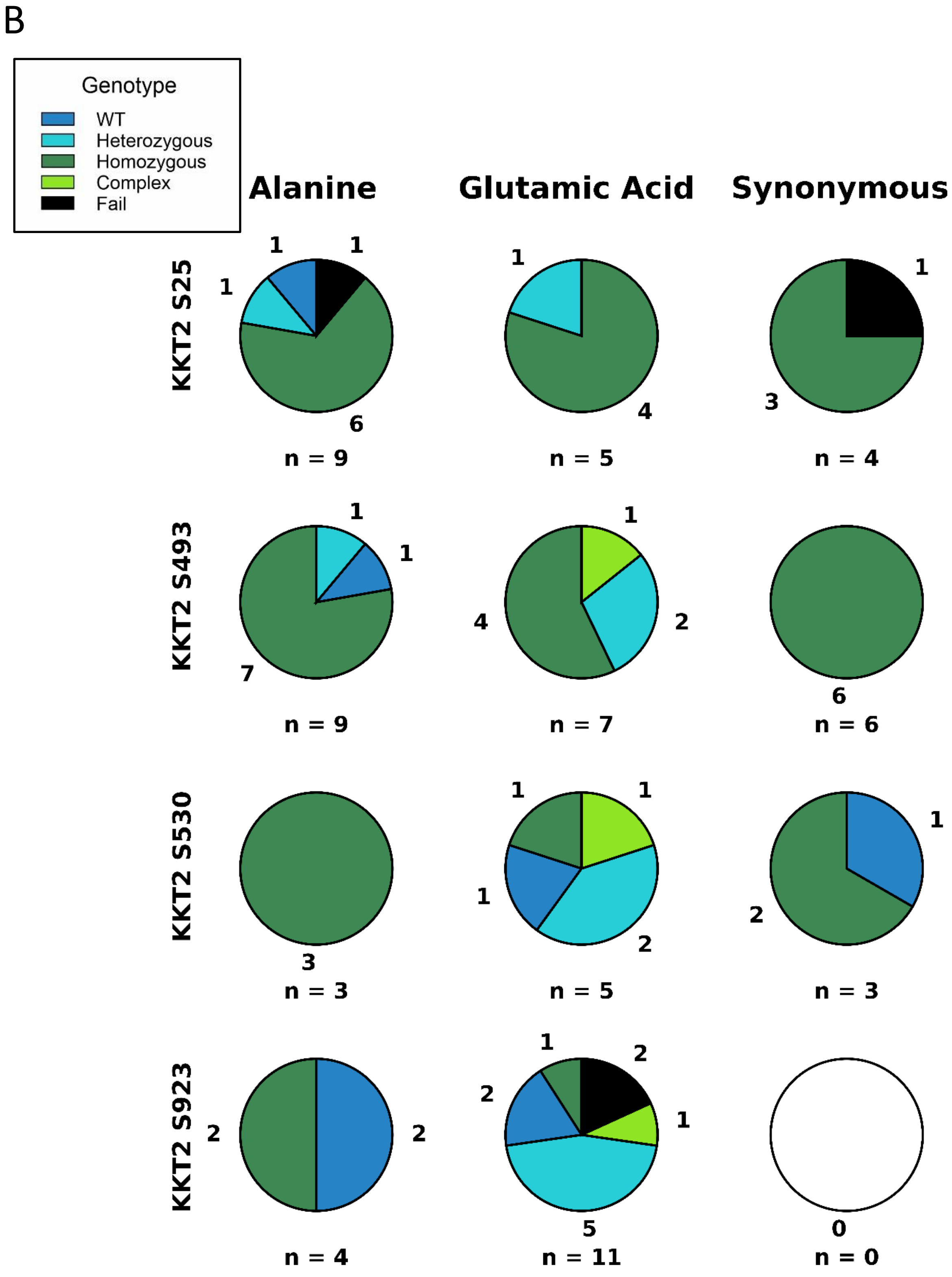
		WT (%)	Heterozygous Mutant (%)	Homozygous Mutant (%)	Complex Mutant (%)	Unclear Result (%)	Failed Result (%)	Total Assessed (%)
Kinetochore Phosphosite Mutants	PCR	122 (48.4)	32 (12.7)	30 (11.9)	-	43 (17.1)	25 (9.9)	252 (100.0)
	Sequencing	12 (5.0)	13 (5.4)	52 (21.7)	5 (2.1)	-	5 (2.1)	87 (36.3)
Synonymous KKT2 Mutants Only	PCR	33 (68.8)	5 (10.4)	7 (14.6)	-	3 (6.3)	0 (0.0)	48 (100.0)
	Sequencing	0 (0.0)	2 (4.2)	8 (16.7)	0 (0.0)	-	0 (0.0)	10 (20.8)
Combined	PCR	155 (51.7)	37 (12.3)	37 (12.3)	-	46 (15.3)	25 (8.3)	300 (100.0)
	Sequencing	12 (4.0)	15 (5.0)	60 (20.0)	5 (1.7)	-	5 (1.7)	97 (32.3)

cells becoming homozygous mutants. This efficiency is an notable improvement over the use of single-stranded DNA repair templates, improving by over 10-fold.

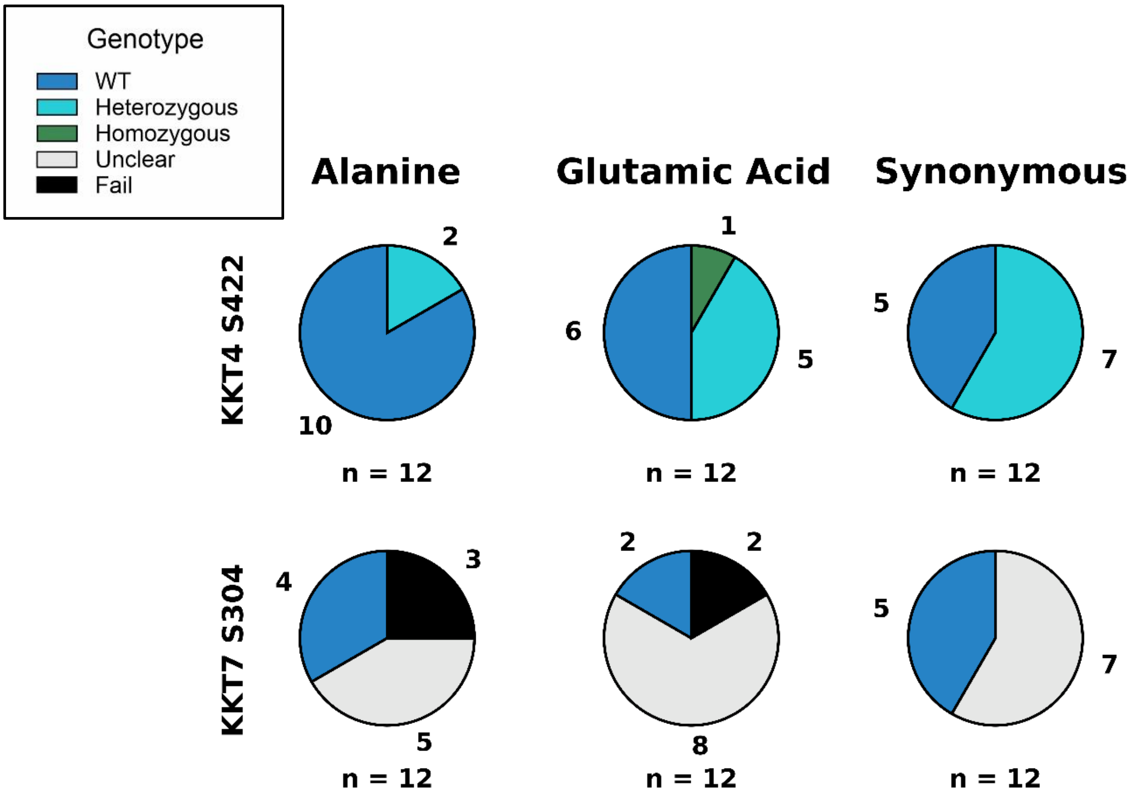
Efficiency varied between transfections, ranging from 1 homozygous mutant within 12 clones (KKT7 S304A, S304E and S304S) – 8.3% integration – to 7 homozygous mutants and 1 heterozygous mutant within 12 clones (KKT2 S493A) – 66.6% integration. Results for each repair template by both PCR and sequencing are in Figure 10. By PCR, KKT2 S25A had the highest percentage of mutations (66.6%). The largest number of mutants detected by PCR was 9 in KKT2 S923E, but as 24 clones were screened, this was equivalent to an editing percentage of 37.5%. In comparison, only one mutant was confirmed by Sanger sequencing in KKT2 S923A, KKT7 S304A, KKT7 S304E and KKT7 S304S.

PCR screening was not clearly able to identify mutants in the transfections for KKT2 S923A, KKT7 S304A, KKT7 S304E and KKT7 S304S. For KKT2 S923A, 2 clones exhibited a single PCR product with high intensity in the mutant PCR and several PCR products of varying intensity in the WT PCR at unexpected product sizes (clones 4 and 11 – Supplementary data in Appendix 7.2.2.2). The other two clones that were unclear had an intense single PCR product in the WT PCR reaction but a lower intensity single PCR product in the mutant PCR reaction (clones 6 and 12). Clones 4 and 11 were both identified by sequencing to be homozygous mutants, suggesting the banding pattern seen in the WT PCR was non-specific amplification of an unknown locus. Clones 6 and 12 were identified by sequencing to be WT, potentially suggesting some non-specific amplification of the WT locus. Similar to clones 6 and 12, a low intensity PCR product was present in the parental reaction when amplified with the mutant PCR primers. But it is unclear why this product was more prominent in these samples than the other WT clones. For the KKT7 mutations, the banding pattern of all the clones, except the parental, showed two PCR products in the WT reaction. Most clones also showed several PCR products in the mutant PCR reactions. These products generally appeared non-specific, but did not correlate with the non-specific products seen in the parental reaction with the same primer pair. However, one clone in each mutation had a PCR product at the same size as the most intense WT PCR product (at the expected size). These clones, as well as one clone with the nonspecific PCR product pattern in the mutant PCR reaction, were sent for sequencing. Only those with the correct size PCR


A
Genotype
WT
Heterozygous
Homozygous
Unclear
Fail
Alanine
Glutamic Acid
Synonymous
KKT2 S25
3
1
1
7
n = 12
3
1
4
4
n = 12
4
8
n = 12
KKT2 S493
3
1
2
6
n = 12
1
2
5
4
n = 12
3
6
3
n = 12
KKT2 S530
1
2
9
n = 12
2
3
19
n = 24
2
1
9
n = 12
KKT2 S923
2
6
4
n = 12
2
8
7
2
5
n = 24
11
13
n = 24


B
Genotype
WT
Heterozygous
Homozygous
Complex
Fail
Alanine
Glutamic Acid
Synonymous
KKT2 S25
1
1
1
6
n = 9
1
4
n = 5
1
3
n = 4
KKT2 S493
1
1
7
n = 9
1
4
2
n = 7
6
n = 6
KKT2 S530
3
n = 3
1
1
1
2
n = 5
1
2
n = 3
KKT2 S923
2
2
n = 4
1
2
2
1
5
n = 11
0
n = 0

C


Genotype
WT
Heterozygous
Homozygous
Unclear
Fail
Alanine
Glutamic Acid
Synonymous
KKT4 S422
KKT7 S304
2
10
n = 12
1
6
5
n = 12
5
7
n = 12
4
3
5
n = 12
2
2
8
n = 12
5
7
n = 12

D

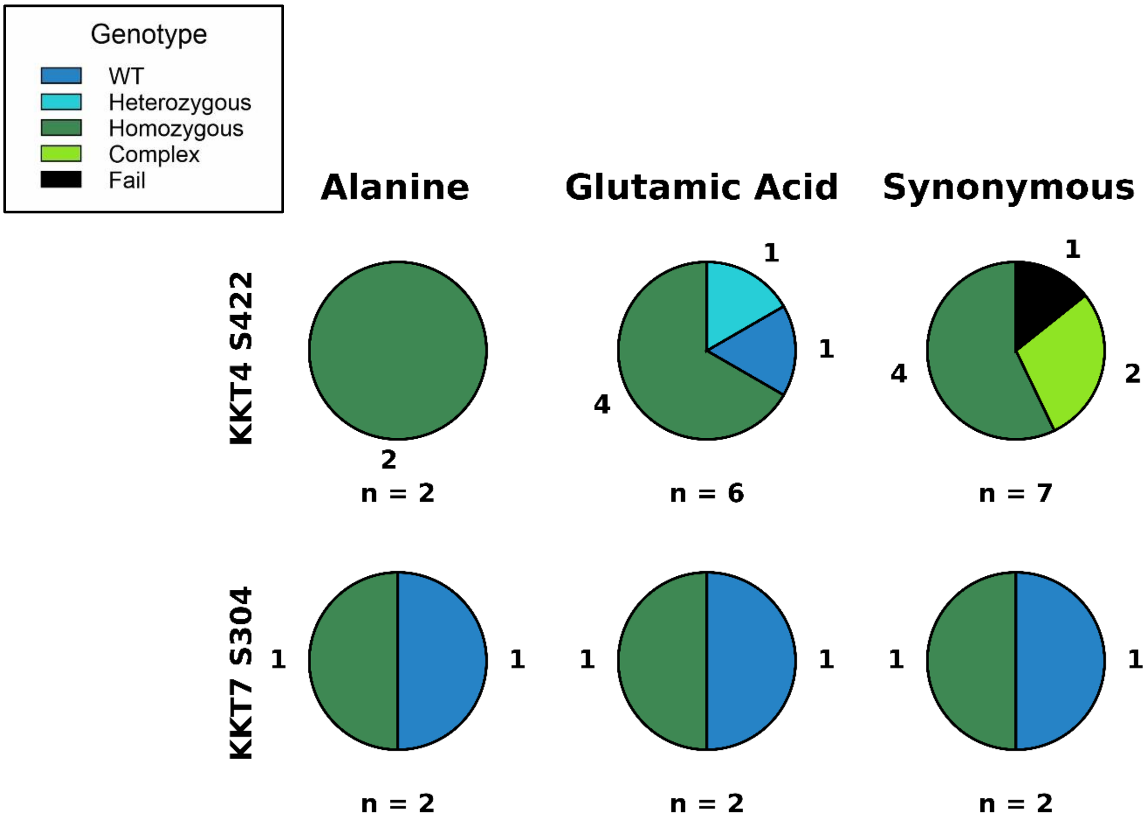

Genotype
WT
Heterozygous
Homozygous
Complex
Fail
Alanine
Glutamic Acid
Synonymous
KKT4 S422
KKT7 S304
2
n = 2
1
1
4
n = 6
1
4
2
n = 7
1
1
n = 2
1
1
n = 2
1
1
n = 2

Figure 10. Screening results of phosphosite mutant clones transfected with dsDNA repair templates. A) KKT2 PCR screening results. B) KKT2 Sanger sequencing results. C) KKT4 S422 and KKT7 S304 PCR screening results. D) KKT4 S422 and KKT7 S304 Sanger sequencing results. Target site is indicated on the left with the target mutation indicated at the top. Number of clones screened is indicated by the n number below each pie chart, with number of clones represented by each slice around the outside, adjacent to their slice. Genotypes in A and C: WT – PCR product was detected in the WT primer set reaction and not in the mutant set reaction; heterozygous – PCR product was detected in both WT and mutant primer set reactions with approximately equivalent intensity; homozygous – PCR product was only detected in mutant primer set reaction; Unclear – PCR product was detected in both WT and mutant primer sets with either differing intensity in each or additional unknown products; Fail – no PCR product was detected in either reaction. Genotypes in B and D: WT – both alleles match the reference sequence; heterozygous – one allele matched the reference sequence, one allele matched the repair template sequence (identified by dual peaks of similar height in the chromatogram); homozygous – both alleles match the repair template sequence; complex – evidence of integration of the repair template either to different extents on each allele, or with unexpected mutations (see main body for more details); Fail – the sequence was unable to align with either the reference sequence or the repair template sequence.

product were mutated (S304A clone 9, S304E clone 12 and S304S clone 5 – Appendix 7.2.2.2).

It was unexpected that KKT2 S923S did not yield a mutant clone within 24 clones when previous work had shown this was possible with the same repair sequence and sgRNA sequences (Figure 9D). It is likely that this result was caused by a technical issue with this transfection and/or screening process. Out of the 24 clones screened, 13 did not generate a PCR product in either the WT or mutant PCR reactions, suggesting a general issue with the collection of DNA, as the WT PCR on the parental DNA (which was from a different DNA extraction) worked as expected and all the reactions shared the same PCR mastermix.

In addition to the expected genotypes of homozygous and heterozygous incorporation, sequencing revealed that 5 clones had integrated the repair template in an unexpected way and have been called “complex”. These were one clone in KKT2 S493E, KKT2 S530E, and KKT2 S923E transfections; as well as 2 clones in the KKT4 S422S transfection. KKT2 S493E clone 9 showed homozygous incorporation of the entire repair template, except it had complete loss of the codon encoding E496, which is in the middle of a recoding region. KKT2 S530E clone 19 seemed to show homozygous incorporation of the entire repair template, except the target codon which showed secondary peaks. The secondary peaks were not as high as the main peaks of the chromatogram, but notably higher than background. In addition, a few other mutated residues also seemed to have background peaks corresponding to the WT bases. This could either be indicative of the presence of a WT copy as well as two mutated copies, or could be suggestive that the cell line was not clonal. KKT2 S923E clone 22 incorporated the entire repair template in a homozygous manner, except the first base of H927 which was heterozygous. This led to a C->G transformation on one allele, causing a mutation to aspartic acid. KKT4 S422S clones 7 and 12 showed an identical genotype, incorporating the repair template in a homozygous manner for the 3’ region of recoding (where the target S422 is) but a heterozygous incorporation of the repair template in the 5’ recoded region.

On the assumption that the editing efficiency of this method is around 30%, when all mutant genotypes are taken into consideration, it is possible to predict the likelihoods of detecting a given number of mutant clones in the future. A binomial distribution can show the probabilities of detecting *x* number of mutant clones when screening 12 clones, to determine if 12 is a suitable number of clones to screen. The binomial distributions for a range of editing efficiencies are shown in Figure 11A. For an editing efficiency of 30%, the most likely scenario is that 3 or 4 mutant clones are detected, but it is much less likely that 7 or more mutant clones are detected. By using the cumulative frequency of these probabilities (Figure 11B), it is possible to infer that at 30% editing efficiency, there is a 90% probability that up to 5 mutant clones are detected when screening 12 clones. When compared with the *in vitro* data (Figure 11C), it is apparent that the detection of larger numbers of clones is not very likely at this editing efficiency, but is more likely at a higher editing efficiency of 40%. In comparison, the large number of transfections with only 1

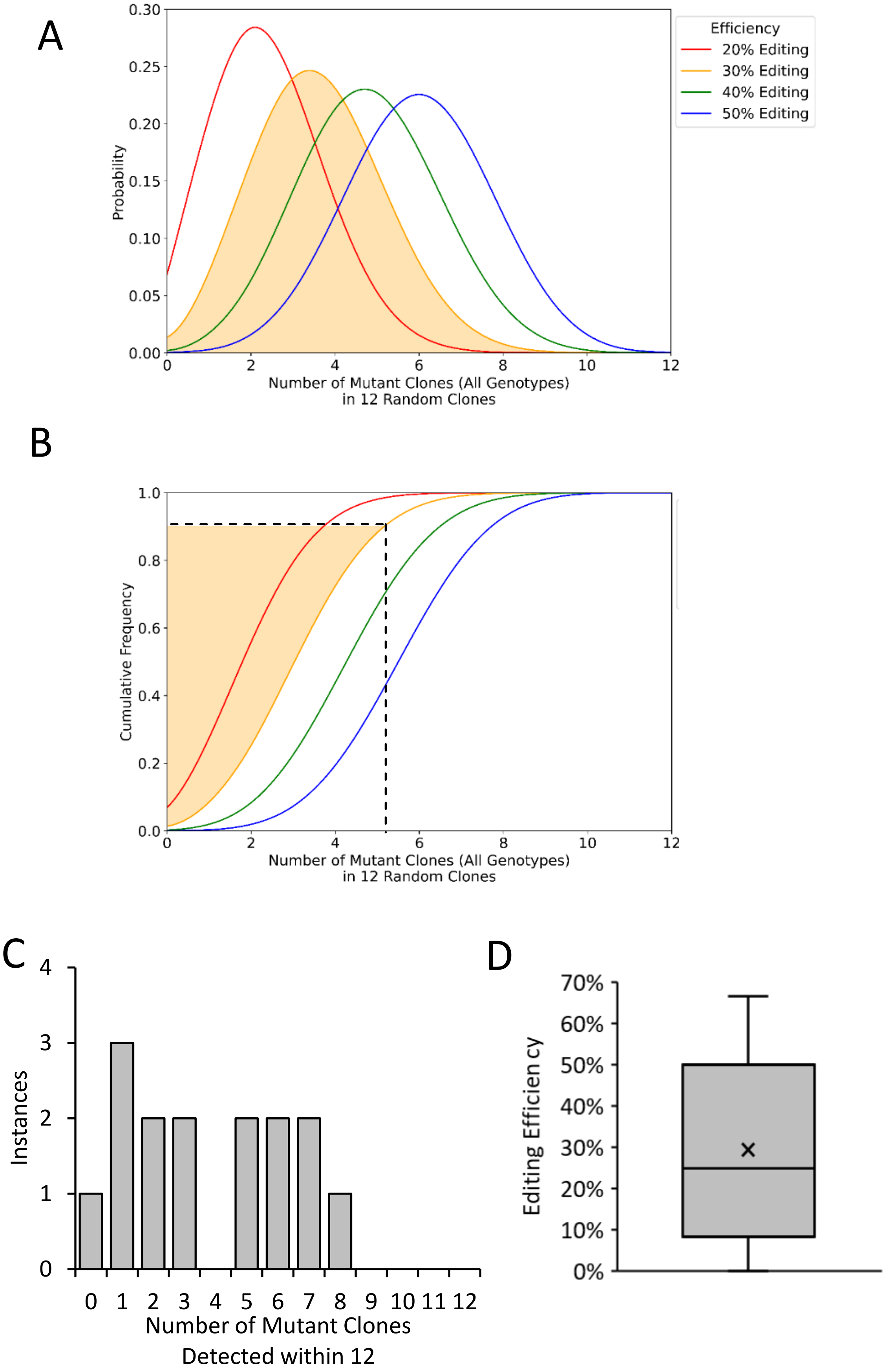

A
0.30
0.25
0.20
0.15
0.10
0.05
0.00
Probability
Efficiency
20% Editing
30% Editing
40% Editing
50% Editing
0
2
4
6
8
10
12
Number of Mutant Clones (All Genotypes)
in 12 Random Clones
B
1.0
0.8
0.6
0.4
0.2
0.0
Cumulative Frequency
0
2
4
6
8
10
12
Number of Mutant Clones (All Genotypes)
in 12 Random Clones
C
4
3
2
1
0
Instances
0 1 2 3 4 5 6 7 8 9 10 11 12
Number of Mutant Clones
Detected within 12
D
70%
60%
50%
40%
30%
20%
10%
0%
Editing Efficien cy

Figure 11. Mathematical analysis of precision editing efficiencies. A) Binomial distribution of the theoretical probabilities of identifying several mutant clones within 12 randomly selected clones for all mutant genotypes combined at a range of editing efficiencies. The theoretical proportion of mutant cells is given in the legend (right). This editing efficiency is a sum of all homozygous, heterozygous and complex mutants. 30% editing efficiency is shaded for clarity, given that the calculated editing efficiency *in vitro* was around 30%. Whilst the probability of finding 8 or more clones appears to be 0 under some conditions, it is never truly 0 but as low as 4 x 10^{-9} for the 20% mutants line (red). B) The cumulative frequency distributions for the same editing efficiencies as in part A. The dotted line indicates the point at which there is a 90% chance for detecting the respective number of clones (approximately 5) or fewer mutant clones with a 30% editing efficiency. This area is highlighted with orange shading. C) A histogram of the frequencies at which several kinetochore mutant clones (all genotypes) were detected when 12 clones were screened. Transfections where 24 clones were screened were omitted. D) Box and whisker plot of the editing efficiencies of all the kinetochore mutations (all genotypes). The mean is indicated with a cross.

detected mutation is more similar to the trend seen when there is only a 20% editing efficiency. This sample size is very small, so drawing conclusions is challenging, but it does appear that there is a split, with some transfections fitting a 20% editing efficiency and some fitting a 40% efficiency (Figure 11C). If a transfection had a high editing efficiency, its sister transfections (i.e. the alternate amino acid replacement but the same target codon) had similarly high efficiency. The same is true of the inverse, i.e. that if one design had poor efficiency then its sister designs also had poor efficiency. For example, KKT2 S25A detected a total of 7 mutant clones within the 12 screened, and S25E and S25S had 5 and 3 respectively. Only one mutant clone was detected in KKT7 S304A, and similarly in S304E and S304S. This would suggest that there is something inherent about either the synonymous recoding in the repair template or sgRNA design that is impacting the integration, as these features are shared between them. Further analysis into the repair template and sgRNA designs is needed to identify the trends, and a larger sample size is needed to infer the “true” editing efficiency with more confidence.

4.5 Growth Analysis of Kinetochore Phosphosite Mutants

As the kinetochore phosphosite mutant clones did not appear to exhibit any visual morphological defects during normal passage and growth, it was hypothesised that the growth rate could be impacted by the phosphosite mutations generated. To assess this, kinetochore phosphosite mutants were grown to mid-log phase, and then 500 cells were loaded into a well of a 96-well plate. After 5 days, the growth of the culture was measured by Alamar blue assay using the fluorescence. A control plate (a duplicate of the 5-day plate to control for inaccurate loading of the small number of cells) was also set up and measured in the same way on day 0, but most of the data points when adjusted for the background were below zero, so this data has not been shown. Two clones were selected at random from the homozygous mutant clones. When two clones were not available, but other non-homozygous clones were available, a second clone was chosen from the available clones. These were KKT2 S530E clone 10 which had a heterozygous genotype at the point of cryostorage (although genotype was not reassessed after thawing), and KKT2 S923E clone 22 which had a heterozygous H927D mutation as well as homozygous S923E.

KKT7 S304A clone 9 was the only cell line to show a significantly different rate of growth, when normalised to T7Cas9. KKT7 S304A clone 9 grew faster than the parental, with a mean growth of 176.3% growth at day 5 compared to the parental - Figure 12.

Whilst no other mutants showed significantly different growth, some clones showed increased growth. Most of the clones that showed an increased growth rate were synonymous mutant clones. The largest of these were KKT2 S530S clone 7, KKT2 S923S clone 10, KKT2 S422S clone 4 which had mean growths of 130.0%, 155.5% and 149.7% respectively. In contrast, the clones that showed a non-significant decrease in growth were mostly KKT2 S25 mutants. KKT2 S25A clone 5, KKT2 S25S clone 5 and KKT2 S25S clone 10 showed decreased growth rates of 83.0%, 83.3% and 79.3% respectively. In addition, KKT2

S493A clone 10 and KKT2 S493E clone 6 also showed reduced growths of 84.0% and 83.3% respectively.

Whilst only KKT7 S304A clone 9 showed a statistically significant growth change, it is apparent that growth rates between clones of the same genotype did not appear to grow similarly. This could be caused by other unknown genetic differences between the clones such as off-target effects or pre-existing genetic diversity from the parental population. However, further investigation is required to assess these differences.

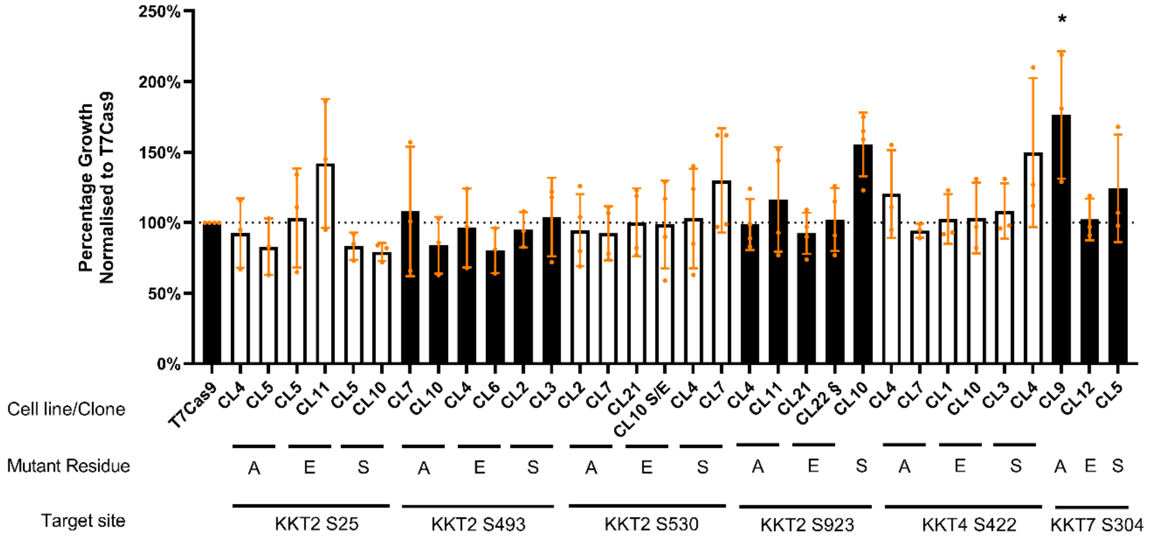


Figure 12. Alamar blue growth assay of kinetochore phosphosite mutants following 5 days of growth. CL = clone. Colours of bars indicate target site groupings. KKT2 S530E clone 10 is a heterozygote for the S530E mutation, indicated by S/E. KKT2 S923E clone 22 had a heterozygous H927D mutation and is indicated by §. Error bars indicate the standard deviation. * is p <0.05. n = 3 for all cell lines except KKT2 S493 and KKT2 S530 mutants which were n = 4.

4.6 Cell Cycle Analysis of Kinetochore Phosphosite Mutants

To assess whether phosphosite mutations influenced cell cycle progression, the quantity of DNA in each cell was measured using propidium iodide flow cytometry in mid-log phase cultures. The proportion of cells in each cell cycle stage was assessed, as well as looking for anomalies in DNA content.

When assessing each cell cycle stage independently, most mutants were not significantly different to the parental T7Cas9 cell line (Figure 13A). Only KKT2 S25S clone 5 showed a significant increase in the number of cells in G1 compared to T7Cas9, but neither S-phase nor G2/M cells showed a significant difference in proportion (Figure 13A). The other KKT2 S25S clone (clone 10) did not mirror this difference. However, it should be noted that only one of the three replicates showed a notable difference in the percentage of G1 cells of 66.5%, whereas the other two were 45.0% and 41.1% respectively.

KKT2 S25E clone 11 and KKT2 S530E clone 21 had to be omitted from this analysis because of the presence of an additional peak with a greater fluorescence than the G2/M peak which the DNA content model fitting was unable to process (Figure 14A). For KKT2 S25E clone 11, this peak was consistent throughout all the replicates, and represented about 4% of the cells using the rough gating shown in Figure 14A. For KKT2 S530E clone 21, this peak grew in proportion with each replicate and passage. Initially, it started similarly to KKT2 S25E clone 11 at around 4.8% but continuously grew in proportion, containing around 13.7% of the cells after several passages. In addition, the apparent G1 peak dropped in proportion with this high intensity peak's increase, starting at around 41% of cells and dropping to 4.6% of cells in the final replicate. Based on the intensity of these additional peaks in both cases, they likely represent a proportion of cells which have become triploid rather than tetraploid.

Following identification of the triploid population in KKT2 S530E clone 21, a fresh batch of cells were thawed from cryostorage, and spit into 4 subpopulations which were grown independently. The flow cytometry analysis was repeated on each subpopulation as before, but the phenotype was not replicated, and showed a normal cell cycle distribution across all replicates (Figure 14B). On receiving this result, it was concluded that repeating KKT2 S25E clone 11 was likely to yield the same result, so was not repeated.

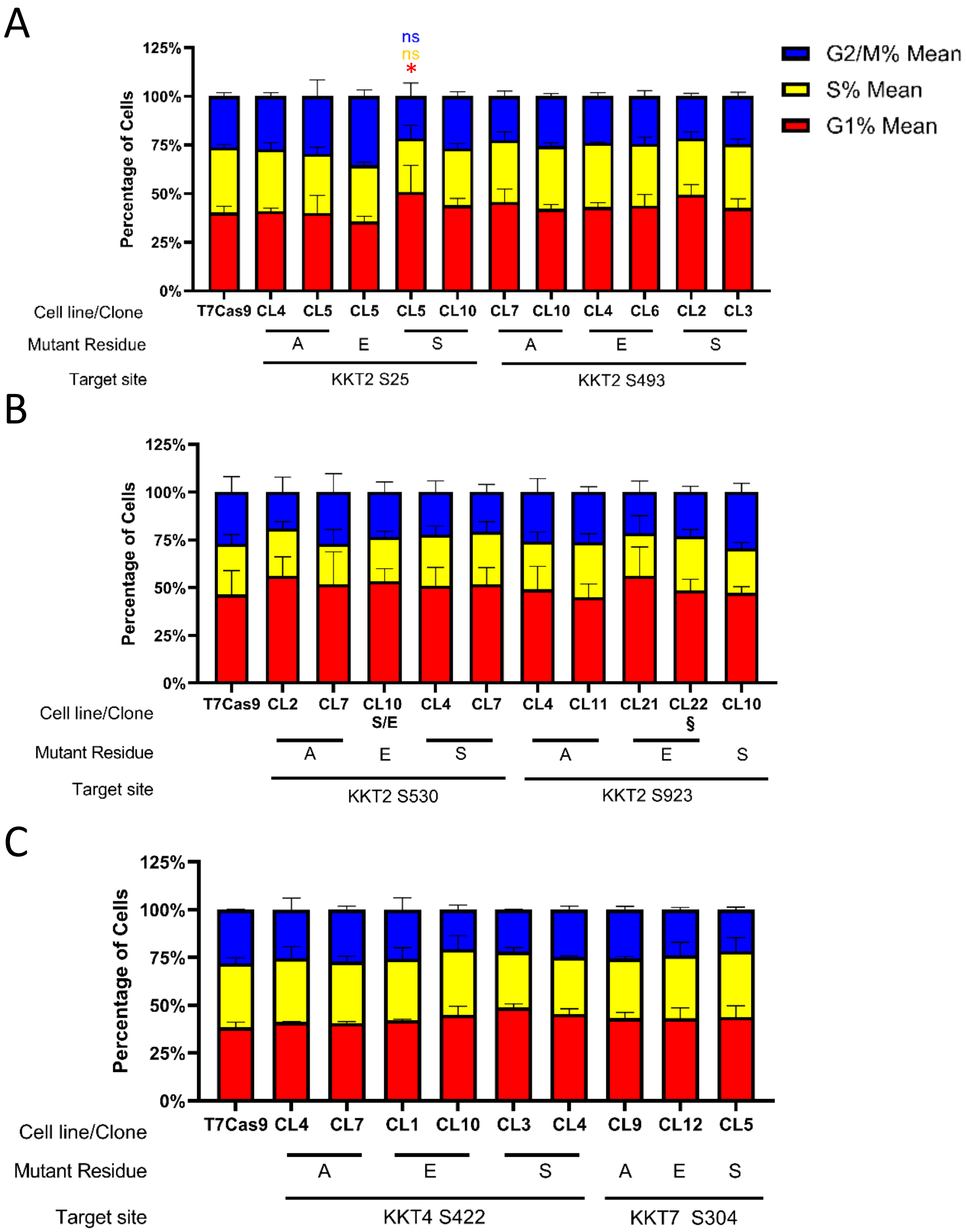
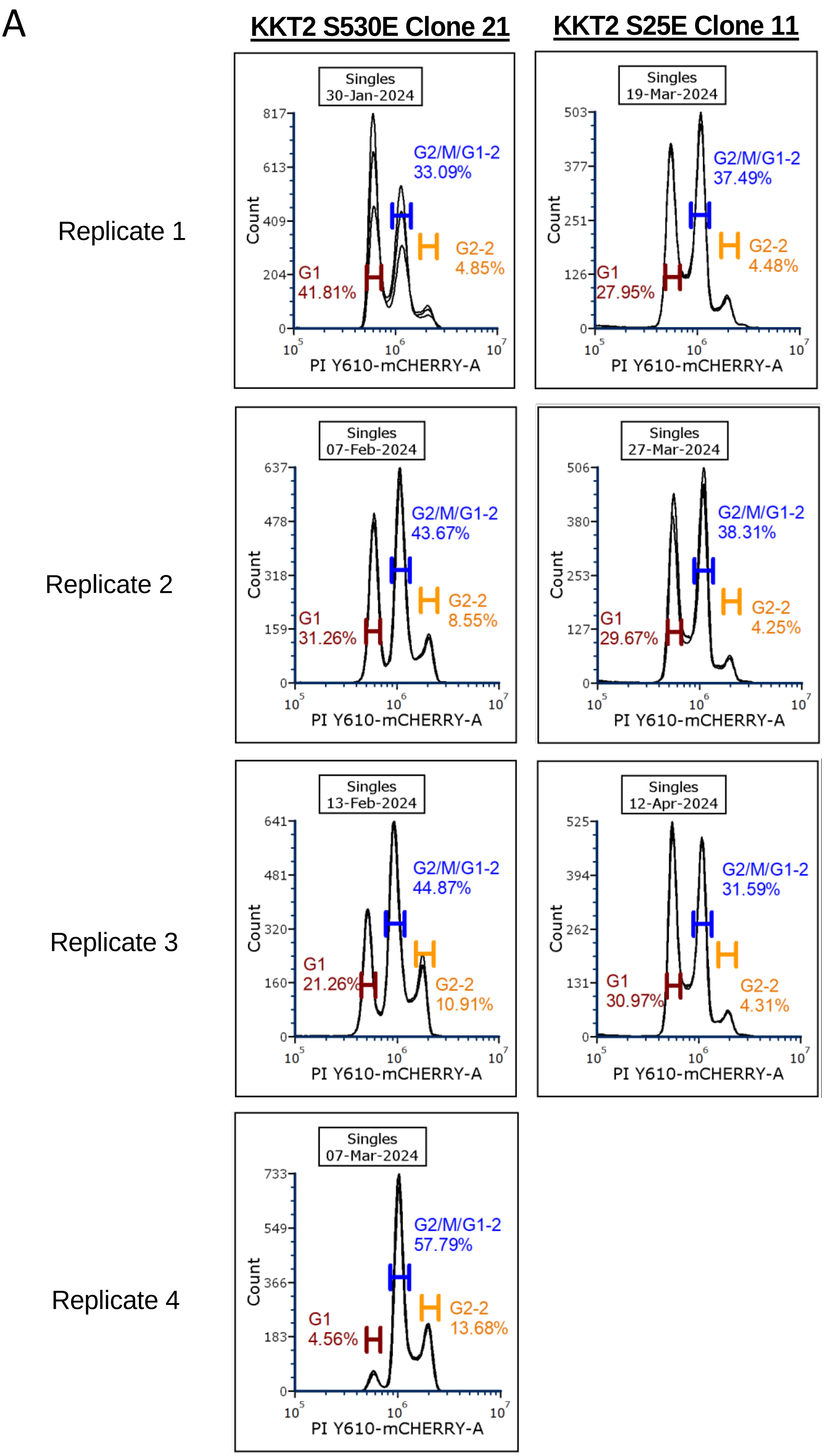


Figure 13. Cell cycle analysis of mid-log phase cultures of kinetochore phosphosite mutants. A) KKT2 S25 and KKT2 S493 mutants, n = 3. B) KKT2 S530 and KKT2 S923 mutants, n= 4. KKT2 S530E clone 10 is a heterozygote for the S530E mutation, indicated by S/E. KKT2 S923E clone 22 had a heterozygous H927D mutation and is indicated by §. C) KKT4 S422 and KKT7 S304 mutants, n =2. * is $p < 0.05$. Error bars indicate the standard deviation in all panels. CL = clone in all panels.


A
KKT2 S530E Clone 21
KKT2 S25E Clone 11
Replicate 1
Singles
30-Jan-2024
817
613
409
204
0
Count
G2/M/G1-2
33.09%
G2-2
4.85%
G1
41.81%
10^5
10^6
10^7
PI Y610-mCHERRY-A
Singles
19-Mar-2024
503
377
251
126
0
Count
G2/M/G1-2
37.49%
G2-2
4.48%
G1
27.95%
10^5
10^6
10^7
PI Y610-mCHERRY-A
Replicate 2
Singles
07-Feb-2024
637
478
318
159
0
Count
G2/M/G1-2
43.67%
G2-2
8.55%
G1
31.26%
10^5
10^6
10^7
PI Y610-mCHERRY-A
Singles
27-Mar-2024
506
380
253
127
0
Count
G2/M/G1-2
38.31%
G2-2
4.25%
G1
29.67%
10^5
10^6
10^7
PI Y610-mCHERRY-A
Replicate 3
Singles
13-Feb-2024
641
481
320
160
0
Count
G2/M/G1-2
44.87%
G1
21.26%
G2-2
10.91%
10^5
10^6
10^7
PI Y610-mCHERRY-A
Singles
12-Apr-2024
525
394
262
131
0
Count
G2/M/G1-2
31.59%
G1
30.97%
G2-2
4.31%
10^5
10^6
10^7
PI Y610-mCHERRY-A
Replicate 4
Singles
07-Mar-2024
733
549
366
183
0
Count
G2/M/G1-2
57.79%
G2-2
13.68%
G1
4.56%
10^5
10^6
10^7
PI Y610-mCHERRY-A

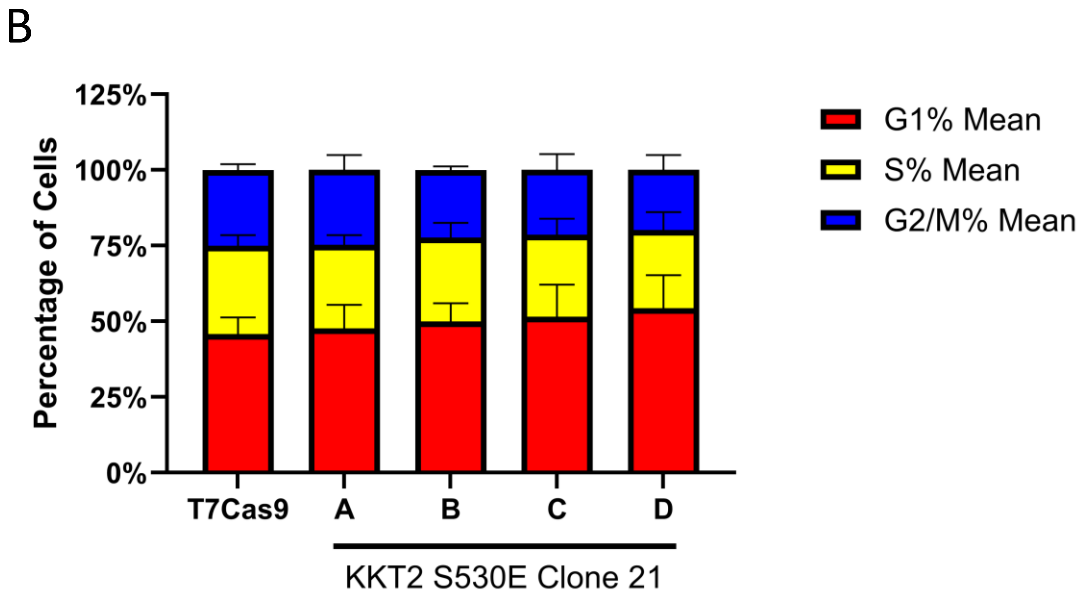


Figure 14. Cell cycle analysis of KKT2 S530E clone 21 and KKT2 S25E clone 11. A) Histograms of propidium iodide intensity for each cell line. In each biological replicate (indicated on the left), three technical replicates are plotted as individual lines. Gates used are identical in width but have been repositioned to fit the exact intensity of the peaks for each biological replicate. Percentages correspond to one representative technical replicate. B) Cell cycle analysis of KKT2 S530E clone 21 repeat when split into four subpopulations (A to D). Error bars indicate the standard deviation.

4.7 Discussion

It is clear from these results that precision editing is possible in *Leishmania mexicana*, but that the methodology used plays a large factor in the success of generating mutants. Broadly speaking, two approaches were investigated in this project – the use of single-stranded DNA (ssDNA) repair templates and double-stranded DNA (dsDNA) repair templates. Single-stranded repair templates were inconsistent in the generation of mutants, and efficiency seemed to be low. The presence of the integrated repair template was demonstrated in populations (Figure 7), but not in clones in most cases (Table 2 and Figure 6). In contrast, double-stranded repair templates were far more consistent in the generation of mutations (Table 4, Table 5, Figure 9, and Figure 10). Based on the frequency that mutated clones were identified, double-stranded repair templates generated a higher proportion of mutant cells within a given population than the single-stranded oligonucleotide repair templates. Other than the "stranded-ness" of the repair templates, the only major differences between the methods were the increases in the length of the homology arms, the inclusion of 6-biopterin in the recovery media, and the larger quantity of sgRNA in the transfection. Whilst it cannot be ruled out that these factors were of more or equal importance than whether the repair template was single- or double-stranded, it seems suggestive that the change in "stranded-ness" is a key factor (Table 5). Whilst the experiments completed in this project are not able to determine why the type of DNA had

Table 5. Comparison of the success of transfections using single-stranded or double-stranded repair templates.

Repair Template Type	**Successful Mutant Generation Out of Total Independent Attempts to Generate Mutants**	
	Clones	*Population*
Single-stranded	3/16 (18.8%)	18/20 (90.0%)
Double-stranded	21/22 (95.5%)	-

such an impact, perhaps it is suggestive that the initiation of the DNA damage response is different between single- and double-stranded repair templates. Further investigation into this process in *Leishmania* may enable even greater improvements in editing efficiency.

Whilst the type of DNA was evidently crucial in generating the desired mutations, it was interesting to reveal that the recoding strategy had little effect on the outcome. The variability in the success of transfections using the ssDNA repair templates was initially attributed to some aspect of the design. To investigate that hypothesis, two targets were chosen, and five repair templates were designed for each of them. These repair templates had a range of different recoding strategies such as using different codons, differences in spacing and quantity of mutations (Table 1). Unexpectedly, almost all the repair templates designed were detected by PCR (Figure 7). It should be noted that the sequences of these PCR products were not determined. However, on the assumption that each PCR exclusively detected the intended repair template, and with the additional data from the dsDNA repair transfections which used the same recoding as some of the ssDNA repair transfections, it is clear that the success of the transfection is not primarily linked to the recoding strategy used in the repair template design. This suggests that there is scope to recode the sequence in ways which enhance usability such as making screening simpler. That being said, this finding should be taken with caution because each of the repair templates tested here only induced ~10-15 SNPs in regions of about 60 bp of one gene in any given cell. Whilst studies in the literature indicate that translation rate is constant between sequences from across the whole genomes of higher eukaryotes, which inevitably vary in composition (Burchmore and Landfear, 1998; Brittingham et al., 2001; Villa et al., 2003; Beetham, Donelson and Dahlin, 2003), there is evidence to suggest that translation rate is dependent on the codons used in trypanosomes (Jeacock, Faria and Horn, 2018; Nascimento et al., 2018). Whilst the repair templates used here were small, it is likely that for larger scale recoding such as whole genes, there will be more consequences on the translation rate and hence health of the cell for diverging from the native sequences. As such, it is advisable to generate synonymous mutant control lines to help distinguish fitness effects caused by the recoding from those caused by the target-mutation, and to choose similarly used codons where possible.

When used cautiously, having the flexibility to choose any synonymous codon has great benefits for designing suitable screening approaches. The *Leishmania* genome has a high GC content of around 59% (Ivens et al., 2005; Peacock et al., 2007; Rogers et al., 2011; Chauhan, Vidyarthi and Poddar, 2011) compared to humans which has an average of around 41% (Lander et al., 2001). This can make primer design extremely challenging in the small regions of interest for precision editing, which can have even higher local GC content. Whilst there are approaches that can be used to amplify high GC content templates, increasing the chance of successful screening on the first attempt has several benefits. Failure to screen on the first attempt can lead to repeated passaging of the cells which can reduce the virulence, as well as requiring more media and consumables which increases the costs. To improve this technique in the future, it may be beneficial to intentionally recode regions with high GC content to increase the AT content (i.e. lower GC content), since editing efficiency does not appear to be impacted. Reducing GC content where the screening primer binds will lower the annealing temperature required during screening PCRs which could reduce instances of unclear genotyping from unexpected PCR results. Additionally, reducing the necessary melting temperature of the screening primer allows a wider range of annealing temperatures to be tested, should initial screening lead to unclear results. As such, repair template design and screening-primer design should be completed in parallel, to ensure annealing temperatures for primers are convenient for use. In addition, ensuring the 3’ base of the screening primer differs between WT and mutant sequence for that region can help to ensure specificity. In the KKT7 S304 PCR screening reactions, the 3’ base of the WT-specific primer was shared between the WT sequence and all the repair templates. The KKT7 S304 transfections had the most uncertain PCR products, with between 5/12 and 8/12 clones generating unclear PCR products each. It is plausible that the WT-specific primer was able to bind sufficiently to both WT and mutant DNA and allow amplification to occur, creating the unclear results. In these PCR reactions, most clones (but not parental DNA) produced two bands in the WT-specific PCR reaction, both of which were close to the expected product size (Appendix 7.2.2.2). This made interpretation of the results harder, as only one clone in each transfection had a single PCR product. Sequencing of clones with single-PCR products revealed they were homozygous mutants, indicating the product that was absent in their reactions was the real WT PCR product. Situations like this demonstrate that it is helpful to incorporate a back-up screening approach into the design such as incorporating a change in common restriction site in the region of interest.

Using a restriction digest strategy as the primary form of screening method was also tested in this project. In most cases, restriction digest tended to be more predictive of the genotype of the clone being assessed than PCR screening, following sequencing. This is with the caveat that this was only true when restriction sites were located in the same continuous stretch of recoded sequence as the mutation of interest. Restriction sites that were located further away from the target site, particularly in cases where there was a break in the recoding, were not good at predicting genotype. This was demonstrated by the presence of clones with a complex genotype, where the repair template was not always integrated in its entirety. A continuous region of recoding tended to be incorporated together, with only occasional failure to incorporate mutations at the end of a series of synonymous mutations. As such, it is highly plausible that clones with partial repair template integration were misidentified as WT if the sequence conferring the restriction site change was not integrated. Primers recognise a much longer sequence in most cases, so had higher chances of detecting a partial integration of the repair template than restriction enzymes which often only recognise a 6 bp long sequence. However, screening by PCR tended to be less accurate and dependent on how specifically the primer bound to either the WT or the mutant sequence. Several clones transfected with dsDNA indicated the presence of a mutant allele by PCR but were identified as WT by Sanger sequencing (5.0%). However, PCR screening was more convenient, and cheaper too. A 5.0% false-positive rate was acceptable to have, since true positive results were also detected. However, it should be considered that of this 5.0% of clones, some PCR results were more suggestive of a particular genotype than others. 17.1% of clones screened by PCR that were transfected with dsDNA had an uncertain genotype following PCR screening (Appendix 7.2.2.3). These included situations where both WT and mutant PCRs had bands but of unequal intensity, or the banding pattern in one or both PCRs were not as expected. As such, some of these PCRs seemed more likely to suggest that the clone was WT and had not integrated the repair template but the result left enough doubt to warrant sequencing. As such, it is not necessarily fair to say that all of these were strict false positives and suggests that the rate of misidentifying WT cells as a mutant genotype is likely less than

5.0%. However, there were instances where PCR screening misidentified mutant genotypes, such as misidentifying a homozygous mutant as a heterozygote. So taking 5.0% as an overall inaccuracy rate of PCR detection is reasonable.

Both in the use of ssDNA and dsDNA, integration of the repair template was not always complete or perfectly faithfully on both alleles. These clones have been designated as "complex" genotypes. Most frequently, "complex" clones showed faithful inclusion of about half of the repair template, typically when there was a break in the recoding in the centre of the repair template. It is likely that in these cells, the WT sequence in the centre of the repair template was used for recombination rather than the intended homology arms. The smallest breaks in recoding were only 11 nt long. At this length of homology, it is more likely that the microhomology-mediated end joining pathway is activated, which can use regions of 5-25 nt in length for double-stranded DNA break repair (Zhang and Matlashewski, 2019). Alternatively, it is possible that this genotype was caused by a failure to induce both double-stranded breaks on one allele by the Cas9 nuclease. In all the repair templates tested, the editing range was only 60 bp. The Cas9 nuclease is 160 kDa, approximately 10 nm x 10 nm x 5 nm in size (Josephs et al., 2015). Cas9 recognises and binds a region of 20 bp (the protospacer), but given its size, it is probable that in some (if not all) of the designs tested here, two Cas9 molecules would not be able to bind to the genomic DNA at the same time to make both breaks due to their proximity. Even though two Cas9 molecules would be unable to make both breaks simultaneously, it is still beneficial to have both sgRNAs. For example, one sgRNA may have better activity than the other, the different break sites may stimulate different repair responses from the cell, or the second may be broken after the first. In some cells, it is plausible that only one dsDNA break occurred, which increased the probability of recombination happening at a recoding break in the repair template. Little is known about the specific factors involved in homologous recombination in *Leishmania* species (Kelso et al., 2017), despite its presence having been demonstrated around 30 years ago (Tobin, Laban and Wirth, 1991). RAD51 (a recombinase that is active during mitosis to repair DNA damage faithfully) is present in *Leishmania* and has been shown to respond to DNA damage (Kelso et al., 2017), as well as having roles in DNA replication (Damasceno et al., 2020). BRAC2 is also present and similar to other organisms, is responsible for localisation and loading of RAD51 onto sites of DNA damage (Kelso et al., 2017). Further research into the other factors involved will hopefully shed light on the most effective way to stimulate the desired form of integration.

Additionally, there were “complex” clones which contained an unexpected change to the amino acid sequence. It is apparent that these nonsynonymous mutations are unlikely to be compensatory, as at least one clone was identified in each transfection with the designed repair template faithfully integrated on both alleles. It is possible that these mutations help the cell manage the induced mutation, but they are clearly not required to survive. It is more likely that these events either stem from unfaithful DNA repair by the cell, or unfaithful production of the repair template (either during oligonucleotide production or during PCR amplification by the polymerase). Unfortunately, with the present data, it is not possible to determine the cause of these mutations.

Of the attempts to use double-stranded repair template that failed to generate mutants, only the repeat of KKT2 S923S failed to identify mutant clones (Figure 10A and B). As this mutation was previously generated in this project (Figure 9D), this result is suggestive that there was a technical issue with this transfection and/or screening process. Whilst there is some evidence to suggest there were technical issue with the DNA extraction or PCR screening step, an alternate hypothesis is that the transfection efficiency dropped because of varying quality and quantity of DNA used. For the second replicate of KKT2 S923S using dsDNA, DNA was prepared in parallel to DNA for other KKT2 transfections in a single PCR plate, but S923 was the last to be used from this. As such, the plate was carefully opened, thawed and re-frozen several times prior to transfection. Doing so left the volumes in these wells to be slightly lower than expected when preparing them for transfection (likely due to evaporation), and perhaps caused some amount of DNA degradation. This could explain why the efficiency, which was 25% when only testing synonymous mutants with freshly made DNA, dropped to 0%.

Whilst synonymous controls do not directly indicate whether a non-synonymous mutation is possible, it is clear from this report that their incorporation as a control in parallel is of great help in determining the reasons for failure to isolate the mutation of interest. Additionally, synonymous mutations are useful as controls in experiments that investigate the effect of the mutation of interest. Clones with synonymous mutations in this context can help to separate the effect of the non-synonymous mutation of interest from any effects caused by the recoding used for screening purposes, as well as off-target effects from the guides used. If the recoded sequence affects things like translation speed, it will be apparent in both clones with synonymous only and non-synonymous mutations. Without the presence of synonymous mutation controls, this may be misidentified as a phenotypic difference between non-synonymous mutation and the parental line.

Unfortunately, the mutants generated here did not appear to have a distinct phenotype. No mutation indicated a significant drop in growth rate, as was hypothesised for cell cycle-dependent proteins. The only significant difference identified was KKT7 S304A clone 9, which grew faster than the parental when assessed by Alamar blue growth assay (Figure 12). Without a second clone showing the same phenotype, it is hard to draw conclusions with certainty that this growth change is directly caused by the mutation in KKT7, or whether it is the result another genetic difference in this clone. Whilst no other mutant clones in this assay showed significant growth changes, it is quite apparent that other pairs of clones with the same mutation did not always grow at a consistent rate. It is not clear why some clones with identical mutations grew at differing rates but is most likely caused by the genetic diversity within the parental cell line. In addition, further validation of this phenotype is needed by a more accurate growth curve, as variance between replicates was high in all cell lines. On the whole, the lack of significantly different growth rates indicates that the phosphosite mutants generated here do not have notable changes to the rate of progression through the cell cycle such that their duplication time is affected, suggesting that none of the phosphosites mutated here have overarching control of the cell cycle.

Additionally, most mutant clones did not exhibit a cell cycle defect. The only identified changes to the cell cycle were KKT2 S25E clone 11 and KKT2 S530E clone 21, which indicated apparent triploid cells (Figure 14A), and a slight increase in the proportion of G1 cells in KKT2 S25S clone 5 (Figure 13A). The triploid phenotype was not replicated when a fresh sample of cells were used (Figure 14B), suggesting this was a random event that was selected for when passaging the cells. It was hence presumed that KKT2 S25E clone 11 shared a similar random event, as the other clone with this mutation lacked this phenotype so was not investigated further. Whether this mutation increases the likelihood of such random events remains to be seen.

Identifying several clones that have integrated the mutation of interest in a homozygous manner has been key to evaluating whether a phenotype is directly caused by the mutation of interest. It is plausible that differences seen between clones are the result of off-target mutations, compensatory mutations, or natural diversity in the parental population. But as none of the full genomes of these mutants have been sequenced, it is not possible to say which is the case with certainty. In the case of some of the mutations, only one clone was identified with the desired genotype. With only one clone it is difficult to be confident that differences seen are the result of the induced mutation alone. Having several clones with the mutation of interest can allow for reasonable scepticism that a phenotypic difference is caused by the mutation, if not all clones exhibit the same phenotype.

The lack of phenotype from the phosphosite mutations generated in this project is not completely unsurprising. Other studies looking at the effect of both ablation of phosphorylation sites and phosphomimetic studies have found little to no phenotypic effect in cell lines with several mutated phosphosites (Hořejší et al., 2010; Yang et al., 2013; Marchand et al., 2022). Whilst these examples are not from kinetoplastids, they demonstrate that it is common for phosphorylation to play a subtle role in controlling protein function. In contrast, there are cases where a single phosphorylation site has significant impact on the cell (Xu et al., 2011; Canton et al., 2012; Keder et al., 2015). However, in the latter case, none of the mutations caused a lethal phenotype. As the kinetochore proteins investigated here are essential genes, complete dysregulation would likely be lethal. As such, it is unsurprising that there was no apparent effect from the loss of individual phosphosites.

It should be noted that in many cases, the phosphosites that were mutated were in proximity to other serine or threonine residues in the primary protein sequence. For example, in KKT2 when looking at the 25 amino acid residues either side of S493, there are 10 other serine or threonine residues. As the kinase that phosphorylates this site is not known, it is unclear whether it would be possible for this particular kinase to phosphorylate a nearby serine or threonine instead. If this upstream kinase is promiscuous enough to be able to phosphorylate alternate residues nearby, the phosphorylation state would likely be the same as a WT KKT2 protein, allowing function to proceed as normal. It has been shown in *T. brucei* that CLK1 can phosphorylate KKT2 at S508, which is thought to be equivalent of

S485 or S487 in *L. mexicana* (Saldivia et al., 2021; Geoghegan et al., 2022). However, CLK1 has not been shown to phosphorylate alternate sites to date. Promiscuous phosphorylation is one potential explanation as to why the mutations generated here had little to no phenotype. Promiscuous kinases such as casein kinase II (Borgo et al., 2021) are known to phosphorylate various parts of the cell cycle machinery in humans (Schweighofer et al., 2024), so it is highly plausible the same could be true for kinetoplastids. However, the phosphorylation status of the mutant proteins generated here was not investigated. Nor is it known whether the phosphorylation state is more or less important than the location of the added phosphate at particular residues.

There have been more phosphorylation sites identified on these kinetochore proteins than were targeted for mutation here (Geoghegan et al., 2022). It is plausible that phosphorylation of several sites has a cumulative effect, and so removal of one phosphorylated residue has minimal impact. But loss of phosphorylation at several sites could cause a noticeable phenotype. Taking the example of KKT2 again, a double mutant was attempted targeting KKT2 S505 and S506 using the ssDNA approach, but no mutant cells were recovered. This is most likely due to the ssDNA method used, so repeating this target with the dsDNA approach would be enlightening. Further study is necessary to determine whether the phosphosites on KKT2, KKT4 and KKT7 have a cumulative effect, or whether phosphorylation plays a different currently unknown role. To determine if this is the case, protein-based approaches would be best to initiate investigation, to assess the range of potential phosphorylation states each kinetochore protein has, before moving to genetic modification to edit those sites. One way would be to use a phospho-protein mass spectrometry approach to identify the different phosphorylation states of each kinetochore protein. This approach could assess whether there are discrete phosphorylation states of high importance or a wider array of potential phosphorylation states with little indication of preference. In the former case, it is more likely that loss of a number of key sites would have an impact on fitness more so than in the latter, which could require loss of most or all phosphorylation sites to have an impact on function.

It should be emphasised that the genes that were precisely edited here were essential. All three kinetochore genes in which the dsDNA method was tested are essential (Akiyoshi and Gull, 2014). The method developed here is likely to be widely applicable for use across the genome, given that it works on essential genes which are most likely to be challenging to manipulate. Of course, not all mutations will be possible. For example, removing catalytic activity of an essential enzyme would likely be impossible to generate in an otherwise WT strain, even if other biochemically similar mutations are tolerated. The mutations induced here were only predicted to play a role on protein regulation, which given the lack of phenotype identified, is either multi-factorial or phosphorylation plays an alternative role. But this method opens up the possibility to unpick different aspects of essential genes that have previously been too technically challenging to investigate, as well as allowing more detailed study into non-essential genes. For example, this method could be used to modify motifs needed for other post-translational modifications, such as removing lysine residues of potential ubiquitination sites, or could be used to disrupt protein-protein interactions. It could also be used to manipulate trafficking signals or to influence drug-sensitivity.

Whilst there are already examples in the literature of precision editing (Zhang and Matlashewski, 2015; Zhang, Lypaczewski and Matlashewski, 2017; Rico et al., 2018; Vasquez et al., 2018; Vergnes et al., 2019), there is little standardisation as to the approach being used (Yagoubat et al., 2020). As such, it is likely that time and resources amongst members of the field are wasted, due to failure to compare approaches. The method presented here is simple and consistent, which has the potential to become the standard in the field. Currently, the method presented here is not suitable for use *en masse* simultaneously. But it is otherwise quite flexible and could be used on any gene of interest, even beyond *Leishmania* species such as in trypanosomes. If there is desire to generate libraries of precisely edited mutants, this method has the potential to enable that, so long as mutants are generated in manageable batches for culturing and screening.

Further research is still needed to adapt this method to become scalable *in vitro* at the point of transfection. To begin to increase the through-put of this method, an automated *in silico* approach to the design has been investigated and is presented in Chapter Four – Python Script. In order to complete the transfections and screening steps necessary to do a large number of precision mutants, it is first necessary to design the repair templates and oligonucleotide primers to create and screen them. Websites such as http://www.leishgedit.net/ (Beneke et al., 2017) have shown that automated design processes can aid in scale-up of mutant generation to allow large-scale projects, such as whole kinome assessment by Baker et al. (2021) to exist. Creating such tools also has the additional benefit of standardisation. Whilst efforts have been made to describe in detail how and when one codon sequence was chosen over another in this report, it is at the end of the day the individual’s choice as to which they use. Using a computer programme to execute this process ensures that the design will always follow the same decision making choices, independent of the researcher designing them. As long as the programme is coded to make design choices that have been shown to work *in vitro*, all non-lethal mutations should be possible to generate. However, the “rate-limiting step” of this precision editing method is the culturing and screening of clones on a transfection-by-transfection basis. Further work is needed to reduce this workload, as this will have the most impact at a high-throughput scale.

In conclusion, a methodology has been developed for consistent precision editing in *Leishmania mexicana* and has been shown to be effective on essential genes. Mutants were successfully generated on KKT2, KKT4 and KKT7, which included phosphomimetic mutants, phosphosite-deficient mutants, and synonymous mutants. Whilst efficiency of the editing varied by transfection, at least one homozygous mutant was recovered in each transfection and overall, 29.2% of mutants screened showed integration of the repair template. None of the mutants generated here showed growth defects or repeatable cell-cycle anomalies, suggesting that these mutations did not have significant impact on the cell cycle.

5 CHAPTER FOUR – PYTHON SCRIPT

5.1 Introduction

Python is an object-oriented high-level programming language with dynamic semantics and intuitive syntax (Python Institute, n.d.). It was created by Guido van Rossum in 1991, a Dutch programmer (Munro, 2024). Whilst Rossum made the first versions, Python has since been worked on by a large community of programmers from around the world, with the newest version (Python 3.0) being released in 2008 (Munro, 2024). This combination of Python being a human-friendly high-level language with intuitive syntax has increased its popularity and has led to significant development in the available packages.

Some packages are used in a wide variety of programmes created for diverse purposes. Examples of these are NumPy (Numerical Python) which is a package designed around mathematical manipulations and handling of arrays of data (Harris et al., 2020); and Pandas (panel data) which can be used for statistical analysis of data and allows information to be represented in a table-like format called a DataFrame (Mckinney, 2010). These DataFrames can be used to store data, but equally they can be used to manipulate or search through data. Both packages allow organisation and manipulation of data, which is useful in many different programmes to perform the necessary calculations to derive the appropriate output.

In contrast, some packages are highly specific to their uses. Biopython is a package designed for molecular biology and bioinformatics (Cock et al., 2009). Biopython's features include (but are not limited to) translation of DNA and RNA to protein, calculating the complement and reverse complement of DNA sequences, and being able to produce and read sequence alignments. Some of the more advanced aspects of Biopython include analysis of large data sets such as handling Next Generation Sequencing reads. Reads can be taken through quality filtering, trimming, assembly into a full genome or analysis against a reference genome to assess gene expression and finally calculating Principal Component Analysis of differentially expressed genes. Development of Python scripts to complete tasks like these also ensure identical analysis of each dataset, allowing consistency between experimental conditions, or even across organisms. As the current version of Biopython has many varied features, the creators have organised its capabilities into groups of smaller packages which can be imported individually. This approach minimises the memory required to run each respective script and keeps the syntax clear. But conveniently, there is cross-compatibility between all sub-packages.

Another Python package for molecular biology is Primer3 (Koressaar and Remm, 2007; Untergasser et al., 2012). Primer3 is a package that can design PCR primers against an input DNA sequence, but can also analyse primer sequences for common features. Primer3 has a wide array of customisable input variables such as desired melting temperatures or GC content of the output primers, and selection or exclusion of certain sequences within the template sequence. It can generate several primer pairs, along with all the associated information such as PCR product length. Originally, Primer3 was developed for command line usage, but has since been adapted into a Python module in 2014 due to its popularity. One consideration when using Primer3 is that it only uses the input sequence for primer design and does not complete any form of cross-reactivity analysis with other parts of the given sequence, nor does it consider the wider genome for similar sequences. As such, caution should be used when designing primers with Primer3 on small reference sequences to ensure specificity. This is particularly important on genes with known homologs of high sequence similarity as primers designed by Primer3 could lack specificity to the intended target.

In this chapter, I will describe and discuss how I created a Python script that can design a repair template in a similar manner to those that were designed and used in the previous chapters. The script is instructed by a simple Excel Spreadsheet "form" (that works as a configuration file), a user-provided a codon usage table, and a FASTA file of their gene. Following execution, two repair template sequences are produced. One repair template contains only synonymous mutations, and the other contains the desired nonsynonymous mutation. Both repair templates have additional identical synonymous mutations needed for screening. In addition, the script designs screening primers to detect the integration of the repair template, and long oligonucleotide primers to produce the respective repair templates. Lastly, the script also provides pairwise sequence alignments and some useful information about the repair templates and primers. All of these outputs are contained in a single text document. This format allows visualisation of the alignments and creates a store for the information the script has calculated.

5.2 Development

A Python script was created using Python 3.10.9, as well as several other packages listed in Table 6, in particular the Biopython package (also known as Bio), Pandas and Primer3 (Koressaar and Remm, 2007; Cock et al., 2009; Mckinney, 2010; Untergasser et al., 2012).

In order to make the script more readable, avoid duplication, and to make it more flexible to modifications in future versions, several files were created which are interlinked. Each file contains a subset of the required code. Apart from the main file, each of the other files creates a series of functions that can be called by other files to execute that portion of code, so act like packages. Comparatively, the main file puts all of these other functions together in succession to achieve all the necessary steps. The file names, their purposes, the

Table 6. Python package versions used in the creation of the Python repair template generating script.

Package	Version
Python	3.10.9
pandas	1.5.3
Biopython (Bio) Bio - SeqIO Bio - Seq Bio - Align	1.81
NumPy	1.23.5
io	N/A
random	N/A
Primer3 Primer3.bindings	2.0.1

shortened name used in the main file, and the appropriate appendices for the full code can be found in Table 7.

One of Biopython's features is that it has inbuilt codon tables, including variant codon usage tables. However, the inbuilt codon table in Biopython did not have capacity to retrieve every possible codon sequence for each amino acid – only one codon sequence was retrieved when Biopython was asked to provide the codons for any amino acid. As such, custom functions were created to call all the codon sequences that code for a given amino acid, with or without the associated frequency usage data for every codon. All the codon sequence and amino acid pairs use the standard genetic code.

Table 7. Repair generator constituent files and purposes.

Filename	Appendix Containing the Code	Main Purposes/Theme	Imported as
main.py	7.2.10	To call each of the other functions in succession to perform the necessary steps to generate the repair template, screening primers and production primers, as well as reading the input Excel spreadsheet configuration file and producing a user-friendly output containing useful information.	-
reading_input_file.py	7.2.11	Interpretation of the input codon usage table and conversion to a Pandas DataFrame.	rif
codon_dictionaries.py	7.2.12	To separate out each codon from the input sequence into identifiable pieces and	cdict

		store them in dictionaries for retrieval and manipulation.	
codon_dataframes.py	7.2.13	To separate out each codon from the input sequence and associate that codon with frequency usage data to allow selection of alternate codons based on usage frequency. These are stored in Pandas DataFrames inside dictionaries.	cdf
formatting_functions.py	7.2.14	To create more readable versions of some of the outputs for the output file.	formats
stitching_functions.py	7.2.15	To break an input sequence into constituent parts and put DNA sequences together to form new sequences.	stitch
validator.py	7.2.16	To confirm the inputs given are consistent with each other e.g. that the specified codon codes for the amino acid listed.	val
primer_functions.py	7.2.17	To design screening and repair template primers with consistent settings that work with a range of target sequences.	primers

Biopython and Pandas packages in particular were used in conjunction to create a script that uses a series of dictionaries to identify the sequence to mutate from a larger DNA sequence, break the sequence down into constituent codon sequences, and then exchange those codon sequences with the replacement codon sequence as dictated by the recoding methodology chosen. Using dictionaries allowed nonsynonymous mutations to be created by removing the key-value pair associated with the target codon's wild-type sequence and replacing it with a new key-value pair corresponding to the desired mutation. Whilst dictionaries do not store data in a guaranteed order, using numbers as part of (or the entirety of) the key ensured codons were retrieved in the order they were in the original input sequence when it was time to recreate a continuous DNA sequence. This ensured that the protein sequence was maintained (except for the target nonsynonymous mutation) and ensured that the chosen synonymous recoding method had actually been applied to each codon, rather than effectively random triplet codes being chosen for each codon regardless of the recoding method selected.

5.2.1 RECODING METHODOLOGIES

One of the main benefits to the script is the automated recoding. Recoding a sequence manually is very time consuming and laborious. The script offers several recoding methodologies to generate both the synonymous recoding (used for screening purposes) and a nonsynonymous target mutation. There are four types of recoding the script can perform. The "matched" setting can only be applied to synonymous mutations, but the other three ("random", "highest" and "lowest") can be applied to both synonymous and nonsynonymous mutations.

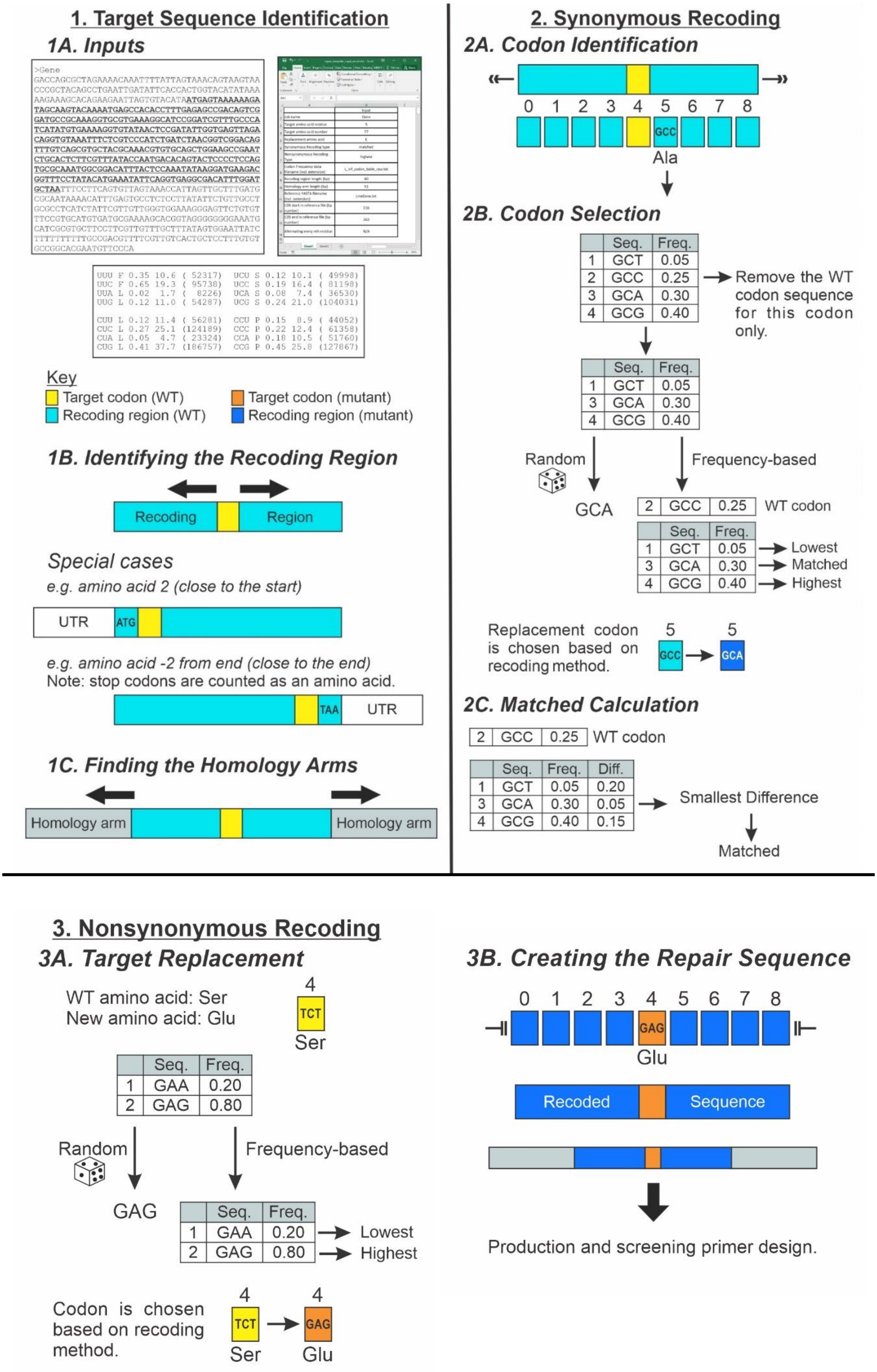
When the chosen recoding method is applied to nonsynonymous mutations, all codon sequences for the respective amino acid are considered in the selection process. However, when the recoding method is applied to synonymous mutations, the WT codon sequence is removed from the available codons to choose from to ensure a mutation occurs. The exceptions to this are the codons for methionine and tryptophan, which only have one codon each in the standard genetic code, so they will always be "replaced" with the same sequence as the WT codon. Similarly, for amino acids that are only encoded by two codons and are being synonymously mutated, the choice after removing the WT codon from the selection leaves only one possible replacement codon sequence. So, these codons will always be recoded predictably to the non-WT codon sequence, regardless of recoding methodology used.

“Random” is as the name suggests, a random unbiased selection of possible codons for the desired amino acid using the random Python package. Unlike the other recoding methods, “random” will cause a different output repair sequence on each execution of the code, when given the same inputs (excluding the exceptions already discussed). If little is known about the impact of different sequence compositions on the target gene or species, or if targeted approaches have been unsuccessful, this method provides a way to generate a repair template without bias in the design to explore options that might not have previously been considered.

“Highest” and “lowest” settings are in reference to the frequency usage of the codon sequences. The user supplies a codon usage table as part of the required inputs (see Figure 17B for an example). The codon usage table provided will dictate which codon is selected for each amino acid, with “highest” referring to the most used codon, and “lowest” the least used codon (see Figure 15, part 2 for more details). These allow the user to bias their recoding to use more common or rarer codon sequences, as desired.

“Matched” is essentially a harmonized codon selection - choosing the codon that is most similarly used to the input codon, and is the most similar to the design strategy used in the previous chapter. To determine which codon is the “matched” codon, a simple subtraction is performed using the values in the “Fraction” column from the supplied codon usage table from https://www.kazusa.or.jp/codon/ (Figure 17C). However, using this approach means that ties can occur fairly frequently. In these instances, the data from the “Number” column (i.e. a count of instances in the selected genome) is used as a tiebreaker, taking the higher of the two. That being said, as this script currently stands, there is the possibility that a tie could persist and if that is the case, the script will output a text based error in the console, and will likely fail to complete.

In all recoding methods, each codon is evaluated independently from other codons encoding the same amino acid. To achieve this, for “matched”, “highest” and “lowest” recoding, frequency usage data is copied from a reference for each codon, and then calculations are performed only on the copy. Similarly in the “random” setting, the list of codon sequences for a given amino acid is copied from a reference list, before adjusting to remove the WT sequence for synonymous mutations. This approach ensures that even in instances where the same amino acid is represented several times, the replacement



1. Target Sequence Identification

1A. Inputs

```
>Gene
GACCAGCGCTAGAAAACAAATTTTATTAGTAAACAGTAAGTAA
CCCGCTACAGCCTGAATTGATATTCACCACTGGTACATATAAA
AAGAAAGCACAGAAGAATTAGTGTACATAATGAGTAAAAAAGA
TAGCAAGTACAAAATGAGCCACACCTTTGAGAGCCGACAGTCG
GATGCCGCAAAGGTGCGTGAAAGGCATCCGGATCGTTTGCCCA
TCATATGTGAAAAGGTGTATAACTCCGATATTGGTGAGTTAGA
CAGGTGTAAATTTCTCGTCCCATCTGATCTAACGGTCGGACAG
TTTGTCAGCGTGCTACGCAAACGTGTGCAGCTGGAAGCCGAAT
CTGCACTCTTCGTTTATACCAATGACACAGTACTCCCCTCCAG
TGCGCAAATGGCGGACATTTACTCCAAATATAAGGATGAAGAC
GGTTTCCTATACATGAAATATTCAGGTGAGGCGACATTTGGAT
GCTAATTTCCTTCAGTGTTAGTAAACCATTAGTTGCTTTGATG
CGCAATAAAACATTTGAGTGCCTCTCCTTATATTCTGTTGCCT
GCGCCTCATCTATTCGTTGTTGGGTGGAAAGGGAGTTCTGTGT
TTCCGTGCATGTGATGCGAAAAGCACGGTAGGGGGGGGAAATG
CATCGCGTGCTTCCTTCGTTGTTTGCTTTATAGTGGAATTATC
TTTTTTTTTTGCCGACGTTTTCGTTGTCACTGCTCCTTTGTGT
GCCGGCACGAATGTTCCCA
```

	Input
Job name	Gene
Target amino acid residue	S
Target amino acid number	77
Replacement amino acid	E
Synonymous Recoding type	matched
Nonsynonymous Recoding Type	highest
Codon Frequency data filename (incl. extension)	L_inf_codon_table_raw.txt
Recoding region length (bp)	60
Homology arm length (bp)	51
Reference FASTA filename (incl. extension)	LmaGene.txt
CDS start in reference file (bp number)	116
CDS end in reference file (bp number)	363
Alternating every nth residue	N/A

```
UUU F 0.35 10.6 ( 52317)  UCU S 0.12 10.1 ( 49998)
UUC F 0.65 19.3 ( 95738)  UCC S 0.19 16.4 ( 81198)
UUA L 0.02  1.7 (  8226)  UCA S 0.08  7.4 ( 36530)
UUG L 0.12 11.0 ( 54287)  UCG S 0.24 21.0 (104031)

CUU L 0.12 11.4 ( 56281)  CCU P 0.15  8.9 ( 44052)
CUC L 0.27 25.1 (124189)  CCC P 0.22 12.4 ( 61358)
CUA L 0.05  4.7 ( 23324)  CCA P 0.18 10.5 ( 51760)
CUG L 0.41 37.7 (186757)  CCG P 0.45 25.8 (127867)
```

Key

🟨	Target codon (WT)	🟧	Target codon (mutant)
🟦	Recoding region (WT)	🟦	Recoding region (mutant)

1B. Identifying the Recoding Region

Special cases

e.g. amino acid 2 (close to the start)

e.g. amino acid -2 from end (close to the end)

Note: stop codons are counted as an amino acid.

1C. Finding the Homology Arms

2. Synonymous Recoding

2A. Codon Identification

2B. Codon Selection

	Seq.	Freq.
1	GCT	0.05
2	GCC	0.25
3	GCA	0.30
4	GCG	0.40

Remove the WT codon sequence for this codon only.

	Seq.	Freq.
1	GCT	0.05
3	GCA	0.30
4	GCG	0.40

Random

GCA

Frequency-based

2	GCC	0.25	WT codon

	Seq.	Freq.	
1	GCT	0.05	Lowest
3	GCA	0.30	Matched
4	GCG	0.40	Highest

Replacement codon is chosen based on recoding method.

2C. Matched Calculation

2	GCC	0.25	WT codon

	Seq.	Freq.	Diff.
1	GCT	0.05	0.20
3	GCA	0.30	0.05
4	GCG	0.40	0.15

Smallest Difference

Matched

3. Nonsynonymous Recoding

3A. Target Replacement

WT amino acid: Ser
New amino acid: Glu

	Seq.	Freq.
1	GAA	0.20
2	GAG	0.80

Random

GAG

Frequency-based

	Seq.	Freq.	
1	GAA	0.20	Lowest
2	GAG	0.80	Highest

Codon is chosen based on recoding method.

3B. Creating the Repair Sequence

Production and screening primer design.

Figure 15. Python repair template generator script workflow (part 1). 1A) The script requires 3 supplemental files (a FASTA file, a codon usage table and the configuration spreadsheet – see Figure 17 for more detailed versions). The configuration spreadsheet is used to determine the location of the CDS (bold underlined) within the supplied FASTA file, and to identify: the target codon, surrounding region to recode (1B), and homology arms (1C). Generally, the recoding region is equally split to have the same number of recoded codons either side of the target codon (however, one side will contain one extra codon in instances where this is not possible). However, the recoding region is adjusted to stay within the CDS if the target codon is in close proximity to the start or end of the CDS (1B). 2A) Each codon from the recoding region is identified and evaluated individually to select a suitable replacement sequence (2B), dictated by the chosen recoding strategy specified in the configuration spreadsheet (1A). The replacement codon will never be the same as the WT sequence, except for methionine and tryptophan codons. 2C) When using the matched recoding setting, the difference in usage of the alternate codons are compared with the WT codon sequence, and the most similarly used codon (either more or less frequently used) is chosen. 3A) For the nonsynonymous mutation, the codon sequence is chosen from any possible codon sequence for that amino acid. The chosen sequence is determined by the recoding method chosen in the configuration spreadsheet (1A). 3C) Lastly, the individual mutated codons are concatenated into the final repair template sequence, with the homology arms added at either end.

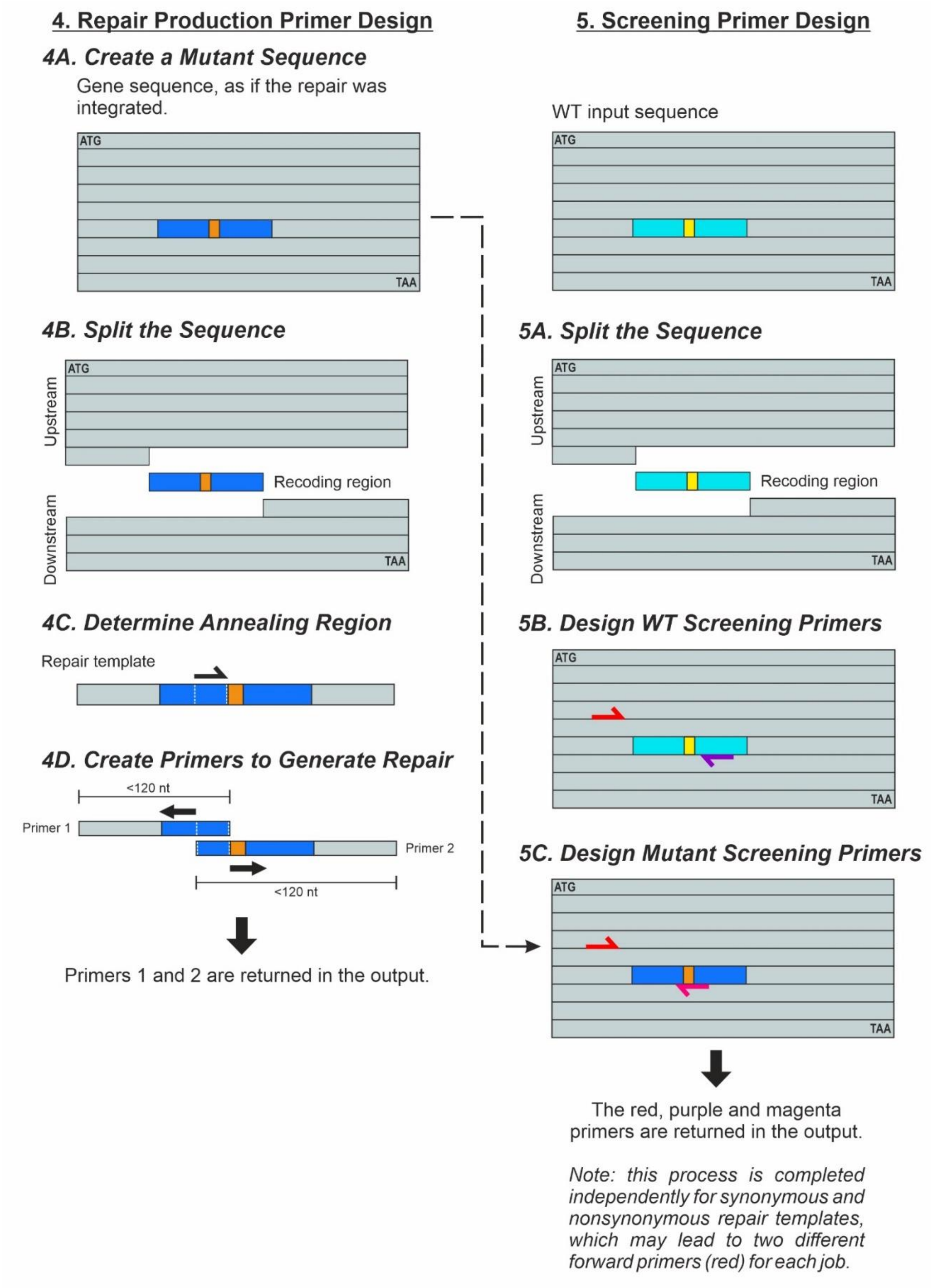
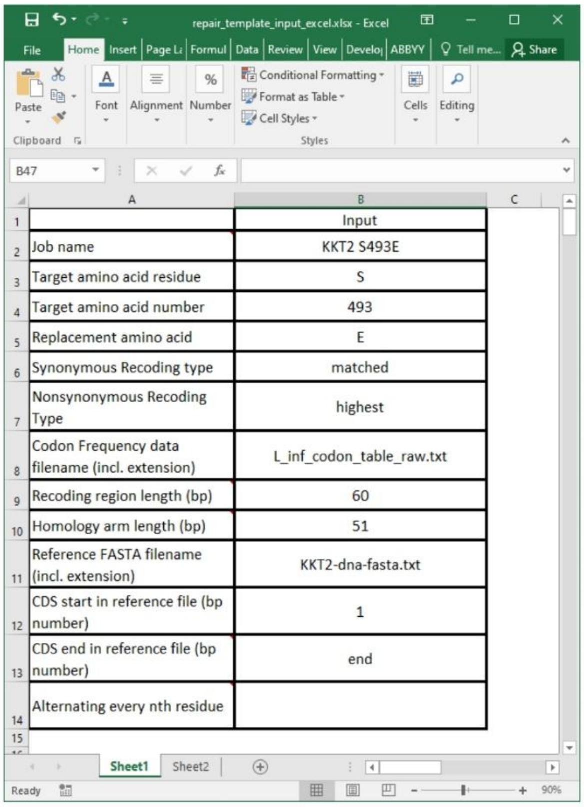

4. Repair Production Primer Design
4A. Create a Mutant Sequence
Gene sequence, as if the repair was integrated.
ATG
TAA
4B. Split the Sequence
Upstream
ATG
Recoding region
Downstream
TAA
4C. Determine Annealing Region
Repair template
4D. Create Primers to Generate Repair
<120 nt
Primer 1
Primer 2
<120 nt
Primers 1 and 2 are returned in the output.
5. Screening Primer Design
WT input sequence
ATG
TAA
5A. Split the Sequence
Upstream
ATG
Recoding region
Downstream
TAA
5B. Design WT Screening Primers
ATG
TAA
5C. Design Mutant Screening Primers
ATG
TAA
The red, purple and magenta primers are returned in the output.
Note: this process is completed independently for synonymous and nonsynonymous repair templates, which may lead to two different forward primers (red) for each job.

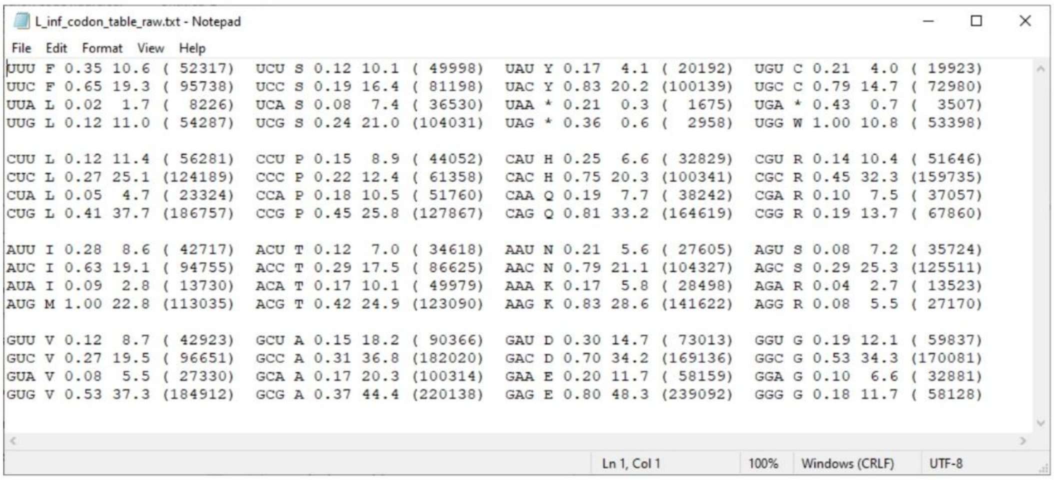
Figure 16. Python repair template generator script workflow (part 2). 4A) To design the primers needed to make the repair template, first an artificial gene sequence is created containing the recoded sequence (dark blue/orange) in place of the WT recoding region. This sequence will also include UTRs if provided. 4B) The sequence is split into 3 parts: upstream of the repair template, the repair template, and downstream of the repair template. 4C) In order to determine a suitable annealing sequence, a primer pair is designed such that the forward primer is within the recoded region/mutated target site of the repair template, and the reverse primer is constrained to the downstream region. The forward primer is used as the annealing sequence but the reverse primer is not used – single primers cannot be designed with Primer3 in this way. 4D) The forward primer sequence is used as the annealing region for the long primers needed to generate the repair template. To complete the long primers, the sequences are extended back to the full length of the repair template on each strand to create primers 1 and 2. Primers 1 and 2 must be less than 120 nt each, otherwise they are redesigned with an alternate annealing region (see Figure 19A for more details). 5A) To design screening primers, the WT sequence is split into 3 parts, similarly to the artificial sequence. 5B) The WT screening primer pair (red/purple) is generated by constraining the forward primer to the upstream region, and constraining the reverse primer to the recoding region (cyan/yellow). 5C) The mutant screening primer pair is designed using the forward primer (red) from the WT screening primer pair, and constraining the reverse primer (magenta) to the recoding region (dark blue/orange) of the artificial sequence containing the repair template generated in part 4A.

A



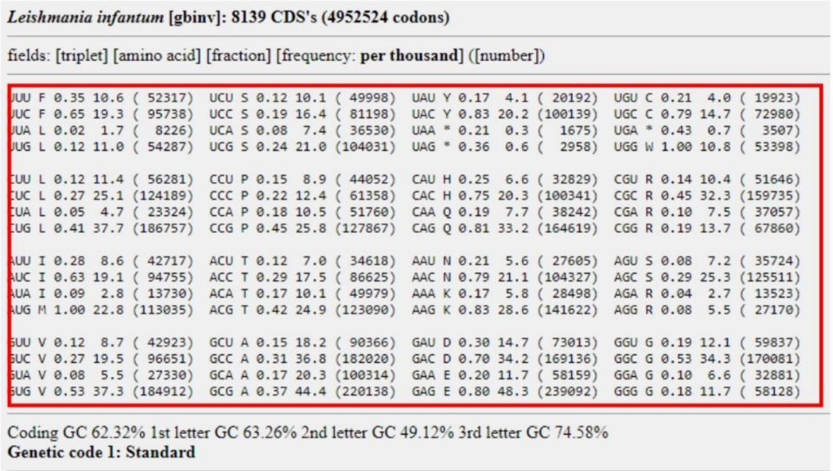
	A	B
1		Input
2	Job name	KKT2 S493E
3	Target amino acid residue	S
4	Target amino acid number	493
5	Replacement amino acid	E
6	Synonymous Recoding type	matched
7	Nonsynonymous Recoding Type	highest
8	Codon Frequency data filename (incl. extension)	L_inf_codon_table_raw.txt
9	Recoding region length (bp)	60
10	Homology arm length (bp)	51
11	Reference FASTA filename (incl. extension)	KKT2-dna-fasta.txt
12	CDS start in reference file (bp number)	1
13	CDS end in reference file (bp number)	end
14	Alternating every nth residue	
15		

B



```
UUU F 0.35 10.6 ( 52317)  UCU S 0.12 10.1 ( 49998)  UAU Y 0.17  4.1 ( 20192)  UGU C 0.21  4.0 ( 19923)
UUC F 0.65 19.3 ( 95738)  UCC S 0.19 16.4 ( 81198)  UAC Y 0.83 20.2 (100139)  UGC C 0.79 14.7 ( 72980)
UUA L 0.02  1.7 (  8226)  UCA S 0.08  7.4 ( 36530)  UAA * 0.21  0.3 (  1675)  UGA * 0.43  0.7 (  3507)
UUG L 0.12 11.0 ( 54287)  UCG S 0.24 21.0 (104031)  UAG * 0.36  0.6 (  2958)  UGG W 1.00 10.8 ( 53398)

CUU L 0.12 11.4 ( 56281)  CCU P 0.15  8.9 ( 44052)  CAU H 0.25  6.6 ( 32829)  CGU R 0.14 10.4 ( 51646)
CUC L 0.27 25.1 (124189)  CCC P 0.22 12.4 ( 61358)  CAC H 0.75 20.3 (100341)  CGC R 0.45 32.3 (159735)
CUA L 0.05  4.7 ( 23324)  CCA P 0.18 10.5 ( 51760)  CAA Q 0.19  7.7 ( 38242)  CGA R 0.10  7.5 ( 37057)
CUG L 0.41 37.7 (186757)  CCG P 0.45 25.8 (127867)  CAG Q 0.81 33.2 (164619)  CGG R 0.19 13.7 ( 67860)

AUU I 0.28  8.6 ( 42717)  ACU T 0.12  7.0 ( 34618)  AAU N 0.21  5.6 ( 27605)  AGU S 0.08  7.2 ( 35724)
AUC I 0.63 19.1 ( 94755)  ACC T 0.29 17.5 ( 86625)  AAC N 0.79 21.1 (104327)  AGC S 0.29 25.3 (125511)
AUA I 0.09  2.8 ( 13730)  ACA T 0.17 10.1 ( 49979)  AAA K 0.17  5.8 ( 28498)  AGA R 0.04  2.7 ( 13523)
AUG M 1.00 22.8 (113035)  ACG T 0.42 24.9 (123090)  AAG K 0.83 28.6 (141622)  AGG R 0.08  5.5 ( 27170)

GUU V 0.12  8.7 ( 42923)  GCU A 0.15 18.2 ( 90366)  GAU D 0.30 14.7 ( 73013)  GGU G 0.19 12.1 ( 59837)
GUC V 0.27 19.5 ( 96651)  GCC A 0.31 36.8 (182020)  GAC D 0.70 34.2 (169136)  GGC G 0.53 34.3 (170081)
GUA V 0.08  5.5 ( 27330)  GCA A 0.17 20.3 (100314)  GAA E 0.20 11.7 ( 58159)  GGA G 0.10  6.6 ( 32881)
GUG V 0.53 37.3 (184912)  GCG A 0.37 44.4 (220138)  GAG E 0.80 48.3 (239092)  GGG G 0.18 11.7 ( 58128)
```

C



Leishmania infantum **[gbinv]: 8139 CDS's (4952524 codons)**

fields: [triplet] [amino acid] [fraction] [frequency: **per thousand**] ([number])

```
UUU F 0.35 10.6 ( 52317)  UCU S 0.12 10.1 ( 49998)  UAU Y 0.17  4.1 ( 20192)  UGU C 0.21  4.0 ( 19923)
UUC F 0.65 19.3 ( 95738)  UCC S 0.19 16.4 ( 81198)  UAC Y 0.83 20.2 (100139)  UGC C 0.79 14.7 ( 72980)
UUA L 0.02  1.7 (  8226)  UCA S 0.08  7.4 ( 36530)  UAA * 0.21  0.3 (  1675)  UGA * 0.43  0.7 (  3507)
UUG L 0.12 11.0 ( 54287)  UCG S 0.24 21.0 (104031)  UAG * 0.36  0.6 (  2958)  UGG W 1.00 10.8 ( 53398)

CUU L 0.12 11.4 ( 56281)  CCU P 0.15  8.9 ( 44052)  CAU H 0.25  6.6 ( 32829)  CGU R 0.14 10.4 ( 51646)
CUC L 0.27 25.1 (124189)  CCC P 0.22 12.4 ( 61358)  CAC H 0.75 20.3 (100341)  CGC R 0.45 32.3 (159735)
CUA L 0.05  4.7 ( 23324)  CCA P 0.18 10.5 ( 51760)  CAA Q 0.19  7.7 ( 38242)  CGA R 0.10  7.5 ( 37057)
CUG L 0.41 37.7 (186757)  CCG P 0.45 25.8 (127867)  CAG Q 0.81 33.2 (164619)  CGG R 0.19 13.7 ( 67860)

AUU I 0.28  8.6 ( 42717)  ACU T 0.12  7.0 ( 34618)  AAU N 0.21  5.6 ( 27605)  AGU S 0.08  7.2 ( 35724)
AUC I 0.63 19.1 ( 94755)  ACC T 0.29 17.5 ( 86625)  AAC N 0.79 21.1 (104327)  AGC S 0.29 25.3 (125511)
AUA I 0.09  2.8 ( 13730)  ACA T 0.17 10.1 ( 49979)  AAA K 0.17  5.8 ( 28498)  AGA R 0.04  2.7 ( 13523)
AUG M 1.00 22.8 (113035)  ACG T 0.42 24.9 (123090)  AAG K 0.83 28.6 (141622)  AGG R 0.08  5.5 ( 27170)

GUU V 0.12  8.7 ( 42923)  GCU A 0.15 18.2 ( 90366)  GAU D 0.30 14.7 ( 73013)  GGU G 0.19 12.1 ( 59837)
GUC V 0.27 19.5 ( 96651)  GCC A 0.31 36.8 (182020)  GAC D 0.70 34.2 (169136)  GGC G 0.53 34.3 (170081)
GUA V 0.08  5.5 ( 27330)  GCA A 0.17 20.3 (100314)  GAA E 0.20 11.7 ( 58159)  GGA G 0.10  6.6 ( 32881)
GUG V 0.53 37.3 (184912)  GCG A 0.37 44.4 (220138)  GAG E 0.80 48.3 (239092)  GGG G 0.18 11.7 ( 58128)
```

Coding GC 62.32% 1st letter GC 63.26% 2nd letter GC 49.12% 3rd letter GC 74.58%
Genetic code 1: Standard

Figure 17. Additional input files needed to execute the Python script repair template generator. A) An example of the configuration Excel spreadsheet used to instruct the Python script. Some cells have data validation activated to provide dropdown menus (cells: B3, B5, B6 and B7). Sheet 2 (not shown) has the corresponding data for the validation. The lengths of the recoding regions and homology arms must be a multiple of three (see main body text for more details). As shown in the example, the word "end" (all lowercase) can be specified instead of the base pair number to signify the end of the CDS is the end of the FASTA file. When an alternating synonymous recoding methodology is selected, the "Alternating every nth residue" cell (B14) must also be filled with an integer value greater than zero. B) An example codon usage table collected from https://www.kazusa.or.jp/codon for *L. infantum*. After choosing the table on https://www.kazusa.or.jp/codon, a genetic code table is also chosen to reveal the translation column and the fraction column (columns 3 and 4 of each group). The filename also corresponds to the one specified in A. C) A screenshot of the codon table from https://www.kazusa.or.jp/codon, after selecting the desired genetic code to use. The red box highlights the data that the user is directed to copy and paste into the text document shown in B.

sequence is chosen in relation to the codon sequence for that residue in the reference sequence, rather than a global change for all codons that code for the same amino acid.

5.2.2 ALTERNATING RECODING

In addition to recoding methods previously described, this script also offers the user the choice to design a repair template that alternates between recoded codons and WT codons. Alternating recoding in this way reduces the number of mutations added to the daughter cell line, whilst retaining a large region of altered sequence for screening purposes. The alternating recoding methods are available in all of the available synonymous recoding strategies and are designated by "alternating" in their name, followed by the type of recoding which the alternation will be.

If an "alternating" method is chosen, the user must also specify the interval (n). The first codon for any "alternating" method is always recoded, followed by the (n+1)th codon until the end of the repair template (Figure 18). The exception to this is that the target codon will always be mutated even if it would normally fall on a codon that is not recoded by the n value. In this situation, the pattern for the synonymous recoding will ignore the target codon, leaving the pattern unaffected (Figure 18, n=2 example). When an alternating recoding method is chosen, the alternating pattern (n value) chosen by the user will also be displayed in the output file.

If no n value is provided or the provided value is 0, the script will prompt the user to rectify the mistake, or else the execution will be cancelled. The user will also be prompted if the script identifies that the user has put in an n value greater than half of the number of codons being recoded. E.g. if the recoding region was 10 codons long and n was set to 7. In

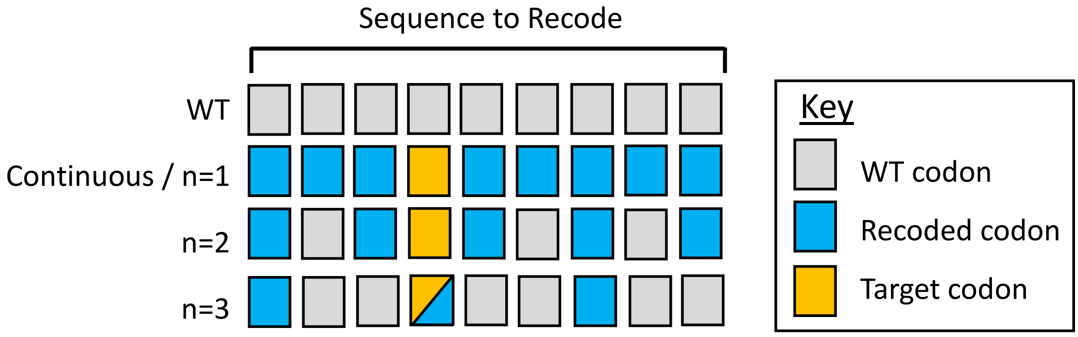


Figure 18. Alternating recoding example. Each rectangle represents a codon. Colours indicate whether the codon is WT sequence (grey), synonymously recoded (blue) or the target codon which is both synonymously and nonsynonymously recoded depending on the repair template (orange). The alternating n value is given for each example. The target codon is always mutated regardless as to whether it would align with a synonymously mutated codon (as in the example of n=3, illustrated by half orange and half blue) or whether it falls between mutated codons (as in the example of n=2).

this instance, only the first codon, the target codon and one other codon would be mutated, which is unlikely to be desired. But if the user confirms that that is what is desired, the repair template will be designed with those settings.

In order to create the “alternating” methods, the codons are initially sorted into two dictionaries – one to recode, and one to remain as the input sequence. The recoding method of choice is applied to the dictionary of the codons that fall within the recoding dictionary, and after mutating, the two dictionaries are recombined into one for reassembly into the final sequence.

5.2.3 FINAL REPAIR SEQUENCE ASSEMBLY

Once the recoding has completed, the new codon sequences are called from their dictionary in order and concatenated to form the recoded region sequences. Then, a check is run to confirm that the target codon translates into the expected amino acid for that repair template. If all is correct, the homology arms are added to the recoded region to complete the repair template. The homology arms are identified by using the co-ordinates of the recoded region in the WT sequence and adjusting them with the length of the homology arms specified by the user (see Figure 15 part 1C and 3B). Lastly, the homology arms and the recoded sequence are concatenated to form each repair template sequence.

5.2.4 PRIMER DESIGN

Once the repair sequences have been created, the script designs oligonucleotide primers to screen for integration and to generate the repair template itself. Both of these tasks use a similar method to design them, by constraining the Primer3 package to design the primers in specific locations of the DNA (Koressaar and Remm, 2007; Untergasser et al., 2012).

For the screening primer design, one primer is always in the region outside of the repair template, with the other inside (Figure 16 part 5). Due to the constraints of the Primer3 package, the primer outside of the repair template is always the forward primer and so is placed upstream of the repair region. As such, the primer design process may become limited if short reference sequences are provided upstream of the target codon. The reverse primer is designed to recognise either the WT sequence in the recoding region or the mutated region of the repair template. It should be noted that the process including the WT primer pair design is completed independently for each repair template (synonymous and nonsynonymous), which can lead to two different WT screening primer sets in some instances. This situation is very rare, as the design process is using the same settings and same input sequence for the WT screening primer pair. However, when it does occur, it is up to the user to evaluate the primers and determine which they wish to use.

To design long oligonucleotides for repair template generation, the forward primer is constrained against the recoded region of the repair template sequence, with the reverse primer downstream of the repair template (Figure 16 part 4). The reverse primer is not required, but Primer3 is only able to complete the design if both regions are specified. Using the downstream DNA rather than a dummy piece of DNA was chosen so that the sequence has more similar properties to that of the region the forward primer is designed against e.g. GC content. The forward primer is used as the annealing sequence for Primer 1 and its reverse-complement for Primer 2 (Figure 16 part 4C and D). The final primer sequences are completed by recounting the sequence from the annealing regions to the ends of the repair template (sense sequence for Primer 1, reverse complement sequence for Primer 2).

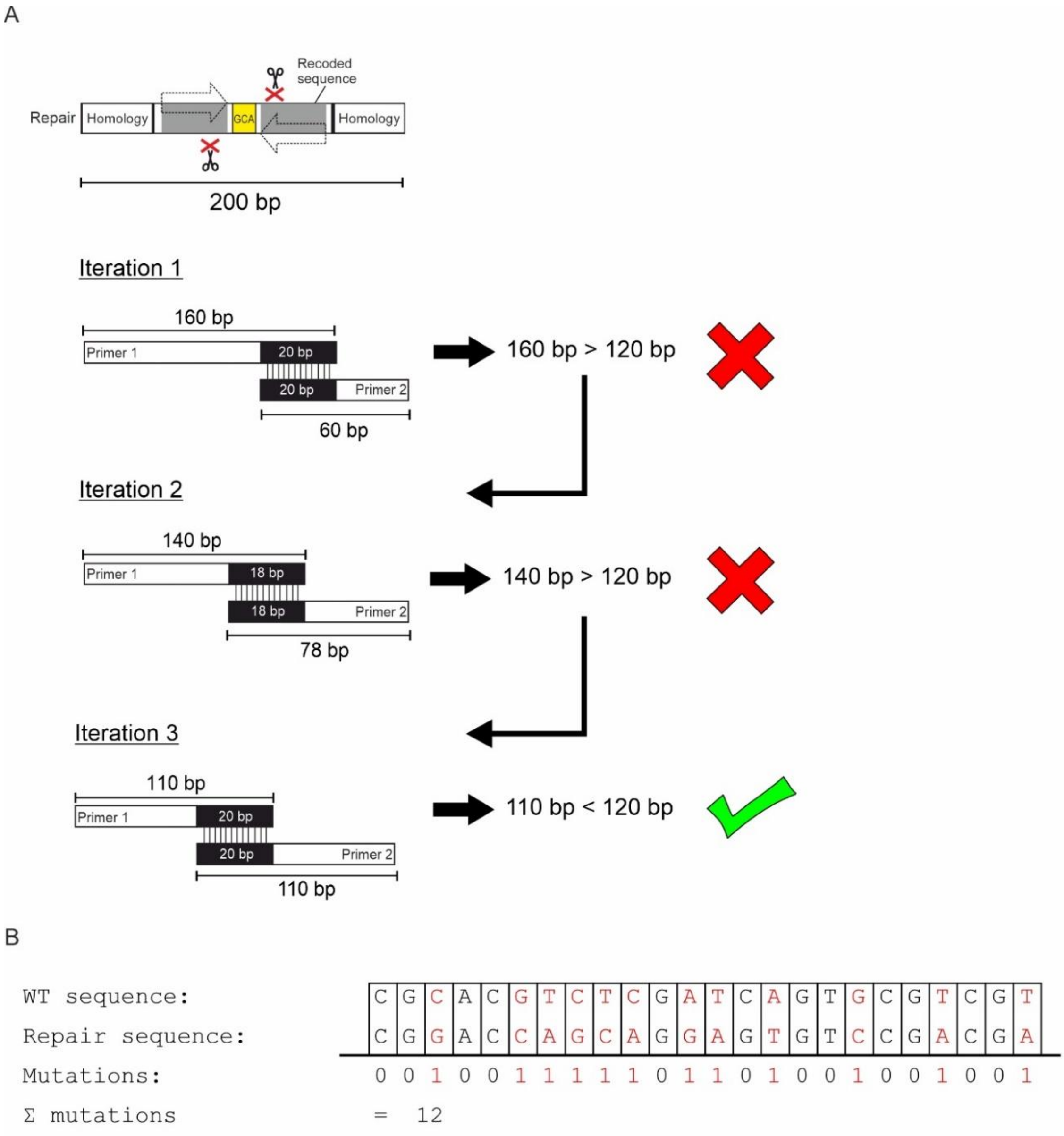
Once completed, both Primer 1 and 2's lengths are evaluated. Most commercial suppliers have a price cut-off for oligonucleotide sequences at 120 nt, and have a notable price increase for those over 120 nt. As such, this script will only accept repair primer sequences where both Primers 1 and 2 are less than or equal to 120 nt long. If this criteria is not met by one or both primers, up to two alternate annealing regions are assessed instead (Figure 19A). This means that successful execution of the script will only be possible for repair templates ≤ 220 bp with a 20 bp annealing region. Generally, the region that can be used to create an annealing region is quite small, and so the settings for Primer3 have been somewhat relaxed compared to the screening primers, to ensure success. That being said, in some tests on the longest repair templates, it was not always possible to design a suitable annealing region, which led to the script failing to complete its execution. Usually, adjusting the design settings will allow repair templates to be generated in this instance. Primers 1 and 2 are included in the output, both given as 5' to 3' sequences, such that they are ready to purchase (Figure 20 green section). A suggested melting temperature is also provided in the output document, as calculated by the Primer3 package.

5.2.5 ALIGNMENT

In the output document, there are two sections containing alignments of the respective sequences (see Figure 20). Firstly, there is a sham alignment nearer the top, which displays the DNA and protein sequences of the WT "repair" sequence, and both repair sequences. This is not a true multiple sequence alignment (MSA), but rather just an alignment of the text characters using spaces and tab characters such that the relevant residues appear in line with each other in the appropriate coding frame. This format is user friendly and serves the same purpose as a MSA in this instance. Because this alignment relies on spacing using text characters, the user is recommended to use a font that has a standard character size such as Courier New, as the alignment of characters will be incorrect when viewed with fonts with variable character sizes such as Calibri or Times New Roman. Biopython does not have the capacity to generate MSAs, although it can interpret them, but it does have the capacity to generate pairwise-sequence alignments (PSAs). Other packages such as ClustalW for Python can create MSAs but ClustalW is not compatible with running on Windows, limiting which devices would be able to run this script. As such, the output also includes PSAs for WT-synonymous repair template and WT-nonsynonymous repair template pairs. As it is expected that every base in both sequences will align without gaps, the open gap and extension gap penalties were set to -10 (from the default suggested of -1) to prevent alignments generating which did not have real biological relevance.

The purpose of including the PSA as well as the sham-MSA is to more clearly highlight where the mutations are to the user and how they are spaced. Providing a PSA allows the user to clearly see which bases are mutated through the symbols. In comparison, reading through the sham-MSA is much harder to spot individual character differences across several lines of text. This format is also more accessible than the use of colours to indicate the differences for those with colour-blindness, which is also not usually possible to include in a text document file.

Additionally, the script provides a count of the number of different bases between each repair template and the WT sequence as well. This function simply compares the nth character from each of the sequences being evaluated and counts the numbers of non-identities (Figure 19B).



WT sequence: C G C A C G T C T C G A T C A G T G C G T C G T

Repair sequence: C G G A C C A G C A G G A G T G T C C G A C G A

Mutations: 0 0 1 0 0 1 1 1 1 1 0 1 1 0 1 0 0 1 0 0 1 0 0 1

Σ mutations = 12

Figure 19. Schematics of the repair primer annealing region design process and the calculation of the number of mutations per repair template. A) The repair primer design analysis iteration. To generate the repair template, two primers are designed which have an overlapping annealing region (black) - see main body section 5.2.4 for more details. The length of these primers are evaluated. If one or both primers are too long, an alternate annealing sequence is evaluated and the primers are redesigned to fit the new annealing sequence. If one or both of the second set of primers are too long, a third and final annealing region is evaluated as before. B) Mutation counter example. The first base of each sequence is compared. If they are not identical (red text), 1 is added to a counter. If

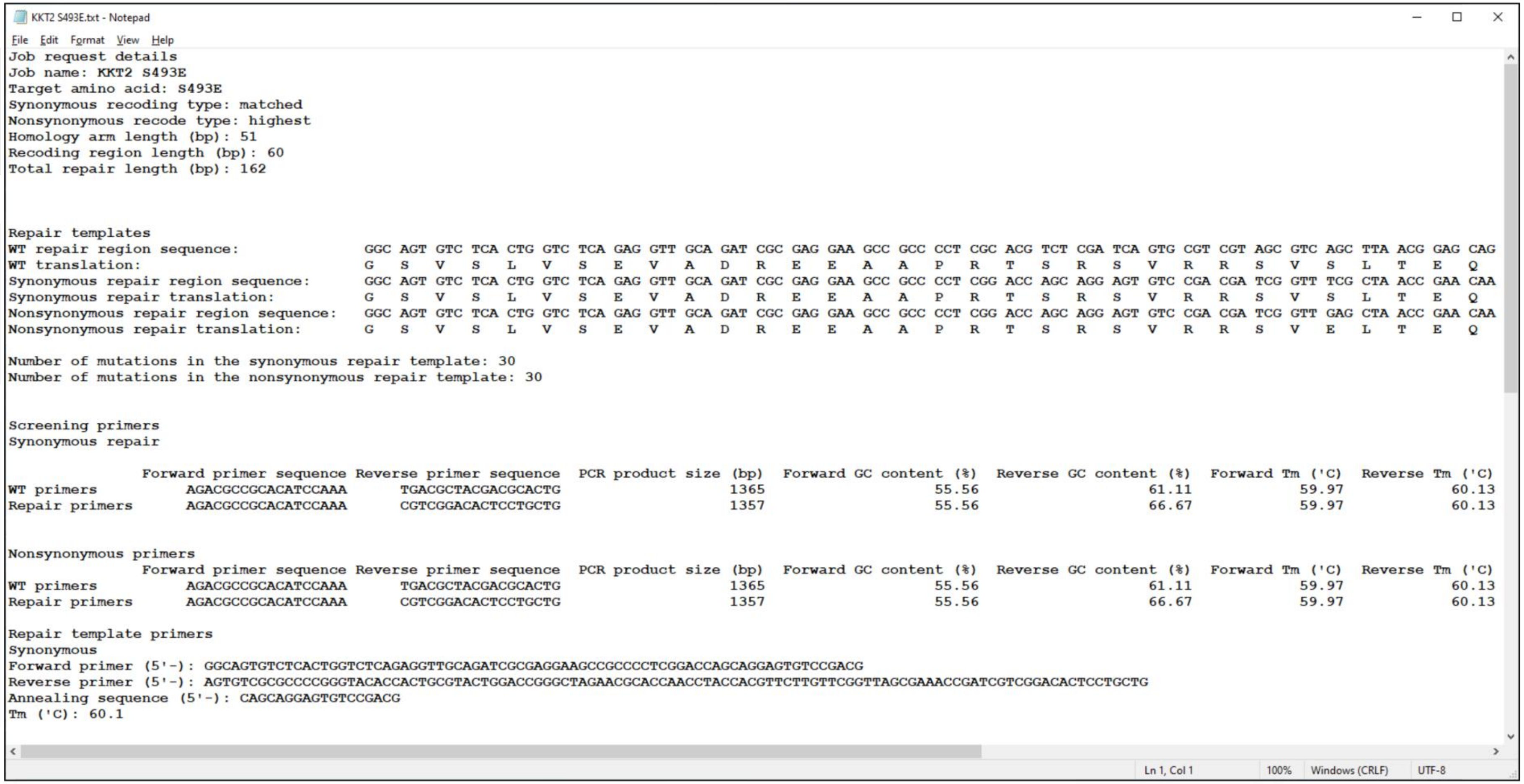
they are identical (black text), nothing is added. The second base of each sequence is then compared in the same way, and so on and so forth. The total number of mutations is then reported in the output file.

5.2.6 COMPLETING THE EXECUTION OF THE CODE AND THE OUTPUT DOCUMENT

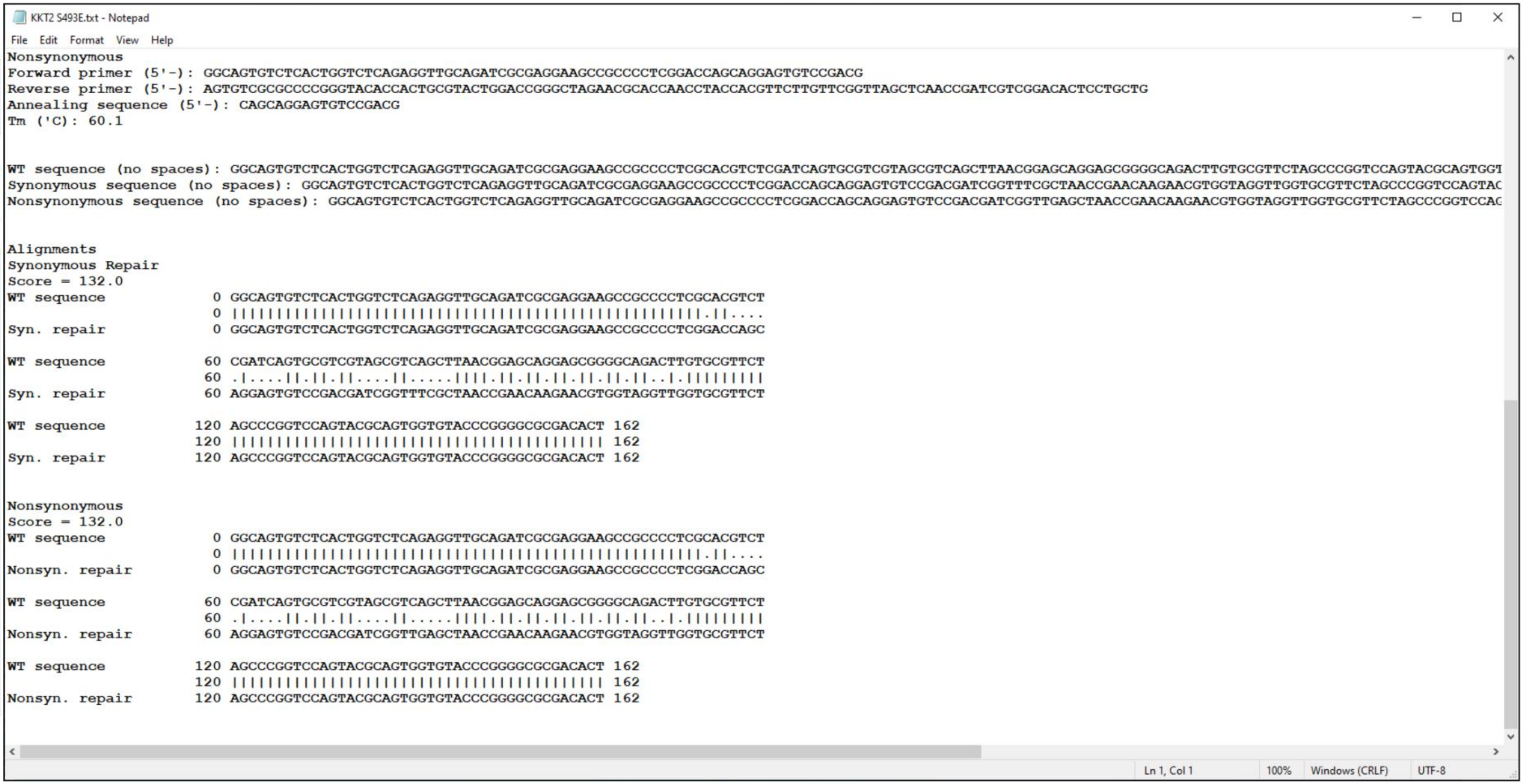
As mentioned in previous sections, the results of the executed script are put into a single file, which is saved as a text document with the job name from the configuration spreadsheet as the filename. Saving the results helps the user organise different designs, and is clearer for reading than displaying the results in the console window of the Python interpreter running the script. However, it should be noted that the script will overwrite any file that has the same name, which could overwrite previous files.

The output document (Figure 20) comprises several sections that each contain different components needed to create precision mutants. Briefly, the output document contains the settings used (for record keeping); the repair template sequences (both with and without coding-frame spacing); screening primer sequences with PCR product sizes, repair primer sequences in a ready-to-purchase 5’ to 3’ format; and pairwise alignments (to visualise the mutations).

On completing the execution of the script, the console also displays a text message to inform the user that it was successful and provides the filename of their output file. Whilst a text document is limited in what information and formatting can be incorporated, this document provides a ready-to-go package of all the sequences and primers needed to generate and screen for precise mutants.



```
Job request details
Job name: KKT2 S493E
Target amino acid: S493E
Synonymous recoding type: matched
Nonsynonymous recode type: highest
Homology arm length (bp): 51
Recoding region length (bp): 60
Total repair length (bp): 162


Repair templates
WT repair region sequence:                GGC AGT GTC TCA CTG GTC TCA GAG GTT GCA GAT CGC GAG GAA GCC GCC CCT CGC ACG TCT CGA TCA GTG CGT CGT AGC GTC AGC TTA ACG GAG CAG
WT translation:                           G   S   V   S   L   V   S   E   V   A   D   R   E   E   A   A   P   R   T   S   R   S   V   R   R   S   V   S   L   T   E   Q
Synonymous repair region sequence:        GGC AGT GTC TCA CTG GTC TCA GAG GTT GCA GAT CGC GAG GAA GCC GCC CCT CGG ACC AGC AGG AGT GTC CGA CGA TCG GTT TCG CTA ACC GAA CAA
Synonymous repair translation:            G   S   V   S   L   V   S   E   V   A   D   R   E   E   A   A   P   R   T   S   R   S   V   R   R   S   V   S   L   T   E   Q
Nonsynonymous repair region sequence:     GGC AGT GTC TCA CTG GTC TCA GAG GTT GCA GAT CGC GAG GAA GCC GCC CCT CGG ACC AGC AGG AGT GTC CGA CGA TCG GTT GAG CTA ACC GAA CAA
Nonsynonymous repair translation:         G   S   V   S   L   V   S   E   V   A   D   R   E   E   A   A   P   R   T   S   R   S   V   R   R   S   V   E   L   T   E   Q

Number of mutations in the synonymous repair template: 30
Number of mutations in the nonsynonymous repair template: 30


Screening primers
Synonymous repair

               Forward primer sequence Reverse primer sequence  PCR product size (bp)  Forward GC content (%)  Reverse GC content (%)  Forward Tm ('C)  Reverse Tm ('C)
WT primers          AGACGCCGCACATCCAAA       TGACGCTACGACGCACTG                    1365                    55.56                   61.11            59.97            60.13
Repair primers      AGACGCCGCACATCCAAA       CGTCGGACACTCCTGCTG                    1357                    55.56                   66.67            59.97            60.13


Nonsynonymous primers
               Forward primer sequence Reverse primer sequence  PCR product size (bp)  Forward GC content (%)  Reverse GC content (%)  Forward Tm ('C)  Reverse Tm ('C)
WT primers          AGACGCCGCACATCCAAA       TGACGCTACGACGCACTG                    1365                    55.56                   61.11            59.97            60.13
Repair primers      AGACGCCGCACATCCAAA       CGTCGGACACTCCTGCTG                    1357                    55.56                   66.67            59.97            60.13

Repair template primers
Synonymous
Forward primer (5'-): GGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCCCCTCGGACCAGCAGGAGTGTCCGACG
Reverse primer (5'-): AGTGTCGCGCCCCGGGTACACCACTGCGTACTGGACCGGGCTAGAACGCACCAACCTACCACGTTCTTGTTCGGTTAGCGAAACCGATCGTCGGACACTCCTGCTG
Annealing sequence (5'-): CAGCAGGAGTGTCCGACG
Tm ('C): 60.1
```



```
Nonsynonymous
Forward primer (5'-): GGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCCCCTCGGACCAGCAGGAGTGTCCGACG
Reverse primer (5'-): AGTGTCGCGCCCCGGGTACACCACTGCGTACTGGACCGGGCTAGAACGCACCAACCTACCACGTTCTTGTTCGGTTAGCTCAACCGATCGTCGGACACTCCTGCTG
Annealing sequence (5'-): CAGCAGGAGTGTCCGACG
Tm ('C): 60.1


WT sequence (no spaces): GGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCCCCTCGCACGTCTCGATCAGTGCGTCGTAGCGTCAGCTTAACGGAGCAGGAGCGGGGCAGACTTGTGCGTTCTAGCCCGGTCCAGTACGCAGTGGT
Synonymous sequence (no spaces): GGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCCCCTCGGACCAGCAGGAGTGTCCGACGATCGGTTTCGCTAACCGAACAAGAACGTGGTAGGTTGGTGCGTTCTAGCCCGGTCCAGTAC
Nonsynonymous sequence (no spaces): GGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCCCCTCGGACCAGCAGGAGTGTCCGACGATCGGTTGAGCTAACCGAACAAGAACGTGGTAGGTTGGTGCGTTCTAGCCCGGTCCAC


Alignments
Synonymous Repair
Score = 132.0
WT sequence              0 GGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCCCCTCGCACGTCT
                         0 |||||||||||||||||||||||||||||||||||||||||||||||||||||.||....
Syn. repair              0 GGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCCCCTCGGACCAGC

WT sequence             60 CGATCAGTGCGTCGTAGCGTCAGCTTAACGGAGCAGGAGCGGGGCAGACTTGTGCGTTCT
                        60 .|....||.||.||....||.....||||.||.||.||.||.||.||..|.|||||||||
Syn. repair             60 AGGAGTGTCCGACGATCGGTTTCGCTAACCGAACAAGAACGTGGTAGGTTGGTGCGTTCT

WT sequence            120 AGCCCGGTCCAGTACGCAGTGGTGTACCCGGGGCGCGACACT 162
                       120 |||||||||||||||||||||||||||||||||||||||||| 162
Syn. repair            120 AGCCCGGTCCAGTACGCAGTGGTGTACCCGGGGCGCGACACT 162


Nonsynonymous
Score = 132.0
WT sequence              0 GGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCCCCTCGCACGTCT
                         0 |||||||||||||||||||||||||||||||||||||||||||||||||||||.||....
Nonsyn. repair           0 GGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCCCCTCGGACCAGC

WT sequence             60 CGATCAGTGCGTCGTAGCGTCAGCTTAACGGAGCAGGAGCGGGGCAGACTTGTGCGTTCT
                        60 .|....||.||.||....||.....||||.||.||.||.||.||.||..|.|||||||||
Nonsyn. repair          60 AGGAGTGTCCGACGATCGGTTGAGCTAACCGAACAAGAACGTGGTAGGTTGGTGCGTTCT

WT sequence            120 AGCCCGGTCCAGTACGCAGTGGTGTACCCGGGGCGCGACACT 162
                       120 |||||||||||||||||||||||||||||||||||||||||| 162
Nonsyn. repair         120 AGCCCGGTCCAGTACGCAGTGGTGTACCCGGGGCGCGACACT 162
```

Figure 20. Example output file generated by the Python script. Colours (left) indicate the different sections. Red – job details specified by the user. Orange – Sham multiple sequence alignment of the WT sequence, and both repair templates, as well as counts of the number of mutations each repair template has. The end of the sequences have been cropped of for legibility. Yellow – Screening primers and corresponding information. Green – Primer sequences to generate the repair template sequences. Cyan – The WT and both repair sequences without any spaces or additional characters. The end of the sequences have been cropped of for legibility. Magenta – Pairwise sequence alignments of the WT sequence with each of the repair templates (| indicates identity, • indicates non-identity). All DNA sequences are in the 5' to 3' orientation. A copy of the text in this file is available in a larger font size in Appendix 7.2.9.

5.3 Results

The script created and described here is able to generate recoded repair templates up to 220 bp long for continuous coding sequences (i.e. no introns). It has been tested on DNA sequences from both *Leishmania mexicana* and *Trypanosoma brucei*, and was successfully able to design them within a matter of seconds (although results will vary with different computer's memory availability). The script is instructed by a configuration Excel spreadsheet and exports the results into a text document, both of which increase accessibility for non-programmers. Additionally, all the required software and packages are freely available.

The script has also been designed to take away tedious jobs from the user, so it is able to manipulate codon tables provided by https://www.kazusa.or.jp/codon/, which are given as RNA sequences, and reformats them into DNA sequences. This only requires the user to go to the website, retrieve the codon usage table for their organism, and then copy, paste and save the data (Figure 17B and C). This reduces the burden on the user, so that fewer mistakes are made. Additionally, once prepared, the codon table is reusable to apply to any target gene in the same organism. Other tedious tasks that the script completes include recoding each triplet code in the desired sequence, visualising each point mutation, and assessing annealing sequences for primers to generate the repair template. Having personally created many repair templates manually, these tasks can take hours in total, especially when including checking for human errors. However, this script is able to perform each of these tasks consistently and far quicker than any human.

The script is (somewhat) able to understand which part of a DNA sequence is coding and which is not, as guided by the user. This allows the user to prepare a single FASTA file for an entire gene sequence, including 5’ and 3’ untranslated regions (UTRs), to generate repair templates for as many sites as they wish in that gene by just providing the target amino acid residue and amino acid number. Whilst it would have been possible only require the amino acid number without the corresponding amino acid identity, requiring the user to provide the identity was intentionally chosen to provide opportunity for the user to identify mistakes, as prompted by the script’s checking mechanisms. If the DNA sequence of the given residue number does not correspond to the input amino acid residue, it will output a text-based error message in the console. In this instance, the script will continue to run, unless it encounters further issues. Likewise, the script will check that the user has correctly specified a coding sequence that is in frame i.e. the coding sequence length is a multiple of three. Currently, the script does not recognise coding sequences by the presence of start and stop codons. This has the benefit that a user can provide only a partial gene sequence, as long as it is in-frame. In this case, the script will treat the specified start and end as the “real” start and end, and will act in accordance with the special cases in Figure 15 part 1B, so it is not recommended to do this. Additionally, if the user supplies a sequence that is a multiple of three base pairs long, but from a nonsense frame, the script will still “recode” in the +1 frame. Hopefully it will be apparent to the user that the protein sequence they expected is not correct in the output, even if the code has not detected a difference in the expected amino acid residue and number.

5.4 Limitations and Future Directions

A key next step for this script is testing the designs it produces *in vitro*. Whilst the script generates sequences that seem sensible, those sequences are only useful if they have real world tractability. As such, designs using a variety of the settings should be tested on targets that have already been shown to be possible to mutate. Using a previously mutated site and the same sgRNAs will ensure that a failure to mutate the site is because of some aspect of the repair template design, rather than leading to questions about its essentiality. Additionally, this experiment could also assess whether one of the design strategies is favoured for incorporation over the others, and whether different sequence compositions have different effects on the cell.

Any computer programme is limited to what its programming tells it to do. In the modern era, we are all familiar with words such as “glitch” and “bug” in reference to programmes not performing the expected task, caused by mistakes in a piece of software’s code. Whilst this script has been tested on a variety of different inputs to remove as many issues as possible, there are still issues with this version of the script. Most of the known issues, regard features that are lacking or imperfect from a biologist’s perspective. However, from a programmer’s perspective, the main issue with the script at present is that it lacks proper error catching mechanisms. Currently, unless an issue arises that means the script is unable to perform a task, the script will continue to completion. The current “error catching” is only simple if clauses which when activated print out error related text. This text can be easy to miss in the console of the user’s Python interpreter and the presence of an output file may lead a user to believe the script has performed the task as expected, when it has not done so. In future versions of this script, it would be prudent to incorporate proper error catching mechanisms into it, which will cancel or stall the script if there are issues with the inputs or with the calculations. These errors are also harder for users to miss, as they involve brightly coloured text and error codes, which the user can use to investigate further.

As for the biological issues with the code, the most major set of issues is with how this script interprets what is a coding sequence, how it should be translated and hence what DNA sequences should or should not be translated. This script does not use coding sequence detection methods to identify which part of the DNA sequence are coding and which are not. Instead, the user specifies where the coding sequence starts and ends, and the script verifies that this specified region has a length that is a multiple of three, corresponding to complete triplet codes. As such, this script interprets any string of A’s, C’s, G’s and T’s that has a length which is divisible by three to be suitable DNA for recoding. Technically, other letter characters may also be translated if they are used to represent combinations of bases such as R for purines and Y for pyrimidines, either yielding "X" or a real amino acid if the character is in the wobble base position. Even if the sequence provided includes several stop codons, which are obvious to any biologist as being either nonsense or out of frame, the script will try to recode it. It only treats a stop codon differently for the specified end of the gene. However, this stop codon will still be recoded to an alternate one, if it is included in the recoding region (see Figure 15 part 1B for this special case). The script is able to detect incorrect target site translations, so it would be expected that in most of these instances, the desired residue to mutate would not match the input target residue. But, there is a possibility that there is a combination of DNA bases that match the target amino acid in the correct position, and hence the code completes. At present, it is hoped that the user will be able to identify an issue has occurred from the displayed coding sequence in the output file being noticeably wrong. Going forwards, it would be best to add an error catching mechanism that halts the progression of the code if the entire recoding region's translation does not match the input. Alternatively, or in conjunction to that, adding coding sequence detection may prevent some of these errors going unnoticed.

On a related note, if the DNA sequence provided is not a multiple of three, text-based error messages are displayed. In some tests, it was noticed that if the repair template's length was not a multiple of three, the code would still continue to completion because the current error catching mechanisms do not prevent the code from completing. In these instances, the last codon of the repair template (i.e. in the homology arm) was presumed to be missing a base and so was not translated. On the assumption that it was in fact the last codon which was missing a base, this response is acceptable and is similar to how most commercially available programs would interpret the sequence. However, if in fact the base was missing from the start of the repair template, the script identifies the codons from the start of the sequence, so in effect "causes" a frame shift before calculating the translation. Going forwards, it would be good to add a feature that can handle repair template lengths that are not a multiple of three, perhaps by extracting the additional bases from the reference sequence, that are removed in the output.

As eluded to, because the present version of the script can only correctly recode in-frame DNA, the user is limited to selecting homology arm lengths and recoding region lengths that are multiples of three to avoid these issues. This is not inherently problematic, but does restrict flexibility and creates opportunities for mistakes that cause the script to fail to complete or produce nonsensical results. So in future versions, it would be good to either restrict the inputs that the user can pick from to being values that are multiples of three, or to find workarounds for the situations when they are not.

An additional restriction the user has on their input DNA sequence is that this script can only handle continuous coding sequences. As such, the script is unable to provide repair templates for non-coding regions (although it can use non-coding DNA for homology arms if the target is in proximity to the start or end of a gene), and it cannot recode coding sequences that contain introns. As this script was designed for use in *Leishmania* species and other kinetoplastids, this is generally not a problem. Few genes in these species contain introns. However, it does mean that this script could not be used for higher eukaryotes that have much more complex gene structures, although it may work in some prokaryotic species (currently untested). Likewise, this script was designed to generate mutations in coding sequences, so it being unable to mutate untranslated sequences is not inherently a problem, but it does restrict its use-case somewhat. It may be possible to design a similar script to modify non-coding DNA sequences, but most likely, this would have to be on a case by case basis, as it is unlikely that mutations of interest for non-coding RNAs would be transferable to splicing signals, centromere sequences, or promoter sequences for example.

Similarly, the script can only recode one continuous block of sequence. As discussed already, it can recode alternating codons from within a continuous block. But at present, it is not possible to design a repair template which has a stretch of recoding, followed by a break in recoding, followed by another region of continuous recoding, as was used for some repair templates tested in this project (for example KKT2 S493 repair templates). During some transfections, certain clones showed evidence of recombination at extended breaks in recoding (≥11 bp of continuous WT sequence) – these have been labelled as “complex”. To minimise chances of this happening, it seems advisable to avoid incorporation of such large blocks of WT sequence (even though the use of the alternating settings when n>3 could produce this). As such, this feature was not developed for this script. However, if there is demand for such a feature from the field, it would be reasonably straightforward to incorporate, using a similar strategy to the alternating recoding options that this script already includes.

As this script was designed for coding sequences, the translations of each triplet code have been manually written into this code. As discussed previously, Biopython does include a range of codon tables, but it does not have full two-way directionality. When requesting the translation for any triplet code, it was able to recall the amino acid. However, when requesting the codons that correspond to an amino acid, it would only produce one of the triplet codes, regardless of how many there are (which could be up to six). So two functions were created to ensure that all triplet codes were recalled – one with, and one without frequency usage data. However, because this was manually coded into the script, this version of the script only recalls codons corresponding to the standard codon table, and does not know how to call abnormal codon usage sequences. In a future version, it would be possible to recreate these functions for all known codon usage tables, and require the user to select which table to translate from, although this is not necessary for kinetoplastids.

Whilst the script is designed for coding mutations, it has been designed to only induce a single nonsynonymous mutation. There are instances where a pair or small number of coding mutations may be necessary in a single repair template, such as was trialled with KKT2 S505 and S506 double mutants using ssDNA in this project. As the main version of the code presented here did not allow for multiple mutations (so would have to be adjusted manually, which also impacts all of the primer designs), a first draft of a modified version of the code which can generate multiple mutations has been created. This version allows up to 5 mutations per repair template and is available in Appendix 7.2.19. This version requires the user to specify the number of mutations up to 5 (so can be used for a single mutation if desired), as well as the amino acid residue and number for each of them. It also has some more complex assessments of how to distribute the recoding region such that all target sites are incorporated if they are unevenly spaced. In short, if the target sites are evenly spaced, the recoding region will be centred around the middle of those sites. But if one target site is distant from the others, the recoding region will be adjusted to ensure that all target mutations are within the coding region, and any additional codons to record are spread as evenly as possible, flanking the target sites. However, this version has not been tested as thoroughly as the main version presented, so may contain some currently unknown bugs.

At present, this script does not automate the entire process of designing a repair template. The largest missing feature is the capacity to design sgRNAs for the target of interest. For this current version of the script, the user has to complete this step themselves, and then inputs the region to be recoded to cover the sgRNAs they have chosen, if they wish to replicate the methods used here. This step is a key part of creating the repair template, so will definitely be investigated for incorporation into future versions.

In order to incorporate sgRNA design into the script, the script also needs to be able to complete BLAST searches of the entire genome to ensure the sgRNA sequence is unique. Primer design would also be improved if BLAST searching were incorporated as part of the design quality checking process, to reduce the possibility of off-target primer binding during the screening PCR. The current screening primers are designed only in reference to the input sequence. This could mean that the screening primers generated are not consistently specific enough to recognise only the target gene, especially if the target gene has very similar homologs in the genome. The Biopython package does offer BLAST searching capabilities using the NCBI (National Center for Biotechnology Information) servers via an internet connection or locally on the computer running the search. There are pros and cons to both local and remote BLAST searching. The script at present runs entirely locally, meaning it can be used without connection to the internet (after initial installation of the relevant packages), so local installation would maintain that aspect of the initial code. In some cases, it may also be faster to run locally as the user is not “competing” with other users for the server memory to complete the search. Additionally, local running retains confidentiality of the sequences (Cock et al., 2009). However, local BLAST searching requires a lot more set up, which would have to be completed on each user’s device to achieve, and that user would also require a database of the genomes to BLAST against on their device too. As such, it is most likely that future versions of this script will use online BLAST searching, to minimise the set up required by the user of the script (Cock et al., 2009). This could have the benefit that the user will be able to access newer versions of genomes when they are published, with minimal extra steps to use.

Another design feature that would be beneficial to have in the script would be either the inclusion of a restriction site change in the sequence or at least an assessment of the new sequence for restriction site changes. Based on the practical experiments completed in this project, restriction enzymes seemed to be more accurate predictors of the genotype than PCRs when restriction sites were in close proximity to the target site. However, the results were sometimes less clear to interpret. In addition, when the designed primers conferred poor specificity at distinguishing WT from mutant sequence, restriction enzymes served as a useful backup. It was generally more laborious to screen by restriction digest, and comes with potentially higher costs (depending on the enzyme). But if the user has this information at the point of design, they can make informed choices about which way they wish to screen their mutants. Alternatively, they may choose a different design that better suits the reagents they have on-hand. It should be noted that the additional labour to screen by restriction digest largely comes from requiring a PCR amplification step to generate the DNA to be digested on all clones being screened. Therefore, if a user intends to use restriction site analysis, an additional primer set would be required. As such, primer design for this should be added to the script also. Alternatively, a separate script could be developed to allow the user to provide the repair template sequences and primer sequences that they have on-hand to suggest a restriction digest strategy, should the user have issues with PCR-based analysis.

To further expand this script into a high-throughput tool, it would be beneficial to add batch job capacity. To tackle this problem, another version of the script was developed with batch capabilities and is available in Appendix 7.2.18. In short, this version uses a for-loop to iterate over the script several times, with each iteration corresponding to a column in a modified version of the Excel configuration form, where each column is a different job to execute. Surprisingly, execution of this script with 7 input columns did not take much longer to run. However, further testing of this version is required to determine the limitations and potential bugs that may exist in the new parts of the code.

Lastly, it would greatly improve accessibility of this script if the script could be hosted on a website. There are already web-based tools for designing sgRNAs (http://grna.ctegd.uga.edu/), and for designing CRISPR-Cas9 edits (http://www.leishgedit.net/Home.html) which have been crucial in the field for much of the recent work involving gene editing. Being web-based allows potentially global access to the script developed here, which could have the effect of standardising this methodology across the field. It would also allow improvement to the user-interface. At present, the use of an Excel spreadsheet “form” was chosen to make using the script less intimidating for non-programmers. However, setting up the script on the user’s device and executing the code still requires users to interact with a programming software, which can be quite intimidating. All of this could be hidden from the user on a website, only requiring them to fill in a more user-friendly form. A website may also allow visualisation options to be created to show the input sequence, its translation, and how the repair templates and primers designed fit in with that.

5.5 Summary and Conclusion

Overall, the script created here completes the tasks it was programmed to do successfully, which can help the user design repair templates and screening primers quickly. It has been tested on DNA sequences from both *L. mexicana* and *T. brucei*. The script does not complete every task necessary to complete this precision editing methodology from scratch, but none of the missing features are untenable, and much of the labour load is reduced for the user when using this script in its current version. The missing features are hoped to be included in future versions to further improve on the work completed here, especially improving accessibility to use this script through hosting it on a website. At present, as long as the user understands the limitations of the script discussed here, it can be used effectively. Further testing is needed to assess whether the repair templates and screening primers designed with this tool are effective *in vitro*.

6 CHAPTER FIVE – GENERAL DISCUSSION

6.1 Discussion

This project set out to establish an efficient precision editing methodology for *Leishmania mexicana* in order to generate kinetochore phosphosite mutants. Two methods were trialled to engineer the desired amino acid substitutions: using 120 nt single-stranded DNA (ssDNA) repair templates and 160 bp double-stranded DNA (dsDNA) repair templates targeting the genomic DNA. Whilst both methods were able to generate some precision edited mutants, there was a stark contrast in the efficiency between them. dsDNA repair templates were about 15-fold more efficient on average than ssDNA repair templates.

Successful generation of a range of kinetochore phosphosite mutants allowed for investigation into the effects of these mutations. Most mutations did not result in a statistically significant change in growth rate or cell cycle progression (Figure 12). Additionally, the kinetochore phosphosite mutants were assessed for cell cycle defects, which largely showed no change compared to WT. However, two clones with mutations in KKT2 showed an apparent triploid DNA content following continual passage (Figure 14), although the secondary clone for each of these cell lines failed to show the same phenotype. Additionally, the phenotype was not replicated when the experiment was repeated using a fresh sample of cells taken from cryo-storage. *Leishmania* are renowned for ploidy changes due to their high genome plasticity, especially in response to stress. The underlying mechanisms as to the drivers of these ploidy changes are not currently known, but reports of such events are common under a wide array of circumstances (Black et al., 2023). Given that the phenotype was not seen in both clones assessed and that it was not repeatable, it suggests that these were likely random events. Whether the mutations induced in KKT2 lead to an increased probability of these events happening remains to be seen.

To enable expansion of the technique developed here into higher throughput systems, a Python script was developed to automate the design process. Not only does the script design the repair template, but also generates PCR screening primers and long primers to generate the repair templates it has designed. The script has several customisation options concerning how the recoding is completed, to allow users to both mimic the strategy used here, but also to try alternate designs should the former approach not work. Whilst this programme is currently lacking a few desired features to complete the entire repair template design process, namely protospacer identification and sgRNA design, in its current version, it is already a functional tool. Two other versions of the tool have also been drafted. The first includes batch design of several repair templates from a single execution of the script (Appendix 7.2.18) and the second generates repair templates containing up to five nonsynonymous mutations in close proximity (Appendix 7.2.19).

The purpose of creating a precision editing methodology that is efficient and convenient was largely to allow investigation into essential genes, without complete loss of the target protein. Currently in *Leishmania*, essential genes can be investigated through inducible deletion such as the DiCre LoxP system (Duncan, Jones and Mottram, 2017), episomal expression prior to genomic deletion, or using *ex vivo/in vitro* approaches such as recombinant protein expression. In the case of using DiCre recombinase, so called "leaky" expression can still be a problem, whereby a small proportion of cells escape deletion of the target locus. It is also a complex process to set up a cell line, typically requiring several rounds of transfection and screening of suitable clones. Whilst other kinetoplastids have RNAi machinery that can be used for inducible deletion, most *Leishmania* species lack RNAi machinery, including *L. mexicana* (Ullu, Tschudi and Chakraborty, 2004). Episomal expression of either WT or mutant versions of a target protein can often have the effect of dysregulated expression, typically in the form of over expression of the target protein, which can have cytotoxic effects. Recombinant expression can allow study of the target protein but takes it out of the cellular context of that protein. Additionally, some proteins are far more challenging to express and purify than others, and there is no way to determine this in advance, especially as many of the kinetochore proteins do not have identifiable protein domains. Moreover, interactors or substrate proteins may need to be expressed to gain any functional insights, only adding work to an already labour-intensive process.

The benefits of precision editing have not gone unnoticed by the community, with examples of targeted editing using small selection-free constructs being used by a wide array of groups (Zhang and Matlashewski, 2015; Crawford et al., 2017; Medeiros et al., 2017; Janssen et al., 2018; Rico et al., 2018; Wall et al., 2018; Lander and Chiurillo, 2019; Pal and Dam, 2022). However, the similarities end there, with each group using different construct lengths, with different homology arm lengths, and some using ssDNA whilst others use dsDNA. Clearly all the methods used were able to generate the desired mutations but with varying levels of success. Broadly speaking the methodology was not the focus for these previous groups' work, but was a means to investigate something of greater interest. Standardising the process could help to reduce time and money wasted in the community, as has been the case through the introduction of tools such as the CRISPR-Cas9 toolkit developed by Beneke et al. (2017). It would also open up the methodology to groups who may have been put off by the laboriousness of the previously published methods which may not even generate the desired mutations.

Uses for the precision editing cover a wide array of different biological questions. As well as investigation into post-translational modifications, as was explored here, precision editing could also be used to explore organelle targeting motifs, catalytic residues of enzymes, protein-protein and protein-non-protein interface interactions, processing signals of pro-proteins, and even potentially the effects of specific residues on protein structure and stability. All of these events require specific amino acid residues in specific locations of the protein, and so modifying those residues allows understanding into why these proteins have evolved to have the sequences and structures that are observed. Doing so in the most native context possible is important to ensure that interpretation of the results is accurate and not due to an artificial situation. For example, an *in vitro* expression of a mutant version of an enzyme could still detect catalytic activity at low levels when the substrates are provided in excess. But that same enzyme could be effectively non-functional in a cellular setting where substrates exist in lower concentrations with temporal control. Or indeed the opposite could be true that in a cellular setting, additional post-translational modifications could increase catalytic activity, which were absent in the recombinantly produced protein e.g. glycosylation is absent if the recombinant protein is made using *Escherichia coli*. Thus, it is important to supplement such *in vitro* experiments with studies *in cellulo*.

Other potential uses of precision editing could include adding small protein tags in a selection free manner, modifying antibody-binding epitopes to enable use of non-kinetoplastid commercial antibodies in molecular biological techniques, or generating a live attenuated vaccine. This project has not attempted to add sequence such as a protein tag into the genome. Given that a 3xHA tag is only 24 amino acids long, equating to 72 bp of DNA sequence, it is not much larger than the constructs used here (editing window of 60 bp, equivalent to 20 amino acids). The difference in size of the construct could affect the editing efficiency, but it stands to reason that this is within the realm of possible, so could offer selection-free tagging for cell lines that already contain larger numbers of antibiotic resistance genes. Especially as similar approaches have already been successful in *T. brucei* (Kovářová et al., 2022). Adding such tags is frequently necessary for techniques such as western blotting and immunofluorescence microscopy in kinetoplastids as most commercial antibodies are against protein epitopes that are not present in the kinetoplastid orthologs of the target proteins. This can become problematic if the gene of interest does not respond well to the addition of an epitope tag, particularly with large disruptions to UTRs from the insertion of antibiotic resistance makers. As such, an alternate approach could be to modify the target protein to become humanised or equivalent so that commercial antibodies could be used against it. This would not be applicable to the most divergent genes in *Leishmania*, but might increase the diversity of usable commercial antibodies. Antibody recognition of proteins is a key defence for the host immune response to a *Leishmania* infection. Since *Leishmania* do not rely on antigenic variation like their *Trypanosoma* cousins, any exposed surface proteins are likely to remain constant throughout infection. This suggests there is an opportunity to create a mutant cell line that could be attenuated during infection for the purposes of vaccine development. Surface proteins are one possible target, but since the majority of the mammalian host infection is intracellular, the immune system has limited opportunity to develop antibodies against metacyclic promastigotes or extracellular amastigotes before they are phagocytosed. Attenuating the parasite’s ability to invade immune cells could allow the immune system to have the time to activate the adaptive immune system to generate antibodies against the *Leishmania* cell. Alternatively, modifying the amastigote’s capacity to manipulate the host macrophage could allow the macrophage to process and present antigens to activate an immune pathway that is less reliant on antibody generation, which may be able to clear the infection.

As more groups use precision editing for more diverse studies, there becomes a greater need for tools to help with the process. For the small number of targets investigated in this project, manual design was sufficient, albeit time consuming. As one of the aims of this project was to investigate methods to scale up this approach, it became increasingly apparent that in order to create libraries worth of mutants, it would first be necessary to design them. Designing potentially hundreds of repair templates and screening primer sets individually was undoubtedly going to be extremely time consuming to do manually, and would likely result in errors in some repair templates. Even though the method developed here would not allow for a bulk library transfection to generate a mix of mutants, a library could still be created by generating mutants and then combining confirmed clones together. As this method is currently only about 30% efficient, combining confirmed clones to create a pool is a wiser approach than using a population of cells that could be largely WT after transfection. To open up this avenue for future studies, a tool was created using Python to generate the repair templates, screening primers and repair production primers. Further work is needed to complete the sgRNA design process, which currently has to be completed manually, but relieving the workload of the other design steps makes this process more tenable than before.

To expand this technique into a bulk approach, significant modification would be needed. One major challenge when doing a library-style transfection is the identification of which cells in a population contain the mutations of interest. In previous studies such as Baker et al. (2021), barcodes were used which can be identified through Illumina sequencing. In this approach, a common sequence surrounding all barcodes allows for amplification of every barcode, no matter where it is integrated in the genome. This allows assessment of all mutations in the population from a single PCR reaction. However, it is not possible to incorporate a barcode that is common to all targets in precisely edited mutants targeting protein coding genes, as this would change the protein sequence encoded by the CDS. Alternatively, DNA encoding the sgRNA can be used as a barcode if it is either incorporated endogenously in a neutral locus, or if it is provided on an episome (plasmid) which is maintained by the cell. Endogenous incorporation of DNA encoding sgRNA for precision CRISPR editing has been done in *T. brucei* (Rico et al., 2018). In this instance, a construct containing the DNA to transcribe into the sgRNA, a T7 promoter, and a hepatitis delta virus

(HDV) ribozyme was integrated into a spacer region of the ribosomal DNA locus, in a cell line with tetracycline-inducible Cas9 expression. The HDV ribozyme is a self-cleaving RNA when transcribed by T7 polymerase, which releases free sgRNA to direct the Cas9 break (Rico et al., 2018). As such, it would be possible to amplify the DNA encoding the sgRNA from common regions of the integrated construct for library-scale assessment of mutations in the population. Along this premise, Engstler and Beneke (2023) transfected four *Leishmania* species with a series of plasmids containing sgRNAs and a Cas9-fusion cytosine base editor protein. Because the plasmids had a common backbone and were under constant selection, the presence of each targeted mutation in the population could be tracked by amplifying and sequencing the region of the plasmid containing the sgRNA target sequence. This suggests that a plasmid based sgRNA strategy could provide options for scaling up this precision editing strategy in a traceable manner. One potential challenge with this method would be developing a plasmid that incorporates the repair template and the sgRNA, without loss of the plasmid following homologous recombination or translocation of the repair template. It is necessary to link both the repair template and sgRNA on a single plasmid to ensure that cells that received the plasmid have the capacity to complete the precision editing, rather than just making a DSB without a repair template. This ensures that detection of their sgRNA sequence represents mutated cells, rather than cells which only have the guide. Whilst it is known that *Leishmania spp.* can produce circular DNA to use for horizontal gene transfer (Douanne et al., 2022), and it is well established that they can maintain circular DNA constructs, it is not clearly established if circular DNA can be incorporated back into the nuclear genome, and what effect that has on the presence of the circular DNA.

A significant challenge of this project has been working with the plasticity of the *L. mexicana* genome (Black et al., 2023). The absence of most of the components of the Non-Homologous End Joining (NHEJ) pathway in *Leishmania spp.* (Passos-Silva et al., 2010) would lead one to believe that repair of double-stranded DNA breaks (DSBs) would favour faithful homology directed repair (HDR). However, the natural plasticity of the genome has in many instances shown that integration of the desired repair template can occur more flexibly than anticipated. Any integration events in this project which did not incorporate the entirety (or near entirety) on any given allele have been deemed "complex". Most frequently, this was observed as integration of part of the repair template on one allele, but complete integration on the other. It is unclear whether the cause of this integration event was due to only one of the two DSBs occurring on that allele, or whether the parasite was able to use sequence in the middle of the repair template as a micro-homology region to alter the incorporation of the repair template. Repair template designs without breaks in recoding such as KKT2 S25A/E/S and KKT2 S923A/E/S did not detect the presence of this form of complex mutants. In contrast, these complex mutants were detected in KKT2 S493A (using ssDNA) and KKT2 S422S (using dsDNA), both of which contained a break in the recoding of either 11 bp or 18 bp respectively (see Appendices 7.2.4.2 and 7.2.4.4 for repair template designs). It should be noted that complex mutants were detected in KKT2 S923E, which did not have a break in the recoding, but were given this designation for incorporation of a single-nucleotide polymorphism (SNP) on one allele which was not part of the original repair template. Taken together, it is apparent that the plasticity of the genome means that sequencing mutant clones becomes paramount to using this technique, as one cannot just expect traditional homozygous and heterozygous genotypes. The mechanisms underlying this diverse integration of the repair template are currently unknown, and with deeper understanding, could potentially be manipulated in a favourable manner.

A lack of an apparent phenotype in the kinetochore phosphosite mutants generated suggests that regulation of the kinetochore complex is not reliant on single phosphorylation events. More likely, this result suggests that regulation of the kinetochore formation and disassembly is more complex, and could potentially include fail-safes to ensure mitosis can occur correctly even if one protein is disrupted. KKT2, investigated here, has been shown to be crucial for kinetochore assembly following phosphorylation by CLK1/KKT10 in *T. brucei* (Saldivia et al., 2021). The results from this project suggests that either CLK1/KKT10 is still able to phosphorylate the KKT2 mutants generated here such that it can correctly localise and initiate kinetochore assembly, or that correct localisation and initiation of kinetochore assembly are independent of CLK1/KKT10 phosphorylation of KKT2 in *L. mexicana*. Unfortunately, no KKT2 S505 or S506 mutants were generated, which Saldivia et al. (2021) suggested were the *L. mexicana* equivalent of the phosphosite targeted by CLK1/KKT10 in *T. brucei* (KKT2 S508). It would be interesting to reattempt generation of these sites using the dsDNA method to see if they are attainable, and perhaps have a defect in kinetochore assembly. As for KKT4 S422 and KKT7 S304 mutants, both are known to be proximal to KKT3, and are more highly phosphorylated in S- and G2/M-phases (Geoghegan et al., 2022). Phosphorylation of both sites are reduced with AB1 treatment, which inhibits CLK1/KKT10 kinase activity and hence KKT2-mediated kinetochore assembly (Saldivia et al., 2021; Geoghegan et al., 2022). But the kinase responsible for phosphorylating KKT4 S422 is unknown. Another phosphosite on KKT4, S477 in *T. brucei* (equivalent to S590 in *L. mexicana*), is known to be phosphorylated by CLK1/KKT10 and CLK2/KKT19 (Ishii and Akiyoshi, 2020; Geoghegan et al., 2022), but was not shown to be impacted by AB1 treatment in *L. mexicana* (Geoghegan et al., 2022). KKT7 is phosphorylated by CLK1/KKT10 and CLK2/KKT19 in *T. brucei*. However, the equivalent phosphosite to S304 in *T. brucei* (T327) (Geoghegan et al., 2022) lacks the consensus sequence needed to be phosphorylated by CLK1/KKT10 or CLK2/KKT19 (Ishii and Akiyoshi, 2020). KKT7 also seems to be important for recruiting CLK1/KKT10 and CLK2/KKT19 to the kinetochore in an apparently phosphorylation independent manner (Ishii and Akiyoshi, 2020). Taken together, it is unclear what role these phosphorylation events play with regards to kinetochore assembly, function and regulation. But similarly to the results of this project, phosphodeficient mutants did not impact the fitness of the cells in other studies in the literature (Ishii and Akiyoshi, 2020).

6.2 Future Directions

Looking forwards, continued research is needed to convert the current methodology into a library-style high throughput screen. Currently, as the sgRNA guides and repair templates are free pieces of DNA, if two or more target sites were combined in one transfection, then a range of different events could happen. In the best-case scenario, it is possible that mutants for each respective gene are recovered. However, there are also possible scenarios where either poly-mutants are created (i.e. multiple mutations occur in the same cell), or more likely, few cells receive the right combination of guides and repairs to generate the desired mutants, and most cells are either unable to make the dsDNA break or unable to repair the break with the mutated repair template. Both of these scenarios would likely lead to a huge drop off in efficiency, as most breaks would be repaired using genomic copies, if any breaks are made at all. In order to achieve the desired result, ideally, the guides and repair templates need to be a single piece of DNA that could be spliced or manipulated by the cell to release the constituent molecules. Designing a construct, either as a linear piece of DNA or a plasmid, will require some investigation to ensure the guide sequences are available to transcribe into the actual sgRNA, but that the repair template is retained as DNA, without additional bases that could cause frame shifts if incorporated. Plasmids have already been used to deliver repair templates for other CRISPR-directed mutations successfully (Sollelis et al., 2015), as well as being used for guide delivery for a Cas9-base editing fusion protein (Engstler and Beneke, 2023). Use of a plasmid also has the advantage that it can confer antibiotic resistance genes to allow for selection of cells that have taken it up, and to continually promote editing to take place, as was shown using the base-editing Cas9 by Engstler and Beneke (2023). However, as this method did not require a repair template, it requires adapting to determine if it is possible to include a repair template as well. Given that it has already been demonstrated that *Leishmania* can integrate DNA into the genome from a plasmid (Sollelis et al., 2015), and that plasmids can carry usable Cas9 guide sequences (Engstler and Beneke, 2023), creating such a plasmid to achieve precision editing is more a matter of "how" than "if". As well as selection, as plasmids can be maintained in *Leishmania,* they also allow for a form of barcoding. One major issue with the current method is that because all the editing takes place within coding sequences, there is no way to include a unique barcode sequence flanked by shared sequences for amplification. If a library was created, every target would require a separate PCR to screen for its presence in the library, making it untenable for more than a handful of mutations to be combined. However, the contents of the plasmid (i.e. the guide or repair template) could be used as a barcode itself, if flanked by sequences to allow amplification. This could allow for more complex assays to be completed on a wide variety of mutants, as well as allowing bulk transfection to generate such libraries, both of which would really take this methodology to the next level.

Once a proof-of-principle experiment has shown a plasmid could achieve a satisfactory level of precision editing in a library style setting, it will then become paramount to have a completed Python script to generate all the appropriate designs. At present, a version of the script has been generated to allow bulk design, but it needs further testing to remove potential bugs. Incorporating some of the other missing features such as sgRNA design (as discussed in more detail in chapter 5.4) is necessary, but more challenging. sgRNA guide design *en masse* is greatly needed for any scaling up of this method beyond a dozen or so target mutations. As previously discussed, this is a more complex addition to the existing script but is undoubtedly worth the work required to do so. Additionally, the designs generated by the current version of the Python script are intended to be used as free linear DNA molecules, so modifications would need to be made to the outputs of the script to ease integration of these sequences into a plasmid. This could simply by done by designing suitable overhangs on the parental plasmid for Gibson assembly or similar methods which could be added onto the ends of the repair template or guides to allow integration. As functions have already been created within the Python script to “stick” sequences together, this would also be a straightforward modification to incorporate.

As well as the additions and changes to the Python script already mentioned, hosting the code on a website is another goal to work towards in the future. Hosting it on a website will allow production of a more user-friendly interface, allow a wider user base to benefit from it and reduce the set-up required by each user. Currently the script runs on a local machine, which means set-up is required on each device, whereas a website would be accessible on any internet-enabled device from anywhere within the world. It will also allow the set-up of more interactive features, such as highlighting issues with the inputs to enable a user to change them; visualising the repair templates and sequences as a whole; and potentially widening usage to those who are put off by having to work with the code directly.

Lastly, prior to release of the Python code in a public forum, it is necessary to investigate whether all the designs it produces are viable when transfected into cells. Currently, the designs produced have not been tested for integration to generate mutant cell lines. In principle, there is no reason that at least the continuous matched recoding would not integrate, as that is the principle used for the repair templates designed and used in the ssDNA and dsDNA repair templates designed here. Whether the other options created in the Python script produce mutant clones at a similar editing efficiency and without additional effects such as altered transcription/translation speed remains to be seen. Establishing whether the different design options confer different rates of success at generating the mutants could also direct whether all the current options in the Python script would be included in a publicly available option. If one recoding setting conferred a far poorer efficiency than the others across several target sites, then it makes sense to remove that option before releasing the Python script to the public to prevent others from getting poor results also.

7 EPILOGUE

7.1 References

Akiyoshi, B. and Gull, K., 2014. Discovery of Unconventional Kinetochores in Kinetoplastids. *Cell* [Online], 156(6), pp.1247–1258. Available from: https://doi.org/10.1016/J.CELL.2014.01.049 [Accessed 19 May 2023].

Altmann, S., Rico, E., Carvalho, S., Ridgway, M., Trenaman, A., Donnelly, H., Tinti, M., Wyllie, S. and Horn, D., 2013. Oligo targeting for profiling drug resistance mutations in the parasitic trypanosomatids. *Nucleic Acids Research* [Online], 1(1256879), pp.13–14. Available from: https://doi.org/10.1093/NAR/GKAC319 [Accessed 9 May 2022].

Altmann, S., Rico, E., Carvalho, S., Ridgway, M., Trenaman, A., Donnelly, H., Tinti, M., Wyllie, S. and Horn, D., 2022. Oligo targeting for profiling drug resistance mutations in the parasitic trypanosomatids. *Nucleic Acids Research* [Online], 50(14), pp.e79–e79. Available from: https://doi.org/10.1093/NAR/GKAC319 [Accessed 18 September 2024].

Baker, N., Catta-Preta, C.M.C., Neish, R., Sadlova, J., Powell, B., Alves-Ferreira, E.V.C., Geoghegan, V., Carnielli, J.B.T., Newling, K., Hughes, C., Vojtkova, B., Anand, J., Mihut, A., Walrad, P.B., Wilson, L.G., Pitchford, J.W., Volf, P. and Mottram, J.C., 2021. Systematic functional analysis of Leishmania protein kinases identifies regulators of differentiation or survival. *Nature Communications 2021 12:1* [Online], 12(1), pp.1–15. Available from: https://doi.org/10.1038/s41467-021-21360-8 [Accessed 23 July 2021].

Ballmer, D., Carter, W., Hooff, J.J.E. van, Tromer, E.C., Ishii, M., Ludzia, P. and Akiyoshi, B., 2024. Kinetoplastid kinetochore proteins KKT14-KKT15 are divergent Bub1/BubR1-Bub3 proteins. *bioRxiv* [Online], p.2024.01.04.574194. Available from: https://doi.org/10.1101/2024.01.04.574194 [Accessed 30 July 2024].

Beetham, J.K., Donelson, J.E. and Dahlin, R.R., 2003. Surface glycoprotein PSA (GP46) expression during short- and long-term culture of Leishmania chagasi. *Molecular and Biochemical Parasitology* [Online], 131(2), pp.109–117. Available from: https://doi.org/10.1016/S0166-6851(03)00197-X [Accessed 13 June 2023].

Beneke, T., Madden, R., Makin, L., Valli, J., Sunter, J. and Gluenz, E., 2017. A CRISPR Cas9 high-throughput genome editing toolkit for kinetoplastids. *Royal Society Open Science* [Online]. Available from: https://doi.org/10.1098/rsos.170095.

Black, J.A., Reis-Cunha, J.L., Cruz, A.K. and Tosi, L.R.O., 2023. Life in plastic, it's fantastic! How Leishmania exploit genome instability to shape gene expression. *Frontiers in Cellular and Infection Microbiology* [Online], 13, p.1102462. Available from: https://doi.org/10.3389/FCIMB.2023.1102462/BIBTEX [Accessed 28 May 2024].

Borgo, C., D'Amore, C., Sarno, S., Salvi, M. and Ruzzene, M., 2021. Protein kinase CK2: a potential therapeutic target for diverse human diseases. *Signal transduction and targeted therapy* [Online], 6(1). Available from: https://doi.org/10.1038/S41392-021-00567-7 [Accessed 18 September 2024].

Brittingham, A., Miller, M.A., Donelson, J.E. and Wilson, M.E., 2001. Regulation of GP63 mRNA stability in promastigotes of virulent and attenuated Leishmania chagasi. *Molecular and Biochemical Parasitology* [Online], 112(1), pp.51–59. Available from: https://doi.org/10.1016/S0166-6851(00)00346-7 [Accessed 13 June 2023].

Brusini, L., D'Archivio, S., McDonald, J. and Wickstead, B., 2021. Trypanosome KKIP1 Dynamically Links the Inner Kinetochore to a Kinetoplastid Outer Kinetochore Complex. *Frontiers in Cellular and Infection Microbiology* [Online], 11, p.159. Available from: https://doi.org/10.3389/FCIMB.2021.641174/BIBTEX [Accessed 24 May 2023].

Burchmore, R.J.S. and Landfear, S.M., 1998. Differential regulation of multiple glucose transporter genes in Leishmania mexicana. *Journal of Biological Chemistry* [Online], 273(44), pp.29118–29126. Available from: https://doi.org/10.1074/jbc.273.44.29118 [Accessed 13 June 2023].

Burza, S., Croft, S.L. and Boelaert, M., 2018. Leishmaniasis. *The Lancet* [Online], 392(10151), pp.951–970. Available from: https://doi.org/10.1016/S0140-6736(18)31204-2 [Accessed 6 September 2019].

Canton, D.A., Keene, C.D., Swinney, K., Langeberg, L.K., Nguyen, V., Pelletier, L., Pawson, T., Wordeman, L., Stella, N. and Scott, J.D., 2012. Gravin Is a Transitory Effector of Polo-like Kinase 1 during Cell Division. *Molecular Cell* [Online], 48(4), pp.547–559. Available from: https://doi.org/10.1016/J.MOLCEL.2012.09.002 [Accessed 22 September 2023].

Chauhan, N., Vidyarthi, A.S. and Poddar, R., 2011. Comparative Multivariate Analysis of Codon and Amino Acid Usage in Three Leishmania Genomes. *Genomics, Proteomics & Bioinformatics* [Online], 9(6), pp.218–228. Available from: https://doi.org/10.1016/S1672-0229(11)60025-9 [Accessed 12 June 2023].

Cock, P.J.A., Antao, T., Chang, J.T., Chapman, B.A., Cox, C.J., Dalke, A., Friedberg, I., Hamelryck, T., Kauff, F., Wilczynski, B. and De Hoon, M.J.L., 2009. Biopython: freely available Python tools for computational molecular biology and bioinformatics. *Bioinformatics (Oxford, England)* [Online], 25(11), pp.1422–1423. Available from: https://doi.org/10.1093/BIOINFORMATICS/BTP163 [Accessed 16 April 2024].

Codeacademy Team, 2020. *What Is a Programming Language?* [Online]. Available from: https://www.codecademy.com/resources/blog/programming-languages/ [Accessed 16 April 2024].

Crawford, E.D., Quan, J., Horst, J.A., Ebert, D., Wu, W. and DeRisi, J.L., 2017. Plasmid-free CRISPR/Cas9 genome editing in Plasmodium falciparum confirms mutations conferring resistance to the dihydroisoquinolone clinical candidate SJ733. *PLoS ONE* [Online], 12(5). Available from: https://doi.org/10.1371/JOURNAL.PONE.0178163 [Accessed 24 May 2024].

Damasceno, J.D., Reis-Cunha, J., Crouch, K., Beraldi, D., Lapsley, C., Tosi, L.R.O., Bartholomeu, D. and McCulloch, R., 2020. Conditional knockout of RAD51-related genes in Leishmania major reveals a critical role for homologous recombination during genome replication. *PLOS Genetics* [Online], 16(7), p.e1008828. Available from: https://doi.org/10.1371/JOURNAL.PGEN.1008828 [Accessed 8 August 2024].

D'Archivio, S. and Wickstead, B., 2017. Trypanosome outer kinetochore proteins suggest conservation of chromosome segregation machinery across eukaryotes. *Journal of Cell Biology* [Online], 216(2), pp.379–391. Available from: https://doi.org/10.1083/JCB.201608043 [Accessed 4 May 2023].

Douanne, N., Dong, G., Amin, A., Bernardo, L., Blanchette, M., Langlais, D., Olivier, M. and Fernandez-Prada, C., 2022. Leishmania parasites exchange drug-resistance genes through extracellular vesicles. *Cell Reports* [Online], 40(3), p.111121. Available from: https://doi.org/10.1016/J.CELREP.2022.111121 [Accessed 5 July 2023].

Duncan, S.M., Jones, N.G. and Mottram, J.C., 2017. Recent advances in Leishmania reverse genetics: Manipulating a manipulative parasite. *Molecular and Biochemical Parasitology* [Online], 216, pp.30–38. Available from: https://doi.org/10.1016/J.MOLBIOPARA.2017.06.005 [Accessed 6 September 2019].

Engstler, M. and Beneke, T., 2023. Gene editing and scalable functional genomic screening in Leishmania species using the CRISPR/Cas9 cytosine base editor toolbox LeishBASEedit. *eLife* [Online], 12, p.e85605. Available from: https://doi.org/10.7554/eLife.85605.

Fiebig, M., Kelly, S. and Gluenz, E., 2015. Comparative Life Cycle Transcriptomics Revises Leishmania mexicana Genome Annotation and Links a Chromosome Duplication with Parasitism of Vertebrates. *PLOS Pathogens* [Online], 11(10), p.e1005186. Available from: https://doi.org/10.1371/JOURNAL.PPAT.1005186 [Accessed 24 February 2023].

Gasiunas, G., Barrangou, R., Horvath, P. and Siksnys, V., 2012. Cas9-crRNA ribonucleoprotein complex mediates specific DNA cleavage for adaptive immunity in bacteria. *Proceedings of the National Academy of Sciences of the United States of America* [Online], 109(39), pp.E2579–E2586. Available from: https://doi.org/10.1073/PNAS.1208507109/SUPPL_FILE/PNAS.201208507SI.PDF [Accessed 14 October 2022].

Geoghegan, V., Carnielli, J.B.T., Jones, N.G., Saldivia, M., Antoniou, S., Hughes, C., Neish, R., Dowle, A. and Mottram, J.C., 2022. CLK1/CLK2-driven signalling at the Leishmania kinetochore is captured by spatially referenced proximity phosphoproteomics. *Communications Biology 2022 5:1* [Online], 5(1), pp.1–17. Available from: https://doi.org/10.1038/s42003-022-04280-1 [Accessed 28 November 2022].

Harris, C.R., Millman, K.J., van der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M.H., Brett, M., Haldane, A., del Río, J.F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C. and Oliphant, T.E., 2020. Array programming with NumPy. *Nature 2020 585:7825* [Online], 585(7825), pp.357–362. Available from: https://doi.org/10.1038/s41586-020-2649-2 [Accessed 16 April 2024].

Hořejší, Z., Takai, H., Adelman, C.A., Collis, S.J., Flynn, H., Maslen, S., Skehel, J.M., de Lange, T. and Boulton, S.J., 2010. CK2 Phospho-Dependent Binding of R2TP Complex to TEL2 Is Essential for mTOR and SMG1 Stability. *Molecular Cell* [Online], 39(6), pp.839–850. Available from: https://doi.org/10.1016/J.MOLCEL.2010.08.037 [Accessed 25 September 2023].

Horn, D., 2022. Genome-scale RNAi screens in African trypanosomes. *Trends in Parasitology* [Online], 38(2), pp.160–173. Available from: https://doi.org/10.1016/J.PT.2021.09.002 [Accessed 4 May 2023].

Hsu, P.D., Scott, D.A., Weinstein, J.A., Ran, F.A., Konermann, S., Agarwala, V., Li, Y., Fine, E.J., Wu, X., Shalem, O., Cradick, T.J., Marraffini, L.A., Bao, G. and Zhang, F., 2013. DNA targeting specificity of RNA-guided Cas9 nucleases. *Nature Biotechnology 2013 31:9* [Online], 31(9), pp.827–832. Available from: https://doi.org/10.1038/nbt.2647 [Accessed 17 March 2025].

Ishii, M. and Akiyoshi, B., 2020. Characterization of unconventional kinetochore kinases KKT10 and KKT19 in Trypanosoma brucei. *Journal of Cell Science* [Online], 133(8). Available from: https://doi.org/10.1242/JCS.240978/266234/AM/CHARACTERIZATION-OF-UNCONVENTIONAL-KINETOCHORE [Accessed 30 May 2024].

Ishii, M., Ludzia, P., Marcianò, G., Allen, W., Nerusheva, O.O. and Akiyoshi, B., 2022. Divergent polo boxes in KKT2 bind KKT1 to initiate the kinetochore assembly cascade in Trypanosoma brucei. *Molecular Biology of the Cell* [Online], 33(14). Available from: https://doi.org/10.1091/MBC.E22-07-0269-T/ASSET/IMAGES/LARGE/MBC-33-AR143-G006.JPEG [Accessed 22 May 2023].

Ivens, A.C., Peacock, C.S., Worthey, E.A., Murphy, L., Aggarwal, G., Berriman, M., Sisk, E., Rajandream, M.A., Adlem, E., Aert, R., Anupama, A., Apostolou, Z., Attipoe, P., Bason, N., Bauser, C., Beck, A., Beverley, S.M., Bianchettin, G., Borzym, K., Bothe, G., Bruschi, C. V., Collins, M., Cadag, E., Ciarloni, L., Clayton, C., Coulson, R.M.R., Cronin, A., Cruz, A.K., Davies, R.M., De Gaudenzi, J., Dobson, D.E., Duesterhoeft, A., Fazelina, G., Fosker, N., Frasch, A.C., Fraser, A., Fuchs, M., Gabel, C., Goble, A., Goffeau, A., Harris, D., Hertz-Fowler, C., Hilbert, H., Horn, D., Huang, Y., Klages, S., Knights, A., Kube, M., Larke, N., Litvin, L., Lord, A., Louie, T., Marra, M., Masuy, D., Matthews, K., Michaeli, S., Mottram, J.C., Müller-Auer, S., Munden, H., Nelson, S., Norbertczak, H., Oliver, K., O'Neil, S., Pentony, M., Pohl, T.M., Price, C., Purnelle, B., Quail, M.A., Rabbinowitsch, E., Reinhardt, R., Rieger, M., Rinta, J., Robben, J., Robertson, L., Ruiz, J.C., Rutter, S., Saunders, D., Schäfer, M., Schein, J., Schwartz, D.C., Seeger, K., Seyler, A., Sharp, S., Shin, H., Sivam, D., Squares, R., Squares, S., Tosato, V., Vogt, C., Volckaert, G., Wambutt, R., Warren, T., Wedler, H., Woodward, J., Zhou, S., Zimmermann, W., Smith, D.F., Blackwell, J.M., Stuart, K.D., Barrell, B. and Myler, P.J., 2005. The genome of the kinetoplastid parasite, Leishmania major. *Science* [Online], 309(5733), pp.436–442. Available from: https://doi.org/10.1126/SCIENCE.1112680/SUPPL_FILE/IVENS_S1_S2.ZIP [Accessed 12 June 2023].

Janssen, B.D., Chen, Y.P., Molgora, B.M., Wang, S.E., Simoes-Barbosa, A. and Johnson, P.J., 2018. CRISPR/Cas9-mediated gene modification and gene knock out in the human-infective parasite Trichomonas vaginalis. *Scientific Reports 2017 8:1* [Online], 8(1), pp.1–14. Available from: https://doi.org/10.1038/s41598-017-18442-3 [Accessed 24 May 2024].

Jeacock, L., Faria, J. and Horn, D., 2018. Codon usage bias controls mRNA and protein abundance in trypanosomatids. *eLife* [Online], 7. Available from: https://doi.org/10.7554/ELIFE.32496 [Accessed 19 February 2025].

Josephs, E.A., Kocak, D.D., Fitzgibbon, C.J., McMenemy, J., Gersbach, C.A. and Marszalek, P.E., 2015. Structure and specificity of the RNA-guided endonuclease Cas9 during DNA interrogation, target binding and cleavage. *Nucleic Acids Research* [Online], 43(18), p.8924. Available from: https://doi.org/10.1093/NAR/GKV892 [Accessed 10 May 2024].

Keder, A., Rives-Quinto, N., Aerne, B.L., Franco, M., Tapon, N. and Carmena, A., 2015. The Hippo Pathway Core Cassette Regulates Asymmetric Cell Division. *Current Biology* [Online], 25(21), pp.2739–2750. Available from: https://doi.org/10.1016/J.CUB.2015.08.064 [Accessed 25 September 2023].

Kelso, A.A., Waldvogel, S.M., Luthman, A.J. and Sehorn, M.G., 2017. Homologous recombination in protozoan parasites and recombinase inhibitors. *Frontiers in Microbiology* [Online], 8(SEP), p.286759. Available from: https://doi.org/10.3389/FMICB.2017.01716/BIBTEX [Accessed 8 August 2024].

Koressaar, T. and Remm, M., 2007. Enhancements and modifications of primer design program Primer3. *Bioinformatics (Oxford, England)* [Online], 23(10), pp.1289–1291. Available from: https://doi.org/10.1093/BIOINFORMATICS/BTM091 [Accessed 16 April 2024].

Kovářová, J., Novotná, M., Faria, J., Rico, E., Wallace, C., Zoltner, M., Field, M.C. and Horn, D., 2022. CRISPR/Cas9-based precision tagging of essential genes in bloodstream form African trypanosomes. *Molecular and Biochemical Parasitology* [Online], 249, p.111476. Available from: https://doi.org/10.1016/J.MOLBIOPARA.2022.111476 [Accessed 10 October 2022].

Lander, N. and Chiurillo, M.A., 2019. State-of-the-art CRISPR/Cas9 Technology for Genome Editing in Trypanosomatids. *Journal of Eukaryotic Microbiology* [Online], 66(6), pp.981–991. Available from: https://doi.org/10.1111/JEU.12747 [Accessed 24 May 2024].

Lee, K.D., 2011. *Python Programming Fundamentals* [Online]. London: Springer London. Available from: https://doi.org/10.1007/978-1-84996-537-8 [Accessed 16 April 2024].

Llauró, A., Hayashi, H., Bailey, M.E., Wilson, A., Ludzia, P., Asbury, C.L. and Akiyoshi, B., 2018. The kinetoplastid kinetochore protein KKT4 is an unconventional microtubule tip–coupling protein. *Journal of Cell Biology* [Online], 217(11), pp.3886–3900. Available from: https://doi.org/10.1083/JCB.201711181 [Accessed 22 May 2023].

Madusanka, R.K., Silva, H. and Karunaweera, N.D., 2022. Treatment of Cutaneous Leishmaniasis and Insights into Species-Specific Responses: A Narrative Review. *Infectious Diseases and Therapy* [Online], 11(2), pp.695–711. Available from: https://doi.org/10.1007/S40121-022-00602-2/METRICS [Accessed 3 May 2023].

Marchand, A., Sarchione, A., Athanasopoulos, P.S., Bauderlique-Le Roy, H., Goveas, L., Magnez, R., Drouyer, M., Emanuele, M., Ho, F.Y., Liberelle, M., Melnyk, P., Lebègue, N., Thuru, X., Nichols, R.J., Greggio, E., Kortholt, A., Galli, T., Chartier-Harlin, M.C. and Taymans, J.M., 2022. A Phosphosite Mutant Approach on LRRK2 Links Phosphorylation and Dephosphorylation to Protective and Deleterious Markers, Respectively. *Cells* [Online], 11(6), p.1018. Available from: https://doi.org/10.3390/CELLS11061018/S1 [Accessed 19 September 2023].

Marcianò, G., Ishii, M., Nerusheva, O.O. and Akiyoshi, B., 2021. Kinetoplastid kinetochore proteins kkt2 and kkt3 have unique centromere localization domains. *Journal of Cell Biology* [Online], 220(8). Available from: https://doi.org/10.1083/JCB.202101022/212224 [Accessed 23 May 2023].

McConville, M.J. and Naderer, T., 2011. Metabolic Pathways Required for the Intracellular Survival of Leishmania. *https://doi.org/10.1146/annurev-micro-090110-102913* [Online], 65, pp.543–561. Available from: https://doi.org/10.1146/ANNUREV-MICRO-090110-102913 [Accessed 3 May 2023].

Mckinney, W., 2010. *Data Structures for Statistical Computing in Python* [Online]. Available from: [Accessed 16 April 2024].

Medeiros, L.C.S., South, L., Peng, D., Bustamante, J.M., Wang, W., Bunkofske, M., Perumal, N., Sanchez-Valdez, F. and Tarleton, R.L., 2017. Rapid, Selection-Free, High-Efficiency Genome Editing in Protozoan Parasites Using CRISPR-Cas9 Ribonucleoproteins. *mBio* [Online], 8(6). Available from: https://doi.org/10.1128/MBIO.01788-17 [Accessed 24 May 2024].

Munro, A., 2024. *Python* [Online]. Available from: https://www.britannica.com/technology/Python-computer-language [Accessed 16 April 2024].

Musacchio, A. and Desai, A., 2017. A Molecular View of Kinetochore Assembly and Function. *Biology* [Online], 6(1). Available from: https://doi.org/10.3390/BIOLOGY6010005 [Accessed 4 May 2023].

Nascimento, J. de F., Kelly, S., Sunter, J. and Carrington, M., 2018. Codon choice directs constitutive mRNA levels in trypanosomes. *eLife* [Online], 7. Available from: https://doi.org/10.7554/ELIFE.32467 [Accessed 19 February 2025].

Nerusheva, O.O. and Akiyoshi, B., 2016. Divergent polo box domains underpin the unique kinetoplastid kinetochore. *Open Biology* [Online], 6(3). Available from: https://doi.org/10.1098/rsob.150206.

Nerusheva, O.O., Ludzia, P. and Akiyoshi, B., 2019. Identification of four unconventional kinetoplastid kinetochore proteins KKT22–25 in Trypanosoma brucei. *Open Biology* [Online], 9(12). Available from: https://doi.org/10.1098/RSOB.190236 [Accessed 24 May 2023].

Pal, S. and Dam, S., 2022. CRISPR-Cas9: Taming protozoan parasites with bacterial scissor. *Journal of Parasitic Diseases: Official Organ of the Indian Society for Parasitology* [Online], 46(4), p.1204. Available from: https://doi.org/10.1007/S12639-022-01534-X [Accessed 23 May 2024].

Passos-Silva, D.G., Rajão, M.A., Nascimento De Aguiar, P.H., Vieira-Da-Rocha, J.P., Machado, C.R. and Furtado, C., 2010. Overview of DNA repair in Trypanosoma cruzi, Trypanosoma brucei, and Leishmania major. *Journal of Nucleic Acids* [Online], 2010. Available from: https://doi.org/10.4061/2010/840768 [Accessed 17 April 2023].

Peacock, C.S., Seeger, K., Harris, D., Murphy, L., Ruiz, J.C., Quail, M.A., Peters, N., Adlem, E., Tivey, A., Aslett, M., Kerhornou, A., Ivens, A., Fraser, A., Rajandream, M.A., Carver, T., Norbertczak, H., Chillingworth, T., Hance, Z., Jagels, K., Moule, S., Ormond, D., Rutter, S., Squares, R., Whitehead, S., Rabbinowitsch, E., Arrowsmith, C., White, B., Thurston, S., Bringaud, F., Baldauf, S.L., Faulconbridge, A., Jeffares, D., Depledge, D.P., Oyola, S.O., Hilley, J.D., Brito, L.O., Tosi, L.R.O., Barrell, B., Cruz, A.K., Mottram, J.C., Smith, D.F. and Berriman, M., 2007. Comparative genomic analysis of three Leishmania species that cause diverse human disease. *Nature Genetics 2007 39:7* [Online], 39(7), pp.839–847. Available from: https://doi.org/10.1038/ng2053 [Accessed 12 June 2023].

Python Institute, n.d. *About Python* [Online]. Available from: https://pythoninstitute.org/about-python [Accessed 16 April 2024].

Rico, E., Jeacock, L., Kovářová, J. and Horn, D., 2018. Inducible high-efficiency CRISPR-Cas9-targeted gene editing and precision base editing in African trypanosomes. *Scientific Reports 2018 8:1* [Online], 8(1), pp.1–10. Available from: https://doi.org/10.1038/s41598-018-26303-w [Accessed 10 October 2022].

Rogers, M.B., Hilley, J.D., Dickens, N.J., Wilkes, J., Bates, P.A., Depledge, D.P., Harris, D., Her, Y., Herzyk, P., Imamura, H., Otto, T.D., Sanders, M., Seeger, K., Dujardin, J.C., Berriman, M., Smith, D.F., Hertz-Fowler, C. and Mottram, J.C., 2011. Chromosome and gene copy number variation allow major structural change between species and strains of Leishmania. *Genome Research* [Online], 21(12), p.2129. Available from: https://doi.org/10.1101/GR.122945.111 [Accessed 6 August 2024].

Rogers, M.E., Chance, M.L. and Bates, P.A., 2002. The role of promastigote secretory gel in the origin and transmission of the infective stage of Leishmania mexicana by the sandfly Lutzomyia longipalpis. *Parasitology* [Online], 124(5), pp.495–507. Available from: https://doi.org/10.1017/S0031182002001439 [Accessed 3 September 2021].

Sacks, D.L., 1989. Metacyclogenesis in Leishmania promastigotes. *Experimental Parasitology* [Online], 69(1), pp.100–103. Available from: https://doi.org/10.1016/0014-4894(89)90176-8 [Accessed 3 May 2023].

Saldivia, M., Fang, E., Ma, X., Myburgh, E., Carnielli, J.B.T., Bower-Lepts, C., Brown, E., Ritchie, R., Lakshminarayana, S.B., Chen, Y.-L., Patra, D., Ornelas, E., Koh, H.X.Y., Williams, S.L., Supek, F., Paape, D., McCulloch, R., Kaiser, M., Barrett, M.P., Jiricek, J., Diagana, T.T., Mottram, J.C. and Rao, S.P.S., 2020. Targeting the trypanosome kinetochore with CLK1 protein kinase inhibitors. *Nature Microbiology* [Online], 5(10), pp.1207–1216. Available from: https://doi.org/10.1038/s41564-020-0745-6.

Saldivia, M., Wollman, A.J.M., Carnielli, J.B.T., Jones, N.G., Leake, M.C., Bower-Lepts, C., Rao, S.P.S. and Mottram, J.C., 2021. A clk1-kkt2 signaling pathway regulating kinetochore assembly in trypanosoma brucei. *mBio* [Online], 12(3). Available from: https://doi.org/10.1128/MBIO.00687-21/SUPPL_FILE/MBIO.00687-21-S0001.DOCX [Accessed 10 January 2023].

Schweighofer, J., Mulay, B., Hoffmann, I., Vogt, D., Pesenti, M.E. and Musacchio, A., 2024. Interactions with multiple inner kinetochore proteins determine mitotic localization of FACT. *bioRxiv* [Online], p.2024.06.14.599021. Available from: https://doi.org/10.1101/2024.06.14.599021 [Accessed 18 September 2024].

Sollelis, L., Ghorbal, M., Macpherson, C.R., Martins, R.M., Kuk, N., Crobu, L., Bastien, P., Scherf, A., Lopez-Rubio, J.J. and Sterkers, Y., 2015. First efficient CRISPR-Cas9-mediated genome editing in Leishmania parasites. *Cellular Microbiology* [Online], 17(10), pp.1405–1412. Available from: https://doi.org/10.1111/CMI.12456/SUPPINFO [Accessed 2 September 2022].

Sridhar, S. and Fukagawa, T., 2022. Kinetochore Architecture Employs Diverse Linker Strategies Across Evolution. *Frontiers in Cell and Developmental Biology* [Online], 10, p.862637. Available from: https://doi.org/10.3389/FCELL.2022.862637/BIBTEX [Accessed 18 July 2024].

Tobin, J.F., Laban, A. and Wirth, D.F., 1991. Homologous recombination in Leishmania enriettii. *Proceedings of the National Academy of Sciences of the United States of America* [Online], 88(3), pp.864–868. Available from: https://doi.org/10.1073/PNAS.88.3.864 [Accessed 8 August 2024].

Trager, W., 1969. Pteridine Requirement of the Hemoflagellate Leishmania tarentolae. *The Journal of Protozoology* [Online], 16(2), pp.372–375. Available from: https://doi.org/10.1111/J.1550-7408.1969.TB02284.X [Accessed 10 July 2024].

Ullu, E., Tschudi, C. and Chakraborty, T., 2004. RNA interference in protozoan parasites. *Cellular Microbiology* [Online], 6(6), pp.509–519. Available from: https://doi.org/10.1111/J.1462-5822.2004.00399.X [Accessed 24 May 2024].

Untergasser, A., Cutcutache, I., Koressaar, T., Ye, J., Faircloth, B.C., Remm, M. and Rozen, S.G., 2012. Primer3—new capabilities and interfaces. *Nucleic Acids Research* [Online], 40(15), p.e115. Available from: https://doi.org/10.1093/NAR/GKS596 [Accessed 16 April 2024].

Vasquez, J.J., Wedel, C., Cosentino, R.O. and Siegel, T.N., 2018. Exploiting CRISPR–Cas9 technology to investigate individual histone modifications. *Nucleic Acids Research* [Online], 46(18), p.e106. Available from: https://doi.org/10.1093/NAR/GKY517 [Accessed 13 June 2023].

Vergnes, B., Gazanion, E., Mariac, C., Du Manoir, M., Sollelis, L., Lopez-Rubio, J.J., Sterkers, Y. and Bañuls, A.L., 2019. A single amino acid substitution (H451Y) in Leishmania calcium-dependent kinase SCAMK confers high tolerance and resistance to antimony. *Journal of Antimicrobial Chemotherapy* [Online], 74(11), pp.3231–3239. Available from: https://doi.org/10.1093/JAC/DKZ334 [Accessed 13 June 2023].

Villa, H., Marcos, A.R.O., Reguera, R.M., Balaña-Fouce, R., García-Estrada, C., Pérez-Pertejo, Y., Tekwani, B.L., Myler, P.J., Stuart, K.D., Bjornsti, M.A. and Ordóñez, D., 2003. A novel active DNA topoisomerase I in Leishmania donovani. *Journal of Biological Chemistry* [Online], 278(6), pp.3521–3526. Available from: https://doi.org/10.1074/jbc.M203991200 [Accessed 13 June 2023].

Wall, R.J., Rico, E., Lukac, I., Zuccotto, F., Elg, S., Gilbert, I.H., Freund, Y., Alley, M.R.K., Field, M.C., Wyllie, S. and Horn, D., 2018. Clinical and veterinary trypanocidal benzoxaboroles target CPSF3. *Proceedings of the National Academy of Sciences of the United States of America* [Online], 115(38), pp.9616–9621. Available from: https://doi.org/10.1073/PNAS.1807915115/SUPPL_FILE/PNAS.1807915115.SAPP.PDF [Accessed 17 September 2024].

Xu, J., Wang, M., Gao, X., Hu, B., Du, Y., Zhou, J., Tian, X. and Huang, X., 2011. Separase Phosphosite Mutation Leads to Genome Instability and Primordial Germ Cell Depletion during Oogenesis. *PLOS ONE* [Online], 6(4), p.e18763. Available from: https://doi.org/10.1371/JOURNAL.PONE.0018763 [Accessed 19 September 2023].

Yagoubat, A., Corrales, R.M., Bastien, P., Lévêque, M.F. and Sterkers, Y., 2020. Gene Editing in Trypanosomatids: Tips and Tricks in the CRISPR-Cas9 Era. *Trends in Parasitology* [Online], 36(9), pp.745–760. Available from: https://doi.org/10.1016/J.PT.2020.06.005 [Accessed 13 June 2023].

Yang, X., Lau, K.Y., Sevim, V. and Tang, C., 2013. Design Principles of the Yeast G1/S Switch. *PLOS Biology* [Online], 11(10), p.e1001673. Available from: https://doi.org/10.1371/JOURNAL.PBIO.1001673 [Accessed 20 September 2023].

Zhang, W.-W., Lypaczewski, P. and Matlashewski, G., 2017. Optimized CRISPR-Cas9 Genome Editing for Leishmania and Its Use To Target a Multigene Family, Induce Chromosomal Translocation, and Study DNA Break Repair Mechanisms . *mSphere* [Online], 2(1). Available from: https://doi.org/10.1128/MSPHERE.00340-16/ASSET/CFBEB13D-5E70-44F4-B94B-255C4E0D799C/ASSETS/GRAPHIC/SPH0011722180007.JPEG [Accessed 7 February 2023].

Zhang, W.W. and Matlashewski, G., 2015. CRISPR-Cas9-mediated genome editing in Leishmania donovani. *mBio* [Online], 6(4), pp.861–876. Available from: https://doi.org/10.1128/MBIO.00861-15/SUPPL_FILE/MBO004152405SF10.DOCX [Accessed 17 October 2022].

Zhang, W.-W. and Matlashewski, G., 2019. Single-Strand Annealing Plays a Major Role in Double-Strand DNA Break Repair following CRISPR-Cas9 Cleavage in Leishmania. *mSphere* [Online], 4(4). Available from: https://doi.org/10.1128/MSPHERE.00408-19 [Accessed 10 May 2024].

7.2 Appendices

7.2.1 LIST OF ABBREVIATIONS

Abbreviation	**Meaning**
ANOVA	Analysis of variance
ATP	Adenosine Triphosphate
BLAST	Basic Local Alignment Search Tool
bp	Base pair(s)
CCAN	Constitutive centromere associated network
CDS	Coding sequence
CENP	CENtromere Protein
CL	Cutaneous Leishmaniasis
CLK	Cdc2-like kinase
CRISPR	Clustered regularly interspaced short palindromic repeats
DAPI	4’6-diamidino-2-phenylindole
DiCre	Dimerizable Cre recombinase
DNA	Deoxyribonucleic acid
DSB	Double-strand break
dsDNA	Double-stranded DNA
EDTA	Ethylenediaminetetraacetic acid
FASTA	FAST-All
FBS	Foetal bovine serum
HA	Haemagglutinin
HDR	Homology-directed repair
HDV	Hepatitis delta virus
kb	Kilobase(s)
KKIP	Kinetoplastid kinetochore interacting protein
KKT	Kinetoplastid kinetochore protein
MSA	Multiple sequence alignment

NHEJ	Non-homologous end joining
nt	Nucleotide(s)
NumPy	Numerical Python
PAM	Protospacer Adjacent Motif
Pandas	Panel data
PBS	Phosphate buffered saline
PCR	Polymerase chain reaction
PFA	Paraformaldehyde
PSA	Pairwise-sequence alignment
RNA	Ribonucleic acid
RNAi	RNA-interference
SAC	Spindle assembly checkpoint
sgRNA	Single-guide RNA
SNP	Single nucleotide polymorphism
ssDNA	Single-stranded DNA
T7 RNAP	T7 RNA polymerase
UTR	Untranslated region
WT	Wild-type

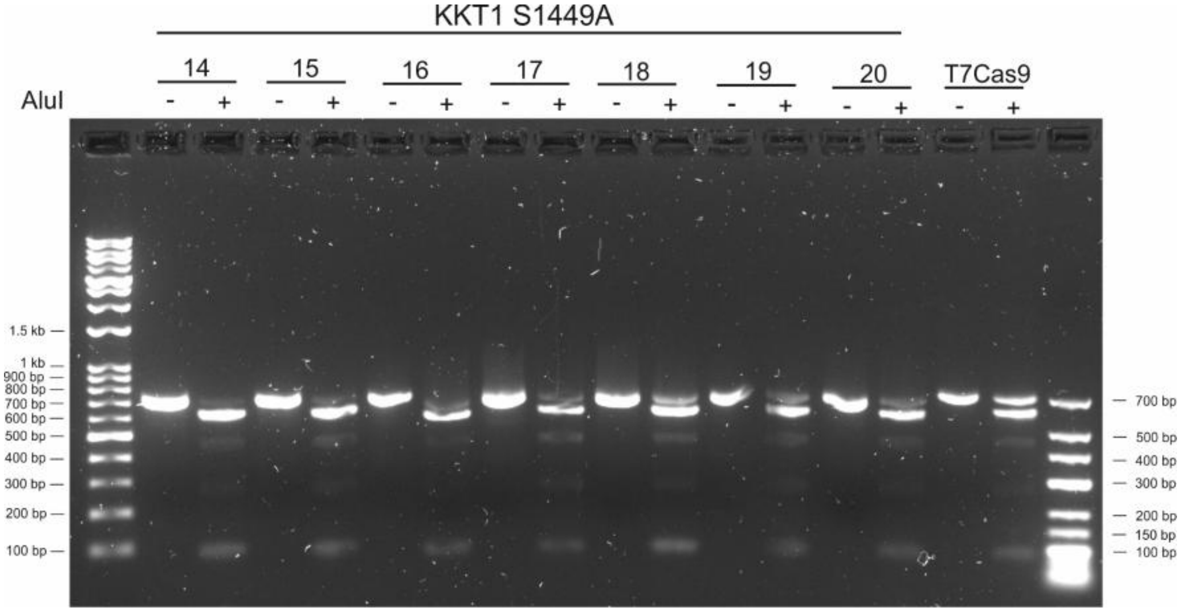
7.2.2 SUPPLEMENTARY DATA

7.2.2.1 Single-stranded Transfection Restriction Digest Screens

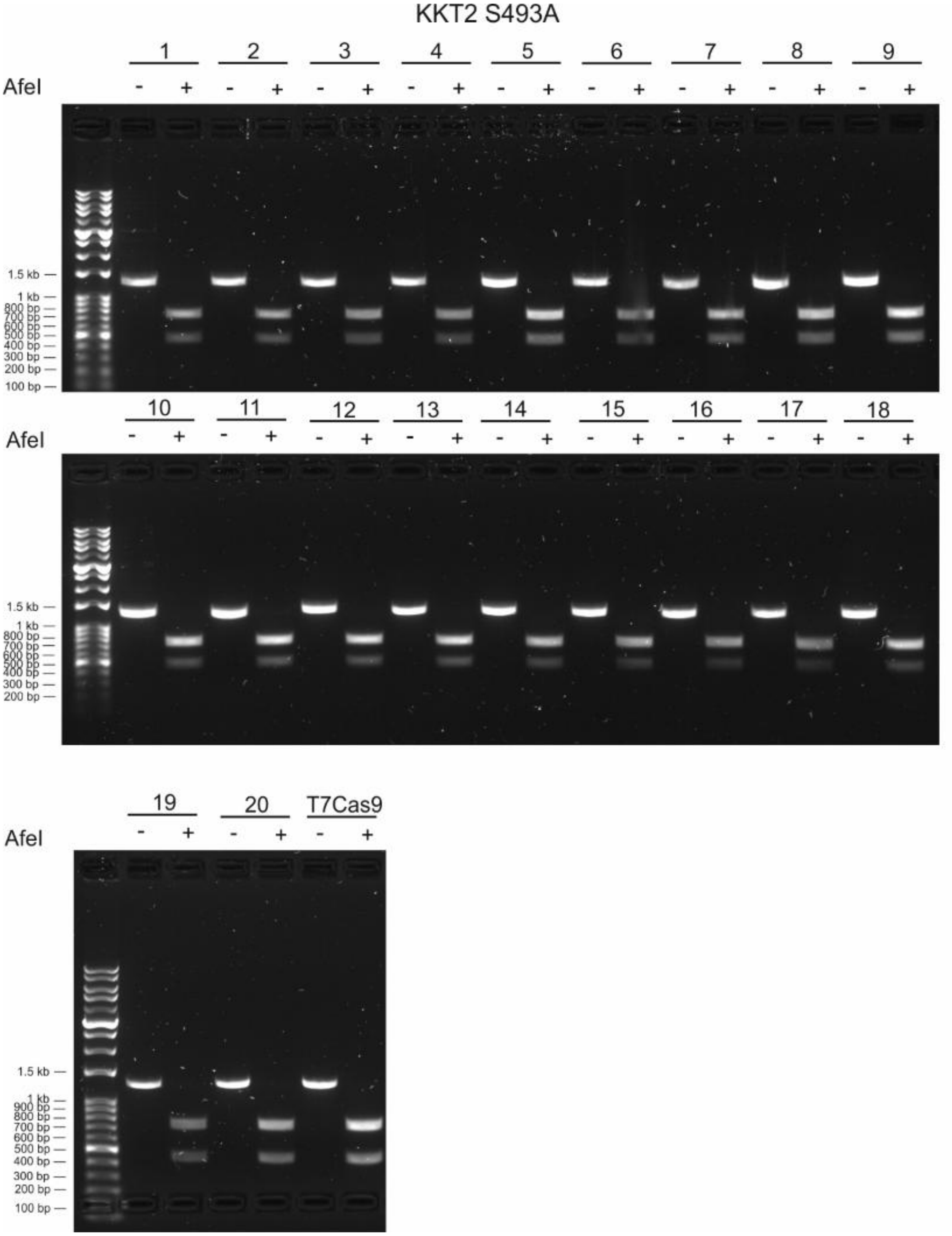
Restriction digests for screening clones to detect genotype following transfections with ssDNA repair templates which confer either phosphodeficient mutations or synonymous mutation equivalent designs. A PCR was designed which encompassed the whole repair region and some of the genomic DNA either side of the repair region. PCRs were purified and the same quantity of PCR product for each clone was digested with the restriction enzyme indicated on the respective agarose gel. The restriction enzyme used corresponded to a restriction site which was engineered into or removed from the repair sequence. Expected digest patterns for each mutation can be found in Appendix 7.2.6. T7Cas9 is the parental cell line. Numbers or CL followed by a number indicate clone number. “-" indicates undigested sample. “+” indicates digested sample.

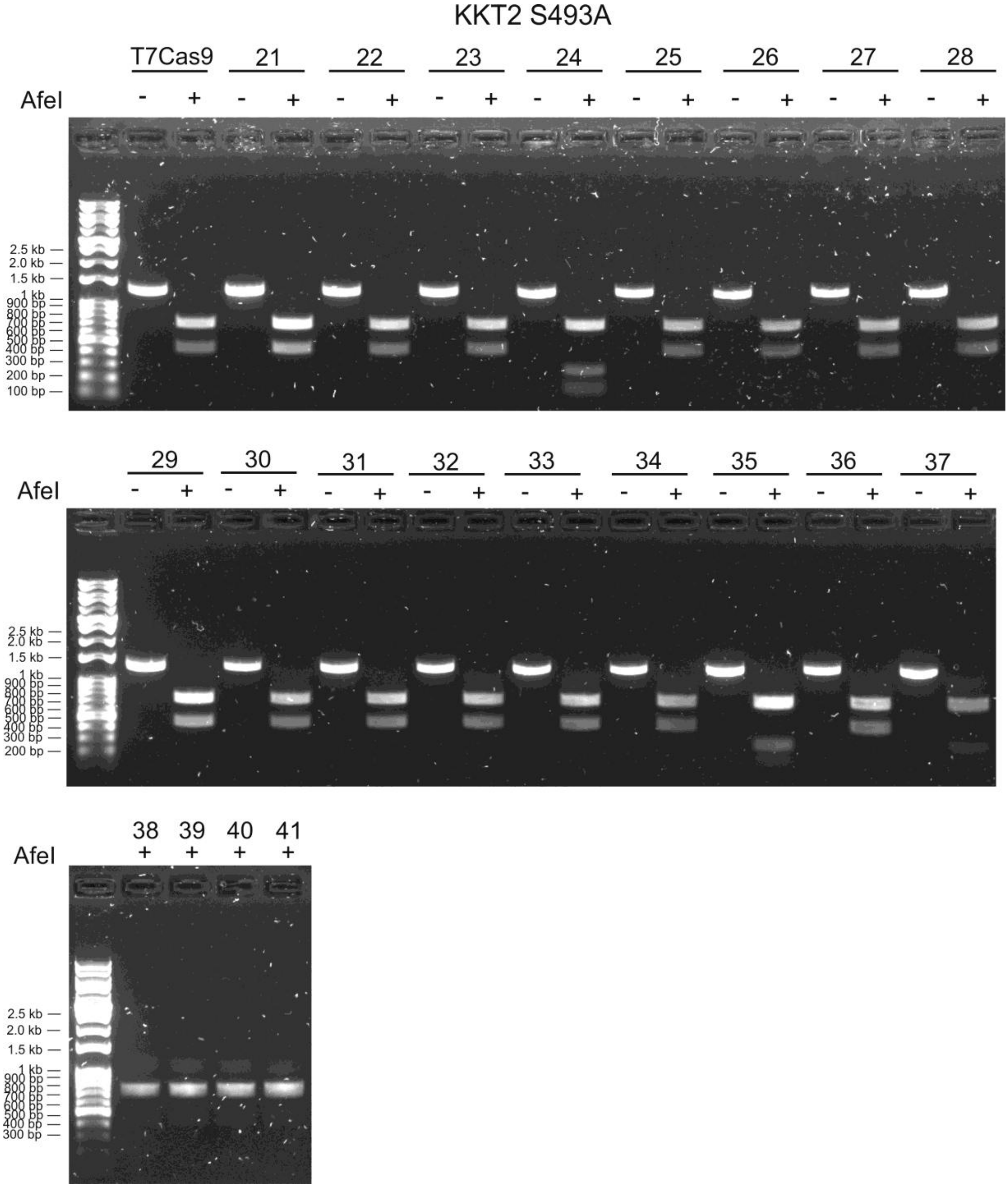
7.2.2.1.1 KKT1 S1449A

For KKT1 S1449A only clones 14-20 were possible to screen.

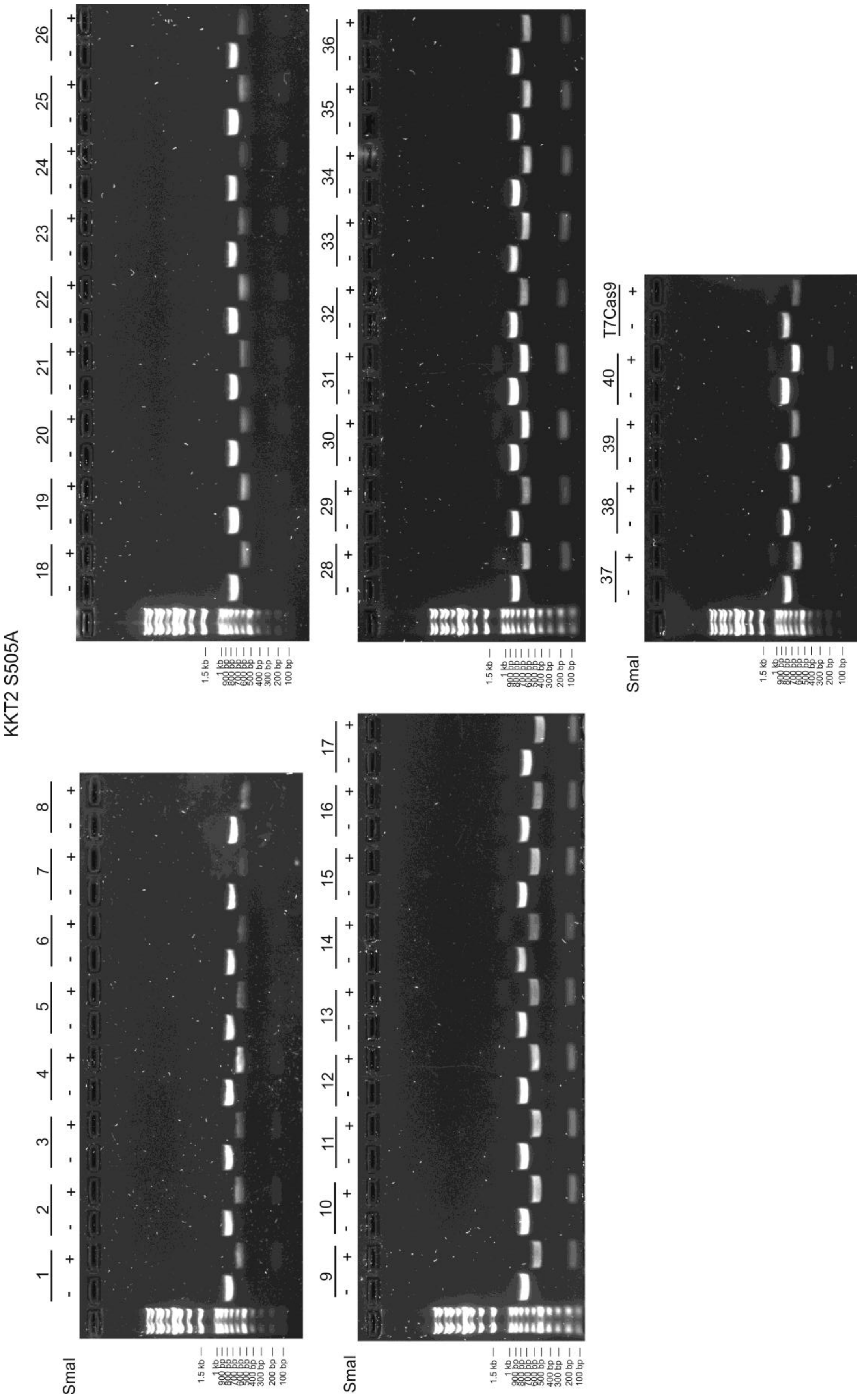


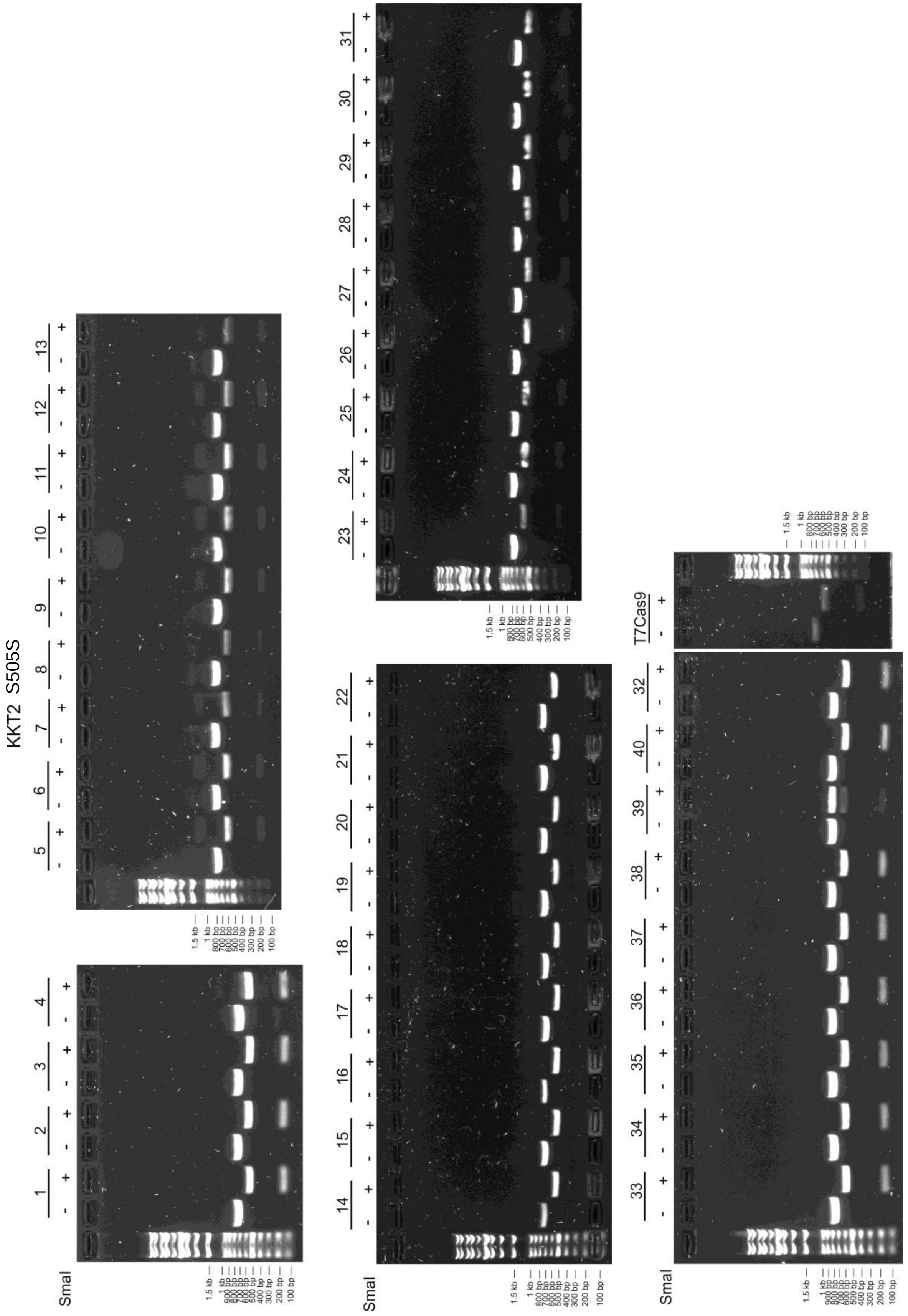
7.2.2.1.2 KKT2 S493




KKT2 S493A
T7Cas9
21
22
23
24
25
26
27
28
AfeI
- + - + - + - + - + - + - + - + - +
2.5 kb
2.0 kb
1.5 kb
1 kb
900 bp
800 bp
700 bp
600 bp
500 bp
400 bp
300 bp
200 bp
100 bp
29
30
31
32
33
34
35
36
37
AfeI
- + - + - + - + - + - + - + - + - +
2.5 kb
2.0 kb
1.5 kb
1 kb
900 bp
800 bp
700 bp
600 bp
500 bp
400 bp
300 bp
200 bp
38
39
40
41
AfeI
+ + + +
2.5 kb
2.0 kb
1.5 kb
1 kb
900 bp
800 bp
700 bp
600 bp
500 bp
400 bp
300 bp

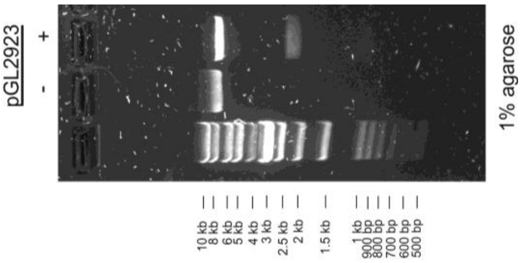
7.2.2.1.3 KKT2 S505

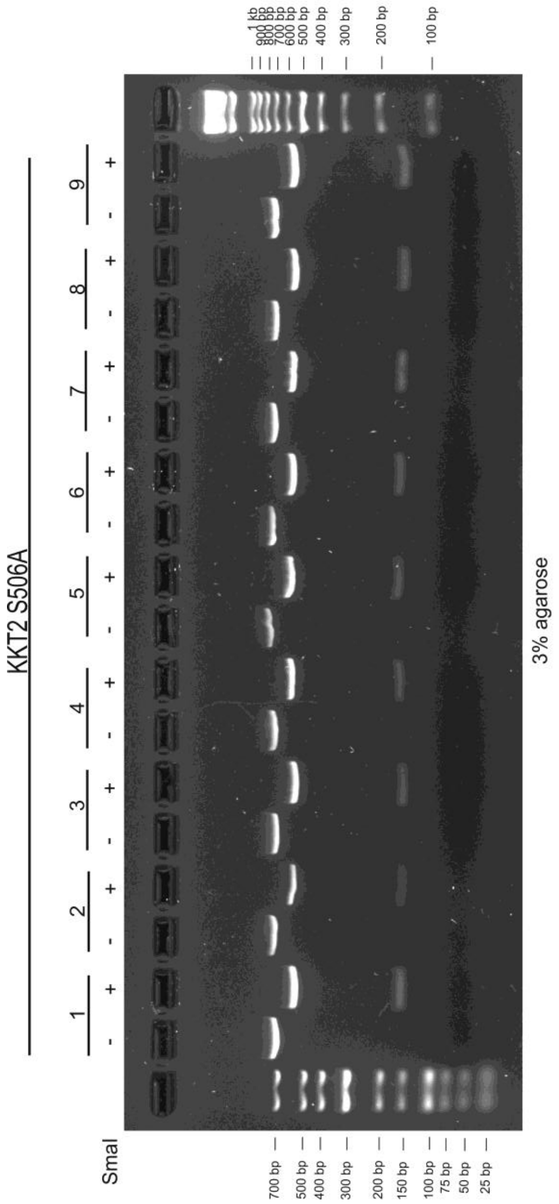


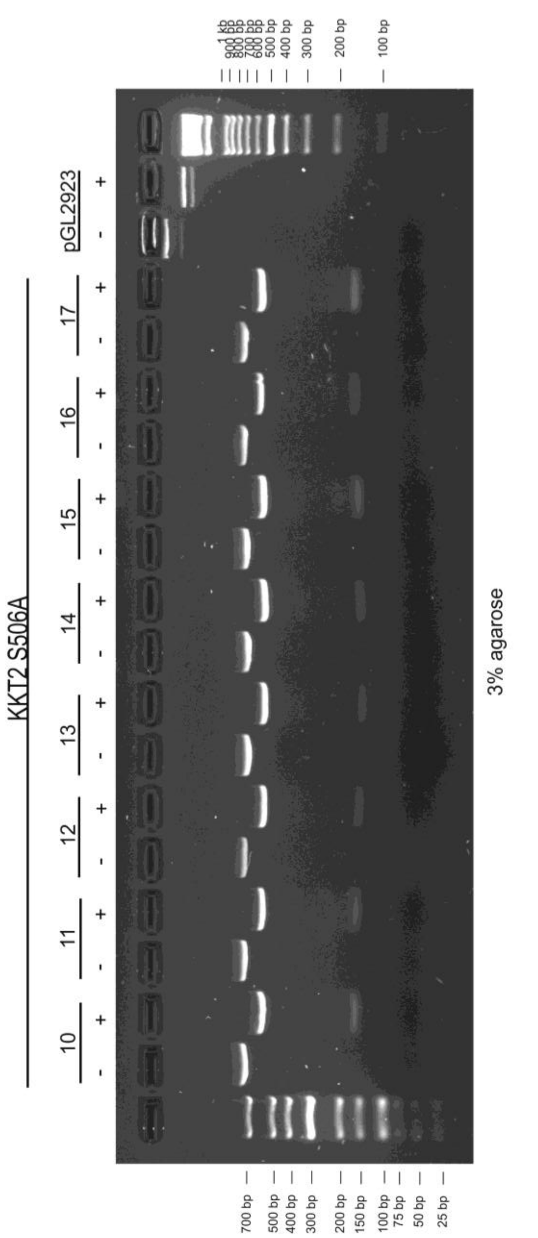

KKT2 S505S
SmaI
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
T7Cas9
- +
1.5 kb
1 kb
900 bp
800 bp
700 bp
600 bp
500 bp
400 bp
300 bp
200 bp
100 bp

7.2.2.1.4 KKT2 S506

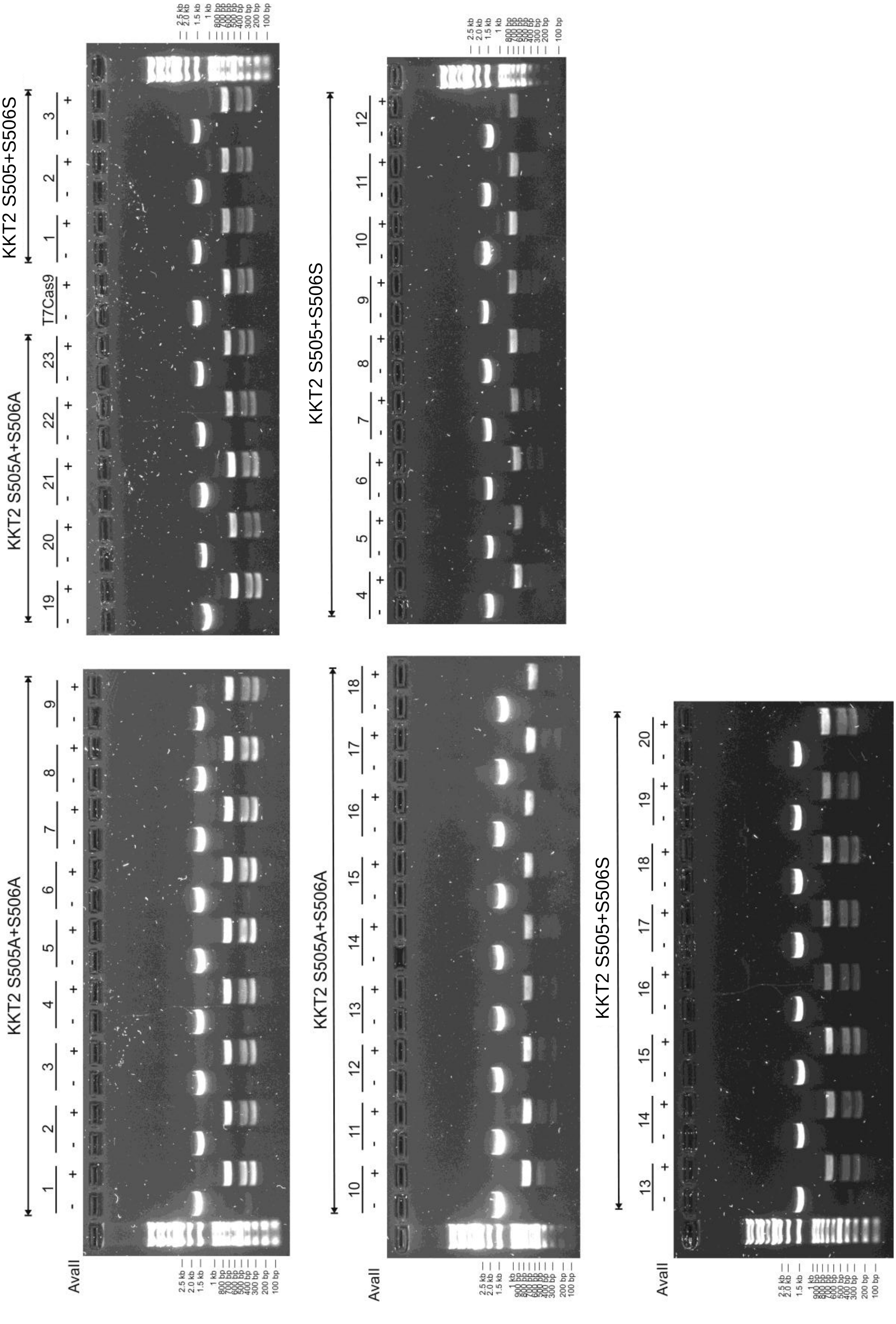
pGL2923 was a plasmid digested as a positive control to confirm enzyme activity.



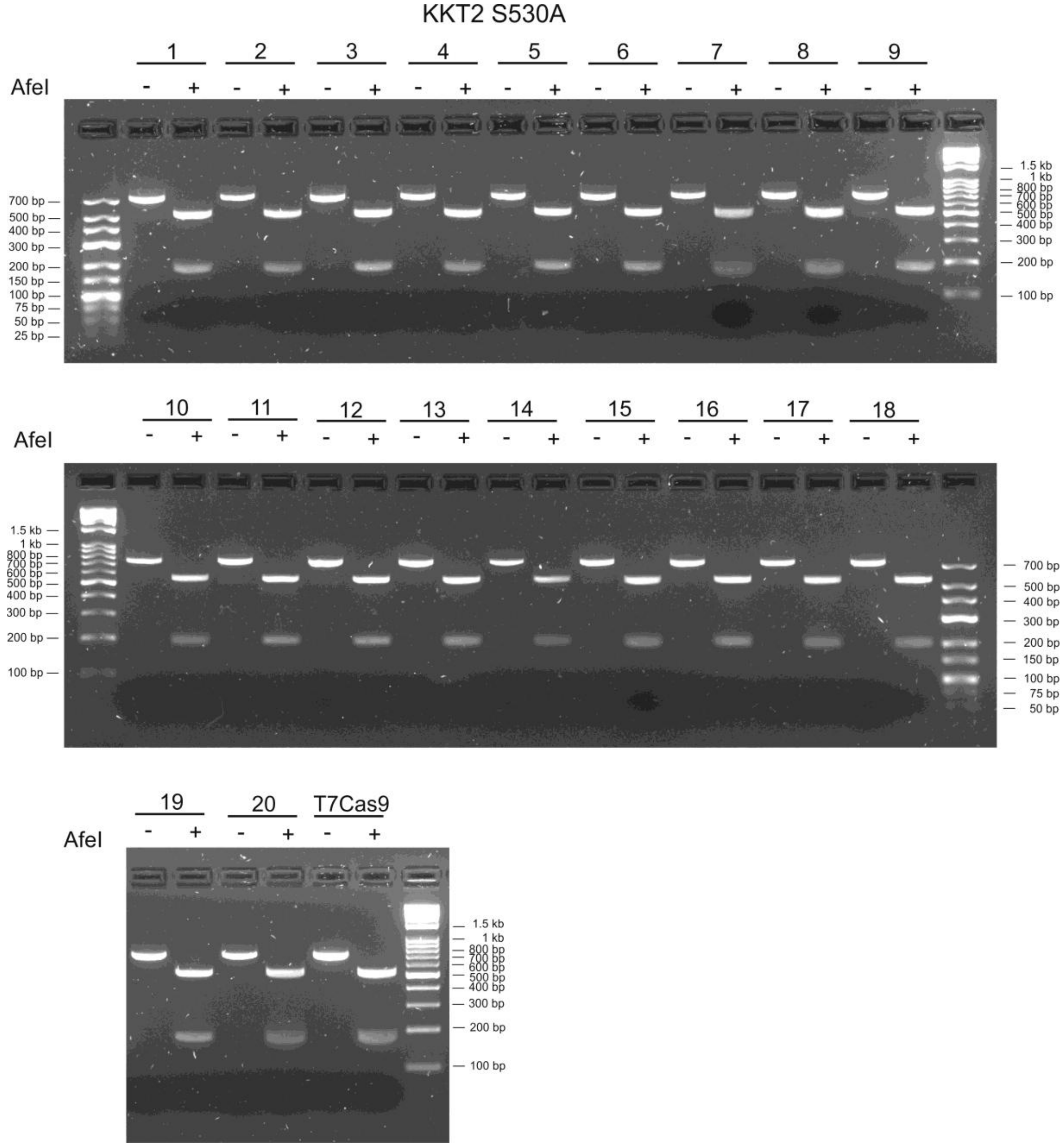




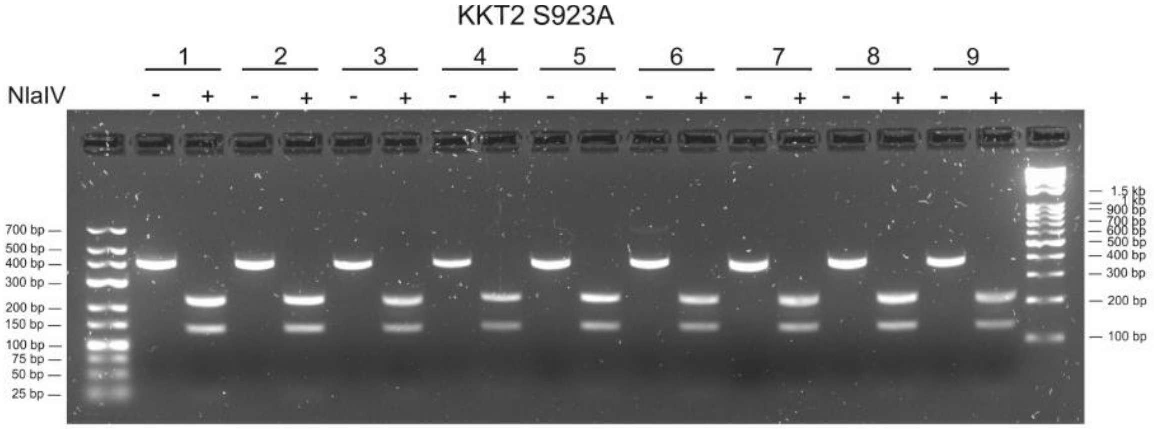
7.2.2.1.5 KKT2 S505 + S506

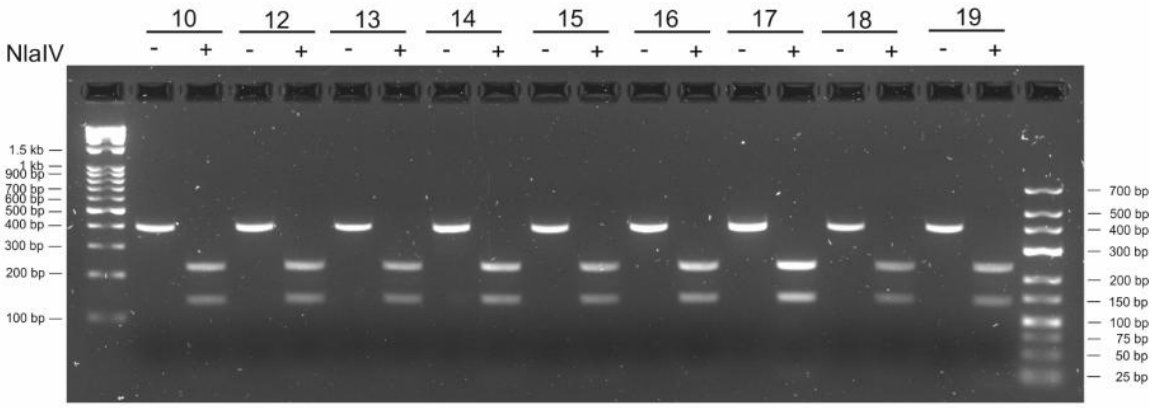


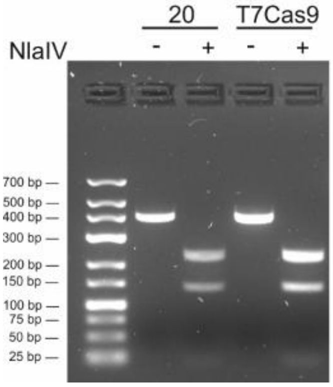
7.2.2.1.6 KKT2 S530

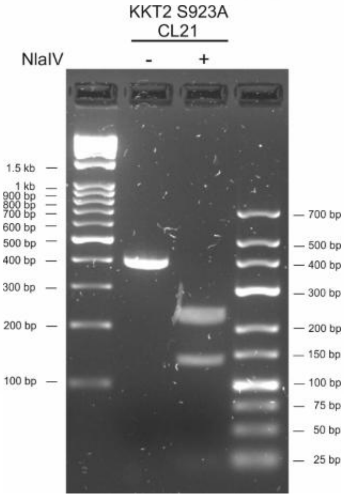


7.2.2.1.7 KKT2 S923



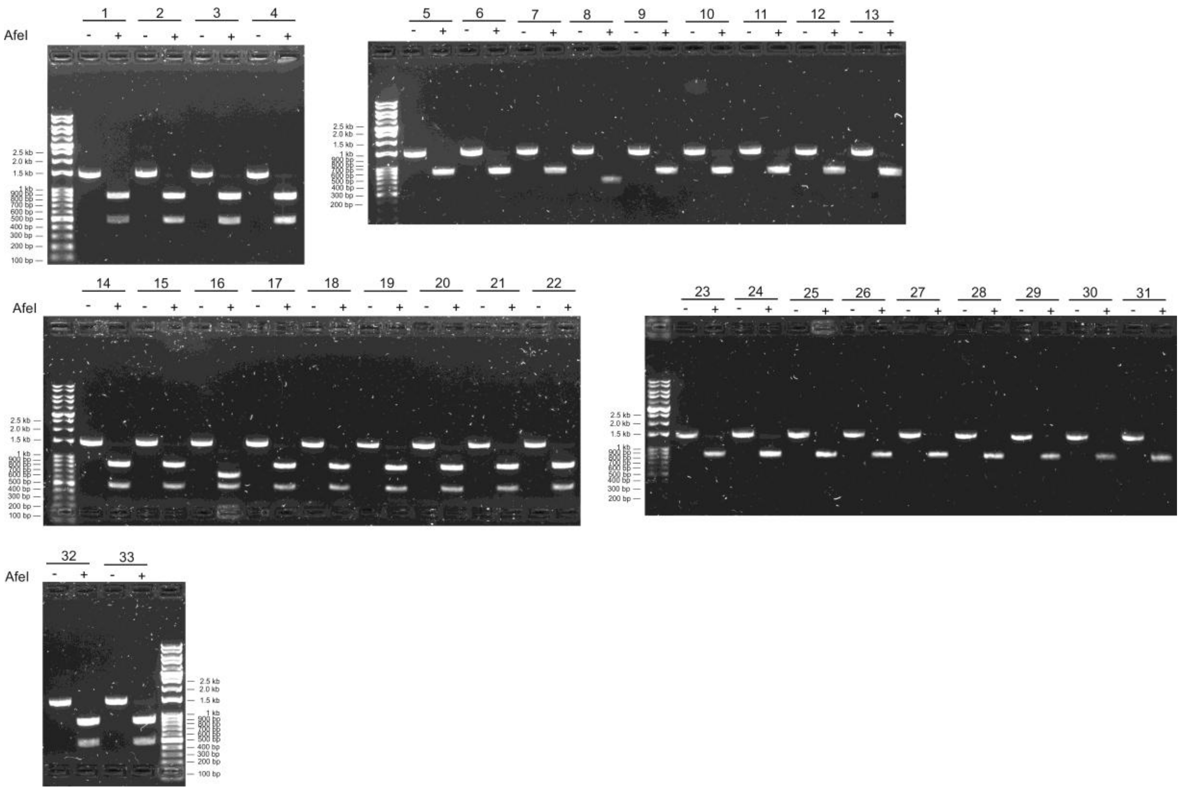




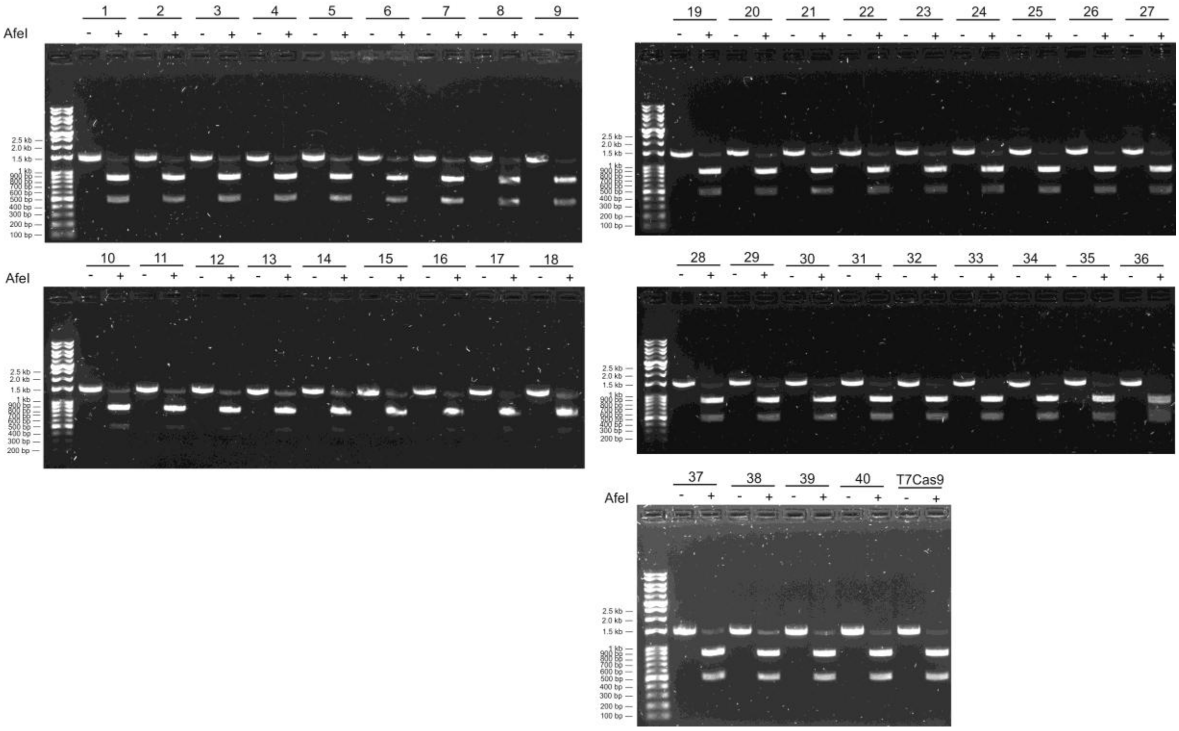


7.2.2.1.8 KKT4 S300

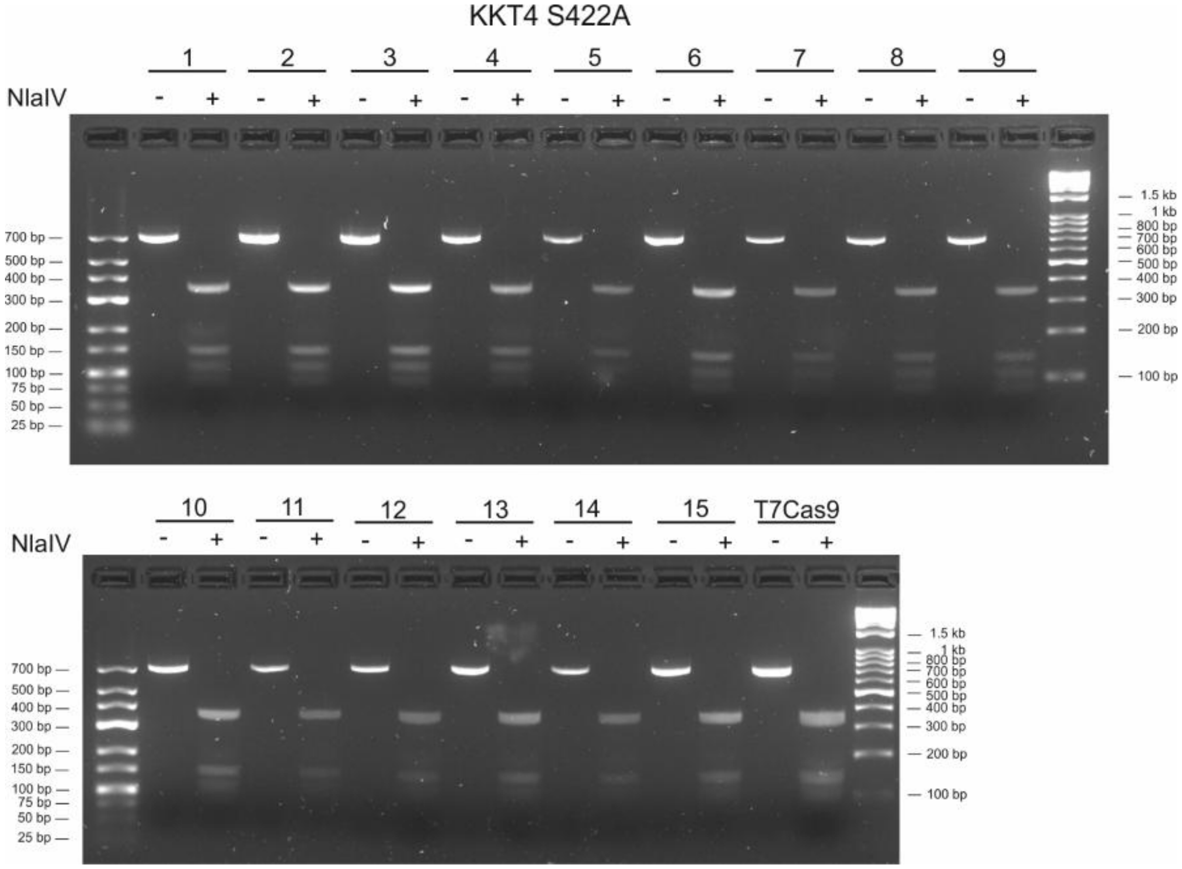
KKT4 S300A



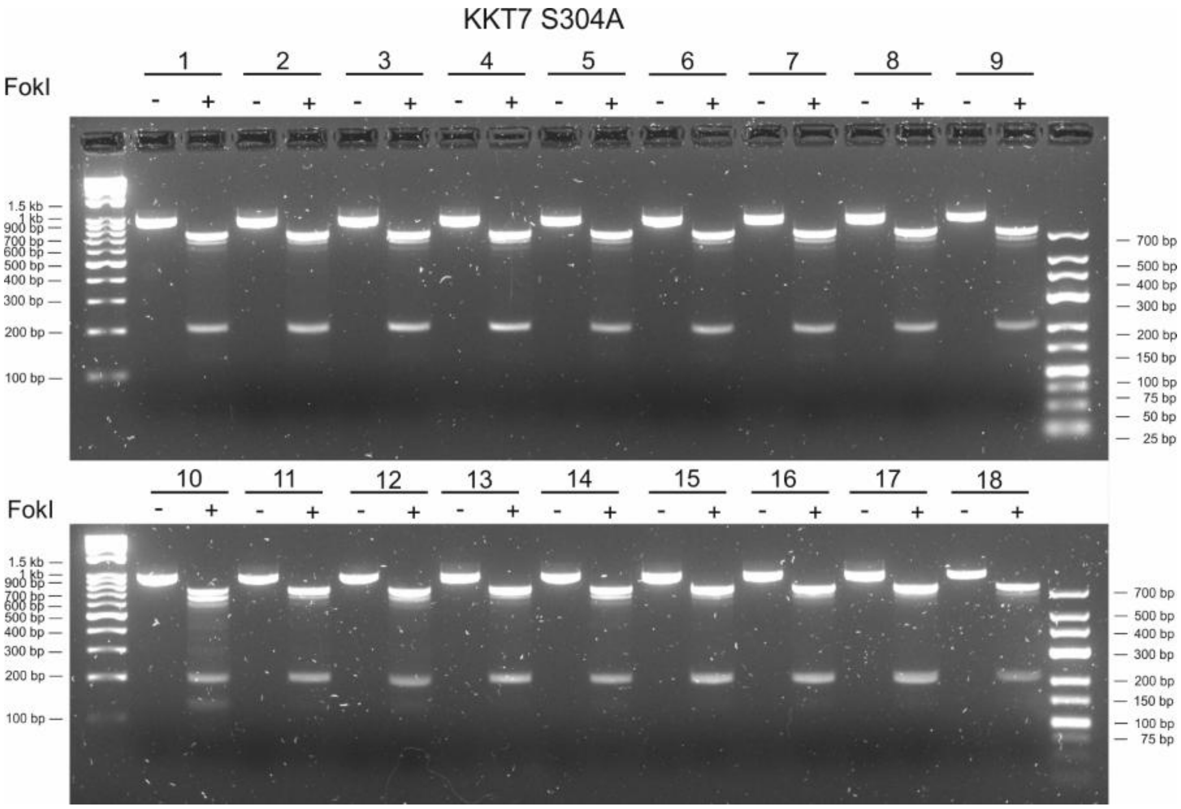
KKT4 S300S

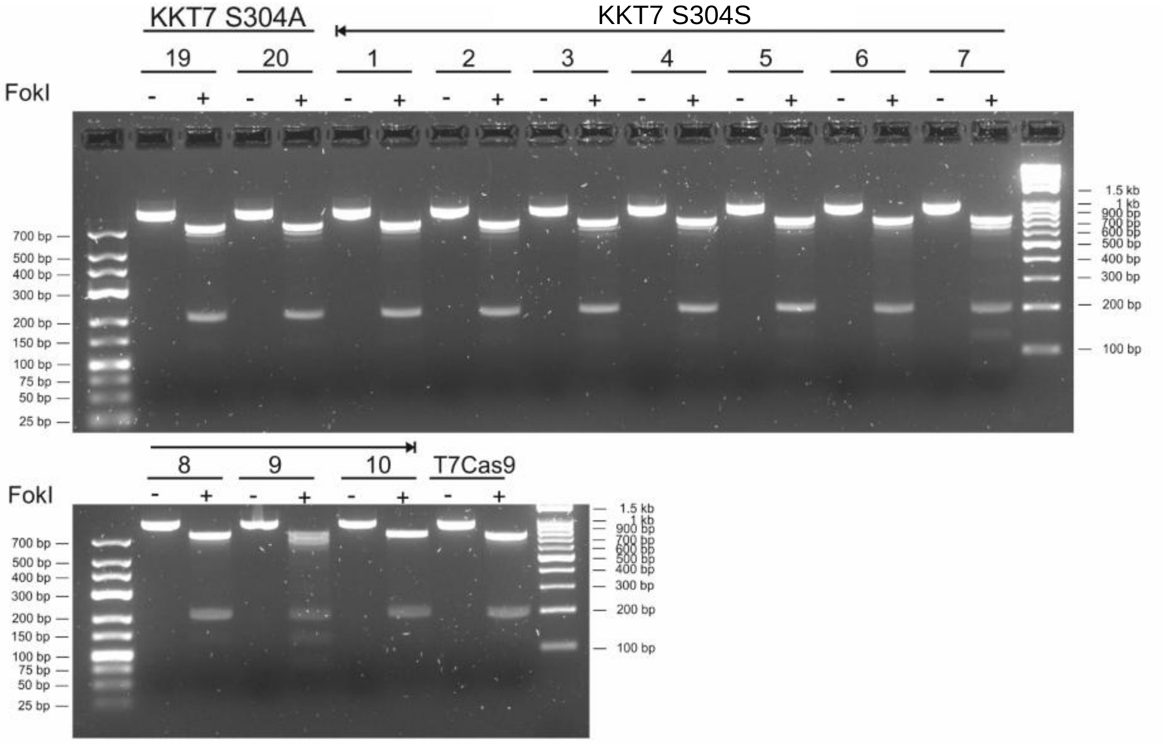


7.2.2.1.9 KKT4 S422



7.2.2.1.10 KKT7 S304

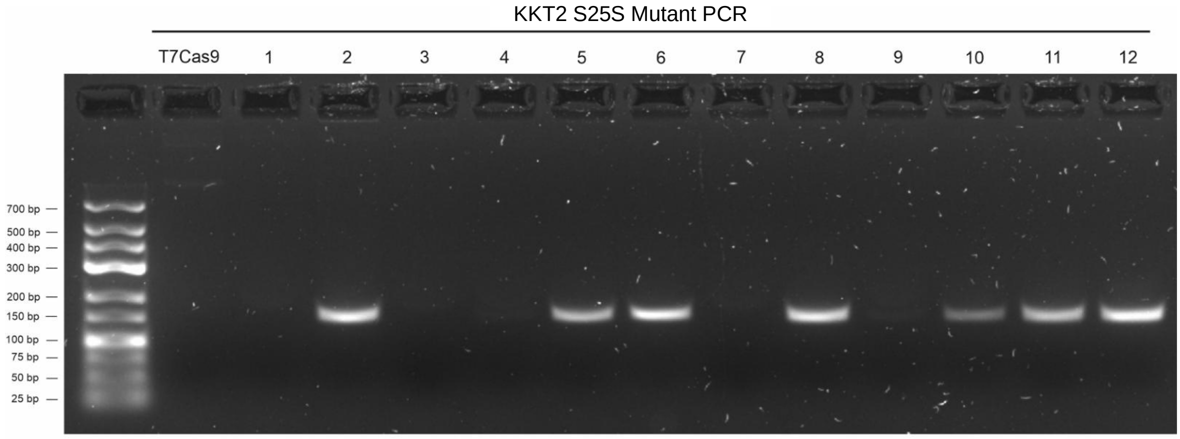


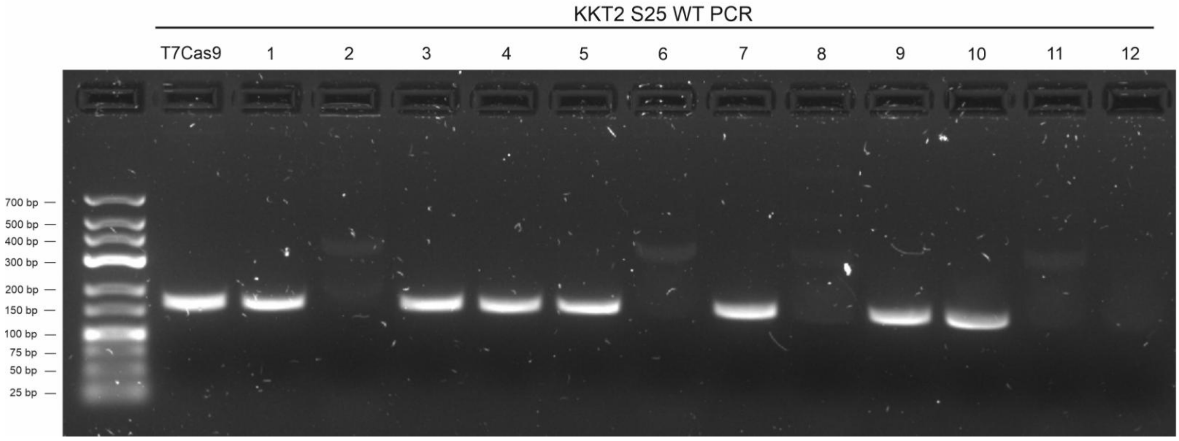


7.2.2.2 KKT2 Synonymous Only Mutant Double-stranded Transfection PCR Screens

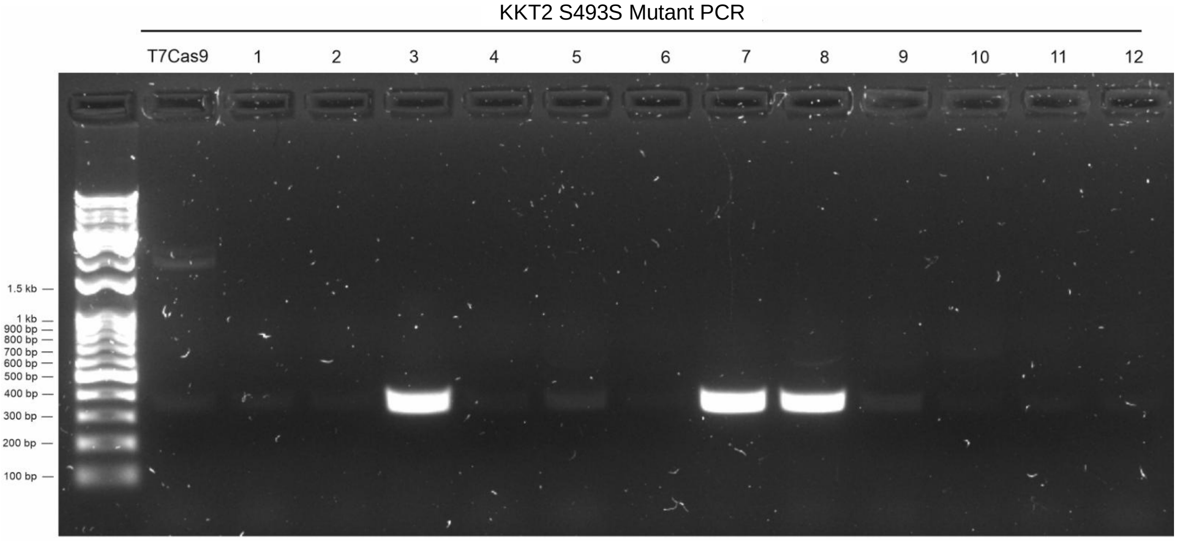
PCR screen for detecting genotype following transfections with dsDNA repair templates which confer synonymous mutations in KKT2. Expected PCR product sizes can be found in Appendix 7.2.5.6.2. Input DNA quantity was not standardised between clones but was consistent between each PCR on the same clone. T7Cas9 is the parental cell line. Numbers or CL followed by a number indicate clone number.

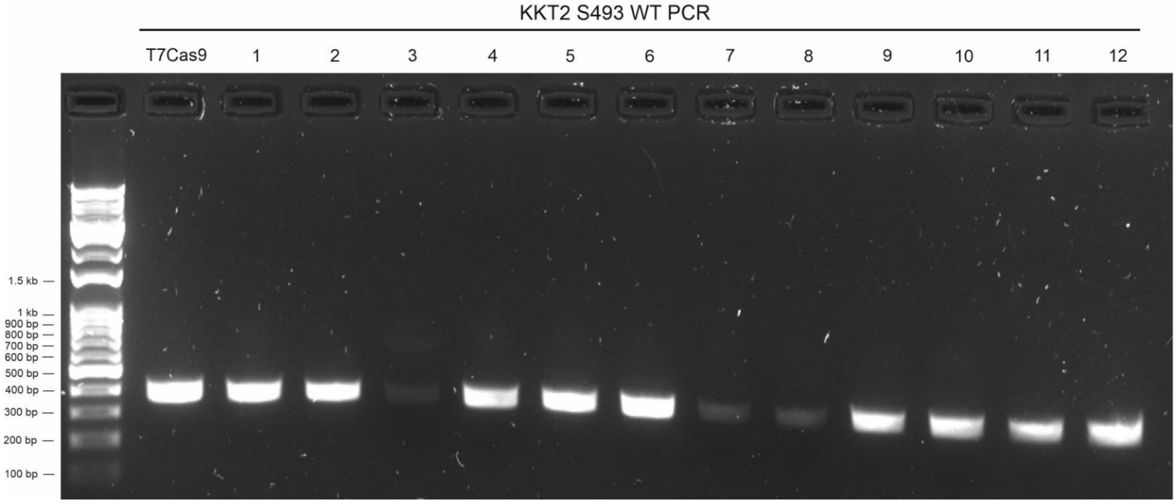
7.2.2.2.1 KKT2 S25S



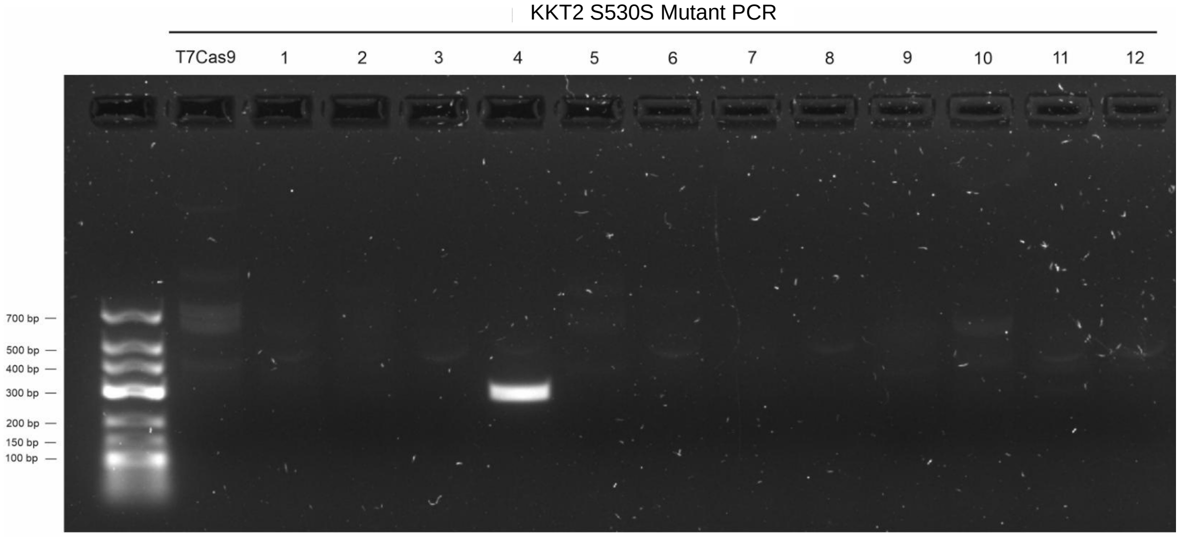


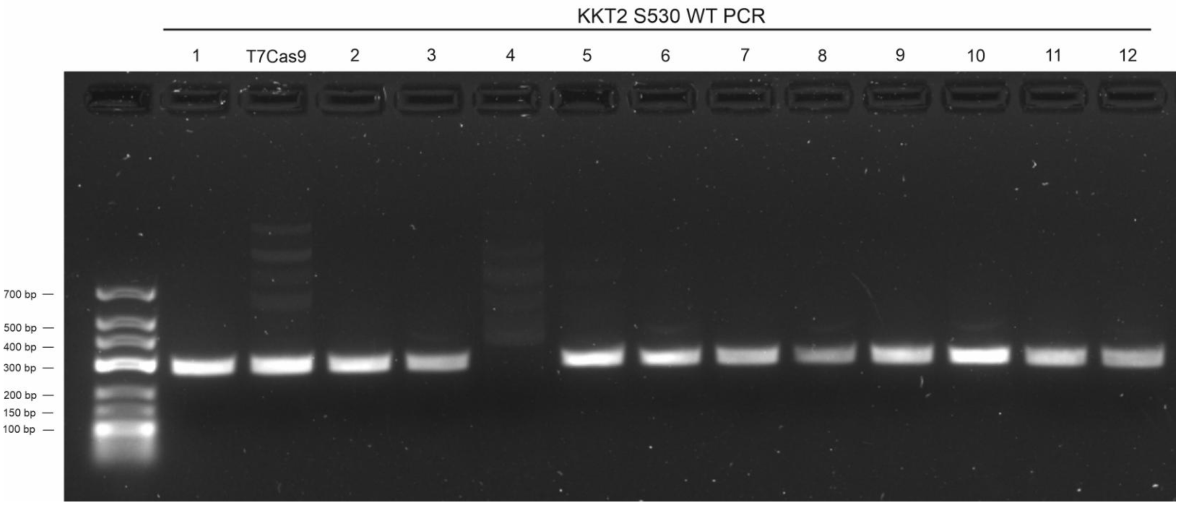
7.2.2.2.2 KKT2 S493S



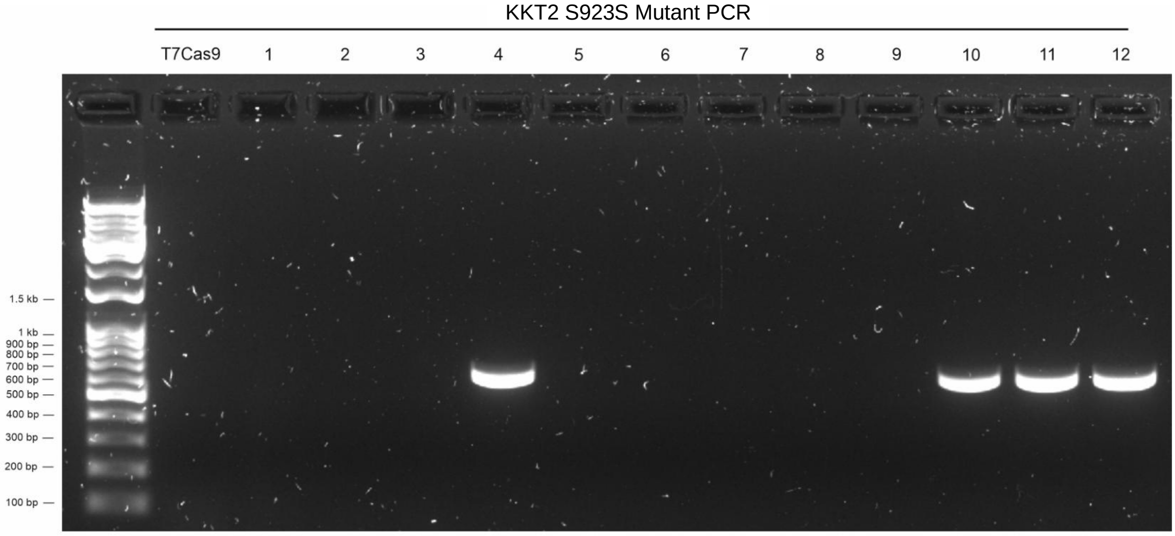


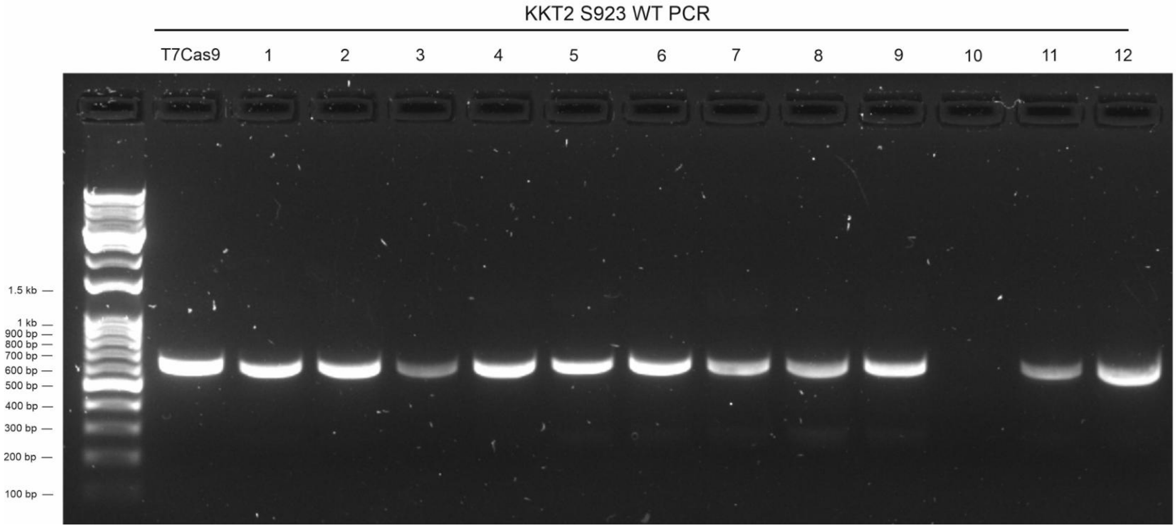
7.2.2.2.3 KKT2 S530S





7.2.2.2.4 KKT2 S923S

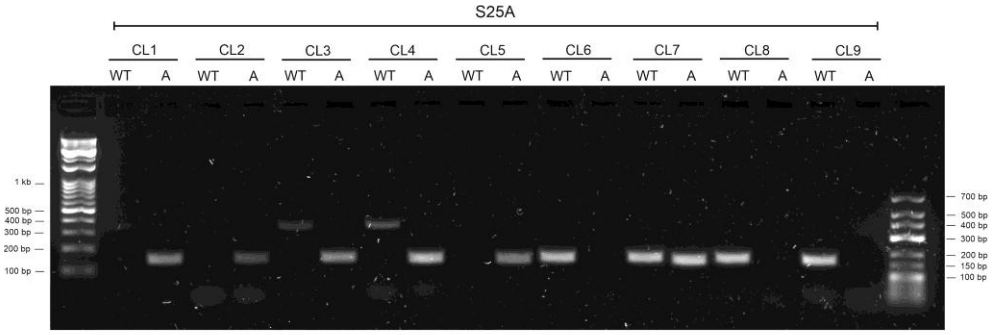


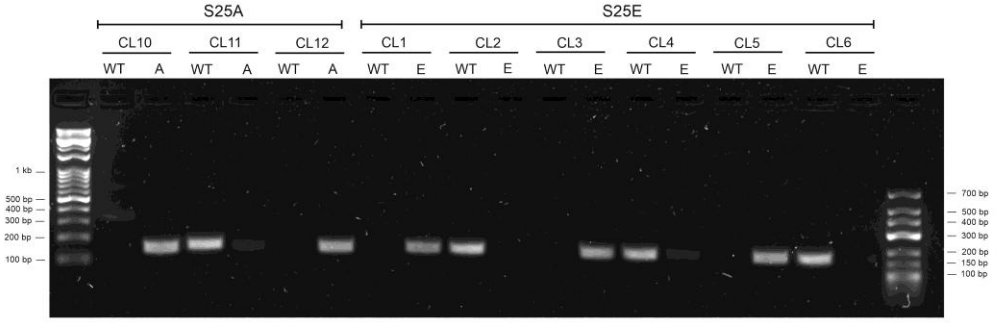


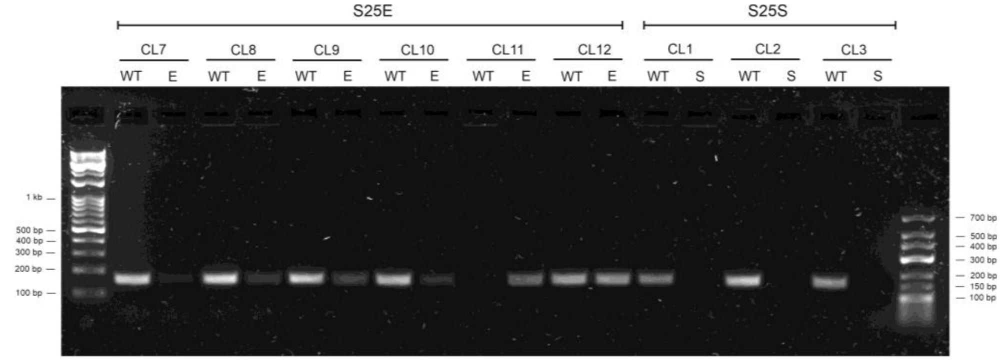
7.2.2.3 Double-stranded Transfection PCR Screens

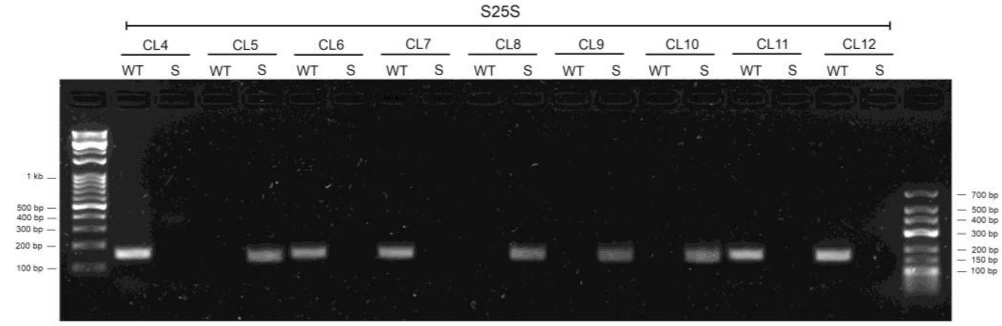
PCR screen for detecting genotype of clones following transfections with dsDNA repair templates which confer either phosphodeficient, phosphomimetic or synonymous mutation equivalent designs. Expected PCR product sizes can be found in Appendix 7.2.5.6.3. Input DNA quantity was not standardised between clones but was consistent between each PCR on the same clone. T7Cas9 is the parental cell line. Numbers or CL followed by a number indicate clone number. WT – WT PCR conditions. A – Alanine mutant specific PCR conditions. E – Glutamic acid mutant specific PCR conditions. S – Synonymous mutant specific PCR conditions. M – mutant PCR conditions (primer recognises a region of shared recoded sequence between alanine/glutamic acid/synonymous mutant repair templates).

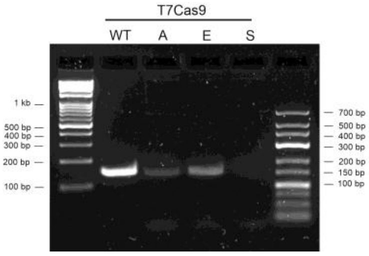
7.2.2.3.1 KKT2 S25



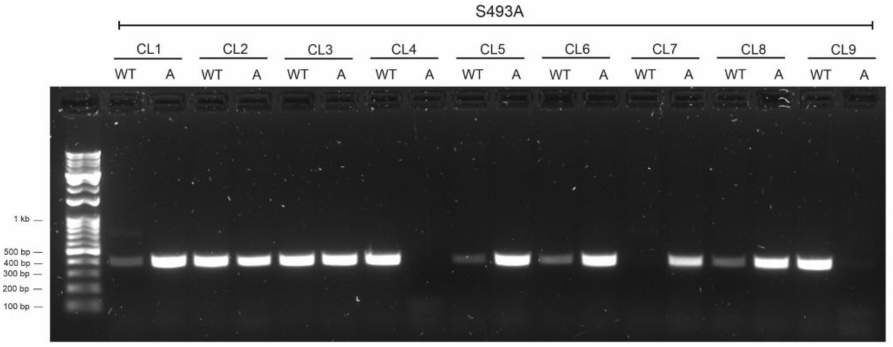


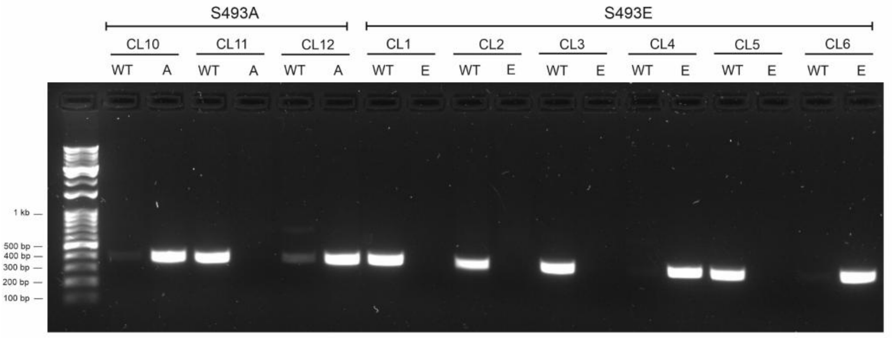


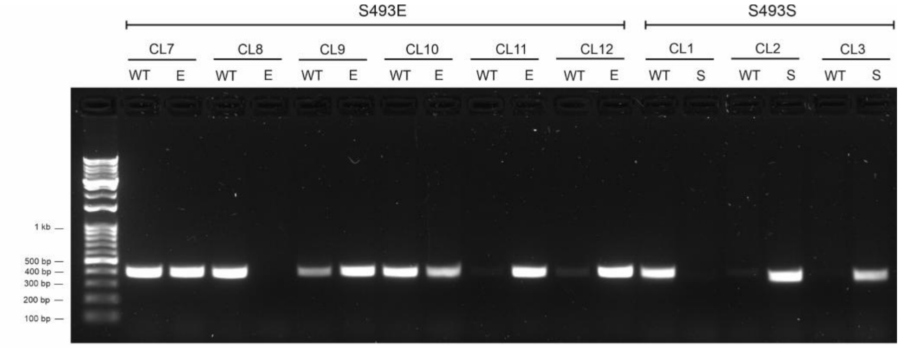


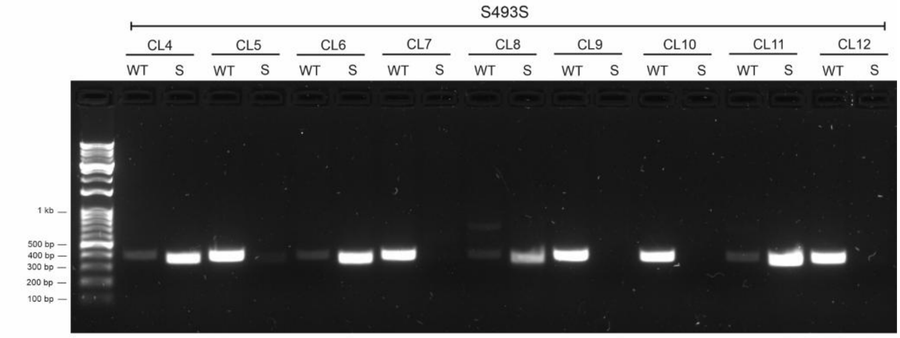


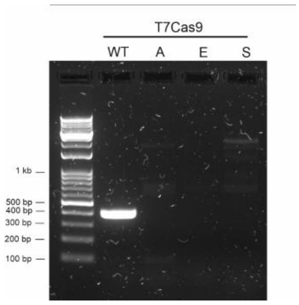
7.2.2.3.2 KKT2 S493


S493A
CL1 CL2 CL3 CL4 CL5 CL6 CL7 CL8 CL9
WT A WT A WT A WT A WT A WT A WT A WT A WT A
1 kb
500 bp
400 bp
300 bp
200 bp
100 bp

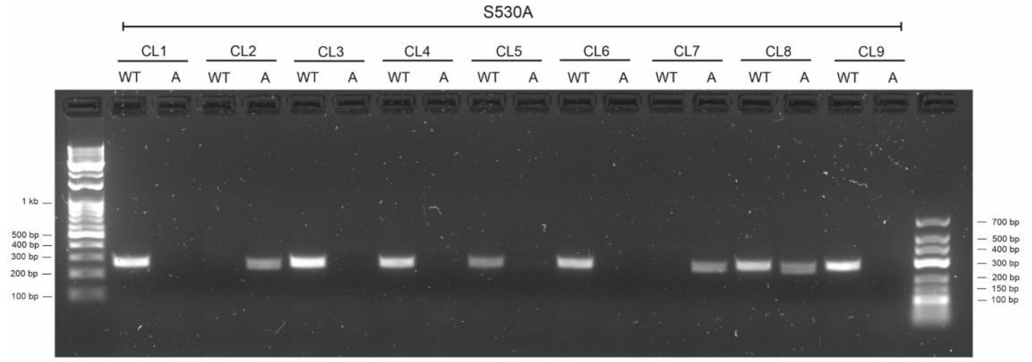

S493A
S493E
CL10 CL11 CL12 CL1 CL2 CL3 CL4 CL5 CL6
WT A WT A WT A WT E WT E WT E WT E WT E WT E
1 kb
500 bp
400 bp
300 bp
200 bp
100 bp

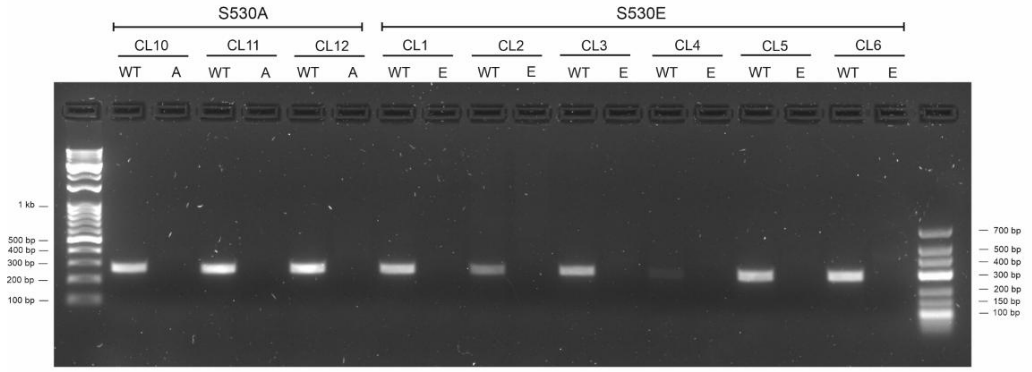

S493E
S493S
CL7 CL8 CL9 CL10 CL11 CL12 CL1 CL2 CL3
WT E WT E WT E WT E WT E WT E WT S WT S WT S
1 kb
500 bp
400 bp
300 bp
200 bp
100 bp

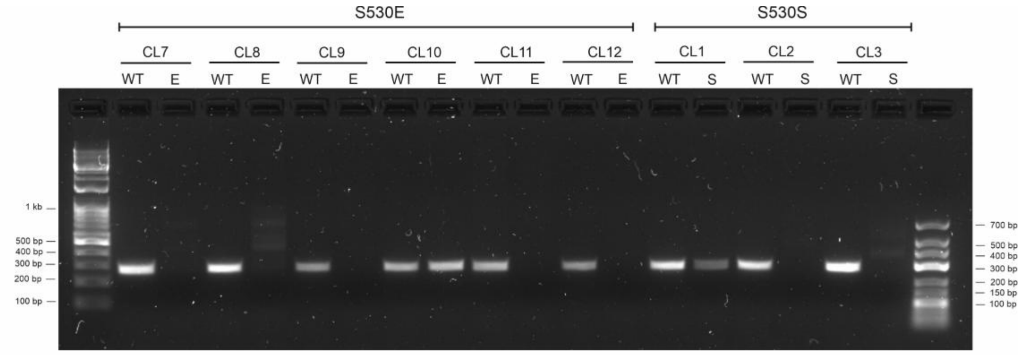

S493S
CL4 CL5 CL6 CL7 CL8 CL9 CL10 CL11 CL12
WT S WT S WT S WT S WT S WT S WT S WT S WT S
1 kb
500 bp
400 bp
300 bp
200 bp
100 bp

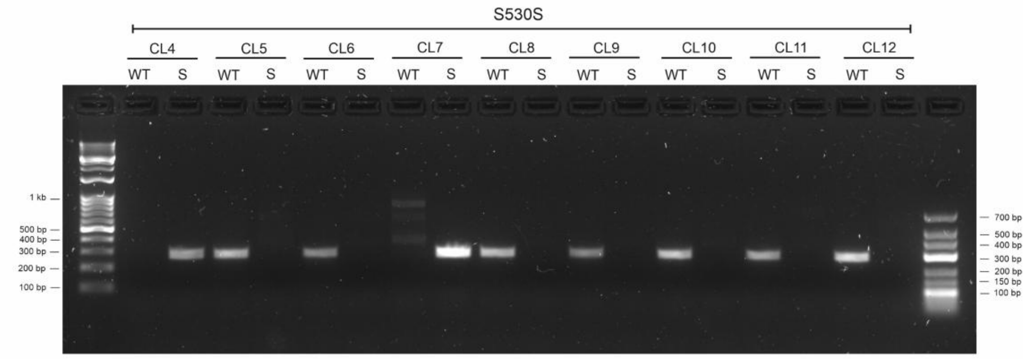


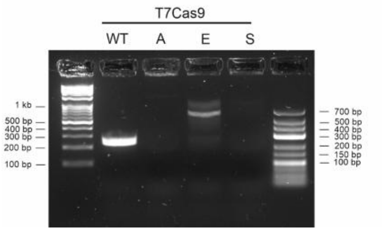
7.2.2.3.3 KKT2 S530

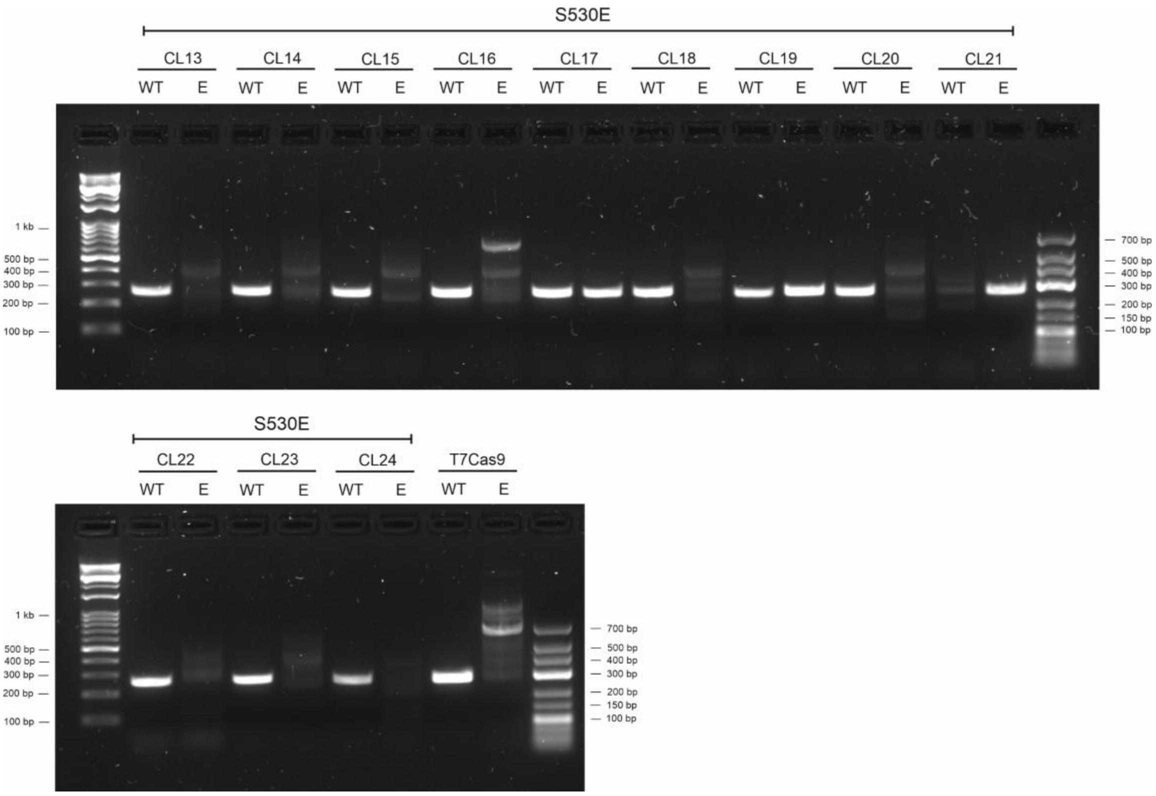




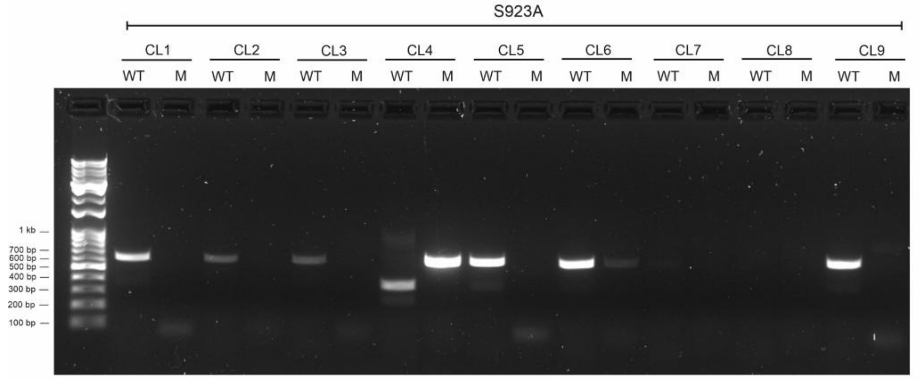


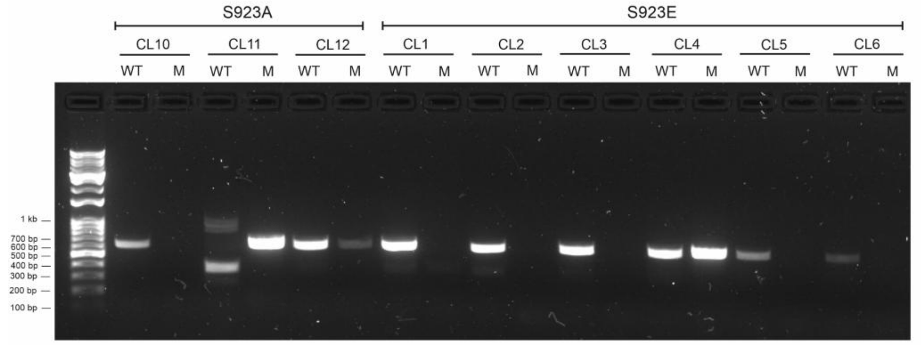

S530S
CL4 CL5 CL6 CL7 CL8 CL9 CL10 CL11 CL12
WT S WT S WT S WT S WT S WT S WT S WT S WT S
1 kb
500 bp
400 bp
300 bp
200 bp
100 bp
700 bp
500 bp
400 bp
300 bp
200 bp
150 bp
100 bp

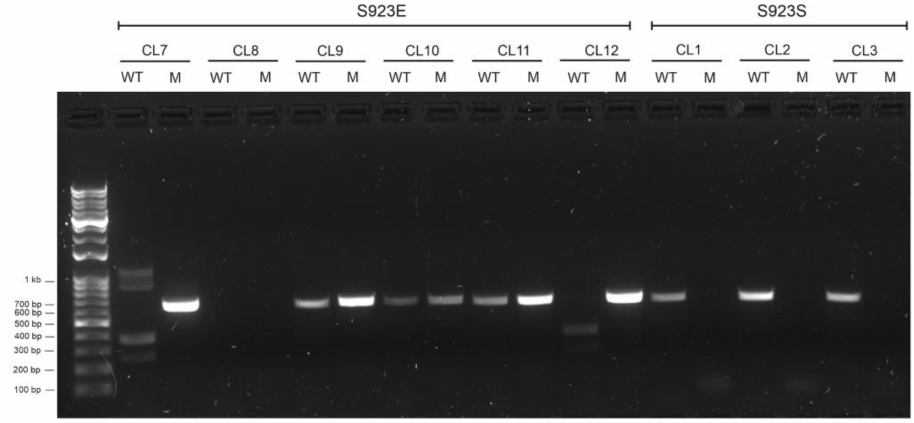


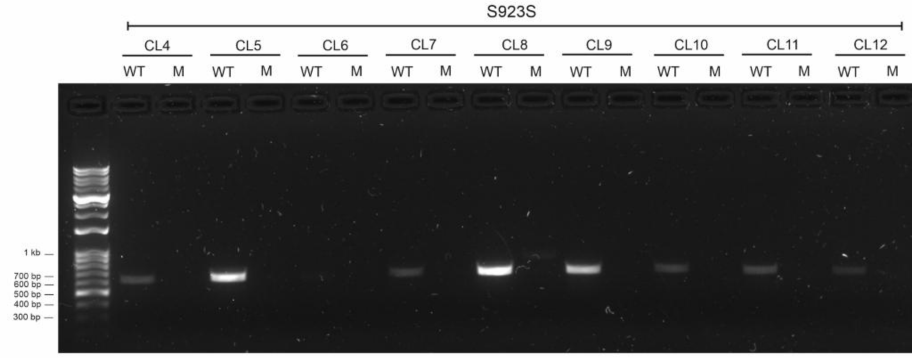

S530E
CL13
CL14
CL15
CL16
CL17
CL18
CL19
CL20
CL21
WT
E
1 kb
500 bp
400 bp
300 bp
200 bp
100 bp
700 bp
150 bp
S530E
CL22
CL23
CL24
T7Cas9
WT
E

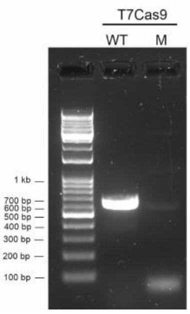
7.2.2.3.4 KKT2 S923

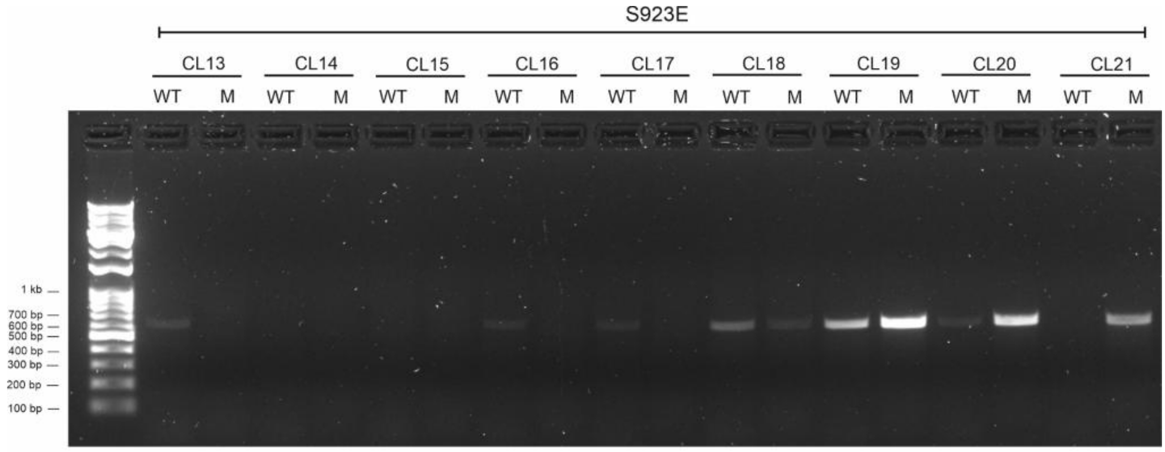


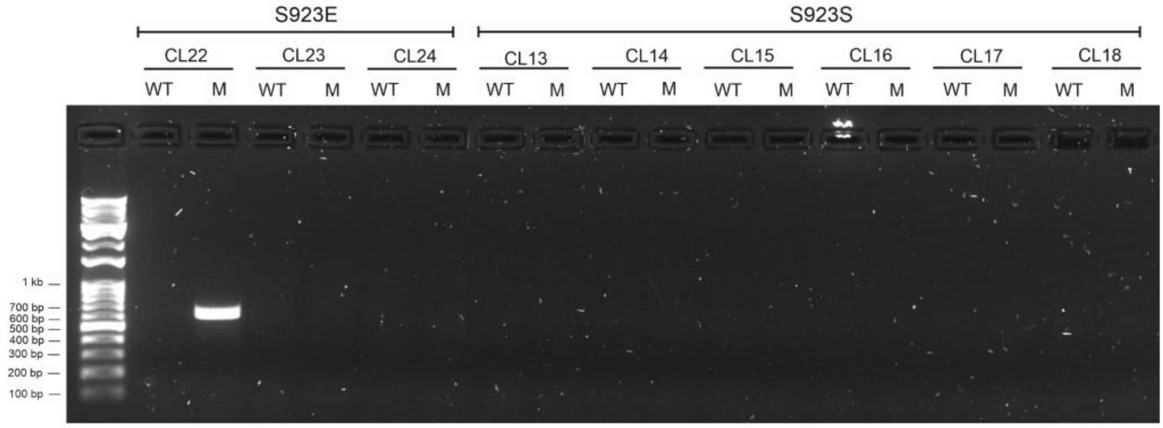


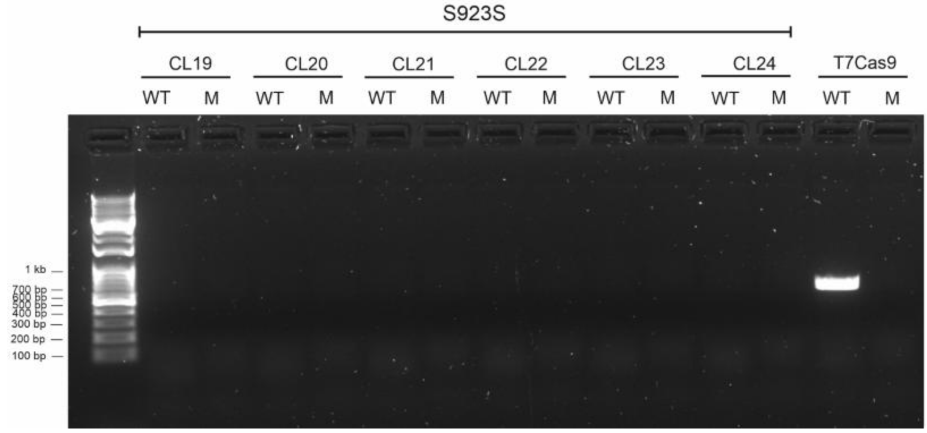




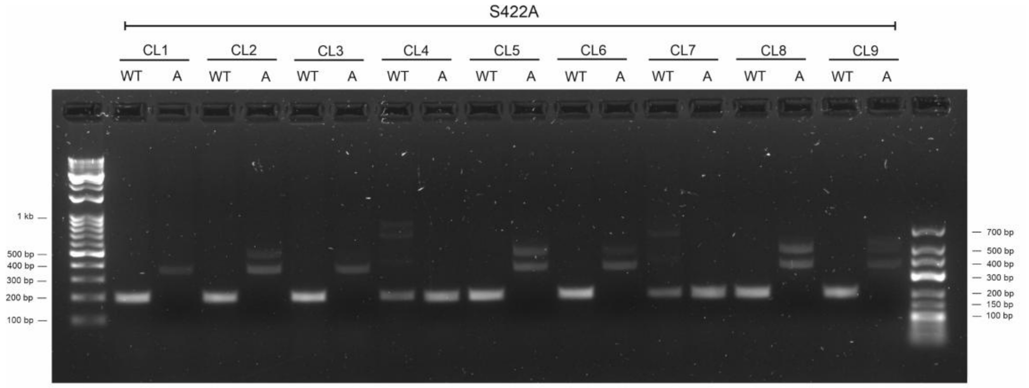


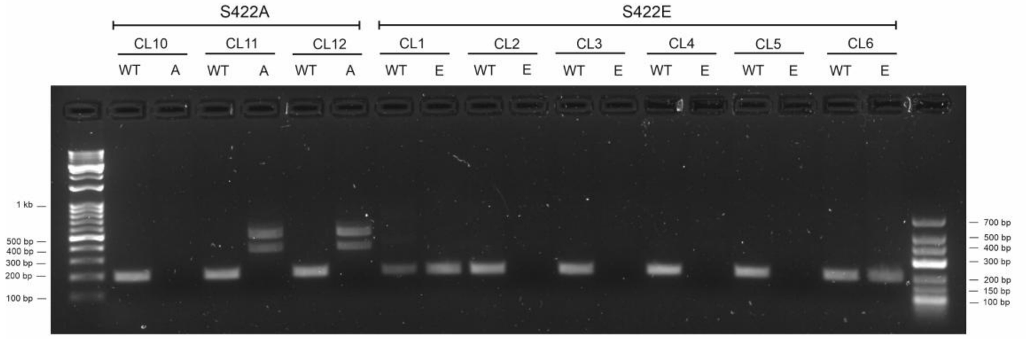

S923E
CL13
CL14
CL15
CL16
CL17
CL18
CL19
CL20
CL21
WT M WT M WT M WT M WT M WT M WT M WT M WT M
1 kb
700 bp
600 bp
500 bp
400 bp
300 bp
200 bp
100 bp

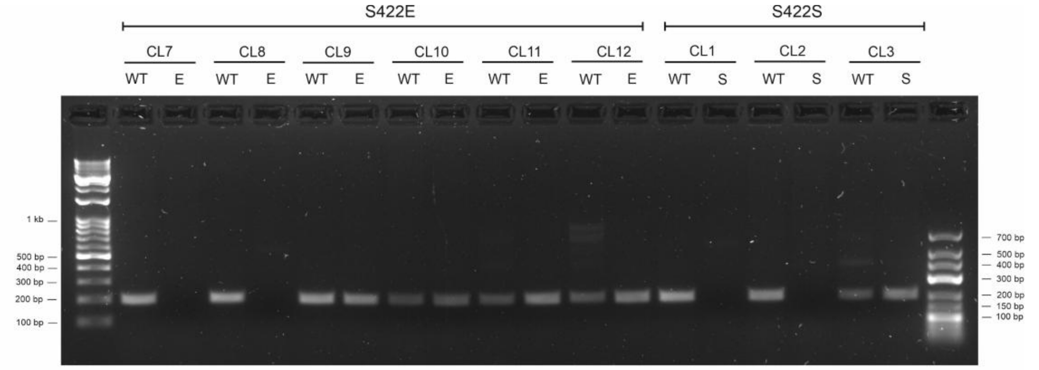

S923E
S923S
CL22
CL23
CL24
CL13
CL14
CL15
CL16
CL17
CL18
WT M WT M WT M WT M WT M WT M WT M WT M WT M
1 kb
700 bp
600 bp
500 bp
400 bp
300 bp
200 bp
100 bp

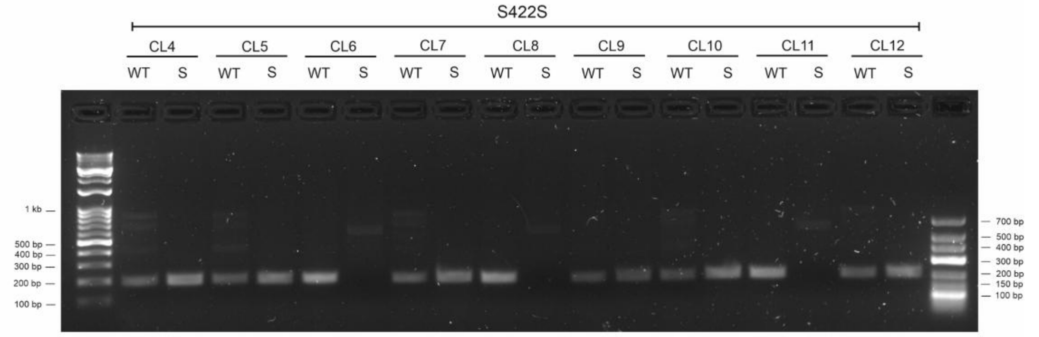

S923S
CL19
CL20
CL21
CL22
CL23
CL24
T7Cas9
WT M WT M WT M WT M WT M WT M WT M
1 kb
700 bp
600 bp
500 bp
400 bp
300 bp
200 bp
100 bp

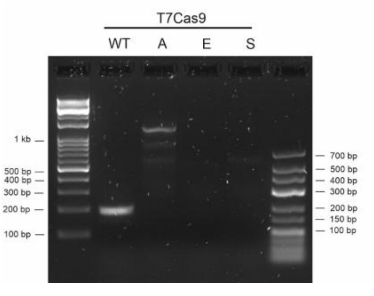
7.2.2.3.5 KKT4 S422



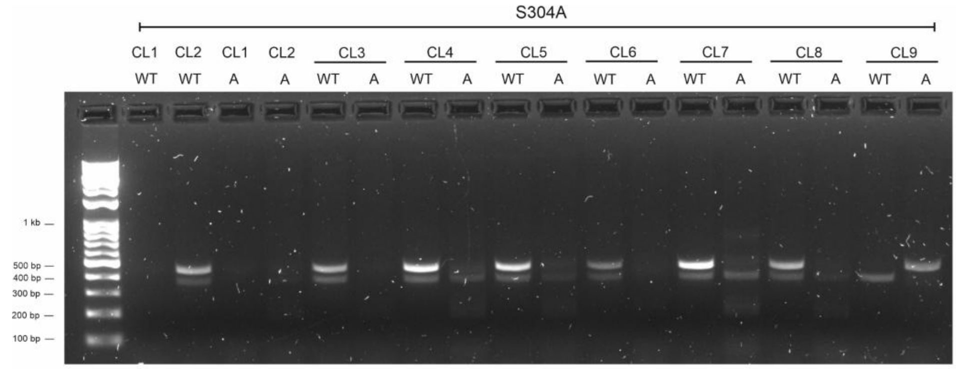


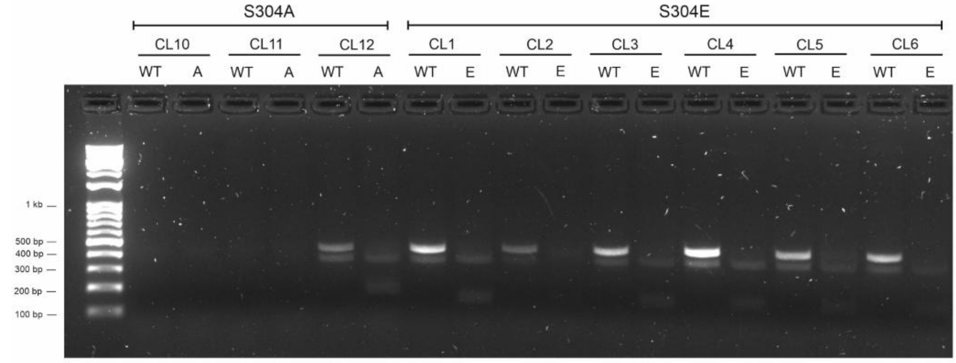


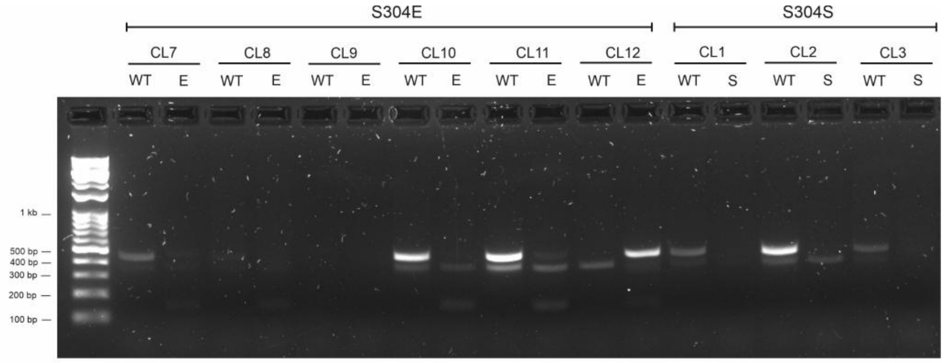

S422S
CL4 CL5 CL6 CL7 CL8 CL9 CL10 CL11 CL12
WT S WT S WT S WT S WT S WT S WT S WT S WT S
1 kb
500 bp
400 bp
300 bp
200 bp
100 bp
700 bp
500 bp
400 bp
300 bp
200 bp
150 bp
100 bp

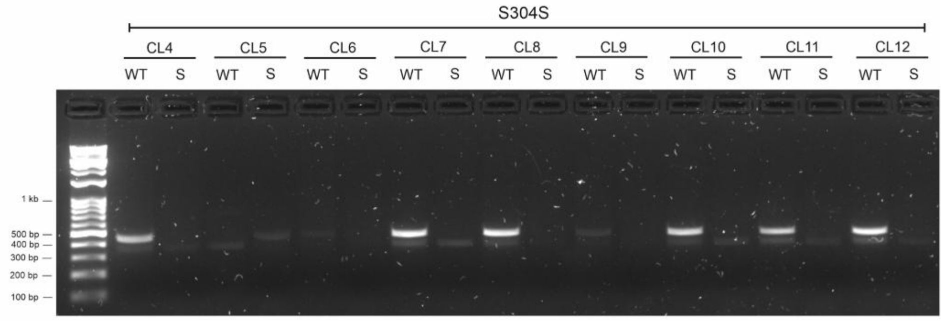


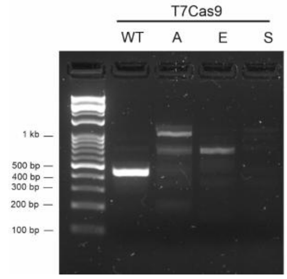
7.2.2.3.6 KKT7 S304


S304A
CL1 CL2 CL1 CL2 CL3 CL4 CL5 CL6 CL7 CL8 CL9
WT WT A A WT A WT A WT A WT A WT A WT A WT A
1 kb
500 bp
400 bp
300 bp
200 bp
100 bp


S304A
S304E
CL10 CL11 CL12 CL1 CL2 CL3 CL4 CL5 CL6
WT A WT A WT A WT E WT E WT E WT E WT E WT E
1 kb
500 bp
400 bp
300 bp
200 bp
100 bp


S304E
S304S
CL7 CL8 CL9 CL10 CL11 CL12 CL1 CL2 CL3
WT E WT E WT E WT E WT E WT E WT S WT S WT S
1 kb
500 bp
400 bp
300 bp
200 bp
100 bp


S304S
CL4 CL5 CL6 CL7 CL8 CL9 CL10 CL11 CL12
WT S WT S WT S WT S WT S WT S WT S WT S WT S
1 kb
500 bp
400 bp
300 bp
200 bp
100 bp



7.2.3 GENES OF INTEREST

7.2.3.1 Names and IDs

Name	Gene ID
KKT1	LmxM.36.1900
KKT2	LmxM.36.5350
KKT4	LmxM.10.0300
KKT7	LmxM.27.0430

7.2.3.2 Genomic DNA Sequences

All DNA sequences are for the CDS of the gene, except KKT2 which includes a region upstream of the start (indicated by underlined text) to show the homology regions of S25 mutants.

7.2.3.2.1 >KKT1

```
ATGGTTCTCAATTTGTTCTCCGGTGCGGCGCTCAACGGGCACGGCAGCACGCACCGTCGCG
GGCGAGCGTCTTCCTCGCTCAACAGCACGGACACGGGGCGCCGGCCTCAGCAGCAGCGTCG
TCAGGCAAGTCGCAGCACGACATACGGTGCATCCATGCAGACGGATGGTGCCGAGCAGTCC
GGATCTGGGCTCCGTGCTGAAGCTGCCGAGGATCGCGTGCTCTTCAACAACTGCGTGGCGC
AGGTGCAGCGCCACCTCAAGACGCACGCGGATTCACCCAGCACGCTCCACACGCTCGCCTC
TTACTACACCAAAACAGAGCCGTTCATCGAGGGCCGCCCCTTTTGCGTGACCCTGAGCTAC
GCCACCTTTCTGTTTCACATGCAAATGGCCCGCATCAGCGTCACGGATGTGGAGCTGTACG
TGCAGCTTCTCACCAGCATCTTGTCGCAGATCACCGAGGATGATCAGCTCCACCACCCGTT
TGTACAGCAGGTGCTTCGCGATCATGTGTTCGGTCTGCCGTCGCCGACCTGCCGCGGCGCG
GCCCACAGCGTGGTGCTCTTGTCACCGCAGCAGTACCGTGCCTTTGCGACGATGACCACTG
CGCTCATTTCCCTCGCCGTGGTGCCGCTCAGCATTGTGTACCAGTTCCACGACCGGCTTGA
GACATACTGCGAGTGCGCCTCACCGCTTGTAGCCAACCGCGCCTTGGCGCTGCTCGTGCAG
ACAGTGGGCGAGGTGCGCATGGATGAGCAGGTCACTGCACTTCAGTACGTCCTGAAGACGA
AGCCGGTGAAGATGAATGTGGACTTCCTCCTCGCTTGCTACGAGCGTCTGAAACGCGCAGT
GATGGATCCGGCGCACGGACCGTCGTTCGGCCGCGCCCTCTCCATCCACTGTAGTGAACTC
TTCCTGCGCTTCCGGTCCCCGGTGCGGCGCGACTACGTGGAACGTTTCCTATACCCGAGTC
TGTGCCACAGCGACATGGCCAGCTTCCTGGAGATACCTGCAACTCGCAAGCACCTGTTGCG
CGAGCTGCTGTCGCAGTGCACGCCGGGCATGGGAACCATGAATCCGTTCTACATGTGCCTC
TGCGCCGTCCTGCAAAGTTGCTTCGACAACGAGACGGACGGCGCGCTCGAGACGGTGGCCC
TCATCAACTGCCATATGCCACACGCCGCTTATTTCATGTCTACCCTGGCTGTTGACTCGCA
CATGTCTGTGCCGATGTTTGCCAAAGTGATGATATCGCTCGCTCGCGGCGCCGGGATGGCT
ATGACGGGTCGCGACACGCCTGACGAGGTGGCCGCCTCGATCAACGAGAACCGCACGAGCG
TGTATAATGTGCTCTTCCTGCTCCGCGAGGTAGTTCGCAGCTGCTCCACCACCGCCTCTCG
CCGCGCCACCGATATGCTTAAGGCGCTGCGGGTCGCTGTGGCTCCAAAGACGATTGAGGCG
CTCGGAAAGCTGTCCAGCGAGGCCTTCGAGGCCGTCAGCGACATCACACTCGATCCGCAGC
TGCTGTGCGCAGAGCTGGCGATGGTGCTGCATCAGGACCATATCGCAGAGGCCATGGACTC
GGCAGTAGAGTACTTTCGCGACGTCAGGTCGAAGTGCCCGTACTGCGCGGCGGCGCGCAGC
TCTTCGCTGCTGTGCCCCGTCAACGGCACGGTGCACGTCGCCGGCCAGTCGTCGGTGAGCC
GCGTGCTGTCGACGCTATCGGAGTGCGCTGGCGCGAAGGCGGTGGAGGAGAAGCTGATCAG
CTATCTGCGCGATCCCGCCTTGCAGATGGAAAGCGCCGTGCACTACCTCATCTACCACATC
```

```
GTCGCGAATGGCGGGCAGCACCGTAATACGCTCTTTGTGGCTGTGGAGCCGTACGTGCGGA
GCACATTACTGGCTTTGGTGAGCGCAGACCGCAGCGGGGTTCGCGGGCTCGTGGACAGCAC
GCTGAAGGCAAACGTGCTCATGCTGCACGTGAAGCTTGTCACCCTCCTCGCCTCCTCCATC
GACCCGTCGTACCTGGAGAGCATCTTGAAAGTCTTCTCGGAGCTGAGGCTGCGCAACAACC
ACGACGCGCTCGCGCTGTGGTACATGGGCAATGTTCTGCTGCGCAGCTGCCGCGGCAACTT
AGAGCTGCTACCCACCGACCCCCAGGAGAACAACTACTGCGTTGCCTTCCCTGGCTGCGCG
CCGGCAAGCGCGACGACCGCCGACAACGCGCAGCTCGTGCTGAAGCTGCTGCACCGCGCGC
ACAGCTTCAGCCCTGAGATGCACAAATTGGTTGGCTGTTGCGTGTGCAAGCTCATCCAGGA
CTTCAACATGCAGGCTCCAAACATTTGCAGCACCTTGCTGTCGCCTTTCGGCTTCTTCCCA
GTCGGACTCGAGTCACTGAACGCCTTCGCCCTTCCGGCAGGCGCCGGCAGCACCTTCTGGA
GCTTCTTTTTGCAGCAGATGCGCAGCTCTGCGCCGGCGCGGACGGCGTTCATGGCGACACT
GGCCAAAAGTCTATCGCGGCGCTTCCGCATTGCATCGCCAATGGACGCGCTCGCCCCGTAC
GGGGTGGAGCCGACAGGGCACTTGTTCGTCATCATGGTCTACGAGGCGATGAAGCGCAACC
CACCACTGGCGCGCGTGCTGCTCTACATGGTGTCGCACTGGATGAAGCAGGCAGGCCACCC
GCCCGGCAAGCTCGCGTGCCTCGTGTACGTGTGCGTGCAGCTCATCACAGTCGTGGTCGAC
CGTGCAGAGGGCCCGGCTGCGGCAGAGGTGGAGGCGGAGACGCCGCAGGATCGGCAGCAGT
TTGACGACGCCGTGAAGAAGGCGGCACGGGTACTGAAGAGTCAGCAGGCGCGCCTTGATAG
ACTGGCCCCGACGGCGCGGCGCGAGAACGTAGAATTCTTCCACCTGCTGCGCCGTTTGCAG
CGTCGCGTGCGCCGGACTGTGGCCACCGCCTCAGGTGAGATTGTTGTTGGCGACGAAGCCG
CGGAGGAGTACGACGACCACGATGACGCCGTTGACGACAGTTCGGCTGGCGGGCATGTGCG
GCAGGATTCCATCACAGACGCAGTTTGTGCCATGCAGGAGCTGCAGAATGCCGCTGACAAC
AGTGTGTTTGACGACTACGCCGATGATGTCGACCAAGAGGACGACGGCGCTTACGGGAACG
ATGAGGGTGCCTGCGACGCTGCTTCTCCAGGGCTTAGGCGTTCCGCGCAGACGGAGGGCAG
CGGCCACAGGGCCGAGGGCCCGCCTGCCCCAATGCAGATTCGCCACTTGCCGCAAGGCATA
ACCAGCATCCTGCGCTCGCCCGCGCAGAGAAGCCCCAACAAGAGCGACAGGGGTGCCGCCG
GTGTGGAGAAGGGCTCGACGACCTCTGTGAACATGTATCGTGAAGCGAACCGGCGAACCGA
TGTCGAGGGCGTCCCGCATGGCGCGGATGGCGATGATGCAGAGATGCGCAGTCGCGATGGC
GAAGCAGCCCACAGCTGTGCGCTGGGAGTTGAGCCTCGTACCCGGTCGACGTCTCGCGGCG
TGCAGACGGACGTGCCTCTGGCGAGCCCCGCGCTGCCCGGGAACGCGCCGCAGCGGAGCGT
GGGGACGTCACCGATACAGCCGGCAGGCACCTCGTCGCAGATCTCGGTCACGCGACGCGAC
GGCACGCAGCTGCCCTGTCGCACACCTGCCGACGTCGGCTCTGCGCACACTCCCTCCTCCT
CCCTCTATCAGCCACAGCGCTCCCACACACGGCCGCCAGAGGCCGATGGCATGCTCAGCGA
GGGTACTCGGACTCCAGCGCAGCGAGGATCGACGTGGCGTGAGCCGGACCTGGCCGACTAC
GTGGACGGTGACACCACCCCGATCGACGACTTCACCGGCGTGCCGCGGCTGCAGGCGACCA
CCACGAGTGACGGTATTGTGCTGCCCTCTGGTATGGTGCTCGAGTACCTGCGCACGCACCA
AGGGATGGACTCGTTGCAGCACGAGCTGAAACAGTTTGACCAGCAGTGGATGGTGCAGCAG
GTTGCTGAGTACGTGTCGCAGAACGGCGGCATGGTCGGCGCTGCTGGTCCGTCGTCGACTA
TTAGGGGCGGTGTGTCGTCTGTGCAGTCCGTTACGGTGGAAGGCCGCGCCAACAACTACAG
CCGCCCGCACGCCGATCCAACTGAGCTTGCGCCGACCCGCACGGTGTGCACAGAGGTGCAC
ATGATAGGGCCAGCCACGTCCTACTCGCGGCCACCTAGACAGGAGGAGCACGGGCGTGTCG
TGGCAGCCGCGCCGGGCCTGCCTGAAGAGGAGGAAGTGAACGTCGTAGATGGCGAACACCC
TATTCGCGCCGTCAGCGGCCCCCCAGACGACAGCGACCTTGCTGGACGCGCAGGTGACGAC
GAAGCGACTAAGCGCCGACGTGTGGAGGCCACCGGAGGCAACGCGACAACTCCGCTGCCAC
CACCAGTCTCCCCCGTAAGCGCTTTCCGCGGCCGCAACTTCTTCTTGAACCAGCATACACA
GCAGGAGGTGGGCTCGACGCTCCAGGACATTCACTACCTGCAGAGAAGGCAGCAGGCGAAC
ATGTCGGCGCTGGCCAAGGCGCAAAGCGCGGCTGAGACGGCGGAGTCAGCAGGTGACGACG
AGGCACCGCGCAAGACACCACACCAGGGTCAGTCCTCTACGGGCGTGGCGGGTGAGGGTGT
GCCGCCGACAACGCCTTACGGACAGGTTATTCTTCCAACTTGGATCGTGGAGCAGCGCAAC
GACACGGCTATCCGTGAGTTGCGACAGGTGATGGGGGCTCACAACCCGAATGACAGTCGAC
TATCGACGAGTGCGGGCAAGCGCAGCCGCATTCGCGGCAGTGGGACAGGCGACGGCAGCGG
```

```
CAACAGCGCTGCGTGGTGGGCTGAGATGAGCTCGGCGCCGATGCCCAACTACGCTGCGGAC
CCTCAGTACTCCATGGAGCTCTTTTAG
```

7.2.3.2.2 >KKT2

```
GGCCTCTGATGTCACACTTTTGCGGCTCGTTGTCGAGGACTCCACCACGGGGTGGGGCGAT
ATCTATGCCGCGTGATTTGTCGCAGACCCCCGCCATCTCTCGACTTGGAAGCACGGTGAAG
ACGCCGCACATCCAAAAATGTGTTGTTGACCAAGCAGAGGATGATGATCATCCACTGGAGC
ACATGACGGTCTATTTTGAAGAGGAGGAGCTTAGAGTAGTTACCACTGGGCTGCTTGGAAA
AGGTGGGTTTGGAAAGGTATTTGATGCCGTTTCGAACAGCGGTGAGGCCTACGCGCTCAAA
GTGTCATCGAAACGCATGAGCGAGAACGACTGGAAGCGACTGAAGGAGGAGGTGACGCTCA
TGAGCCATTTCTCGCGCCATCCCAACATTGTCAAATTCTACGGTGCTGGTAGGGATGAAGA
TCGCGCCTACGTAGTAATGGAGCGGTGCGCAGGCAAGTCGCTTCACGACGTCATAGCCAGC
AGGAGTCTTGATGTGCCGGAGATTTTGTGGATTGGGTGGGCCCTGGTGAACACCATCTCCT
ACATTCATTCCAAAGGCTGCATTCACCGCGACCTGAAGCCACAGAATCTCTTGTTTGACAA
CGAAGGTAATTTGAAGATAACAGATTTTGGACTTTCCAGCCGCATATCAGAGGCGCATCCT
CGCAAGACGGTTGCCGGTACAGCAATGTACATGGCGCCTGAAATGGCAACTGAGGTTTACA
AGCGAATGACAAAAAACTCAGAAGCCCCTTCGCTGAGCTACGGCAAGGAGGTGGACACGTG
GAGTATTGGTGTGGTCCTCTACGTGCTCTTGACACGCATGAATCCGTATCTCGAGGCGATA
GAACAGAAAGGTATGCGCCAACTGGACAAAGAGCACAAATCGCTTGCCCTCTTCAACGCTG
TAGCGGGTGCCGCGTGGAGTTGGCCAAGGGAGTGGAGGGGAGATCCACAGCTCTGCGGACT
TGTGGAGCGCATGTTGCACCGCGAGCCGTCGCGGCGCGCCACGCTGATGGAGGTGCTCGAG
GACTCTGTGTGGAACCGCCGGCCACTGTCCTGCCCACTTTCGCTGCTCCAGAAGCTCAACT
TGCTGGAACCTTCGCCGTCGAGTGGCCTGCCTCTGAACAACCTTGCCGAGAATTTACAGTT
CCGCCCGAAGCGCTCGGCGGAGGCGGTGCTGCGCGAAGGACTAGAGCGCGTCGAAGCCACT
GAGCAGCGCGGTCGTGCGCAGCTGGAGCTTGAGTACTACGAAACCTACAATGTCCTCTGGA
GCCTTCTTACTCTGGCGCGGGCGGAAGAGGACGCCAGAGCTGACATCCTCCAGTCCGAGGA
GGTGCAGCGAGGCAAGCTGCGCAATCAGTCTCTTGCTCGCCAGTCTGCCCGTCGGAGGTGT
GGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCCCCTCGCACGTCTC
GATCAGTGCGTCGTAGCGTCAGCTTAACGGAGCAGGAGCGGGGCAGACTTGTGCGTTCTAG
CCCGGTCCAGTACGCAGTGGTGTACCCGGGGCGCGACACTGCCACTCGTTGGAACCTTCGC
GCCGTAGTATCGCTGCCACGCGACATGACGGACGAGATCGAGCGCGAGTTCAAGTGCATGA
ACGGGCACGTAATGACAAAGTTGACCTCGATGCCGCACGGCTACAACGGCTTCGACTGCAA
TGTGTGCGATCGCGGCATTCTTACGATCACGGCCGAGTCACCAGCCTTCCGATGCTACAAG
TGCGACTACGATGTTTGCATGAAATGTGCGTACTCCGGCAAGTTCAAAGACGTTAACTTTG
TCTGTGTGACATGCGCGAAGCGCTTCACCTCAACCGCAAAGCTGCAGGGGCATTCCTTGCG
CTGTCGCGGCCCGAGCGAAAGTCCCTCACCGCGGCGATCGTCGCGCATGAACACGATGCTC
TGGGACGAGCCGAAGAGACCGAGCCTGCTGGAGGTACAGCTGCCTGAGGCGCCCCAGAGCG
AGCGGAAGCTGCGCGCCAGCCGCTGCCGCTCTGGACGCCCCACGTACAACCGCACATCGAC
CGGTGGCCGCATTAGCATTGGAGACTCGAATGCGCACAGTGTGGTGGACTTCGACGCAATG
GTGGCCTCGCACCGCGAGGCTGACTTTCCCAAGGTGAGCACACGCGCGTCTGCCACCGGCC
GCGAATCCTCGCAGAGACGGGAGCGCACGGGTAGTGGGCGTGGCCGACCATCCACCTCGTC
TAGCGGCAGCCTTTCACTTGACTTGCCGCCGCAGGTGCAGGTACCAAGCAAAGAGTCGCGC
CCACAGGTGCAGCCGCGTAGTTCCGCTGAGCTGCGCGATATCATGGAGGAGGTGGAGCAGC
GGAAGCAGGCACTGCCCCGTGACCCCCTCTTGTCCGCGCCGGCCACACCGCCGCAGTACAA
CTGCAACGGTGAGATCATCGGCATTTCTGCTCGTCGCCGCGCAGAGAGCCTGGAGATGGCG
CGCGCGGAAGTCATCACGATCCGCGCCGAGGTCGCGGACCGGCCGCGCGAGCTGCAGCATC
AGCCGCGTGTGCCGCGCAGCGCCTCCTCATCGAGAGCGGAGAAGGGGCTTCCGAGCCCCCA
CAAACGCCGTCGTGAGGAGTGGCAGCAGCCCGCGCATGCGCCGTCTCCATCCGGTACGGCG
AAGCGAGCCGCTGTGGAGGAGCACGTGGTGAAGCAAGCCATCATGCCGCCTCAGGTGCCAC
GCGGACGAGCACAGCAGCCACGTGCCCCCTCCGTCAGCGGGCACACCGCACAGGGCGGTCC
```

```
GCCACTGCCGCGCCGCGGCCCAGCTGCGCCATCTCCTGCAGCCGCTTTGAAGGCGCACCTC
AGTCCCTTCCAGGCCCCCGCTGCGATTCCTCCCAAGAACTTTGCCTCCATCCTGCAGTCGC
GCTACTCCATGACGAACGCAATGGCGCCCACGTGCAACACTTCGACCACAAGGCCGGCGGG
CGGAGCAGGAGCTGCGACCGCTGCTCTTGGCCAGGGGGGTGCTCCGACATACAGCCATGCG
CTGAGCCGTCCCAACGGCGCCTTTTTGGCGTTGCCGCGCGAGGAGCGGAATCGCCAGCAAT
TCTTGGACGACTTTCTTAGTGGTGGCTGGGTGCGCTTCTACTCTTTCACAAACGAGGACAC
CGTCGTCATGTACTACTCACTGCAGCCTGGTCGCTACGGGGCCATGTTTCCCACCGAGGCA
GGCGTCGGCACTGCTGTGTTGGACGTGTACTCGAAACTGGTCCTCTATGTGCCGTGCATGA
ACAACGAGAGCACGAACCGCAGTCAACCCCACCCACACGTACAAACGTTCTACGACGAAGA
GGCGCGCATTCTTAGCTTGCCGGAGGCGCAGCGGTACCTGGGCGGCGTGCTACGCTGCATC
ACTGGATTTGTAGATGAGTTTAGCCGCTTGAAGGCTGAGGGCCTTACTCCAGCGGCGGTGC
ATGCTGCCTACATCCACCACCGTAGCATGTCCCACGTGCCGCGGGATACGAAGTTCGTGTA
CATTCGCAAAGTATTCCCTGACCCGGCTGGGTCTTTCACGCTTTTCCGCCTGTCGAACCTG
CGCTCGCAAGTCGTTTGCAACGCTATGGTGGACATTCGCTGGCAGAGTGACCGGCGCCACA
ACGTTGGCCAAAAGTATTACATCAACGCGGACGGCACCGCTGAGCCTTTTCTCGTCGATCA
AACCGGAATTCTGTCGCAGCTGGAGACGGTCCTCAACAACAATTTCCGGAGATGA
```

7.2.3.2.3 >KKT4

```
ATGAGCACCGACGCCCAGGAGCTGGTGCGCCAGCTCACGGAAAACCCAGAGGTTCTGGAGA
GCATGCAGCACATGATCTCTCTACTGCGTGCCAATCCTCCGCGTATCTCCGGCAGCAACAA
CGGTGGAGGTCTTGGCAACGCGGAGACTAACGGCCCTGAGAGAGGTGCACCGCAGTGTGTG
CGACCACCGCGCCGCGGATATGGCGCTGACGTTGATTGCGATCACCACCAGCCCACAACCA
GGCGGAAGCTGCGCAGCAGTGATGGCACCGCCCACAGCGCCACTTCCCTGTCTGCGTCGTC
GTTGACGCAGGAGGCGCACTCCTTCTATGGTGACGACAGGGTTGGTGCGCGCACCACCGTC
AGTGATCACAACGGCACCACCGGCGGCGCCTCTTCGCCTACGCCAAGCTTCGTCAGCACAG
GATCCCGCGCAGCGCCTCAGGTGGTCACTGCGGCCTCACGGCACGCGCCGCGCCGCTCCTC
GCTTCTCCCGAGCCCGCACGAGCATCGCCCCACCACAGCTCCCGATGAGCAGCTGATGGCC
ACCGCCAACAAGCTGACGGAGGCGCAGCGGCGCATTGCAGAACTGGAGAAGGAGCTCCAGC
GCACCACGCAGCGGGTGGACCAGTTGTCCGATGTGGTGCAGCGGCAGAAGGACGAGCTACA
GGCCGCGAAGGATCGACATGCGCTAGAGATGGAGGAAACACGACACGCCTACAACGCCGTG
ATTCACCGCAAAGACGAGGTGCAAGAGGAGGCGCTGCGCCAGCTGCTCAAATCCCGCCAGC
TGATGGTGTCGGCAGCCAGGTACGAGGCCGTCGTGGCGGCGAAGAAGCTTCACGCTCAGCG
GTTGGAAAAGGAGAACAACACCGGCGCCGATGATGCGATGGGAAGCCCGAAGGGGCTAGCA
GGCGTACAGGCAAGCGCGAACCCCAACGAGCGCGGCACTCACCCCGGGCTGGCGCCAAGTC
AGACATCAGTGAACGCGCGGCACTCTTCGACGCTCGGCTACGGGTCGGGCACGACAGCCAA
GTACAGCAGCGCTCTAAAGCGTGACCGCCAGAATGACGAGGGGGACCTTGTTGACGATGCC
GGCGTCGAGACTGGCGCACACGAGCCTGGTGAGGCGCGATACGGGGAAGCAGCTCACCACC
ATCCGCCAGTGAAGCGCACCACGTTGGACACGTCTCGTCTGCAGGGCAGCGCCGATCGTGT
CGTGCAAGGACGGAGGGGCGTTGCGGCGACCAAGGCGGAGACGTCTCCGGCGTACATCACC
ACCCCCACGCCGGCCGGCAAGGCGTCCACCGCGCTCGTCGGCACGCGCACTCAGTCAAGCA
GCGCGCGCAAGCGCCGCACCCCGCGCACGCCGAGCCGCACCAACGCTGAACGCATCGCCGG
GTCGGTGGCAGAGAACAGAATCCGCTCGCAACAGCGCCTGCCTGGGACAACGTCGCTGAAG
ATCGAAAGTCCCACGCCTGTGGTGAGCACCGCGTGGACGGCGGACCGTTCTCTCACGGGCA
GTCGTACGCCGCCGCCGTCCAGCGCTGGCGTGTGCACGGTGTCCGAGGCAGTGACCAAGCA
TCATCAACTTTACCCCCAGCAGCAAGTGCATCAAGTTCCGTCCACGAGACCGCCGCTCATG
CAGCGTGCAGCGGGTCGTCTTCCTCCAGCTCCGCACCGCACCGCGGCTGCCTCGACGGCGG
TGCCCAACACGCGAAGTGGTACCTCTTCCATTGCCTCCGGTGGCCCGACGCGGTCACCGTC
GCCTGTGAACCCGAAGCGTGGCGCCATGCTGCCGCGCCGCTTCATCTTCACAGGTCTGAAA
GACCATGAGCCTCAGCGGCTGGTTAGCGCAATAGCCGCGGTCGGCGAGGATGCCGCGGCAC
TGGCGAGCGACCTCGATGAGCCGCCGCCAAGCAGCACGACTCACATTGTGCTGCGCGGGAC
```

```
GCCGCGTAGCGTCAAAGCCCTCTGCGGGGTGGTATCGGGCAAGTGGCTTGTCTCTCCCGAA
TACGTGTACAACAGCCAGCAAAGCGGCTTCTGGCTAGACGAGCTCGAGGAGGGCGGTCTGC
GCATCTTTCCGCCGCCGCTAAAGTGCCAGCGTTTTCTTCTGACGGTGGAGCACCCAGGCAT
CCGGGCGAAACTGGCGCAGGTGATCGAGTACGGCGGTGGCGAGGTTTTGGCAAGCGGCAGT
GACAAGCGTGGCCCTGGCGCCGGCGACACTGTGGCGCAGGACGTGGTCGTGATCACCTCTG
GTGATGACCTCTTGCGATATGCGACGCAAGACCGCGTGTAA
```

7.2.3.2.4 >KKT7

```
ATGACGGACGTAACCTCTTCGCTCCGCCCGTCGTCGCGCCAGGGCTCCCCGGTGCCGCGCC
GGCAGCTCGGCATTCTGCCTGTGAACCAGCGCTCCTACTCGCGTGTGGGCTCCAAGGGCAT
GATTGGCGACGACTCGCCGCTCATGTCACCCTTGCCCTACTATCCGCGTCGTCGCAGTGTC
ACCTTTGCCGGTGACCAGAGCGTGAGAGAGGAGCGACCCAACTACAACGCCGCATATTCCG
CTTCGGCTCCCGTTTCCCCGGCGCGTCACGGCTCACCGCCGCCGGTCTCCATCCTCAAGTC
GAACTTGTCGTTTCCGGCGGCAGAGGAGGAAGACAGCGGCGCTGCGCCGGCGTACCAGGCT
GCTGCGGCCACAGTGAGTGGTGTCTTGGACCGCAAGGACCGCGCGCGCAACTCTCCGGTGC
CGGTGCGCGGCCGCTCCAATAGTCGTCAGCGCCTTGCGGCGCGGCGCAAGGAGGCGCAGCT
GCATCGCAGCTTCTACGATGACAGCTTCGTGGAGGAGTATGTGCTACGAGCCAAGACGGAG
CTGGAGGAGGAGGAGGCAGAGCAACGCCGAATGCAGGAGCAGCTGAGGGCCGAACAGGAGA
GGGCGAAGAGGGCAGAGCGCCGCGTCTCGGAGGCAACGGAGAAGATCAACGCCCTGCAACA
CGCGAAAGAGGTGCTGATGGCGGCCACGGTGCGCCGCCACACCTCTGTGACGCCGTCTCCG
CAGCGTGCGCCTGCCGAAAAATCGAAGCGCAACTCCAGCCTTTTGCGGGAGCTCGAAGAGG
ACCCCGACCCAGAGGTGCAGGCAGCGCTGAAGGAGCTCGCACGCAACTCCATGGCGAAGCA
ACAGAGTCGCGTTCACTCTTCTGCCCATCAGCGTCGTCGGTCGATATCCATTGTCTCCGCC
GACGCCCTCGCGAAGAGCGGCGAGGACGAAGACGGTGACGACAACGACACCCGCAAGCGCG
CGCGTCTAGAGAAGATCGTCTCCACGCTGCTTGCGAAGAAGGCCAAGAGCAAGAGCAAGCG
TAGCGTGATGGTTATCGACTGGTCCGATCTCGACTCCGACGCCGACGGCAACACCTCGACC
ACTGATGAGGATGGGGAGGAGACTGCGGTGGGCCTCAAGCGACAACGCGGCCGCCCTGCCA
AGAGCCGCAGTATAGCGTTGGGGACCGAGGCGACACTGGTGTCATCGGCGAAGCATGTACA
GAAGCCGTCCACGAAGCGCGCAGCCTCGTCCCGTAAGCGCCATGTCAGCGCAGAGCCGGAG
TTGGGCGATTCGCTTCTTTTTGAGGATGAAGCGGAGCAGCCGATTTTGCTTCCTCGCCGGC
AGAACACGCGACCGGCTCCGACTCGGTCTATCTCGTACATCGAAATGGGTGGCGACGATGA
CCTGCTGAGGGATGCTTCCAGCGTTGAGCGTGTGGTGCGGCGACCACCTCGTGCCACGCGC
GCACCGGCCACGCGGCAGCGCCGCGGCCGTCTTGCATCTACTAGCACCCGCGAAGGTGCAG
AGGTCATGTCGTCTTTCACAGGCACCACCGCCTTGCGAGGACGTGCGTCGCAGCCACCAGC
CGCGCCGACAGGGGGCCCGACCGGCGTCCCGCCTCGCCGGCGCCGCGGGTCCGTCCAGCGC
GCAGACCCCAATGATCCCATGGCCGTTTTTTTTGAGGCTGCCTTTCCGAGTCCTTCGAAGT
TTGACGAGATGATGATGCAGGCTGGCGGCCTGCCAGAGACCCGTCGTGGCGGCGGTGGCGG
CGGGCGAGGACAGGGACGGCATCCCAACTTGGTGCTGCCTAGCTCCATTGGACGCCGCCGC
TGA
```

7.2.3.3 Native Protein Sequences

As with the DNA sequences, the TriTrypDB start methionine for LmxM.31.0120 is indicated in underlined text. The protein sequence listed is the extended one.

Sites targeted in this project are highlighted in yellow for each protein.

7.2.3.3.1 >KKT1

```
MVLNLFSGAALNGHGSTHRRGRASSSLNSTDTGRRPQQQRRQASRSTTYGASMQTDGAEQS
GSGLRAEAAEDRVLFNNCVAQVQRHLKTHADSPSTLHTLASYYTKTEPFIEGRPFCVTLSY
ATFLFHMQMARISVTDVELYVQLLTSILSQITEDDQLHHPFVQQVLRDHVFGLPSPTCRGA
AHSVVLLSPQQYRAFATMTTALISLAVVPLSIVYQFHDRLETYCECASPLVANRALALLVQ
TVGEVRMDEQVTALQYVLKTKPVKMNVDFLLACYERLKRAVMDPAHGPSFGRALSIHCSEL
FLRFRSPVRRDYVERFLYPSLCHSDMASFLEIPATRKHLLRELLSQCTPGMGTMNPFYMCL
CAVLQSCFDNETDGALETVALINCHMPHAAYFMSTLAVDSHMSVPMFAKVMISLARGAGMA
MTGRDTPDEVAASINENRTSVYNVLFLLREVVRSCSTTASRRATDMLKALRVAVAPKTIEA
LGKLSSEAFEAVSDITLDPQLLCAELAMVLHQDHIAEAMDSAVEYFRDVRSKCPYCAAARS
SSLLCPVNGTVHVAGQSSVSRVLSTLSECAGAKAVEEKLISYLRDPALQMESAVHYLIYHI
VANGGQHRNTLFVAVEPYVRSTLLALVSADRSGVRGLVDSTLKANVLMLHVKLVTLLASSI
DPSYLESILKVFSELRLRNNHDALALWYMGNVLLRSCRGNLELLPTDPQENNYCVAFPGCA
PASATTADNAQLVLKLLHRAHSFSPEMHKLVGCCVCKLIQDFNMQAPNICSTLLSPFGFFP
VGLESLNAFALPAGAGSTFWSFFLQQMRSSAPARTAFMATLAKSLSRRFRIASPMDALAPY
GVEPTGHLFVIMVYEAMKRNPPLARVLLYMVSHWMKQAGHPPGKLACLVYVCVQLITVVVD
RAEGPAAAEVEAETPQDRQQFDDAVKKAARVLKSQQARLDRLAPTARRENVEFFHLLRRLQ
RRVRRTVATASGEIVVGDEAAEEYDDHDDAVDDSSAGGHVRQDSITDAVCAMQELQNAADN
SVFDDYADDVDQEDDGAYGNDEGACDAASPGLRRSAQTEGSGHRAEGPPAPMQIRHLPQGI
TSILRSPAQRSPNKSDRGAAGVEKGSTTSVNMYREANRRTDVEGVPHGADGDDAEMRSRDG
EAAHSCALGVEPRTRSTSRGVQTDVPLASPALPGNAPQRSVGTSPIQPAGTSSQISVTRRD
GTQLPCRTPADVGSAHTPSSSLYQPQRSHTRPPEADGMLSEGTRTPAQRGSTWREPDLADY
VDGDTTPIDDFTGVPRLQATTTSDGIVLPSGMVLEYLRTHQGMDSLQHELKQFDQQWMVQQ
VAEYVSQNGGMVGAAGPSSTIRGGVSSVQSVTVEGRANNYSRPHADPTELAPTRTVCTEVH
MIGPATSYSRPPRQEEHGRVVAAAPGLPEEEEVNVVDGEHPIRAVSGPPDDSDLAGRAGDD
EATKRRRVEATGGNATTPLPPPVSPVSAFRGRNFFLNQHTQQEVGSTLQDIHYLQRRQQAN
MSALAKAQSAAETAESAGDDEAPRKTPHQGQSSTGVAGEGVPPTTPYGQVILPTWIVEQRN
DTAIRELRQVMGAHNPNDSRLSTSAGKRSRIRGSGTGDGSGNSAAWWAEMSSAPMPNYAAD
PQYSMELF
```

7.2.3.3.2 >KKT2

```
MSHFCGSLSRTPPRGGAISMPRDLSQTPAISRLGSTVKTPHIQKCVVDQAEDDDHPLEHMT
VYFEEEELRVVTTGLLGKGGFGKVFDAVSNSGEAYALKVSSKRMSENDWKRLKEEVTLMSH
FSRHPNIVKFYGAGRDEDRAYVVMERCAGKSLHDVIASRSLDVPEILWIGWALVNTISYIH
SKGCIHRDLKPQNLLFDNEGNLKITDFGLSSRISEAHPRKTVAGTAMYMAPEMATEVYKRM
TKNSEAPSLSYGKEVDTWSIGVVLYVLLTRMNPYLEAIEQKGMRQLDKEHKSLALFNAVAG
AAWSWPREWRGDPQLCGLVERMLHREPSRRATLMEVLEDSVWNRRPLSCPLSLLQKLNLLE
PSPSSGLPLNNLAENLQFRPKRSAEAVLREGLERVEATEQRGRAQLELEYYETYNVLWSLL
TLARAEEDARADILQSEEVQRGKLRNQSLARQSARRRCGSVSLVSEVADREEAAPRTSRSV
RRSVSLTEQERGRLVRSSPVQYAVVYPGRDTATRWNLRAVVSLPRDMTDEIEREFKCMNGH
VMTKLTSMPHGYNGFDCNVCDRGILTITAESPAFRCYKCDYDVCMKCAYSGKFKDVNFVCV
TCAKRFTSTAKLQGHSLRCRGPSESPSPRRSSRMNTMLWDEPKRPSLLEVQLPEAPQSERK
```

```
LRASRCRSGRPTYNRTSTGGRISIGDSNAHSVVDFDAMVASHREADFPKVSTRASATGRES
SQRRERTGSGRGRPSTSSSGSLSLDLPPQVQVPSKESRPQVQPRSSAELRDIMEEVEQRKQ
ALPRDPLLSAPATPPQYNCNGEIIGISARRRAESLEMARAEVITIRAEVADRPRELQHQPR
VPRSASSSRAEKGLPSPHKRRREEWQQPAHAPSPSGTAKRAAVEEHVVKQAIMPPQVPRGR
AQQPRAPSVSGHTAQGGPPLPRRGPAAPSPAAALKAHLSPFQAPAAIPPKNFASILQSRYS
MTNAMAPTCNTSTTRPAGGAGAATAALGQGGAPTYSHALSRPNGAFLALPREERNRQQFLD
DFLSGGWVRFYSFTNEDTVVMYYSLQPGRYGAMFPTEAGVGTAVLDVYSKLVLYVPCMNNE
STNRSQPHPHVQTFYDEEARILSLPEAQRYLGGVLRCITGFVDEFSRLKAEGLTPAAVHAA
YIHHRSMSHVPRDTKFVYIRKVFPDPAGSFTLFRLSNLRSQVVCNAMVDIRWQSDRRHNVG
QKYYINADGTAEPFLVDQTGILSQLETVLNNNFRR
```

7.2.3.3.3 >KKT4

```
MSTDAQELVRQLTENPEVLESMQHMISLLRANPPRISGSNNGGGLGNAETNGPERGAPQCV
RPPRRGYGADVDCDHHQPTTRRKLRSSDGTAHSATSLSASSLTQEAHSFYGDDRVGARTTV
SDHNGTTGGASSPTPSFVSTGSRAAPQVVTAASRHAPRRSSLLPSPHEHRPTTAPDEQLMA
TANKLTEAQRRIAELEKELQRTTQRVDQLSDVVQRQKDELQAAKDRHALEMEETRHAYNAV
IHRKDEVQEEALRQLLKSRQLMVSAARYEAVVAAKKLHAQRLEKENNTGADDAMGSPKGLA
GVQASANPNERGTHPGLAPSQTSVNARHSSTLGYGSGTTAKYSSALKRDRQNDEGDLVDDA
GVETGAHEPGEARYGEAAHHHPPVKRTTLDTSRLQGSADRVVQGRRGVAATKAETSPAYIT
TPTPAGKASTALVGTRTQSSSARKRRTPRTPSRTNAERIAGSVAENRIRSQQRLPGTTSLK
IESPTPVVSTAWTADRSLTGSRTPPPSSAGVCTVSEAVTKHHQLYPQQQVHQVPSTRPPLM
QRAAGRLPPAPHRTAAASTAVPNTRSGTSSIASGGPTRSPSPVNPKRGAMLPRRFIFTGLK
DHEPQRLVSAIAAVGEDAAALASDLDEPPPSSTTHIVLRGTPRSVKALCGVVSGKWLVSPE
YVYNSQQSGFWLDELEEGGLRIFPPPLKCQRFLLTVEHPGIRAKLAQVIEYGGGEVLASGS
DKRGPGAGDTVAQDVVVITSGDDLLRYATQDRV
```

7.2.3.3.4 >KKT7

```
MTDVTSSLRPSSRQGSPVPRRQLGILPVNQRSYSRVGSKGMIGDDSPLMSPLPYYPRRRSV
TFAGDQSVREERPNYNAAYSASAPVSPARHGSPPPVSILKSNLSFPAAEEEDSGAAPAYQA
AAATVSGVLDRKDRARNSPVPVRGRSNSRQRLAARRKEAQLHRSFYDDSFVEEYVLRAKTE
LEEEEAEQRRMQEQLRAEQERAKRAERRVSEATEKINALQHAKEVLMAATVRRHTSVTPSP
QRAPAEKSKRNSSLLRELEEDPDPEVQAALKELARNSMAKQQSRVHSSAHQRRRSISIVSA
DALAKSGEDEDGDDNDTRKRARLEKIVSTLLAKKAKSKSKRSVMVIDWSDLDSDADGNTST
TDEDGEETAVGLKRQRGRPAKSRSIALGTEATLVSSAKHVQKPSTKRAASSRKRHVSAEPE
LGDSLLFEDEAEQPILLPRRQNTRPAPTRSISYIEMGGDDDLLRDASSVERVVRRPPRATR
APATRQRRGRLASTSTREGAEVMSSFTGTTALRGRASQPPAAPTGGPTGVPPRRRRGSVQR
ADPNDPMAVFFEAAFPSPSKFDEMMMQAGGLPETRRGGGGGGRGQGRHPNLVLPSSIGRRR
```

7.2.4 REPAIR TEMPLATE DESIGNS

From appendix 7.2.4.2 onwards, WT sequences and repair template designs for mutation are shown in the translated frame. Translations of the region are shown at the top. Target sites are highlighted in yellow. Black text indicates native sequence. Synonymously recoded regions are indicated in orange text.

7.2.4.1 *Codon Usage of Leishmania Infantum from https://www.kazusa.or.jp/codon/*

		Second Base																					
		U					C					A					G						
		Triplet code	Amino Acid	Fraction	Freq. per thousand	Number	Triplet code	Amino Acid	Fraction	Freq. per thousand	Number	Triplet code	Amino Acid	Fraction	Freq. per thousand	Number	Triplet code	Amino Acid	Fraction	Freq. per thousand	Number		
First Base	U	UUU	F	0.35	10.6	-52317	UCU	S	0.12	10.1	-49998	UAU	Y	0.17	4.1	-20192	UGU	C	0.21	4.0	-19923	U	Third Base
		UUC	F	0.65	19.3	-95738	UCC	S	0.19	16.4	-81198	UAC	Y	0.83	20.2	-100139	UGC	C	0.79	14.7	-72980	C	
		UUA	L	0.02	1.7	-8226	UCA	S	0.08	7.4	-36530	UAA	*	0.21	0.3	-1675	UGA	*	0.43	0.7	-3507	A	
		UUG	L	0.12	11.0	-54287	UCG	S	0.24	21.0	-104031	UAG	*	0.36	0.6	-2958	UGG	W	1.00	10.8	-53398	G	
	C	CUU	L	0.12	11.4	-56281	CCU	P	0.15	8.9	-44052	CAU	H	0.25	6.6	-32829	CGU	R	0.14	10.4	-51646	U	
		CUC	L	0.27	25.1	-124189	CCC	P	0.22	12.4	-61358	CAC	H	0.75	20.3	-100341	CGC	R	0.45	32.3	-159735	C	
		CUA	L	0.05	4.7	-23324	CCA	P	0.18	10.5	-51760	CAA	Q	0.19	7.7	-38242	CGA	R	0.10	7.5	-37057	A	
		CUG	L	0.41	37.7	-186757	CCG	P	0.45	25.8	-127867	CAG	Q	0.81	33.2	-164619	CGG	R	0.19	13.7	-67860	G	
	A	AUU	I	0.28	8.6	-42717	ACU	T	0.12	7.0	-34618	AAU	N	0.21	5.6	-27605	AGU	S	0.08	7.2	-35724	U	
		AUC	I	0.63	19.1	-94755	ACC	T	0.29	17.5	-86625	AAC	N	0.79	21.1	-104327	AGC	S	0.29	25.3	-125511	C	
		AUA	I	0.09	2.8	-13730	ACA	T	0.17	10.1	-49979	AAA	K	0.17	5.8	-28498	AGA	R	0.04	2.7	-13523	A	
		AUG	M	1.00	22.8	-113035	ACG	T	0.42	24.9	-123090	AAG	K	0.83	28.6	-141622	AGG	R	0.08	5.5	-27170	G	
	G	GUU	V	0.12	8.7	-42923	GCU	A	0.15	18.2	-90366	GAU	D	0.30	14.7	-73013	GGU	G	0.19	12.1	-59837	U	
		GUC	V	0.27	19.5	-96651	GCC	A	0.31	36.8	-182020	GAC	D	0.70	34.2	-169136	GGC	G	0.53	34.3	-170081	C	
		GUA	V	0.08	5.5	-27330	GCA	A	0.17	20.3	-100314	GAA	E	0.20	11.7	-58159	GGA	G	0.10	6.6	-32881	A	
		GUG	V	0.53	37.3	-184912	GCG	A	0.37	44.4	-220138	GAG	E	0.80	48.3	-239092	GGG	G	0.18	11.7	-58128	G	

7.2.4.2 Single-Stranded Repair Templates

Left hand side

Name																										
KKT1 S1449 Translation		L	P	E	E	E	E	V	N	V	V	D	G	E	H	P	I	R	A	V	S/A	G	P	P	D	D
KKT1 S1449 WT sequence		CTG	CCT	GAA	GAG	GAG	GAA	GTG	AAC	GTC	GTA	GAT	GGC	GAA	CAC	CCT	ATT	CGC	GCC	GTC	AGC	GGC	CCC	CCA	GAC	GAC
KKT1 S1449A		CTG	CCT	GAA	GAG	GAG	GAA	GTG	AAC	GTC	GTA	GAT	GGC	GAA	CAT	CCA	ATA	CGG	GCT	GTT	GCG	GGT	CCA	CCT	GAT	GAT
KKT1 S1449S		CTG	CCT	GAA	GAG	GAG	GAA	GTG	AAC	GTC	GTA	GAT	GGC	GAA	CAT	CCA	ATA	CGG	GCT	GTT	TCG	GGT	CCA	CCT	GAT	GAT
KKT2 M146 Translation	-	S	H	F	S	R	H	P	N	I	V	K	F	Y	G	A	G	R	D	E	D	R	A	Y	V	V
KKT2 M146 WT sequence	G	AGC	CAT	TTC	TCG	CGC	CAT	CCC	AAC	ATT	GTC	AAA	TTC	TAC	GGT	GCT	GGT	AGG	GAT	GAA	GAT	CGC	GCC	TAC	GTA	GTA
KKT2 M146G	G	AGC	CAT	TTC	TCG	CGC	CAT	CCC	AAC	ATT	GTC	AAG	TTT	TAT	GGA	GCG	GGC	CGC	GAC	GAG	GAC	CGA	GCG	TAT	GTG	GTG
KKT2 S505/S506 Translation		S	R	S	V	R	R	S	V	S	L	T	E	Q	E	R	G	R	L	V	R	S/A	S/A	P	V	Q
KKT2 S505/S506 WT sequence		TCT	CGA	TCA	GTG	CGT	CGT	AGC	GTC	AGC	TTA	ACG	GAG	CAG	GAG	CGG	GGC	AGA	CTT	GTG	CGT	TCT	AGC	CCG	GTC	CAG
KKT2 S505A		TCT	CGA	TCA	GTG	CGT	CGT	AGC	GTC	AGC	CTA	ACC	GAA	CAA	GAA	CGT	GGC	AGA	CTT	GTG	CGT	GCG	AGC	CCG	GTG	CAA
KKT2 S505S		TCT	CGA	TCA	GTG	CGT	CGT	AGC	GTC	AGC	CTA	ACC	GAA	CAA	GAA	CGT	GGC	AGA	CTT	GTG	CGT	TCG	AGC	CCG	GTG	CAA
KKT2 S506A		TCT	CGA	TCA	GTG	CGT	CGT	AGC	GTC	AGC	CTA	ACC	GAA	CAA	GAA	CGT	GGC	AGA	CTT	GTG	CGT	TCT	GCC	CCG	GTG	CAA
KKT2 S506S		TCT	CGA	TCA	GTG	CGT	CGT	AGC	GTC	AGC	CTA	ACC	GAA	CAA	GAA	CGT	GGC	AGA	CTT	GTG	CGT	TCT	AGT	CCG	GTG	CAA
KKT2 S505A+S506A Double		TCT	CGA	TCA	GTG	CGT	CGT	AGC	GTC	AGC	TTA	ACC	GAA	CAA	GAA	CGT	GGT	AGG	CTC	GTC	CGG	GCG	GCG	CCC	GTG	CAA
KKT2 S505S+S506S Double		TCT	CGA	TCA	GTG	CGT	CGT	AGC	GTC	AGC	TTA	ACC	GAA	CAA	GAA	CGT	GGT	AGG	CTC	GTC	CGG	AGC	TCG	CCC	GTG	CAA

KKT2 S493 Translation	V	S	E	V	A	D	R	E	E	A	A	P	R	T	S	R	S	V	R	R	S	V	S/A	L	T
KKT2 S493 WT sequence	GTC	TCA	GAG	GTT	GCA	GAT	CGC	GAG	GAA	GCC	GCC	CCT	CGC	ACG	TCT	CGA	TCA	GTG	CGT	CGT	AGC	GTC	AGC	TTA	ACG
KKT2 S493A	GTC	TCA	GAG	GTT	GCA	GAT	CGC	GAG	GAA	GCC	GCG	CCA	CGG	ACC	TCA	CGT	TCT	GTC	CGT	CGT	AGC	GTA	GCG	CTA	ACC
KKT2 S493S	GTC	TCA	GAG	GTT	GCA	GAT	CGC	GAG	GAA	GCC	GCG	CCA	CGG	ACC	TCA	CGT	TCT	GTC	CGT	CGT	AGC	GTA	TCG	CTA	ACC

KKT2 S530 Translation	V	V	Y	P	G	R	D	T	A	T	R	W	N	L	R	A	V	V	S/A	L	P	R	D	M	T
KKT2 S530 WT sequence	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAC	CTT	CGC	GCC	GTA	GTA	TCG	CTG	CCA	CGC	GAC	ATG	ACG
KKT2 S530A	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAT	TTG	CGG	GCG	GTT	GTA	GCG	CTC	CCT	CGG	GAT	ATG	ACC
KKT2 S530S	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAT	TTG	CGG	GCG	GTT	GTA	TCC	CTC	CCT	CGG	GAT	ATG	ACC

KKT2 S530 1 guide Translation	A	V	V	Y	P	G	R	D	T	A	T	R	W	N	L	R	A	V	V	S/A	L	P	R	D	M
KKT2 S530 1 guide WT sequence	GCA	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAC	CTT	CGC	GCC	GTA	GTA	TCG	CTG	CCA	CGC	GAC	ATG
KKT2 S530A 1 guide	GCA	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCG	ACA	CGG	TGG	AAT	TTG	CGG	GCG	GTT	GTA	GCG	CTC	CCT	CGG	GAT	ATG
KKT2 S530S 1 guide	GCA	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCG	ACA	CGG	TGG	AAT	TTG	CGG	GCG	GTT	GTA	TCC	CTC	CCT	CGG	GAT	ATG

KKT2 S923 Translation	K	Q	A	I	M	P	P	Q	V	P	R	G	R	A	Q	Q	P	R	A	P	S/A	V	S	G	H
KKT2 S923 WT sequence	AAG	CAA	GCC	ATC	ATG	CCG	CCT	CAG	GTG	CCA	CGC	GGA	CGA	GCA	CAG	CAG	CCA	CGT	GCC	CCC	TCC	GTC	AGC	GGG	CAC
KKT2 S923A	AAG	CAA	GCC	ATC	ATG	CCG	CCT	CAA	GTG	CCA	CGC	GGA	CGA	GCA	CAG	CAG	CCA	CGT	GCG	CCA	GCG	GTT	TCG	GGT	CAT
KKT2 S923S	AAG	CAA	GCC	ATC	ATG	CCG	CCT	CAA	GTG	CCA	CGC	GGA	CGA	GCA	CAG	CAG	CCA	CGT	GCG	CCA	TCG	GTT	TCG	GGT	CAT

KKT4 S300 Translation	A	A	K	K	L	H	A	Q	R	L	E	K	E	N	N	T	G	A	D	D	A	M	G	S/A	P
KKT4 S300 WT sequence	GCG	GCG	AAG	AAG	CTT	CAC	GCT	CAG	CGG	TTG	GAA	AAG	GAG	AAC	AAC	ACC	GGC	GCC	GAT	GAT	GCG	ATG	GGA	AGC	CCG
KKT4 S300A	GCG	GCG	AAG	AAG	CTT	CAC	GCT	CAG	CGG	TTG	GAG	AAA	GAA	AAT	AAT	ACG	GGC	GCC	GAT	GAT	GCG	ATG	GGA	GCG	CCC
KKT4 S300S	GCG	GCG	AAG	AAG	CTT	CAC	GCT	CAG	CGG	TTG	GAG	AAA	GAA	AAT	AAT	ACG	GGC	GCC	GAT	GAT	GCG	ATG	GGA	TCG	CCC
KKT4 S422 Translation	L	Q	G	S	A	D	R	V	V	Q	G	R	R	G	V	A	A	T	K	A	E	T	S/A	P	A
KKT4 S422 WT sequence	CTG	CAG	GGC	AGC	GCC	GAT	CGT	GTC	GTG	CAA	GGA	CGG	AGG	GGC	GTT	GCG	GCG	ACC	AAG	GCG	GAG	ACG	TCT	CCG	GCG
KKT4 S422A	CTG	CAG	GGC	AGC	GCC	GAT	CGT	GTC	GTC	CAG	GGG	CGT	CGT	GGC	GTT	GCG	GCG	ACC	AAG	GCG	GAG	ACG	GCG	CCG	GCC
KKT4 S422S	CTG	CAG	GGC	AGC	GCC	GAT	CGT	GTC	GTC	CAG	GGG	CGT	CGT	GGC	GTT	GCG	GCG	ACC	AAG	GCG	GAG	ACG	TCA	CCG	GCC
KKT7 S304 Translation	A	K	Q	Q	S	R	V	H	S	S	A	H	Q	R	R	R	S	I	S	I	V	S/A	A	D	A
KKT7 S304 WT sequence	GCG	AAG	CAA	CAG	AGT	CGC	GTT	CAC	TCT	TCT	GCC	CAT	CAG	CGT	CGT	CGG	TCG	ATA	TCC	ATT	GTC	TCC	GCC	GAC	GCC
KKT7 S304A	GCG	AAG	CAA	CAG	AGT	CGC	GTT	CAC	TCT	TCT	GCG	CAC	CAA	CGG	CGG	CGT	AGC	ATT	AGC	ATT	GTC	GCG	GCG	GAT	GCG
KKT7 S304S	GCG	AAG	CAA	CAG	AGT	CGC	GTT	CAC	TCT	TCT	GCG	CAC	CAA	CGG	CGG	CGT	AGC	ATT	AGC	ATT	GTC	TCG	GCG	GAT	GCG

Right hand side

															Name
S	D	L	A	G	R	A	G	D	D	E	A	T	K	R	KKT1 S1449 Translation
AGC	GAC	CTT	GCT	GGA	CGC	GCA	GGT	GAC	GAC	GAA	GCG	ACT	AAG	CGC	KKT1 S1449 WT sequence
TCG	GAT	CTA	GCT	GGA	CGC	GCA	GGT	GAC	GAC	GAA	GCG	ACT	AAG	CGC	KKT1 S1449A
TCG	GAT	CTA	GCT	GGA	CGC	GCA	GGT	GAC	GAC	GAA	GCG	ACT	AAG	CGC	KKT1 S1449S
M/G	E	R	C	A	G	K	S	L	H	D	V	I	A	-	KKT2 M146 Translation
ATG	GAG	CGG	TGC	GCA	GGC	AAG	TCG	CTT	CAC	GAC	GTC	ATA	GCC	AG	KKT2 M146 WT sequence
GCG	GAA	CGT	TGT	GCA	GGC	AAG	TCG	CTT	CAC	GAC	GTC	ATA	GCC	AG	KKT2 M146G
Y	A	V	V	Y	P	G	R	D	T	A	T	R	W	N	KKT2 S505/S506 Translation
TAC	GCA	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAC	KKT2 S505/S506 WT sequence
TAT	GCT	GTC	GTC	TAC	CCA	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAC	KKT2 S505A
TAT	GCT	GTC	GTC	TAC	CCA	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAC	KKT2 S505S
TAT	GCT	GTC	GTC	TAC	CCA	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAC	KKT2 S506A
TAT	GCT	GTC	GTC	TAC	CCA	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAC	KKT2 S506S
TAT	GCC	GTC	GTC	TAT	CCG	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAC	KKT2 S505A+S506A Double
TAT	GCC	GTC	GTC	TAT	CCG	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAC	KKT2 S505S+S506S Double

E	Q	E	R	G	R	L	V	R	S	S	P	V	Q	Y	KKT2 S493 Translation
GAG	CAG	GAG	CGG	GGC	AGA	CTT	GTG	CGT	TCT	AGC	CCG	GTC	CAG	TAC	KKT2 S493 WT sequence
GAA	CAA	GAA	CGT	GGC	AGA	CTT	GTG	CGT	TCT	AGC	CCG	GTC	CAG	TAC	KKT2 S493A
GAA	CAA	GAA	CGT	GGC	AGA	CTT	GTG	CGT	TCT	AGC	CCG	GTC	CAG	TAC	KKT2 S493S

D	E	I	E	R	E	F	K	C	M	N	G	H	V	M	KKT2 S530 Translation
GAC	GAG	ATC	GAG	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	ATG	KKT2 S530 WT sequence
GAT	GAA	ATC	GAG	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	ATG	KKT2 S530A
GAT	GAA	ATC	GAG	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	ATG	KKT2 S530S

T	D	E	I	E	R	E	F	K	C	M	N	G	H	V	KKT2 S530 1 guide Translation
ACG	GAC	GAG	ATC	GAG	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	KKT2 S530 1 guide WT sequence
ACC	GAT	GAA	ATT	GAA	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	KKT2 S530A 1 guide
ACC	GAT	GAA	ATT	GAA	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	KKT2 S530S 1 guide

T	A	Q	G	G	P	P	L	P	R	R	G	P	A	A	KKT2 S923 Translation
ACC	GCA	CAG	GGC	GGT	CCG	CCA	CTG	CCG	CGC	CGC	GGC	CCA	GCT	GCG	KKT2 S923 WT sequence
ACG	GCT	CAA	GGT	GGT	CCG	CCA	CTG	CCG	CGC	CGC	GGC	CCA	GCT	GCG	KKT2 S923A
ACG	GCT	CAA	GGT	GGT	CCG	CCA	CTG	CCG	CGC	CGC	GGC	CCA	GCT	GCG	KKT2 S923S

K	G	L	A	G	V	Q	A	S	A	N	P	N	E	R	KKT4 S300 Translation
AAG	GGG	CTA	GCA	GGC	GTA	CAG	GCA	AGC	GCG	AAC	CCC	AAC	GAG	CGC	KKT4 S300 WT sequence
AAA	GGT	CTT	GCT	GGG	GTA	CAG	GCA	AGC	GCG	AAC	CCC	AAC	GAG	CGC	KKT4 S300A
AAA	GGT	CTT	GCT	GGG	GTA	CAG	GCA	AGC	GCG	AAC	CCC	AAC	GAG	CGC	KKT4 S300S
Y	I	T	T	P	T	P	A	G	K	A	S	T	A	L	KKT4 S422 Translation
TAC	ATC	ACC	ACC	CCC	ACG	CCG	GCC	GGC	AAG	GCG	TCC	ACC	GCG	CTC	KKT4 S422 WT sequence
TAT	ATT	ACG	ACG	CCC	ACG	CCC	GCC	GGC	AAG	GCG	TCC	ACC	GCG	CTC	KKT4 S422A
TAT	ATT	ACG	ACG	CCC	ACG	CCC	GCC	GGC	AAG	GCG	TCC	ACC	GCG	CTC	KKT4 S422S
L	A	K	S	G	E	D	E	D	G	D	D	N	D	T	KKT7 S304 Translation
CTC	GCG	AAG	AGC	GGC	GAG	GAC	GAA	GAC	GGT	GAC	GAC	AAC	GAC	ACC	KKT7 S304 WT sequence
CTT	GCC	AAA	TCG	GGC	GAG	GAC	GAA	GAC	GGT	GAC	GAC	AAC	GAC	ACC	KKT7 S304A
CTT	GCC	AAA	TCG	GGC	GAG	GAC	GAA	GAC	GGT	GAC	GAC	AAC	GAC	ACC	KKT7 S304S

7.2.4.3 Pooled Single-Stranded Repair Template Designs - Left hand side.

Name																														
KKT2 S493 Translation	V	S	E	V	A	D	R	E	E	A	A	P	R	T	S	R	S	V	R	R	S	V	S/A	L	T	E	Q	E	R	Y
KKT2 S493 WT sequence	GTC	TCA	GAG	GTT	GCA	GAT	CGC	GAG	GAA	GCC	GCC	CCT	CGC	ACG	TCT	CGA	TCA	GTG	CGT	CGT	AGC	GTC	AGC	TTA	ACG	GAG	CAG	GAG	CGG	TAC
KKT2 S493A	GTC	TCA	GAG	GTT	GCA	GAT	CGC	GAG	GAA	GCC	GCG	CCA	CGG	ACC	TCA	CGT	TCT	GTC	CGT	CGT	AGC	GTA	GCG	CTA	ACC	GAA	CAA	GAA	CGT	TAC
KKT2 S493S design 1	GTC	TCA	GAG	GTT	GCA	GAT	CGC	GAG	GAA	GCC	GCG	CCA	CGG	ACC	TCA	CGT	TCT	GTC	CGT	CGT	AGC	GTA	TCG	CTA	ACC	GAA	CAA	GAA	CGT	TAC
KKT2 S493A design 2	GTC	TCA	GAG	GTT	GCA	GAT	CGC	GAG	GAA	GCC	GCG	CCT	CGA	ACG	TCT	CGA	TCA	GTG	CGT	CGT	AGC	GTA	GCG	CTC	ACG	GAG	CAG	GAG	CGT	TAC
KKT2 S493S design 2	GTC	TCA	GAG	GTT	GCA	GAT	CGC	GAG	GAA	GCC	GCG	CCT	CGA	ACG	TCT	CGA	TCA	GTG	CGT	CGT	AGC	GTA	TCG	CTC	ACG	GAG	CAG	GAG	CGT	TAC
KKT2 S493A design 3	GTC	TCA	GAG	GTT	GCA	GAT	CGC	GAG	GAA	GCC	GCG	CCA	CGC	ACG	TCT	CGA	TCA	GTG	CGT	CGT	TCG	GTA	GCG	CTA	ACG	GAG	CAA	GAA	CGT	TAC
KKT2 S493S design 3	GTC	TCA	GAG	GTT	GCA	GAT	CGC	GAG	GAA	GCC	GCG	CCA	CGC	ACG	TCT	CGA	TCA	GTG	CGT	CGT	TCG	GTA	AGT	CTA	ACG	GAG	CAA	GAA	CGT	TAC
KKT2 S493A design 4	GTC	TCA	GAG	GTT	GCA	GAT	CGC	GAG	GAA	GCC	GCG	CCT	AGG	ACG	AGT	CGA	AGC	GTG	AGG	CGT	AGC	GTA	GCG	CTC	ACG	GAG	CAA	GAG	AGA	TAC
KKT2 S493S design 4	GTC	TCA	GAG	GTT	GCA	GAT	CGC	GAG	GAA	GCC	GCG	CCT	AGG	ACG	AGT	CGA	AGC	GTG	AGG	CGT	AGC	GTA	TCC	CTC	ACG	GAG	CAA	GAG	AGA	TAC
KKT2 S493A design 5	GTC	TCA	GAG	GTT	GCA	GAT	CGC	GAG	GAA	GCC	GCA	CCC	CGT	ACA	TCT	CGA	TCA	GTG	CGT	CGT	TCC	GTA	GCG	CTC	ACA	GAG	CAG	GAG	AGG	TAC
KKT2 S493S design 5	GTC	TCA	GAG	GTT	GCA	GAT	CGC	GAG	GAA	GCC	GCA	CCC	CGT	ACA	TCT	CGA	TCA	GTG	CGT	CGT	TCC	GTA	TCT	CTC	ACA	GAG	CAG	GAG	AGG	TAC
KKT2 S530 1 guide Translation	A	V	V	Y	P	G	R	D	T	A	T	R	W	N	L	R	A	V	V	S/A	L	P	R	D	M	T	D	E	I	V
KKT2 S530 WT sequence	GCA	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAC	CTT	CGC	GCC	GTA	GTA	TCG	CTG	CCA	CGC	GAC	ATG	ACG	GAC	GAG	ATC	GTA
KKT2 S530A design 1	GCA	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCG	ACA	CGG	TGG	AAT	TTG	CGG	GCG	GTT	GTA	GCG	CTC	CCT	CGG	GAT	ATG	ACC	GAT	GAA	ATT	GTA
KKT2 S530S design 1	GCA	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCG	ACA	CGG	TGG	AAT	TTG	CGG	GCG	GTT	GTA	TCC	CTC	CCT	CGG	GAT	ATG	ACC	GAT	GAA	ATT	GTA
KKT2 S530A design 2	GCA	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAT	TTG	CGA	GCC	GTA	GTA	GCG	CTT	CCA	CGT	GAC	ATG	ACG	GAC	GAG	ATC	GTA
KKT2 S530S design 2	GCA	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAT	TTG	CGA	GCC	GTA	GTA	TCA	CTT	CCA	CGT	GAC	ATG	ACG	GAC	GAG	ATC	GTA
KKT2 S530A design 3	GCA	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAT	TTG	CGG	GCG	GTA	GTG	GCG	CTC	CCA	CGA	GAT	ATG	ACG	GAC	GAG	ATC	GTA
KKT2 S530S design 3	GCA	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAT	TTG	CGG	GCG	GTA	GTG	TCA	CTC	CCA	CGA	GAT	ATG	ACG	GAC	GAG	ATC	GTA
KKT2 S530A design 4	GCA	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAT	TTG	AGA	GCC	GTA	GTT	GCG	TTA	CCA	AGG	GAT	ATG	ACG	GAC	GAG	ATC	GTA
KKT2 S530S design 4	GCA	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAT	TTG	AGA	GCC	GTA	GTT	AGT	TTA	CCA	AGG	GAT	ATG	ACG	GAC	GAG	ATC	GTA
KKT2 S530A design 5	GCA	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAT	TTA	AGG	GCT	GTC	GTA	GCG	TTG	CCT	CGT	GAC	ATG	ACA	GAC	GAG	ATA	GTA
KKT2 S530S design 5	GCA	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAT	TTA	AGG	GCT	GTC	GTA	TCT	TTG	CCT	CGT	GAC	ATG	ACA	GAC	GAG	ATA	GTA

Right hand side.

											Name
G	R	L	V	R	S	S	P	V	Q	Y	KKT2 S493 Translation
GGC	AGA	CTT	GTG	CGT	TCT	AGC	CCG	GTC	CAG	TAC	KKT2 S493 WT sequence
GGC	AGA	CTT	GTG	CGT	TCT	AGC	CCG	GTC	CAG	TAC	KKT2 S493A
GGC	AGA	CTT	GTG	CGT	TCT	AGC	CCG	GTC	CAG	TAC	KKT2 S493S design 1
GGC	AGA	CTT	GTG	CGT	TCT	AGC	CCG	GTC	CAG	TAC	KKT2 S493A design 2
GGC	AGA	CTT	GTG	CGT	TCT	AGC	CCG	GTC	CAG	TAC	KKT2 S493S design 2
GGC	AGA	CTT	GTG	CGT	TCT	AGC	CCG	GTC	CAG	TAC	KKT2 S493A design 3
GGC	AGA	CTT	GTG	CGT	TCT	AGC	CCG	GTC	CAG	TAC	KKT2 S493S design 3
GGC	AGA	CTT	GTG	CGT	TCT	AGC	CCG	GTC	CAG	TAC	KKT2 S493A design 4
GGC	AGA	CTT	GTG	CGT	TCT	AGC	CCG	GTC	CAG	TAC	KKT2 S493S design 4
GGC	AGA	CTT	GTG	CGT	TCT	AGC	CCG	GTC	CAG	TAC	KKT2 S493A design 5
GGC	AGA	CTT	GTG	CGT	TCT	AGC	CCG	GTC	CAG	TAC	KKT2 S493S design 5
E	R	E	F	K	C	M	N	G	H	V	KKT2 S530 1 guide Translation
GAG	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	KKT2 S530 WT sequence
GAA	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	KKT2 S530A design 1
GAA	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	KKT2 S530S design 1
GAG	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	KKT2 S530A design 2
GAG	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	KKT2 S530S design 2
GAG	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	KKT2 S530A design 3
GAG	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	KKT2 S530S design 3
GAG	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	KKT2 S530A design 4
GAG	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	KKT2 S530S design 4
GAG	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	KKT2 S530A design 5
GAG	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	KKT2 S530S design 5

7.2.4.3.1 Pooled single-stranded repair template sgRNA and repair template primer IDs.

	S493A	**S493S**	**S530A**	**S530S**
sgRNA Primers	OL12987 OL12988	OL12987 OL12988	OL12902 OL12903	OL12902 OL12903
Repair Templates	OL12999 OL13651 OL13653 OL13655 OL13657	OL13000 OL13652 OL13654 OL13656 OL13658	OL13369 OL13659 OL13661 OL13663 OL13665	OL13370 OL13660 OL13662 OL13664 OL13666

7.2.4.4 Double-Stranded Repair Template Designs

Plus strand sequence only. Left hand side.

Name																										
KKT2 S25 Translation	-	P	L	M	S	H	F	C	G	S	L	S	R	T	P	P	R	G	G	A	I	S	M	P	R	D
KKT2 S25 WT sequence	GG	CCT	CTG	ATG	TCA	CAC	TTT	TGC	GGC	TCG	TTG	TCG	AGG	ACT	CCA	CCA	CGG	GGT	GGG	GCG	ATA	TCT	ATG	CCG	CGT	GAT
KKT2 S25A	GC	CCT	CTG	ATG	TCA	CAC	TTT	TGC	GGC	TCG	TTG	TCG	AGG	ACT	CCA	CCA	CGG	GGT	GGG	GCG	ATA	TCT	ATG	CCC	AGA	GAC
KKT2 S25E	CC	CCT	CTG	ATG	TCA	CAC	TTT	TGC	GGC	TCG	TTG	TCG	AGG	ACT	CCA	CCA	CGG	GGT	GGG	GCG	ATA	TCT	ATG	CCC	AGA	GAC
KKT2 S25S	GG	CCT	CTG	ATG	TCA	CAC	TTT	TGC	GGC	TCG	TTG	TCG	AGG	ACT	CCA	CCA	CGG	GGT	GGG	GCG	ATA	TCT	ATG	CCC	AGA	GAC
KKT2 S493 Translation	-	C	G	S	V	S	L	V	S	E	V	A	D	R	E	E	A	A	P	R	T	S	R	S	V	R
KKT2 S493 WT sequence	GG	TGT	GGC	AGT	GTC	TCA	CTG	GTC	TCA	GAG	GTT	GCA	GAT	CGC	GAG	GAA	GCC	GCC	CCT	CGC	ACG	TCT	CGA	TCA	GTG	CGT
KKT2 S493A	GG	TGT	GGC	AGT	GTC	TCA	CTG	GTC	TCA	GAG	GTT	GCA	GAT	CGC	GAG	GAA	GCC	GCG	CCA	CGG	ACC	TCA	CGT	TCT	GTC	CGT
KKT2 S493E	GG	TGT	GGC	AGT	GTC	TCA	CTG	GTC	TCA	GAG	GTT	GCA	GAT	CGC	GAG	GAA	GCC	GCG	CCA	CGG	ACC	TCA	CGT	TCT	GTC	CGT
KKT2 S493S	GG	TGT	GGC	AGT	GTC	TCA	CTG	GTC	TCA	GAG	GTT	GCA	GAT	CGC	GAG	GAA	GCC	GCG	CCA	CGG	ACC	TCA	CGT	TCT	GTC	CGT
KKT2 S530 Translation		R	S	S	P	V	Q	Y	A	V	V	Y	P	G	R	D	T	A	T	R	W	N	L	R	A	V
KKT2 S530 WT sequence		CGT	TCT	AGC	CCG	GTC	CAG	TAC	GCA	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCC	ACT	CGT	TGG	AAC	CTT	CGC	GCC	GTA
KKT2 S530A		CGT	TCT	AGC	CCG	GTC	CAG	TAC	GCA	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCG	ACA	CGG	TGG	AAT	TTG	CGG	GCG	GTT
KKT2 S530E		CGT	TCT	AGC	CCG	GTC	CAG	TAC	GCA	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCG	ACA	CGG	TGG	AAT	TTG	CGG	GCG	GTT
KKT2 S530S		CGT	TCT	AGC	CCG	GTC	CAG	TAC	GCA	GTG	GTG	TAC	CCG	GGG	CGC	GAC	ACT	GCG	ACA	CGG	TGG	AAT	TTG	CGG	GCG	GTT
KKT2 S923 Translation	-	V	E	E	H	V	V	K	Q	A	I	M	P	P	Q	V	P	R	G	R	A	Q	Q	P	R	A
KKT2 S923 WT sequence	CT	GTG	GAG	GAG	CAC	GTG	GTG	AAG	CAA	GCC	ATC	ATG	CCG	CCT	CAG	GTG	CCA	CGC	GGA	CGA	GCA	CAG	CAG	CCA	CGT	GCC
KKT2 S923A	CT	GTG	GAG	GAG	CAC	GTG	GTG	AAG	CAA	GCC	ATC	ATG	CCG	CCT	CAA	GTG	CCA	CGC	GGA	CGA	GCA	CAG	CAG	CCA	CGT	GCG
KKT2 S923E	CT	GTG	GAG	GAG	CAC	GTG	GTG	AAG	CAA	GCC	ATC	ATG	CCG	CCT	CAA	GTG	CCA	CGC	GGA	CGA	GCA	CAG	CAG	CCA	CGT	GCG
KKT2 S923S	CT	GTG	GAG	GAG	CAC	GTG	GTG	AAG	CAA	GCC	ATC	ATG	CCG	CCT	CAA	GTG	CCA	CGC	GGA	CGA	GCA	CAG	CAG	CCA	CGT	GCG

KKT4 S422 Translation	-	T	T	L	D	T	S	R	L	Q	G	S	A	D	R	V	V	Q	G	R	R	G	V	A	A	T
KKT4 S422 WT Sequence	AC	ACC	ACG	TTG	GAC	ACG	TCT	CGT	CTG	CAG	GGC	AGC	GCC	GAT	CGT	GTC	GTG	CAA	GGA	CGG	AGG	GGC	GTT	GCG	GCG	ACC
KKT4 S422A	AC	ACC	ACG	TTG	GAC	ACG	TCT	CGT	CTG	CAG	GGC	AGC	GCC	GAT	CGT	GTC	GTG	CAG	GGT	CGC	CGT	GGT	GTG	GCC	GCG	ACC
KKT4 S422E	AC	ACC	ACG	TTG	GAC	ACG	TCT	CGT	CTG	CAG	GGC	AGC	GCC	GAT	CGT	GTC	GTG	CAG	GGT	CGC	CGT	GGT	GTG	GCC	GCG	ACC
KKT4 S422S	AC	ACC	ACG	TTG	GAC	ACG	TCT	CGT	CTG	CAG	GGC	AGC	GCC	GAT	CGT	GTC	GTG	CAG	GGT	CGC	CGT	GGT	GTG	GCC	GCG	ACC

KKT7 S304 Translation	-	L	A	R	N	S	M	A	K	Q	Q	S	R	V	H	S	S	A	H	Q	R	R	R	S	I	S
KKT7 S304 WT Sequence	CT	CTC	GCA	CGC	AAC	TCC	ATG	GCG	AAG	CAA	CAG	AGT	CGC	GTT	CAC	TCT	TCT	GCC	CAT	CAG	CGT	CGT	CGG	TCG	ATA	TCC
KKT7 S304A	CT	CTC	GCA	CGC	AAC	TCC	ATG	GCG	AAG	CAA	CAG	AGT	CGC	GTT	CAC	TCT	TCT	GCT	CAC	CAA	CGC	CGA	CGC	TCC	ATC	TCC
KKT7 S304E	CT	CTC	GCA	CGC	AAC	TCC	ATG	GCG	AAG	CAA	CAG	AGT	CGC	GTT	CAC	TCT	TCT	GCT	CAC	CAA	CGC	CGA	CGC	TCC	ATC	TCC
KKT7 S304S	CT	CTC	GCA	CGC	AAC	TCC	ATG	GCG	AAG	CAA	CAG	AGT	CGC	GTT	CAC	TCT	TCT	GCT	CAC	CAA	CGC	CGA	CGC	TCC	ATC	TCC

Plus strand sequence only. Right hand side

																													Name
L	S/A /E	Q	T	P	A	I	S	R	L	G	S	T	V	K	T	P	H	I	Q	K	C	V	V	D	Q	A	-		KKT2 S25 Translation
TTG	TCG	CAG	ACC	CCC	GCC	ATC	TCT	CGA	CTT	GGA	AGC	ACG	GTG	AAG	ACG	CCG	CAC	ATC	CAA	AAA	TGT	GTT	GTT	GAC	CAA	GCA	GA		KKT2 S25 WT sequence
CTT	GCG	CAA	ACA	CCA	GCG	ATT	TCA	CGC	CTG	GGG	AGC	ACG	GTG	AAG	ACG	CCG	CAC	ATC	CAA	AAA	TGT	GTT	GTT	GAC	CAA	GCA	GA		KKT2 S25A
CTT	GAG	CAA	ACA	CCA	GCG	ATT	TCA	CGC	CTG	GGG	AGC	ACG	GTG	AAG	ACG	CCG	CAC	ATC	CAA	AAA	TGT	GTT	GTT	GAC	CAA	GCA	GA		KKT2 S25E
CTT	AGT	CAA	ACA	CCA	GCG	ATT	TCA	CGC	CTG	GGG	AGC	ACG	GTG	AAG	ACG	CCG	CAC	ATC	CAA	AAA	TGT	GTT	GTT	GAC	CAA	GCA	GA		KKT2 S25S
R	S	V	S/A /E	L	T	E	Q	E	R	G	R	L	V	R	S	S	P	V	Q	Y	A	V	V	Y	P	G	-		KKT2 S493 Translation
CGT	AGC	GTC	AGC	TTA	ACG	GAG	CAG	GAG	CGG	GGC	AGA	CTT	GTG	CGT	TCT	AGC	CCG	GTC	CAG	TAC	GCA	GTG	GTG	TAC	CCG	GGG	CG		KKT2 S493 WT sequence
CGT	AGC	GTA	GCG	CTA	ACC	GAA	CAA	GAA	CGT	GGC	AGA	CTT	GTG	CGT	TCT	AGC	CCG	GTC	CAG	TAC	GCA	GTG	GTG	TAC	CCG	GGG	CG		KKT2 S493A
CGT	AGC	GTA	GAG	CTA	ACC	GAA	CAA	GAA	CGT	GGC	AGA	CTT	GTG	CGT	TCT	AGC	CCG	GTC	CAG	TAC	GCA	GTG	GTG	TAC	CCG	GGG	CG		KKT2 S493E
CGT	AGC	GTA	TCT	CTA	ACC	GAA	CAA	GAA	CGT	GGC	AGA	CTT	GTG	CGT	TCT	AGC	CCG	GTC	CAG	TAC	GCA	GTG	GTG	TAC	CCG	GGG	CG		KKT2 S493S
V	S/A /E	L	P	R	D	M	T	D	E	I	E	R	E	F	K	C	M	N	G	H	V	M	T	K	L	T	S	-	KKT2 S530 Translation
GTA	TCG	CTG	CCA	CGC	GAC	ATG	ACG	GAC	GAG	ATC	GAG	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	ATG	ACA	AAG	TTG	ACC	TCG	A	KKT2 S530 WT sequence
GTA	GCG	CTC	CCT	CGG	GAT	ATG	ACC	GAT	GAA	ATT	GAA	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	ATG	ACA	AAG	TTG	ACC	TCG	A	KKT2 S530A
GTA	GAG	CTC	CCT	CGG	GAT	ATG	ACC	GAT	GAA	ATT	GAA	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	ATG	ACA	AAG	TTG	ACC	TCG	A	KKT2 S530E
GTA	AGT	CTC	CCT	CGG	GAT	ATG	ACC	GAT	GAA	ATT	GAA	CGC	GAG	TTC	AAG	TGC	ATG	AAC	GGG	CAC	GTA	ATG	ACA	AAG	TTG	ACC	TCG	A	KKT2 S530S
P	S/A /E	V	S	G	H	T	A	Q	G	G	P	P	L	P	R	R	G	P	A	A	P	S	P	A	A	A	-		KKT2 S923 Translation
CCC	TCC	GTC	AGC	GGG	CAC	ACC	GCA	CAG	GGC	GGT	CCG	CCA	CTG	CCG	CGC	CGC	GGC	CCA	GCT	GCG	CCA	TCT	CCT	GCA	GCC	GCT	TT		KKT2 S923 WT sequence
CCA	GCG	GTT	TCG	GGT	CAT	ACG	GCT	CAA	GGT	GGT	CCG	CCA	CTG	CCG	CGC	CGC	GGC	CCA	GCT	GCG	CCA	TCT	CCT	GCA	GCC	GCT	TT		KKT2 S923A
CCA	GAG	GTT	TCG	GGT	CAT	ACG	GCT	CAA	GGT	GGT	CCG	CCA	CTG	CCG	CGC	CGC	GGC	CCA	GCT	GCG	CCA	TCT	CCT	GCA	GCC	GCT	TT		KKT2 S923E
CCA	AGT	GTT	TCG	GGT	CAT	ACG	GCT	CAA	GGT	GGT	CCG	CCA	CTG	CCG	CGC	CGC	GGC	CCA	GCT	GCG	CCA	TCT	CCT	GCA	GCC	GCT	TT		KKT2 S923S

K	A	E	T	S/A /E	P	A	Y	I	T	T	P	T	P	A	G	K	A	S	T	A	L	V	G	T	R	T	-	KKT4 S422 Translation
AAG	GCG	GAG	ACG	TCT	CCG	GCG	TAC	ATC	ACC	ACC	CCC	ACG	CCG	GCC	GGC	AAG	GCG	TCC	ACC	GCG	CTC	GTC	GGC	ACG	CGC	ACT	CA	KKT4 S422 WT Sequence
AAG	GCG	GAG	ACG	GCG	CCC	GCC	TAT	ATT	ACG	ACA	CCC	ACG	CCG	GCC	GGC	AAG	GCG	TCC	ACC	GCG	CTC	GTC	GGC	ACG	CGC	ACT	CA	KKT4 S422A
AAG	GCG	GAG	ACG	GAG	CCC	GCC	TAT	ATT	ACG	ACA	CCC	ACG	CCG	GCC	GGC	AAG	GCG	TCC	ACC	GCG	CTC	GTC	GGC	ACG	CGC	ACT	CA	KKT4 S422E
AAG	GCG	GAG	ACG	AGC	CCC	GCC	TAT	ATT	ACG	ACA	CCC	ACG	CCG	GCC	GGC	AAG	GCG	TCC	ACC	GCG	CTC	GTC	GGC	ACG	CGC	ACT	CA	KKT4 S422S

I	V	S/A /E	A	D	A	L	A	K	S	G	E	D	E	D	G	D	D	N	D	T	R	K	R	A	R	L	-	KKT7 S304 Translation
ATT	GTC	TCC	GCC	GAC	GCC	CTC	GCG	AAG	AGC	GGC	GAG	GAC	GAA	GAC	GGT	GAC	GAC	AAC	GAC	ACC	CGC	AAG	CGC	GCG	CGT	CTA	GA	KKT7 S304 WT Sequence
ATT	GTC	GCA	GCG	GAT	GCA	CTG	GCC	AAA	TCG	GGC	GAG	GAC	GAA	GAC	GGT	GAC	GAC	AAC	GAC	ACC	CGC	AAG	CGC	GCG	CGT	CTA	GA	KKT7 S304A
ATT	GTC	GAA	GCG	GAT	GCA	CTG	GCC	AAA	TCG	GGC	GAG	GAC	GAA	GAC	GGT	GAC	GAC	AAC	GAC	ACC	CGC	AAG	CGC	GCG	CGT	CTA	GA	KKT7 S304E
ATT	GTC	TCA	GCG	GAT	GCA	CTG	GCC	AAA	TCG	GGC	GAG	GAC	GAA	GAC	GGT	GAC	GAC	AAC	GAC	ACC	CGC	AAG	CGC	GCG	CGT	CTA	GA	KKT7 S304S

7.2.5 PRIMERS

All primer sequences are given in the 5’ to 3’ orientation.

7.2.5.1 sgRNA Primers

Capital letters indicate the protospacer recognition sequence. Note: OL12825’s protospacer sequence was later identified to be incorrect (copied from the repair template not the WT sequence), but not before ssDNA repair transfections were completed using it. It was replaced with OL14600 for dsDNA transfections.

Name	Target	Sequence	Description
OL6137	-	aaaagcaccgactcggtgccactttttcaagttgataacggactagccttattttaacttgctatttctagctctaaaac	Universal sgRNA Primer (G00)
OL12985	KKT1	gaaattaatacgactcactataggGCAAGGTCGCTGTCGTCTGGgttttagagctagaaatagc	S1449 Guide 1
OL12986	KKT1	gaaattaatacgactcactataggCACCCTATTCGCGCCGTCAGgttttagagctagaaatagc	S1449 Guide 2
OL12632	KKT2	gaaattaatacgactcactataggGGGGTCTGCGACAAATCACGgttttagagctagaaatagc	S25 Guide 1
OL14011	KKT2	gaaattaatacgactcactataggACCCCCGCCATCTCTCGACTgttttagagctagaaatagc	S25 Guide 2
OL12987	KKT2	gaaattaatacgactcactataggGCACTGATCGAGACGTGCGAgttttagagctagaaatagc	S493 Guide 1
OL12988	KKT2	gaaattaatacgactcactataggAGCTTAACGGAGCAGGAGCGgttttagagctagaaatagc	S493 Guide 2
OL12775	KKT2	gaaattaatacgactcactataggCCGGGTACACCACTGCGTACgttttagagctagaaatagc	S505/S506 Guide 1
OL12898	KKT2	gaaattaatacgactcactataggAGCTTAACGGAGCAGGAGCGgttttagagctagaaatagc	S506A Guide 2
OL12902	KKT2	gaaattaatacgactcactataggATCTCGTCCGTCATGTCGCGgttttagagctagaaatagc	S530 Guide 1
OL12903	KKT2	gaaattaatacgactcactataggGCAGCGATACTACGGCGCGAgttttagagctagaaatagc	S530 Guide 2
OL12778	KKT2	gaaattaatacgactcactataggGGTGTGCCCGCTGACGGAGGgttttagagctagaaatagc	S923 Guide 1
OL12899	KKT2	gaaattaatacgactcactataggAGCGGGCACACCGCACAGGGgttttagagctagaaatagc	S923 Guide 2
OL12989	KKT4	gaaattaatacgactcactataggGTACGCCTGCTAGCCCCTTCgttttagagctagaaatagc	S300 Guide 1
OL12990	KKT4	gaaattaatacgactcactataggTTGGAAAAGGAGAACAACACgttttagagctagaaatagc	S300 Guide 2
OL12764	KKT4	gaaattaatacgactcactataggATCGTGTCGTGCAAGGACGGgttttagagctagaaatagc	S422 Guide 1

OL12900	KKT4	gaaattaatacgactcactataggGTACATCACCACCCCCACGCgttttagagctagaaatagc	S422 Guide 2
OL12825	KKT7	gaaattaatacgactcactataggGCTCTTCGCGAGGGCATCCGCCGgttttagagctagaaatagc	S304 Guide 1
OL12901	KKT7	gaaattaatacgactcactataggATATCGACCGACGACGCTGAgttttagagctagaaatagc	S304 Guide 2
OL14600	KKT7	gaaattaatacgactcactataggGCTCTTCGCGAGGGCGTCGGgttttagagctagaaatagc	S304 Guide 3 (replacement for guide 1)

7.2.5.2 Single-Stranded Repair Primers

Name	Target	Sequence	Description
OL12997	KKT1	GCGCTTAGTCGCTTCGTCGTCACCTGCGCGTCCAGCTAGATCCGAATCATCAGGTGGACCCGCAACAGCCCGTATTGGATGTTCGCCATCTACGACGTTCACTTCCTCCTCTTCAGGCAG	S1449A
OL12998	KKT1	GCGCTTAGTCGCTTCGTCGTCACCTGCGCGTCCAGCTAGATCCGAATCATCAGGTGGACCCGAAACAGCCCGTATTGGATGTTCGCCATCTACGACGTTCACTTCCTCCTCTTCAGGCAG	S1449S
OL12999	KKT2	GTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCGCCACGGACCTCACGTTCTGTCCGTCGTAGCGTAGCGCTAACCGAACAAGAACGTGGCAGACTTGTGCGTTCTAGCCCGGTCCAGTAC	S493A
OL13000	KKT2	GTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCGCCACGGACCTCACGTTCTGTCCGTCGTAGCGTATCGCTAACCGAACAAGAACGTGGCAGACTTGTGCGTTCTAGCCCGGTCCAGTAC	S493S
OL13001	KKT2	TCTCGATCAGTGCGTCGTAGCGTCAGCCTAACCGAACAAGAACGTGGCAGACTTGTGCGTGCGAGCCCGGTGCAATATGCTGTCGTCTACCCAGGGCGCGACACTGCCACTCGTTGGAAC	S505A
OL13002	KKT2	TCTCGATCAGTGCGTCGTAGCGTCAGCCTAACCGAACAAGAACGTGGCAGACTTGTGCGTTCGAGCCCGGTGCAATATGCTGTCGTCTACCCAGGGCGCGACACTGCCACTCGTTGGAAC	S505S
OL13369	KKT2	GCAGTGGTGTACCCGGGGCGCGACACTGCGACACGGTGGAATTTGCGGGCGGTTGTAGCGCTCCCTCGGGATATGACCGATGAAATTGAACGCGAGTTCAAGTGCATGAACGGGCACGTA	S530A one guide
OL13370	KKT2	GCAGTGGTGTACCCGGGGCGCGACACTGCGACACGGTGGAATTTGCGGGCGGTTGTATCCCTCCCTCGGGATATGACCGATGAAATTGAACGCGAGTTCAAGTGCATGAACGGGCACGTA	S530S one guide
OL13651	KKT2	GTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCGCCTCGAACGTCTCGATCAGTGCGTCGTAGCGTAGCGCTCACGGAGCAGGAGCGTGGCAGACTTGTGCGTTCTAGCCCGGTCCAGTAC	KKT2 S493A design 2

OL13652	KKT2	GTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCGCCTCGAACGTCTCGATCAGTGCGTCGTAGCGTATCGCTCACGGAGCAGGAGCGTGGCAGACTTGTGCGTTCTAGCCCGGTCCAGTAC	KKT2 S493S design 2
OL13653	KKT2	GTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCGCCACGCACGTCTCGATCAGTGCGTCGTTCGGTAGCGCTAACGGAGCAAGAACGTGGCAGACTTGTGCGTTCTAGCCCGGTCCAGTAC	KKT2 S493A design 3
OL13654	KKT2	GTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCGCCACGCACGTCTCGATCAGTGCGTCGTTCGGTAAGTCTAACGGAGCAAGAACGTGGCAGACTTGTGCGTTCTAGCCCGGTCCAGTAC	KKT2 S493S design 3
OL13655	KKT2	GTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCGCCTAGGACGAGTCGAAGCGTGAGGCGTAGCGTAGCGCTCACGGAGCAAGAGAGAGGCAGACTTGTGCGTTCTAGCCCGGTCCAGTAC	KKT2 S493A design 4
OL13656	KKT2	GTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCGCCTAGGACGAGTCGAAGCGTGAGGCGTAGCGTATCCCTCACGGAGCAAGAGAGAGGCAGACTTGTGCGTTCTAGCCCGGTCCAGTAC	KKT2 S493S design 4
OL13657	KKT2	GTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCACCCCGTACATCTCGATCAGTGCGTCGTTCCGTAGCGCTCACAGAGCAGGAGAGGGGCAGACTTGTGCGTTCTAGCCCGGTCCAGTAC	KKT2 S493A design 5
OL13658	KKT2	GTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCACCCCGTACATCTCGATCAGTGCGTCGTTCCGTATCTCTCACAGAGCAGGAGAGGGGCAGACTTGTGCGTTCTAGCCCGGTCCAGTAC	KKT2 S493S design 5
OL13659	KKT2	GCAGTGGTGTACCCGGGGCGCGACACTGCCACTCGTTGGAATTTGCGAGCCGTAGTAGCGCTTCCACGTGACATGACGGACGAGATCGAGCGCGAGTTCAAGTGCATGAACGGGCACGTA	KKT2 S530A design 2
OL13660	KKT2	GCAGTGGTGTACCCGGGGCGCGACACTGCCACTCGTTGGAATTTGCGAGCCGTAGTATCACTTCCACGTGACATGACGGACGAGATCGAGCGCGAGTTCAAGTGCATGAACGGGCACGTA	KKT2 S530S design 2
OL13661	KKT2	GCAGTGGTGTACCCGGGGCGCGACACTGCCACTCGTTGGAATTTGCGGGCGGTAGTGGCGCTCCCACGAGATATGACGGACGAGATCGAGCGCGAGTTCAAGTGCATGAACGGGCACGTA	KKT2 S530A design 3
OL13662	KKT2	GCAGTGGTGTACCCGGGGCGCGACACTGCCACTCGTTGGAATTTGCGGGCGGTAGTGTCACTCCCACGAGATATGACGGACGAGATCGAGCGCGAGTTCAAGTGCATGAACGGGCACGTA	KKT2 S530S design 3

OL13663	KKT2	GCAGTGGTGTACCCGGGGCGCGACACTGCCACTCGTTGGAATTTGAGAGCCGTAGTTGCGTTACCAAGGGATATGACGGACGAGATCGAGCGCGAGTTCAAGTGCATGAACGGGCACGTA	KKT2 S530A design 4
OL13664	KKT2	GCAGTGGTGTACCCGGGGCGCGACACTGCCACTCGTTGGAATTTGAGAGCCGTAGTTAGTTTACCAAGGGATATGACGGACGAGATCGAGCGCGAGTTCAAGTGCATGAACGGGCACGTA	KKT2 S530S design 4
OL13665	KKT2	GCAGTGGTGTACCCGGGGCGCGACACTGCCACTCGTTGGAATTTAAGGGCTGTCGTAGCGTTGCCTCGTGACATGACAGACGAGATAGAGCGCGAGTTCAAGTGCATGAACGGGCACGTA	KKT2 S530A design 5
OL13666	KKT2	GCAGTGGTGTACCCGGGGCGCGACACTGCCACTCGTTGGAATTTAAGGGCTGTCGTATCTTTGCCTCGTGACATGACAGACGAGATAGAGCGCGAGTTCAAGTGCATGAACGGGCACGTA	KKT2 S530S design 5
RC kkt2 M146G	KKT2	GAGCCATTTCTCGCGCCATCCCAACATTGTCAAGTTTTATGGAGCGGGCCGCGACGAGGACCGAGCGTATGTGGTGGGCGAACGTTGTGCAGGCAAGTCGCTTCACGACGTCATAGCCAG	M146G
OL12909	KKT2	CAGTACGCAGTGGTGTACCCGGGGCGCGACACTGCCACTCGTTGGAATCTTCGGGCCGTAGTAGCGCTGCCACGGGATATGACGGACGAGATCGAGCGCGAGTTCAAGTGCATGAACGGG	S530A
OL12928	KKT2	TCTCGATCAGTGCGTCGTAGCGTCAGCCTAACCGAACAAGAACGTGGCAGACTTGTGCGTTCTGCCCCGGTGCAATATGCTGTCGTCTACCCAGGGCGCGACACTGCCACTCGTTGGAAC	S506A
OL12929	KKT2	TCTCGATCAGTGCGTCGTAGCGTCAGCCTAACCGAACAAGAACGTGGCAGACTTGTGCGTTCTAGTCCGGTGCAATATGCTGTCGTCTACCCAGGGCGCGACACTGCCACTCGTTGGAAC	S506S
OL12930	KKT2	GTGGTGTACCCGGGGCGCGACACTGCCACTCGTTGGAATTTGCGGGCGGTTGTAGCGCTCCCTCGGGATATGACCGATGAAATCGAGCGCGAGTTCAAGTGCATGAACGGGCACGTAATG	S530A
OL12931	KKT2	GTGGTGTACCCGGGGCGCGACACTGCCACTCGTTGGAATTTGCGGGCGGTTGTATCCCTCCCTCGGGATATGACCGATGAAATCGAGCGCGAGTTCAAGTGCATGAACGGGCACGTAATG	S530S
OL12932	KKT2	AAGCAAGCCATCATGCCGCCTCAAGTGCCACGCGGACGAGCACAGCAGCCACGTGCGCCAGCGGTTTCGGGTCATACGGCTCAAGGTGGTCCGCCACTGCCGCGCCGCGGCCCAGCTGCG	S923A

OL12933	KKT2	AAGCAAGCCATCATGCCGCCTCAAGTGCCACGCGGACGAGCACAGCAGCCACGTGCGCCATCGGTTTCGGGTCATACGGCTCAAGGTGGTCCGCCACTGCCGCGCCGCGGCCCAGCTGCG	S923S
OL13351	KKT2	TCTCGATCAGTGCGTCGTAGCGTCAGCTTAACCGAACAAGAACGTGGTAGGCTCGTCCGGGCGGCGCCCGTGCAATATGCCGTCGTCTATCCGGGGCGCGACACTGCCACTCGTTGGAAC	S505A+S506A double mutant
OL13352	KKT2	TCTCGATCAGTGCGTCGTAGCGTCAGCTTAACCGAACAAGAACGTGGTAGGCTCGTCCGGAGCTCGCCCGTGCAATATGCCGTCGTCTATCCGGGGCGCGACACTGCCACTCGTTGGAAC	S505S+S506S double mutant
OL13003	KKT4	GCGGCGAAGAAGCTTCACGCTCAGCGGTTGGAGAAAGAAAATAATACGGGCGCCGATGATGCGATGGGAGCGCCCAAAGGTCTTGCTGGGGTACAGGCAAGCGCGAACCCCAACGAGCGC	S300A
OL13004	KKT4	GCGGCGAAGAAGCTTCACGCTCAGCGGTTGGAGAAAGAAAATAATACGGGCGCCGATGATGCGATGGGATCGCCCAAAGGTCTTGCTGGGGTACAGGCAAGCGCGAACCCCAACGAGCGC	S300S
OL12934	KKT4	CTGCAGGGCAGCGCCGATCGTGTCGTCCAGGGGCGTCGTGGCGTTGCGGCGACCAAGGCGGAGACGGCGCCGGCCTATATTACGACGCCCACGCCCGCCGGCAAGGCGTCCACCGCGCTC	S422A
OL12935	KKT4	CTGCAGGGCAGCGCCGATCGTGTCGTCCAGGGGCGTCGTGGCGTTGCGGCGACCAAGGCGGAGACGTCACCGGCCTATATTACGACGCCCACGCCCGCCGGCAAGGCGTCCACCGCGCTC	S422S
OL12936	KKT7	GCGAAGCAACAGAGTCGCGTTCACTCTTCTGCGCACCAACGGCGGCGTAGCATTAGCATTGTCGCGGCGGATGCGCTTGCCAAATCGGGCGAGGACGAAGACGGTGACGACAACGACACC	S304A
OL12937	KKT7	GCGAAGCAACAGAGTCGCGTTCACTCTTCTGCGCACCAACGGCGGCGTAGCATTAGCATTGTCTCGGCGGATGCGCTTGCCAAATCGGGCGAGGACGAAGACGGTGACGACAACGACACC	S304S

7.2.5.3 Single-Stranded Screening and Sequencing Primers

Name	Target	Sequence	Description
OL12991	KKT1	TGTCGCAACTCACGGATAGC	S1449
OL12992	KKT1	CTATTAGGGGCGGTGTGTCG	S1449
OL13055	KKT1	GACCCTGGTGTGGTGTCTTG	S1449
OL13055	KKT1	GACCCTGGTGTGGTGTCTTG	S1449
OL11617	KKT2	CTTCGCGTTAACGTGGATTT	M146
OL11618	KKT2	TGCAACCTCTGAGACCAGTG	M146
OL12993	KKT2	TACGGTGCTGGTAGGGATGA	S493
OL12994	KKT2	TGTCATTACGTGCCCGTTCA	S493
OL12904	KKT2	GACTTGTGGAGCGCATGTTG	S505/S506/S530
OL12905	KKT2	CACACATTGCAGTCGAAGCC	S505/S506/S530
OL13353	KKT2	ACGACTGGAAGCGACTGAAG	S505+S506 Double Mutant
OL13354	KKT2	CTTTGCGGTTGAGGTGAAGC	S505+S506 Double Mutant
OL12868	KKT2	CGGAAGTCATCACGATCCGC	S923
OL12869	KKT2	TTCTTGGGAGGAATCGCAGC	S923
OL12995	KKT4	CCGTGATTCACCGCAAAGAC	S300
OL12996	KKT4	CCACCGTCAGAAGAAAACGC	S300
OL12906	KKT4	ATTCACCGCAAAGACGAGGT	S422
OL12871	KKT4	TGTTGCGAGCGGATTCTGTT	S422
OL12870	KKT4	AGGGGGACCTTGTTGACGAT	S422
OL12907	KKT7	CATTCTGCCTGTGAACCAGC	S304
OL12908	KKT7	GCTCTTGCTCTTGGCCTTCT	S304
OL12878	KKT7	CGCCTGCCGAAAAATCGAAG	S304
OL12879	KKT7	TCCCCATCCTCATCAGTGGT	S304

7.2.5.4 Single-Stranded Pooled Experiment Screening Primers

Name	Target	Sequence	Description
OL13861	KKT2	GCTCCAGAAGCTCAACTTGC	S493 shared mutant screening primer
OL13864	KKT2	GCTTTGCGGTTGAGGTGAAG	S530 shared mutant screening primer
OL13964	KKT2	CTCCGTTAAGCTGACGCTAC	S493 WT screening
OL13965 (OL13862)	KKT2	CTTGTTCGGTTAGCGCTACG	S493A design 1 screening
OL13966	KKT2	GATCGAGACGTTCGAGGCG	S493A design 2 screening
OL13967	KKT2	CTTGCTCCGTTAGCGCTACC	S493A design 3 screening
OL13968	KKT2	CACGCTTCGACTCGTCCTAG	S493A design 4 screening
OL13969	KKT2	CTCTGTGAGCGCTACGGAAC	S493A design 5 screening
OL13970	KKT2	GTTCGGTTAGCGATACGCTAC	S493S design 1 screening
OL13971	KKT2	CTCCGTGAGCGATACGCTAC	S493S design 2 screening
OL13972	KKT2	CTTGCTCCGTTAGACTTACC	S493S design 3 screening
OL13973	KKT2	CTCCGTGAGGGATACGCTAC	S493S design 4 screening
OL13974	KKT2	CTGTGAGAGATACGGAACGAC	S493S design 5 screening
OL13975	KKT2	CCTTCGCGCCGTAGTATCGC	S530 WT screening
OL13976	KKT2	GCGCTCCCTCGGGATATG	S530A design 1 screening
OL13977	KKT2	GTAGTAGCGCTTCCACGTGAC	S530A design 2 screening
OL13978	KKT2	GGCGCTCCCACGAGATATG	S530A design 3 screening
OL13979	KKT2	CCGTAGTTGCGTTACCAAGG	S530A design 4 screening
OL13980	KKT2	GGGCTGTCGTAGCGTTG	S530A design 5 screening
OL13981	KKT2	GTATCCCTCCCTCGGGATATG	S530S design 1 screening
OL13982	KKT2	CGTAGTATCACTTCCACGTG	S530S design 2 screening
OL13983	KKT2	GGTAGTGTCACTCCCACGAG	S530S design 3 screening
OL13984	KKT2	GGAATTTGAGAGCCGTAGTTAG	S530S design 4 screening
OL13985	KKT2	GGGCTGTCGTATCTTTG	S530S design 5 screening

7.2.5.5 *Double-Stranded Repair Primers*

7.2.5.5.1 Primer Sequences

Name	**Target**	**Sequence**	**Description**
OL14224	KKT2	GGCCTCTGATGTCACACTTTTGCGGCTCGTTGTCGAGGACTCCACCACGGGGTGGGGCGATATCTATGCCCAGAGACCTTGCGCAAACACCAGCGATTTCACG	S25A repair
OL14225	KKT2	TCTGCTTGGTCAACAACACATTTTTGGATGTGCGGCGTCTTCACCGTGCTCCCCAGGCGTGAAATCGCTGGTGTTTG	S25A/E shared repair
OL14226	KKT2	GGCCTCTGATGTCACACTTTTGCGGCTCGTTGTCGAGGACTCCACCACGGGGTGGGGCGATATCTATGCCCAGAGACCTTGAGCAAACACCAGCGATTTCACG	S25E repair
OL14145	KKT2	GGCCTCTGATGTCACACTTTTGCGGCTCGTTGTCGAGGACTCCACCACGGGGTGGGGCGATATCTATGCCCAGAGACCTTAGTCAAACACC	S25S mutant forward
OL14146	KKT2	TCTGCTTGGTCAACAACACATTTTTGGATGTGCGGCGTCTTCACCGTGCTCCCCAGGCGTGAAATCGCTGGTGTTTGACTAAGGTCTCT	S25S mutant reverse
OL14228	KKT2	CGCCCCGGGTACACCACTGCGTACTGGACCGGGCTAGAACGCACAAGTCTGCCACGTTCTTGTTCGGTTAGCGCTACGCTACGACGGACAGAACGTG	S493A repair
OL14227	KKT2	GGTGTGGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCGCCACGGACCTCACGTTCTGTCCGTCGTAGC	S493A/E shared repair
OL14229	KKT2	CGCCCCGGGTACACCACTGCGTACTGGACCGGGCTAGAACGCACAAGTCTGCCACGTTCTTGTTCGGTTAGCTCTACGCTACGACGGACAGAACGTG	S493E repair

OL14147	KKT2	GGTGTGGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCGCCACGGACCTCACGTTCTGTCCGTCGTAGCGTATCTCTAAC	S493S mutant forward
OL14148	KKT2	CGCCCCGGGTACACCACTGCGTACTGGACCGGGCTAGAACGCACAAGTCTGCCACGTTCTTGTTCGGTTAGAGATACGCTACGACG	S493S mutant reverse
OL14230	KKT2	CGTTCTAGCCCGGTCCAGTACGCAGTGGTGTACCCGGGGCGCGACACTGCGACACGGTGGAATTTGCGGGCGGTTGTAGCGCTCCCTCGGGATATGACCGATG	S530A repair
OL14231	KKT2	TCGAGGTCAACTTTGTCATTACGTGCCCGTTCATGCACTTGAACTCGCGTTCAATTTCATCGGTCATATCCCGAGGG	S530A/E shared repair
OL14232	KKT2	CGTTCTAGCCCGGTCCAGTACGCAGTGGTGTACCCGGGGCGCGACACTGCGACACGGTGGAATTTGCGGGCGGTTGTAGAGCTCCCTCGGGATATGACCGATG	S530E repair
OL14149	KKT2	CGTTCTAGCCCGGTCCAGTACGCAGTGGTGTACCCGGGGCGCGACACTGCGACACGGTGGAATTTGCGGGCGGTTGTAAGTCTCCCTCGGGATATG	S530S mutant forward
OL14150	KKT2	TCGAGGTCAACTTTGTCATTACGTGCCCGTTCATGCACTTGAACTCGCGTTCAATTTCATCGGTCATATCCCGAGGGAGACTTAC	S530S mutant reverse
OL14233	KKT2	CTGTGGAGGAGCACGTGGTGAAGCAAGCCATCATGCCGCCTCAAGTGCCACGCGGACGAGCACAGCAGCCACGTGCGCCAGCGGTTTCGGGTCATACGGCTCAAG	S923A repair
OL14234	KKT2	AAAGCGGCTGCAGGAGATGGCGCAGCTGGGCCGCGGCGCGGCAGTGGCGGACCACCTTGAGCCGTATGACCCGAA	S923A/E shared repair

OL14235	KKT2	CTGTGGAGGAGCACGTGGTGAAGCAAGCCATCATGCCGCCTCAAGTGCCACGCGGACGAGCACAGCAGCCACGTGCGCCAGAGGTTTCGGGTCATACGGCTCAAG	S923E repair
OL14151	KKT2	CTGTGGAGGAGCACGTGGTGAAGCAAGCCATCATGCCGCCTCAAGTGCCACGCGGACGAGCACAGCAGCCACGTGCGCCAAGTGTTTCGGGTCATAC	S923S mutant forward
OL14152	KKT2	AAAGCGGCTGCAGGAGATGGCGCAGCTGGGCCGCGGCGCGGCAGTGGCGGACCACCTTGAGCCGTATGACCCGAAACACTTGG	S923S mutant reverse
OL14595	KKT4	TGAGTGCGCGTGCCGACGAGCGCGGTGGACGCCTTGCCGGCCGGCGTGGGTGTCGTAATATAGGCG	KKT4 S422 shared repair
OL14592	KKT4	GCACCACGTTGGACACGTCTCGTCTGCAGGGCAGCGCCGATCGTGTCGTGCAGGGTCGCCGTGGTGTGGCCGCGACCAAGGCGGAGACGGCGCCCGCCTATATTACGACACCC	KKT4 S422A repair
OL14593	KKT4	GCACCACGTTGGACACGTCTCGTCTGCAGGGCAGCGCCGATCGTGTCGTGCAGGGTCGCCGTGGTGTGGCCGCGACCAAGGCGGAGACGGAGCCCGCCTATATTACGACACCC	KKT4 S422E repair
OL14599	KKT7	AGCTCGCACGCAACTCCATGGCGAAGCAACAGAGTCGCGTTCACTCTTCTGCTCACCAACGCCGACGCTCCATCTCCATTGTC	KKT7 S304 shared repair
OL14598	KKT7	TCTAGACGCGCGCGCTTGCGGGTGTCGTTGTCGTCACCGTCTTCGTCCTCGCCCGATTTGGCCAGTGCATCCGCTGAGACAATGGAGATGGAGCGTC	KKT7 S304S repair
OL14596	KKT7	TCTAGACGCGCGCGCTTGCGGGTGTCGTTGTCGTCACCGTCTTCGTCCTCGCCCGATTTGGCCAGTGCATCCGCTGCGACAATGGAGATGGAGCGTC	KKT7 S304A repair
OL14597	KKT7	TCTAGACGCGCGCGCTTGCGGGTGTCGTTGTCGTCACCGTCTTCGTCCTCGCCCGATTTGGCCAGTGCATCCGCTTCGACAATGGAGATGGAGCGTC	KKT7 S304E repair

OL14594	KKT4	GCACCACGTTGGACACGTCTCGTCTGCAGGGCAGCGCCGATCGTGTCGTGCAGGGTCGCCGTGGTGTGGCCGCGACCAAGGCGGAGACGAGCCCCGCCTATATTACGACACCC	KKT4 S422S repair

7.2.5.5.2 Primer Combinations and Conditions

	KKT2 S25	KKT2 S493	KKT2 S530	KKT2 S923	KKT4 S422	KKT7 S304
Shared Primer	OL14225	OL14227	OL14231	OL14234	OL14595	OL14599
Alanine Mutant Primer	OL14224	OL14228	OL14230	OL14233	OL14592	OL14596
Glutamic Mutant Primer	OL14226	OL14229	OL14232	OL14235	OL14593	OL14597
Synonymous Mutant Primer(s)	OL14145 OL14146	OL14147 OL14148	OL14149 OL14150	OL14151 OL14152	OL14594	OL14598

		KKT2 Synonymous Mutants Only				All Mutants
		KKT2 S25S	KKT2 S493S	KKT2 S530S	KKT2 S923S	
1 cycle	98°C	30 sec	30 sec	30 sec	30 sec	30 sec
45 cycles	98°C	10 sec	10 sec	10 sec	10 sec	10 sec
	Tm	62°C	62°C	64°C	64°C	55°C
	30 sec	30 sec	30 sec	30 sec	30 sec	30 sec
	72°C	15 sec	15 sec	15 sec	15 sec	15 sec
1 cycle	72°C	10 mins	10 mins	10 mins	10 mins	10 mins
hold	4°C	hold	hold	hold	hold	hold

7.2.5.6 Double-Stranded Screening Primers

7.2.5.6.1 Primer Sequences

Name	**Target**	**Sequence**	**Description**
OL13866	KKT2	GCCAGCAATTCTTGGACGAC	KKT2 mutant screening control
OL13867	KKT2	TCACTCTGCCAGCGAATGTC	KKT2 mutant screening control
OL13860	KKT2	CAGCCCAGTGGTAACTACTC	S25
OL14209	KKT2	CTATGCCGCGTGATTTGTCG	S25 WT
OL13859	KKT2	CAGAGACCTTGCGCAAACAC	S25A
OL14242	KKT2	CTATGCCCAGAGACCTTGAG	S25E
OL14205	KKT2	GACCTTAGTCAAACACCAGCG	S25S
OL13861	KKT2	GCTCCAGAAGCTCAACTTGC	S493
OL13862	KKT2	CTTGTTCGGTTAGCGCTACG	S493A
OL14243	KKT2	CGTTCTTGTTCGGTTAGCTC	S493E
OL14206	KKT2	GACAGAACGTGAGGTCCGTG	S493S
OL13864	KKT2	GCTTTGCGGTTGAGGTGAAG	S530
OL13976	KKT2	GCGCTCCCTCGGGATATG	S530A
OL13863	KKT2	GACCGATGAAATTGAACGCG	S530A/S
OL14286	KKT2	ATTTGCGGGCGGTTGTAGAG	S530E
OL14207	KKT2	ATTTGCGGGCGGTTGTAAGT	S530S
OL7631	KKT2	CTGACTTTCCCAAGGTGAGC	S923
OL14210	KKT2	GTGCCCGCTGACGGAG	S923 WT
OL13865	KKT2	CACCTTGAGCCGTATGACCC	S923A/S
OL14208	KKT2	GCCGTATGACCCGAAACACT	S923S
OL14612	KKT4	GTGGTGATGTACGCCGGAGA	S422 WT
OL14613	KKT4	GTCGTAATATAGGCGGGCGC	S422A
OL14614	KKT4	GTCGTAATATAGGCGGGCTC	S422E
OL14615	KKT4	GTCGTAATATAGGCGGGGCT	S422S
OL14616	KKT7	CCAATAGTCGTCAGCGCCTT	S304
OL14617	KKT7	TCGGCGGAGACAATGGATA	S304 WT
OL14618	KKT7	GCCAGTGCATCCGCTGC	S304A
OL14619	KKT7	GGCCAGTGCATCCGCTTC	S304E
OL14620	KKT7	GCCAGTGCATCCGCTGA	S304S

7.2.5.6.2 Synonymous KKT2 Mutants Only Screening Conditions

Using Q5 polymerase.

		KKT2 DNA Control	**KKT2 S25S**	**KKT2 S493S**	**KKT2 S530S**	**KKT2 S923S**
WT Primers		OL13866 OL13867	OL14209 OL13860	OL13964 OL13861	OL13975 OL13864	OL14210 OL7631
Mutant Primers		-	OL14205 OL13860	OL14206 OL13861	OL14207 OL13864	OL14208 OL7631
Length (bp)		547	166	419	260	648
1 cycle	98°C	1 min	5 min	5 min	5 min	5 min
35 cycles	98°C	30 sec	30 sec	30 sec	30 sec	30 sec
	Tm	68°C 30 sec	66°C 30 sec	67°C 30 sec	68°C 30 sec	67°C 30 sec
	72°C	20 sec	8 sec	20 sec	8 sec	20 sec
1 cycle	72°C	10 min	10 min	10 min	10 min	10 min
hold	4°C	hold	hold	hold	hold	hold

7.2.5.6.3 All Mutant Screening Conditions

Using VeriFi polymerase.

	KKT2 S25	**KKT2 S493**	**KKT2 S530**	**KKT2 S923**	**KKT4 S422**	**KKT7 S304**
Shared primer (paired with all others)	OL13860	OL13861	OL13864	OL7631	OL12870	OL14616
WT primer	OL14209	OL13964	OL13975	OL14210	OL14612	OL14617
Alanine mutant primer	OL13859	OL13862	OL13976	OL13865	OL14613	OL14618
Glutamic acid mutant primer	OL14242	OL14243	OL14286	OL13865	OL14614	OL14619
Synonymous mutant primer	OL14205	OL14206	OL14207	OL13865	OL14615	OL14620
Tm used for screening with VeriFi (°C)	65	64	68	67	66	67
Expected Product Size (bp)	166	419	277	648	208	475

7.2.5.7 *Double-Stranded Sequencing PCR Amplification Conditions and Sanger Sequencing Primers*

		KKT2 S25	**KKT2 S493**	**KKT2 S530**	**KKT2 S923**	**KKT4 S422**	**KKT7 S304**
PCR Amplification		OL12128 OL12616				OL12871 OL12906	OL12907 OL12908
Length (bp)		4396				704	761
1 cycle	95°C	1 min				1 min	1 min
35 – 40 cycles	95°C	15 sec				15 sec	15 sec
	Tm	68°C				68°C	68°C
		2 min 30 sec				24 sec	30 sec
	72°C	20 sec				20 sec	20 sec
1 cycle	72°C	2 min				2 min	2 min
hold	4°C	hold				hold	hold
Sequencing Primers		OL13860 OL12128	OL12904	OL12905	OL12868 OL12869	OL12906	OL12908

7.2.6 SINGLE-STRANDED SCREENING CONDITIONS

Highlights indicate restriction digestion patterns that are shared between WT and the synonymous control mutant.

		KKT2 S505/506A	KKT2 S530A	KKT2 S923A	KKT4 S422A	KKT7 S304A
Primers		OL12904 OL12905	OL12904 OL12905	OL12868 OL12869	OL12871 OL12906	OL12907 OL12908
Length (bp)		742	742	400	704	961
1 cycle	95°C	1 min	1 min	1 min	1 min	1 min
35 cycles	95°C	15 sec	15 sec	15 sec	15 sec	15 sec
	Tm	68°C 15 sec	68°C 15 sec	69°C 15 sec	68°C 15 sec	68°C 15 sec
	72°C	24 sec	24 sec	12 sec	24 sec	30 sec
1 cycle	72°C	2 min	2 min	2 min	2 min	2 min
hold	4°C	hold	hold	hold	hold	hold
Restriction Enzyme		SmaI	AfeI	NlaIV	NlaIV	FokI
Native Digestion		580 162	544 198	235 141 24	355 147 113 89	752 209
Synonymous Control Digestion		742	544 198	376 24	355 147 113 89	648 209 104
S->A Mutant Digestion		742	427 198 117	376 24	184 171 147 113 89	648 209 104

		KKT1 S1449A	KKT1 S1449A	KKT2 S493A	KKT4 S300A	KKT2 M146A
Primers		OL12991 OL12992	OL13055 OL12992	OL12993 OL12994	OL12995 OL12996	OL11617 OL11618
Length (bp)		700	582	1260	1395	1920
1 cycle	95°C	1 min	1 min	1 min	1 min	1 min
35 cycles	95°C	15 sec	15 sec	15 sec	15 sec	15 sec
	Tm	68°C 15 sec	69°C 15 sec	68°C 15 sec	68°C 15 sec	64°C 15 sec
	72°C	22 sec	18 sec	38 sec	42 sec	58 sec
1 cycle	72°C	2 min	2 min	2 min	2 min	2 min
hold	4°C	hold	hold	hold	hold	hold
Restriction Enzyme		AluI	AluI	AfeI	BseYI	SinI or AvaII
Native Digestion		611 89	493 89	766 494	874 521	1293 627
Synonymous Control Digestion		412 199 89	294 199 89	766 494	688 521 186	-
S->A Mutant Digestion		412 199 89	294 199 89	766 316 178	688 521 186	912 627 381

7.2.7 POOLED REPAIR SCREENING CONDITIONS

7.2.7.1 Primer Combinations

Design Specific Target	Length	Shared Primer	Specific Primer	Tm (°C)
S493 WT screening	419 bp	OL13861	OL13964	66
S493A design 1			OL13965 (OL13862)	67
S493A design 2			OL13966	68
S493A design 3			OL13967	68
S493A design 4			OL13968	68
S493A design 5			OL13969	68
S493S design 1			OL13970	66
S493S design 2			OL13971	68
S493S design 3			OL13972	63
S493S design 4			OL13973	68
S493S design 5			OL13974	65
S530 WT screening	260 bp	OL13864	OL13975	68
S530A design 1			OL13976	68
S530A design 2			OL13977	68
S530A design 3			OL13978	68
S530A design 4			OL13979	67
S530A design 5			OL13980	68
S530S design 1			OL13981	67
S530S design 2			OL13982	63
S530S design 3			OL13983	68
S530S design 4			OL13984	64
S530S design 5			OL13985	67

7.2.7.2 Cycling Conditions

Using Q5 polymerase.

Step	Temperature	Time
1 cycle	98°C	5 minutes
35 Cycles	98°C	30 seconds
	S493A/S – 63-68°C S530A/S – 63-68°C	30 seconds
	72°C	S493A/S – 14 seconds S530A/S – 8 seconds
1 cycle	72°C	10 minutes
Hold	4°C	

7.2.8 POOLED REPAIR TEMPLATE RECODING LISTS

Design was as described in Methods 3.7.1 and Results 4.2. Each design’s list is sorted alphabetically by amino acid single letter code. Some lists may not include certain amino acids as per the methodology for that specific design.

Design 2						**Design 3**			**Design 4**			**Design 5**		
Preferred List			Reserve List											
Amino Acid	WT triplet codon	Mutant codon	Amino Acid	WT triplet codon	Mutant codon	Amino Acid	WT triplet codon	Mutant codon	Amino Acid	WT triplet codon	Mutant codon	Amino Acid	WT triplet codon	Mutant codon
A	GCT	GCG	A	GCA	GCT	A	GCA	GCT	A	GCA	GCT	A	GCA	GCC
			A	GCC	GCG	A	GCC	GCG	A	GCC	GCG	A	GCA	GCG
C	TGT	TGC	A	GCG	GCC	A	GCG	GCC	A	GCG	GCC	A	GCC	GCT
			A	GCG	GCA	A	GCG	GCA	A	GCG	GCA	A	GCC	GCA
F	TTT	TTC				A	GCT	GCG	A	GCT	GCG	A	GCG	GCT
F	TTC	TTT	C	TGC	TGT							A	GCT	GCA
						C	TGC	TGT	C	TGC	TGT	A	GCT	GCC
G	GGT	GGC	D	GAC	GAT	C	TGT	TGC	C	TGT	TGC			
G	GGT	GGA	D	GAT	GAC							G	GCA	GGT
						D	GAC	GAT	D	GAT	GAC	G	GGA	GGC
K	AAA	AAG	E	GAA	GAG	D	GAT	GAC	D	GAC	GAT	G	GGG	GGA
			E	GAG	GAA							G	GGG	GGC
L	CTG	CTT				E	GAA	GAG	E	GAA	GAG	G	GGT	GGG
L	CTT	CTG	G	GGA	GGG	E	GAG	GAA	E	GAG	GAA			
L	TTA	CTC	G	GGC	GGG							H	CAT	CAC
L	TTG	CTT	G	GGC	GGT	F	TTC	TTT	F	TTT	TTC			
			G	GGC	GGA	F	TTT	TTC	F	TTC	TTT	I	ATA	ATT
R	AGG	CGC	G	GGG	GGT							I	ATC	ATA
R	CGA	CGC				G	GGA	GGG	G	GGA	GGG	I	ATT	ATA
R	CGC	CGA	H	CAC	CAT	G	GGC	GGG	G	GGC	GGG			

R	CGG	CGT				G	GGC	GGT	G	GGC	GGT	L	CTA	TTG
			I	ATC	ATT	G	GGC	GGA	G	GGC	GGA	L	CTA	TTA
T	ACC	ACT	I	ATT	ATC	G	GGG	GGT	G	GGG	GGT	L	CTA	CTG
T	ACC	ACA				G	GGT	GGC	G	GGT	GGC	L	CTC	TTG
			K	AAG	AAA	G	GGT	GGA	G	GGT	GGA	L	CTC	CTA
V	GTA	GTG										L	CTC	TTA
V	GTT	GTA	L	CTA	CTT	H	CAC	CAT	H	CAC	CAT	L	CTG	TTA
			L	CTC	CTT							L	CTG	TTG
Y	TAT	TAC	L	CTC	CTG	I	ATC	ATT	I	ATC	ATT	L	CTT	CTA
			L	CTG	CTA	I	ATT	ATC	I	ATT	ATC	L	CTT	TTG
			L	CTG	CTC							L	CTT	CTC
			L	TTA	CTA	K	AAA	AAG	K	AAA	AAG	L	CTT	TTA
			L	TTG	CTC	K	AAG	AAA	K	AAG	AAA	L	TTA	CTT
												L	TTA	TTG
			N	AAC	AAT	L	TTA	CTA	L	TTA	CTA	L	TTA	CTG
						L	CTA	CTT	L	TTA	CTC	L	TTG	TTA
			P	CCC	CCA	L	CTC	CTT	L	TTA	CTG	L	TTG	CTA
			P	CCG	CCC	L	CTC	CTG	L	TTA	CTT	L	TTG	CTG
			P	CCT	CCA	L	CTG	CTT	L	TTG	CTA			
						L	CTG	CTA	L	TTG	CTC	N	AAC	AAT
			Q	CAG	CAA	L	CTG	CTC	L	TTG	CTT			
						L	CTT	CTG	L	TTG	CTG	P	CCA	CCC
			R	CGA	CGT	L	TTA	CTC	L	CTA	TTA	P	CCA	CCT
			R	CGC	CGG	L	TTG	CTC	L	CTA	TTG	P	CCA	CCG
						L	TTG	CTT	L	CTC	TTA	P	CCC	CCT
			S	AGC	AGT				L	CTC	TTG	P	CCC	CCG
			S	AGC	TCG	N	AAC	AAT	L	CTG	TTA	P	CCG	CCT
			S	TCA	TCT				L	CTG	TTG	P	CCG	CCA

S	TCT	TCA	P	CCC	CCA	L	CTT	TTA	P	CCT	CCG
			P	CCG	CCC	L	CTT	TTG	P	CCT	CCC
T	ACA	ACG	P	CCT	CCA						
T	ACC	ACG				N	AAC	AAT	Q	CAA	CAG
T	ACG	ACC	Q	CAG	CAA						
T	ACG	ACT				P	CCC	CCA	R	AGA	CGT
			R	AGG	CGC	P	CCG	CCC	R	AGA	CGC
V	GTA	GTT	R	CGA	CGT	P	CCT	CCA	R	AGA	CGA
V	GTC	GTA	R	CGA	CGC				R	AGA	CGG
V	GTC	GTT	R	CGC	CGA	Q	CAG	CAA	R	AGA	AGG
V	GTG	GTC	R	CGC	CGG				R	AGG	CGT
V	GTT	GTG	R	CGG	CGT	R	AGA	CGA	R	AGG	AGA
						R	AGA	CGC	R	AGG	CGG
Y	TAC	TAT	S	AGC	AGT	R	AGA	CGG	R	AGG	CGA
			S	AGC	TCG	R	AGA	CGT	R	CGA	AGA
			S	TCA	TCT	R	AGG	CGA	R	CGA	AGG
			S	TCT	TCA	R	AGG	CGC	R	CGC	AGA
						R	AGG	CGG	R	CGC	CGT
			T	ACA	ACG	R	AGG	CGT	R	CGC	AGG
			T	ACC	ACT	R	CGA	AGA	R	CGG	AGA
			T	ACC	ACG	R	CGA	AGG	R	CGG	AGG
			T	ACC	ACA	R	CGC	AGA	R	CGG	CGC
			T	ACG	ACC	R	CGC	AGG	R	CGG	CGT
			T	ACG	ACT	R	CGG	AGA	R	CGT	AGG
						R	CGG	AGG	R	CGT	AGA
			V	GTA	GTG	R	CGT	AGA	R	CGT	CGC
			V	GTA	GTT	R	CGT	AGG	R	CGT	CGG
			V	GTC	GTA						

V	GTC	GTT	S	AGT	TCA	S	AGC	TCA
V	GTG	GTC	S	AGT	TCC	S	AGC	TCC
V	GTT	GTA	S	AGT	TCG	S	AGC	TCT
V	GTT	GTG	S	AGT	TCT	S	AGT	AGC
			S	AGC	TCA	S	AGT	TCT
Y	TAC	TAT	S	AGC	TCC	S	AGT	TCC
Y	TAT	TAC	S	AGC	TCG	S	AGT	TCA
			S	AGC	TCT	S	AGT	TCG
			S	TCA	AGC	S	TCA	AGT
			S	TCA	AGT	S	TCA	TCG
			S	TCC	AGC	S	TCA	TCC
			S	TCC	AGT	S	TCA	AGC
			S	TCG	AGC	S	TCC	AGC
			S	TCG	AGT	S	TCC	TCG
			S	TCT	AGC	S	TCC	AGT
			S	TCT	AGT	S	TCC	TCA
						S	TCC	TCT
			T	ACA	ACG	S	TCG	AGC
			T	ACC	ACT	S	TCG	TCC
			T	ACC	ACG	S	TCG	AGT
			T	ACC	ACA	S	TCG	TCA
			T	ACG	ACC	S	TCG	TCT
			T	ACG	ACT	S	TCT	AGC
						S	TCT	TCG
			V	GTA	GTG	S	TCT	AGT
			V	GTA	GTT	S	TCT	TCC
			V	GTC	GTA			
			V	GTC	GTT	T	ACA	ACT

V	GTG	GTC	T	ACA	ACC
V	GTT	GTA	T	ACC	ACT
V	GTT	GTG	T	ACG	ACA
			T	ACT	ACA
Y	TAC	TAT	T	ACT	ACG
Y	TAT	TAC			
			V	GTA	GTC
			V	GTC	GTG
			V	GTG	GTA
			V	GTG	GTT
			V	GTT	GTC

7.2.9 EXAMPLE OUTPUT FILE TEXT FROM THE PYTHON SCRIPT

7.2.9.1 Page 1 (left hand side)

```
Job request details
Job name: KKT2 S493E
Target amino acid: S493E
Synonymous recoding type: matched
Nonsynonymous recode type: highest
Homology arm length (bp): 51
Recoding region length (bp): 60
Total repair length (bp): 162



Repair templates
WT repair region sequence:           GGC AGT GTC TCA CTG GTC TCA GAG GTT GCA GAT CGC GAG GAA GCC GCC CCT CGC ACG TCT CGA TCA GTG CGT CGT AGC
WT translation:                      G   S   V   S   L   V   S   E   V   A   D   R   E   E   A   A   P   R   T   S   R   S   V   R   R   S
Synonymous repair region sequence:   GGC AGT GTC TCA CTG GTC TCA GAG GTT GCA GAT CGC GAG GAA GCC GCC CCT CGG ACC AGC AGG AGT GTC CGA CGA TCG
Synonymous repair translation:       G   S   V   S   L   V   S   E   V   A   D   R   E   E   A   A   P   R   T   S   R   S   V   R   R   S
Nonsynonymous repair region sequence: GGC AGT GTC TCA CTG GTC TCA GAG GTT GCA GAT CGC GAG GAA GCC GCC CCT CGG ACC AGC AGG AGT GTC CGA CGA TCG
Nonsynonymous repair translation:    G   S   V   S   L   V   S   E   V   A   D   R   E   E   A   A   P   R   T   S   R   S   V   R   R   S

Number of mutations in the synonymous repair template: 30
Number of mutations in the nonsynonymous repair template: 30


Screening primers
Synonymous repair

               Forward primer sequence Reverse primer sequence  PCR product size (bp)  Forward GC content (%)  Reverse GC content (%)
WT primers          AGACGCCGCACATCCAAA      TGACGCTACGACGCACTG                   1365                    55.56                   61.11
Repair primers      AGACGCCGCACATCCAAA      CGTCGGACACTCCTGCTG                   1357                    55.56                   66.67



Nonsynonymous primers
               Forward primer sequence Reverse primer sequence  PCR product size (bp)  Forward GC content (%)  Reverse GC content (%)
WT primers          AGACGCCGCACATCCAAA      TGACGCTACGACGCACTG                   1365                    55.56                   61.11
Repair primers      AGACGCCGCACATCCAAA      CGTCGGACACTCCTGCTG                   1357                    55.56                   66.67
```

7.2.9.2 Page 1 (right hand side)

```
GTC AGC TTA ACG GAG CAG GAG CGG GGC AGA CTT GTG CGT TCT AGC CCG GTC CAG TAC GCA GTG GTG TAC CCG GGG CGC GAC ACT
V   S   L   T   E   Q   E   R   G   R   L   V   R   S   S   P   V   Q   Y   A   V   V   Y   P   G   R   D   T
GTT TCG CTA ACC GAA CAA GAA CGT GGT AGG TTG GTG CGT TCT AGC CCG GTC CAG TAC GCA GTG GTG TAC CCG GGG CGC GAC ACT
V   S   L   T   E   Q   E   R   G   R   L   V   R   S   S   P   V   Q   Y   A   V   V   Y   P   G   R   D   T
GTT GAG CTA ACC GAA CAA GAA CGT GGT AGG TTG GTG CGT TCT AGC CCG GTC CAG TAC GCA GTG GTG TAC CCG GGG CGC GAC ACT
V   E   L   T   E   Q   E   R   G   R   L   V   R   S   S   P   V   Q   Y   A   V   V   Y   P   G   R   D   T
```

```
Forward Tm ('C)  Reverse Tm ('C)
59.97            60.13
59.97            60.13
```

```
Forward Tm ('C)  Reverse Tm ('C)
59.97            60.13
59.97            60.13
```

7.2.9.3 Page 2

```
Repair template primers
Synonymous
Forward primer (5'-): GGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCCCCTCGGACCAGCAGGAGTGTCCGACG
Reverse primer (5'-): AGTGTCGCGCCCCGGGTACACCACTGCGTACTGGACCGGGCTAGAACGCACCAACCTACCACGTTCTTGTTCGGTTAGCGAAACCGATCGTCGGACACTCCTGCTG
Annealing sequence (5'-): CAGCAGGAGTGTCCGACG
Tm ('C): 60.1


Nonsynonymous
Forward primer (5'-): GGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCCCCTCGGACCAGCAGGAGTGTCCGACG
Reverse primer (5'-): AGTGTCGCGCCCCGGGTACACCACTGCGTACTGGACCGGGCTAGAACGCACCAACCTACCACGTTCTTGTTCGGTTAGCTCAACCGATCGTCGGACACTCCTGCTG
Annealing sequence (5'-): CAGCAGGAGTGTCCGACG
Tm ('C): 60.1


WT sequence (no spaces):
GGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCCCCTCGCACGTCTCGATCAGTGCGTCGTAGCGTCAGCTTAACGGAGCAGGAGCGGGGCAGACTTGTGCGTTCTAGCCCGGTCCAGTACGCAGTGGTGT
ACCCGGGGCGCGACACT
Synonymous sequence (no spaces):
GGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCCCCTCGGACCAGCAGGAGTGTCCGACGATCGGTTTCGCTAACCGAACAAGAACGTGGTAGGTTGGTGCGTTCTAGCCCGGTCCAGTACGCAGTGGTGT
ACCCGGGGCGCGACACT
Nonsynonymous sequence (no spaces):
GGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCCCCTCGGACCAGCAGGAGTGTCCGACGATCGGTTGAGCTAACCGAACAAGAACGTGGTAGGTTGGTGCGTTCTAGCCCGGTCCAGTACGCAGTGGTGT
ACCCGGGGCGCGACACT


Alignments
Synonymous Repair
Score = 132.0
WT sequence            0 GGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCCCCTCGCACGTCT
                       0 |||||||||||||||||||||||||||||||||||||||||||||||||||||.||....
Syn. repair            0 GGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCCCCTCGGACCAGC

WT sequence           60 CGATCAGTGCGTCGTAGCGTCAGCTTAACGGAGCAGGAGCGGGGCAGACTTGTGCGTTCT
                      60 .|....||.||.||....||.....||||.||.||.||.||.||.||..|.|||||||||
Syn. repair           60 AGGAGTGTCCGACGATCGGTTTCGCTAACCGAACAAGAACGTGGTAGGTTGGTGCGTTCT

WT sequence          120 AGCCCGGTCCAGTACGCAGTGGTGTACCCGGGGCGCGACACT 162
                     120 |||||||||||||||||||||||||||||||||||||||||| 162
Syn. repair          120 AGCCCGGTCCAGTACGCAGTGGTGTACCCGGGGCGCGACACT 162
```

```
Nonsynonymous
Score = 132.0
WT sequence            0 GGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCCCCTCGCACGTCT
                       0 |||||||||||||||||||||||||||||||||||||||||||||||||||||.||....
Nonsyn. repair         0 GGCAGTGTCTCACTGGTCTCAGAGGTTGCAGATCGCGAGGAAGCCGCCCCTCGGACCAGC

WT sequence           60 CGATCAGTGCGTCGTAGCGTCAGCTTAACGGAGCAGGAGCGGGGCAGACTTGTGCGTTCT
                      60 .|....||.||.||....||.....||||.||.||.||.||.||.||..|.|||||||||
Nonsyn. repair        60 AGGAGTGTCCGACGATCGGTTTCGCTAACCGAACAAGAACGTGGTAGGTTGGTGCGTTCT

WT sequence          120 AGCCCGGTCCAGTACGCAGTGGTGTACCCGGGGCGCGACACT 162
                     120 |||||||||||||||||||||||||||||||||||||||||| 162
Nonsyn. repair       120 AGCCCGGTCCAGTACGCAGTGGTGTACCCGGGGCGCGACACT 162
```

7.2.10 MAIN CODE

```
# -*- coding: utf-8 -*-
"""
Created on Mon Jan 29 15:34:37 2024

@author: ceh560
"""
#packages used in this file and/or the feeder files
import pandas as pd
from Bio import SeqIO
from Bio import Seq
from Bio import Align
import numpy as np
import io
import random
import primer3
import primer3.bindings

#custom files to import
import codon_dataframes as cdf
import codon_dictionaries as cdict
import formatting_functions as formats
import primer_functions as primers
import reading_input_file as rif
import validator as val
import stitching_functions as stitch




#read input files

input_data = pd.read_excel("repair_template_input_excel.xlsx",
index_col = 0, header = 0)


pd.set_option('display.max_columns', 20)
pd.set_option('display.max_rows', None)
pd.set_option("display.width", 1000)
pd.options.display.float_format = "{:,.2f}".format


job_name = input_data.loc["Job name"][0]
target_AA = input_data.loc["Target amino acid residue"][0]
target_res_num = input_data.loc["Target amino acid number"][0]
output_AA = input_data.loc["Replacement amino acid"][0]
syn_recode_type = input_data.loc["Synonymous Recoding type"][0]
nonsyn_recode_type = input_data.loc["Nonsynonymous Recoding
Type"][0]
codon_freq_input_file = input_data.loc["Codon Frequency data
filename (incl. extension)"][0]
recode_region_length = input_data.loc["Recoding region length
(bp)"][0]
hom_arm_length = input_data.loc["Homology arm length (bp)"][0]
```

```
ref_file_name = input_data.loc["Reference FASTA filename (incl. 
extension)"][0]
CDS_start = input_data.loc["CDS start in reference file (bp 
number)"][0]
CDS_end = input_data.loc["CDS end in reference file (bp 
number)"][0]
alternating_repeat = input_data.loc["Alternating every nth 
residue"][0]

#read input fasta file and process as necessary
gene_name = job_name

target_res_base_nums = [((target_res_num-1)*3), 
(target_res_num*3)]


num_of_codons_to_recode = recode_region_length / 3
target_codon_no = int(num_of_codons_to_recode/2)

if recode_region_length % 2 == 0:
    recode_start = int(target_res_base_nums[0] - 
(recode_region_length/2))

else:
    half_codon_percent = target_codon_no / num_of_codons_to_recode
    back_bases = recode_region_length * half_codon_percent
    recode_start = int(target_res_base_nums[0] - back_bases)


recode_end = recode_start + recode_region_length

#need some special cases for close to the start or end of the CDS
#near the start special case

if num_of_codons_to_recode > target_res_num:
    recode_start = 0
    recode_end = recode_region_length
    target_codon_no = target_res_num - 1



for gene_name in SeqIO.parse(ref_file_name,"fasta"):
    #print(gene_name.id)
    print(gene_name.description)
    print(repr(gene_name.seq))
    print("Gene sequence length: ", len(gene_name), "bp")
    print("\n")

if CDS_end == "end":
    CDS_end = len(gene_name.seq)
else:
    CDS_end = CDS_end


if CDS_start > 1:
    CDS_start = CDS_start - 1
    WT_CDS_seq = gene_name.seq[(CDS_start):CDS_end]
    recode_start_whole = recode_start + CDS_start
    recode_end_whole = recode_end + CDS_start

else:
    WT_CDS_seq = gene_name.seq[:CDS_end]
    recode_start_whole = recode_start
    recode_end_whole = recode_end



#check input is a length divisible by 3
val.triplet_checker(WT_CDS_seq)

#check that the input given is correct and that the target codes
for the expected residue
val.translate_checker(WT_CDS_seq, target_res_num, target_AA)




#near the end special case
total_num_AAs = len(WT_CDS_seq.translate())

if target_res_num > (total_num_AAs - num_of_codons_to_recode):
    recode_end = len(WT_CDS_seq)

    recode_start = len(WT_CDS_seq) - recode_region_length

    if CDS_start > 1:
        recode_end_whole = recode_end + CDS_start
        recode_start_whole = recode_start + CDS_start
    else:
        recode_end_whole = recode_end
        recode_start_whole = recode_start

    num_of_codons_to_recode = int((recode_end - recode_start + 1)
/ 3)

    target_codon_no = num_of_codons_to_recode - (total_num_AAs -
target_res_num) - 1




#establish the sequence to replace, and sequences before and after
to stay the same
WT_template_seq =
gene_name.seq[recode_start_whole:recode_end_whole]
upstream_dna = gene_name.seq[:recode_start_whole]
downstream_dna = gene_name.seq[recode_end_whole:]
```

```
#make dictionary of codons with number keys and one with numbers
and amino acids

codons_to_recode = cdict.codon_dict_maker(WT_template_seq,
key_format= "number")
codons_to_recode_let_num = cdict.codon_dict_maker(WT_template_seq,
key_format= "letter-number")


#make reference dictionaries for all the amino acids
ref_codon_table_df =
rif.codon_table_processor(codon_freq_input_file)

ref_codons = cdf.ref_codon_table_freqs(ref_codon_table_df)

if syn_recode_type == "matched":

    #use that dictionary to create a new one with the specific
frequency values
    codons_to_recode_freqs =
cdf.codon_frequency_collector(input_dict = codons_to_recode,
reference_dict = ref_codons, type = "value")

    #create a dictionary with all the frequencies for the amino
acids in this sequence for each codon
    codons_to_recode_all_freqs =
cdf.codon_frequency_collector(input_dict = codons_to_recode,
reference_dict = ref_codons, type = "dataframe")


    #calculate the differences for each possible codon to the
original
    recode_freq_diffs =
cdf.codon_frequency_difference_calc(codons_to_recode_let_num,
ref_codons)

    #add the differences in frequency to "the" dataframe
    codons_to_recode_abs_diffs =
cdf.codon_freq_diff_adder(codons_to_recode_let_num
,codons_to_recode_all_freqs, recode_freq_diffs)

    #choose which codons to use for synonymous recoding
    codons_to_use_syn =
cdf.codon_freq_selector(codons_to_recode_abs_diffs)


if syn_recode_type == "highest" or syn_recode_type == "lowest":

    #use that dictionary to create a new one with the specific
frequency values
    codons_to_recode_freqs =
cdf.codon_frequency_collector(input_dict = codons_to_recode,
reference_dict = ref_codons, type = "value")
```

```
    #create a dictionary with all the frequencies for the amino
acids in this sequence for each codon
    codons_to_recode_all_freqs =
cdf.codon_frequency_collector(input_dict = codons_to_recode,
reference_dict = ref_codons, type = "dataframe")

    codons_to_recode_choices_freqs = {}
    #remove input codon from list
    for let_num, seq in codons_to_recode_let_num.items():
        if seq == Seq.Seq("ATG") or seq == Seq.Seq("TGG"):

            codons_to_recode_choices_freqs[let_num] =
ref_codon_table_df.loc[ref_codon_table_df["DNA"] == str(seq)]
        else:
            current_df = codons_to_recode_all_freqs[let_num]
            codons_to_recode_choices_freqs[let_num] =
current_df.loc[current_df["DNA"] != str(seq)]

#make the list of codons to use depending on recoding type
    codons_to_use_syn = {}

    if syn_recode_type == "highest":

        for codon_num_let, seq_df in
codons_to_recode_choices_freqs.items():
            max_freq_codon = max(seq_df["Fraction"])
            mutated_res_df_chosen = seq_df.loc[seq_df["Fraction"]
== max_freq_codon, "DNA"]

            #tie breaker for instances with same fraction usage -
hopefully number won't ever have duplicate values
            if len(mutated_res_df_chosen) > 1:
                max_number_codon = max(seq_df["Number"])
                max_number_codon_seq = seq_df.loc[seq_df["Number"]
== max_number_codon, "DNA"].item()
                codons_to_use_syn[codon_num_let] =
max_number_codon_seq

            else:
                codons_to_use_syn[codon_num_let] =
seq_df.loc[seq_df["Fraction"] == max_freq_codon, "DNA"].item()

    if syn_recode_type == "lowest":

        for codon_num_let, seq_df in
codons_to_recode_choices_freqs.items():
            min_freq_codon = min(seq_df["Fraction"])
            mutated_res_df_chosen = seq_df.loc[seq_df["Fraction"]
== min_freq_codon, "DNA"]

            #tie breaker
            if len(mutated_res_df_chosen) > 1:
                min_number_codon = max(seq_df["Number"])
                min_number_codon_seq = seq_df.loc[seq_df["Number"]
== min_number_codon, "DNA"].item()
```

```
                codons_to_use_syn[codon_num_let] =
min_number_codon_seq

            else:
                codons_to_use_syn[codon_num_let] =
seq_df.loc[seq_df["Fraction"] == min_freq_codon, "DNA"].item()


if syn_recode_type == "alternating matched" or syn_recode_type ==
"alternating random" or syn_recode_type == "alternating highest"
or syn_recode_type == "alternating lowest":
    #check input has been given suitably
    if alternating_repeat == "N/A" or alternating_repeat <= 0 or
pd.isna(alternating_repeat) == True:
        print("\n\n\n***ERROR: No value or an invalid value was
set for the alternating pattern of the codons to
recode.***\n\n\n")
        alternating_repeat = int(input("Please enter a positive
integrer for the alternating repeat value: "))

    if alternating_repeat > (0.5 * num_of_codons_to_recode):
        proceed_alt = input("The chosen repeat value is greater
than half of the total number of codons being recoded so only 2 or
fewer codons will be mutated.\n\nDo you wish to proceed? Y/N \n")

        if proceed_alt == "N" or proceed_alt == "n" or proceed_alt
== "NO" or proceed_alt == "No" or proceed_alt == "no":
            alternating_repeat = int(input("Please enter a
positive integer for the alternating repeat value: "))

        elif proceed_alt == "Y" or proceed_alt =="y" or
proceed_alt =="YES" or proceed_alt == "Yes" or proceed_alt ==
"yes":
           pass
        else:
            proceed_alt = input("\n\nThe input given is not valid.
Please try again.\n\nThe chosen repeat value is greater than half
of the total number of codons being recoded so only 2 or fewer
codons will be mutated.\n\nDo you wish to proceed? Y/N \n")
            if proceed_alt == "N" or proceed_alt == "n" or
proceed_alt == "NO" or proceed_alt == "No" or proceed_alt == "no":
                alternating_repeat = int(input("\nPlease enter a
positive integer for the alternating repeat value: "))
            elif proceed_alt == "Y" or proceed_alt =="y" or
proceed_alt =="YES" or proceed_alt == "Yes" or proceed_alt ==
"yes":
               pass
            else:
                proceed_alt = input("\n\nThe input given is not
valid. Please try again.\n\nThe chosen repeat value is greater
than half of the total number of codons being recoded so only 2 or
fewer codons will be mutated.\n\nDo you wish to proceed? Y/N \n")
                if proceed_alt == "N" or proceed_alt == "n" or
proceed_alt == "NO" or proceed_alt == "No" or proceed_alt == "no":
                    alternating_repeat = int(input("\nPlease enter
a positive integer for the alternating repeat value: "))
```

```
                elif proceed_alt == "Y" or proceed_alt =="y" or proceed_alt =="YES" or proceed_alt == "Yes" or proceed_alt == "yes":
                   pass
                else:
                    print("\n------------------------------------------------------------")
                    print(f"\n***Warning, your input was invalid so the code will continue with the value given. Your repair template will recode every {alternating_repeat} codons. If you do not want this, modify the input spreadsheet and rerun the programme.***")


    if alternating_repeat == "N/A" or alternating_repeat <= 0 or pd.isna(alternating_repeat) == True:
        print("\n\n\n***ERROR: An invalid value was set for the alternating pattern of the codons to recode.***\n\n\n")
        alternating_repeat = int(input("Please enter a positive integer for the alternating repeat value: "))

    if alternating_repeat == "N/A" or alternating_repeat <= 0 or pd.isna(alternating_repeat) == True:
        print("\n\n\n***ERROR: An invalid value was set for the alternating pattern of the codons to recode.***\n\n\n")
        alternating_repeat = int(input("Last chance - please enter a positive integer for the alternating repeat value: "))

    if alternating_repeat == "N/A" or alternating_repeat <= 0 or pd.isna(alternating_repeat) == True:
        print("\n\n\n\nYou failed to provide an appropriate input so the programme will be cancelled.\n\nIf you wish to try again, either modify the input spreadsheet or provide a suitable value when prompted in the console.\n")
        raise SystemExit


if syn_recode_type == "alternating matched" or syn_recode_type == "alternating random":


    #determine which codon numbers in range are to be mutated and which are not
    num_of_codons_to_mutate = int(num_of_codons_to_recode / alternating_repeat)
    n_terms = list(range(num_of_codons_to_mutate))
    codon_nums_to_recode = []

    for n in n_terms:
        codon_num = n * alternating_repeat
        codon_nums_to_recode.append(codon_num)

    #ensure that target codon is always recoded even if it doesn't fit the alternating pattern
    if target_codon_no not in codon_nums_to_recode:
        codon_nums_to_recode.append(target_codon_no)
    codon_nums_all = list(codons_to_recode.keys())

    #split the codons to be mutated into a separate dictionary
from the ones to stay the same
    codons_to_keep_WT = {}
    specific_codons_to_recode = {}

    for numbers in codon_nums_all:
        if numbers not in codon_nums_to_recode:
            codons_to_keep_WT[numbers] = codons_to_recode[numbers]

        if numbers in codon_nums_to_recode:
            specific_codons_to_recode[numbers] =
codons_to_recode[numbers]


    for numbers in codon_nums_to_recode:
        if numbers not in codon_nums_to_recode:
            codons_to_keep_WT = codons_to_recode[numbers]

    if syn_recode_type == "alternating matched":
        #on only the codons to recode
        #use that dictionary to create a new one with the specific
frequency values
        codons_to_recode_freqs =
cdf.codon_frequency_collector(input_dict = codons_to_recode,
reference_dict = ref_codons, type = "value")

        #create a dictionary with all the frequencies for the
amino acids in this sequence for each codon
        codons_to_recode_all_freqs =
cdf.codon_frequency_collector(input_dict = codons_to_recode,
reference_dict = ref_codons, type = "dataframe")


        #calculate the differences for each possible codon to the
original
        recode_freq_diffs =
cdf.codon_frequency_difference_calc(codons_to_recode_let_num,
ref_codons)

        #add the differences in frequency to "the" dataframe
        codons_to_recode_abs_diffs =
cdf.codon_freq_diff_adder(codons_to_recode_let_num
,codons_to_recode_all_freqs, recode_freq_diffs)

        #choose which codons to use for synonymous recoding
        codons_to_use_syn =
cdf.codon_freq_selector(codons_to_recode_abs_diffs)


    if syn_recode_type == "alternating random":

        #add letters to dictionary
        specific_codons_to_recode_let_num = {}

        for keys, seq in specific_codons_to_recode.items():
            let_num = str(seq.translate()) + str(keys)
            specific_codons_to_recode_let_num[let_num] = seq


        #make a dictionary of the alternate codons to the input
sequence
        alt_codons_to_recode =
cdict.alt_codons(specific_codons_to_recode_let_num)

        #randomly select which of these to use for each codon
        codons_to_use_syn =
cdict.Syn_random_recoder(alt_codons_to_recode)


    #combine the unchanged codons with the changed codons

    codons_to_keep_WT_let_num = {}

    for codon_num, seq in codons_to_keep_WT.items():
        translation = seq.translate()
        codon_num_let = str(translation) + str(codon_num)

        codons_to_keep_WT_let_num[codon_num_let] = seq

    codons_to_use_syn.update(codons_to_keep_WT_let_num)


if syn_recode_type == "alternating highest" or syn_recode_type ==
"alternating lowest":

    num_of_codons_to_mutate = int(num_of_codons_to_recode /
alternating_repeat)
    n_terms = list(range(num_of_codons_to_mutate))
    codon_nums_to_recode = []

    for n in n_terms:
        codon_num = n * alternating_repeat
        codon_nums_to_recode.append(codon_num)

    if target_codon_no not in codon_nums_to_recode:
        codon_nums_to_recode.append(target_codon_no)


    codon_nums_all = list(codons_to_recode.keys())

    codons_to_keep_WT = {}
    specific_codons_to_recode = {}

    for numbers in codon_nums_all:
        if numbers not in codon_nums_to_recode:
            translate = codons_to_recode[numbers].translate()
            let_num = str(translate) + str(numbers)
            codons_to_keep_WT[let_num] = codons_to_recode[numbers]
        if numbers in codon_nums_to_recode:
            #translate = codons_to_recode[numbers].translate()
            #let_num = str(translate) + str(numbers)
            specific_codons_to_recode[numbers] = 
codons_to_recode[numbers]



    for numbers in codon_nums_to_recode:
        if numbers not in codon_nums_to_recode:
            codons_to_keep_WT = codons_to_recode[numbers]

    #use that dictionary to create a new one with the specific 
frequency values
    codons_to_recode_freqs = 
cdf.codon_frequency_collector(input_dict = 
specific_codons_to_recode, reference_dict = ref_codons, type = 
"value")

    #create a dictionary with all the frequencies for the amino 
acids in this sequence for each codon
    codons_to_recode_all_freqs = 
cdf.codon_frequency_collector(input_dict = 
specific_codons_to_recode, reference_dict = ref_codons, type = 
"dataframe")

    codons_to_recode_choices_freqs = {}
    #remove input codon from list unless it's Met or Trp
    for let_num, df in codons_to_recode_all_freqs.items():
        input_codon = codons_to_recode_let_num[let_num]
        if input_codon == Seq.Seq("ATG") or input_codon == 
Seq.Seq("TGG"):

            codons_to_recode_choices_freqs[let_num] = 
ref_codon_table_df.loc[ref_codon_table_df["DNA"] == 
str(input_codon)]
        else:
            current_df = codons_to_recode_all_freqs[let_num]
            codons_to_recode_choices_freqs[let_num] = 
current_df.loc[current_df["DNA"] != str(input_codon)]

    #recode based on input type
    codons_to_use_syn = {}

    if syn_recode_type == "alternating highest":

        for codon_num_let, seq_df in 
codons_to_recode_choices_freqs.items():
            max_freq_codon = max(seq_df["Fraction"])
            max_freq_codon_seq = seq_df.loc[seq_df["Fraction"] == 
max_freq_codon, "DNA"]
            if len(max_freq_codon_seq) > 1:
                max_number_codon = max(seq_df["Number"])
                max_freq_codon_seq = seq_df.loc[seq_df["Number"] 
== max_number_codon, "DNA"].item()
```

```
                codons_to_use_syn[codon_num_let] = 
max_freq_codon_seq

            else:
             codons_to_use_syn[codon_num_let] = 
max_freq_codon_seq.item()



    if syn_recode_type == "alternating lowest":

        for codon_num_let, seq_df in 
codons_to_recode_choices_freqs.items():
            min_freq_codon = min(seq_df["Fraction"])
            min_freq_codon_seq = seq_df.loc[seq_df["Fraction"] == 
min_freq_codon, "DNA"]
            if len(min_freq_codon_seq) > 1:
                min_number_codon = min(seq_df["Number"])
                min_freq_codon_seq = seq_df.loc[seq_df["Number"] 
== min_number_codon, "DNA"].item()
                codons_to_use_syn[codon_num_let] = 
min_freq_codon_seq

            else:
             codons_to_use_syn[codon_num_let] = 
min_freq_codon_seq.item()




    #combine the unchanged codons with the changed codons

    codons_to_keep_WT_let_num = {}

    for codon_num, seq in codons_to_keep_WT.items():
        codons_to_keep_WT_let_num[codon_num] = seq

    codons_to_use_syn.update(codons_to_keep_WT_let_num)


if syn_recode_type == "random":
    #make a dictionary of the alternate codons to the input 
sequence
    alt_codons_to_recode = 
cdict.alt_codons(codons_to_recode_let_num)

    #randomly select which of these to use for each codon
    codons_to_use_syn = 
cdict.Syn_random_recoder(alt_codons_to_recode)


#add in the nonsynonymous mutation

if nonsyn_recode_type == "highest" or nonsyn_recode_type == 
"lowest":
    nonsyn_ref_dict = ref_codons

if nonsyn_recode_type == "random":
    nonsyn_ref_dict = cdict.alt_codons(codons_to_recode_let_num)
    nonsyn_ref_dict = {output_AA :
cdict.ref_codon_table(output_AA)}


codons_to_use_nonsyn = cdf.non_syn_mutator(target_AA,
target_codon_no, new_AA = output_AA, input_dict =
codons_to_use_syn, type = nonsyn_recode_type, ref_dict =
nonsyn_ref_dict )



#construct the final recoded sequences

synonymous_repair = stitch.sequence_constructor(codons_to_use_syn,
type = "letter-number")
nonsynonymous_repair =
stitch.sequence_constructor(codons_to_use_nonsyn, type = "letter-
number")

#check all the modifications were as expected
#adjust target codon number to what it would be by normal counting
rather than python counting
target_codon_no_not_py = target_codon_no + 1


val.translate_checker(synonymous_repair, target_codon_no_not_py,
target_AA)

val.translate_checker(nonsynonymous_repair,
target_codon_no_not_py, output_AA)



#create the final repair sequence including the homology arms

upstream_hom_arm = gene_name.seq[(recode_start_whole -
hom_arm_length):recode_start_whole]
downstream_hom_arm = gene_name.seq[recode_end_whole:
(recode_end_whole + hom_arm_length)]

WT_entire_repair_region = upstream_hom_arm + WT_template_seq +
downstream_hom_arm
entire_syn_repair = upstream_hom_arm + synonymous_repair +
downstream_hom_arm
entire_nonsyn_repair = upstream_hom_arm + nonsynonymous_repair +
downstream_hom_arm



#construct "gene" sequences for primer design
integrated_synonymous, WT_recode_region =
stitch.mut_seq_integrator(repair_seq = synonymous_repair, ref_seq
= gene_name.seq, repair_start = recode_start_whole, repair_end =
recode_end_whole, WT_repair_seq= "Yes")
integrated_nonsynonymous = stitch.mut_seq_integrator(repair_seq =
nonsynonymous_repair, ref_seq = gene_name.seq, repair_start =
recode_start_whole, repair_end = recode_end_whole, WT_repair_seq=
"No")

#design screening primers
screening_primers_df_syn  =
primers.screening_primer_designer(gene_name.seq,
integrated_synonymous, recode_start_whole, recode_end_whole)
screening_primers_df_nonsyn  =
primers.screening_primer_designer(gene_name.seq,
integrated_nonsynonymous, recode_start_whole, recode_end_whole)

#design primers to generate the repair template
syn_repair_template_primers =
primers.repair_primer_designer(entire_syn_repair, hom_arm_length,
downstream_dna)
nonsyn_repair_template_primers =
primers.repair_primer_designer(entire_nonsyn_repair,
hom_arm_length, downstream_dna)

#repair_template_primers = [syn_repair_template_primers,
nonsyn_repair_template_primers]

#repair_template_primers_df =
pd.DataFrame(repair_template_primers)
#repair_template_primers_df.index = ["Synonymous repair",
"Nonsynonymous repair"]

#do an alignment

#create a pariwise alignment object
aligner = Align.PairwiseAligner(target_internal_open_gap_score = -
10.0, query_internal_open_gap_score = -10.0)



syn_alignment = aligner.align(WT_entire_repair_region,
entire_syn_repair)
for alignment1 in sorted(syn_alignment):
    #print("Score = %.1f:" % alignment1.score)
    #print(alignment1)
    syn_score = alignment1.score
alignment_str_syn = str(alignment1)
alignment_str_syn = alignment_str_syn.replace("target", "WT
sequence").replace("query", "Syn. repair").replace("\n
", "\n              ")
alignment_str_syn = alignment_str_syn.replace("Syn. repair
", "Syn. repair         ")
#print(alignment_str_syn)

nonsyn_alignment = aligner.align(WT_entire_repair_region,
entire_nonsyn_repair)
for alignment2 in sorted(syn_alignment):
    #print("Score = %.1f:" % alignment2.score)
    nonsyn_score = alignment2.score
alignment_str_nonsyn = str(alignment2)
alignment_str_nonsyn = alignment_str_nonsyn.replace("target", "WT 
sequence").replace("query", "Nonsyn. repair").replace("\n
", "\n                ")
alignment_str_nonsyn = alignment_str_nonsyn.replace("Nonsyn. 
repair           ", "Nonsyn. repair       ")
#print(alignment_str_nonsyn)




#format some outputs

WT_repair_seq_spaced = 
formats.codon_spacing(WT_entire_repair_region)
syn_repair_spaced = formats.codon_spacing(entire_syn_repair)
nonsyn_repair_spaced = formats.codon_spacing(entire_nonsyn_repair)

WT_repair_translate = WT_entire_repair_region.translate()
syn_repair_translate = entire_syn_repair.translate()
nonsyn_repair_translate = entire_nonsyn_repair.translate()

WT_repair_translate_spaced = 
formats.protein_align_codon(WT_repair_translate)
syn_repair_translate_spaced = 
formats.protein_align_codon(syn_repair_translate)
nonsyn_repair_translate_spaced = 
formats.protein_align_codon(nonsyn_repair_translate)

syn_repair_mutations_count = 
val.mutation_counter(entire_syn_repair, WT_entire_repair_region)
nonsyn_repair_mutations_count = 
val.mutation_counter(entire_nonsyn_repair, 
WT_entire_repair_region)

syn_repair_primers_output = ""

for category, item in syn_repair_template_primers.items():
    if type(item) == float:
        item = '{:.1f}'.format(item)
    syn_repair_primers_output += category
    syn_repair_primers_output += ": "
    syn_repair_primers_output += str(item)
    syn_repair_primers_output += "\n"

nonsyn_repair_primers_output = ""

for category, item in nonsyn_repair_template_primers.items():
    if type(item) == float:
        item = '{:.1f}'.format(item)
    nonsyn_repair_primers_output += category
    nonsyn_repair_primers_output += ": "
    nonsyn_repair_primers_output += str(item)
    nonsyn_repair_primers_output += "\n"
```

```
if syn_recode_type == "alternating matched" or syn_recode_type ==
"alternating highest" or syn_recode_type == "alternating lowest"
or syn_recode_type == "alternating random":
    alternating_info = f"Alternating recoding every
{alternating_repeat} codons"
else:
    alternating_info = ""


output_file = open(f"{job_name}.txt", "w")

file_lines = ["Job request details\n",
              f"Job name: {job_name}\n",
              f"Target amino acid:
{target_AA}{target_res_num}{output_AA}\n",
              f"Synonymous recoding type: {syn_recode_type}\n",
              f"Nonsynonymous recode type:
{nonsyn_recode_type}\n",
              f"Homology arm length (bp): {hom_arm_length}\n",
              f"Recoding region length (bp):
{recode_region_length}\n",
              f"Total repair length (bp): {(2*hom_arm_length) +
recode_region_length}\n",
              f"{alternating_info}\n",
              "\n",
              "\n",
              "Repair templates\n",
              f"WT repair region sequence:
\t\t{WT_repair_seq_spaced}\n",
              f"WT translation:
\t\t\t{WT_repair_translate_spaced}\n",
              f"Synonymous repair region sequence:
\t{syn_repair_spaced}\n",
              f"Synonymous repair translation:
\t\t{syn_repair_translate_spaced}\n",
              f"Nonsynonymous repair region sequence:
\t{nonsyn_repair_spaced}\n",
              f"Nonsynonymous repair translation:
\t{nonsyn_repair_translate_spaced}\n",
              "\n",
              f"Number of mutations in the synonymous repair
template: {syn_repair_mutations_count}\n",
              f"Number of mutations in the nonsynonymous repair
template: {nonsyn_repair_mutations_count}\n",
              "\n",
              "\n",
              "Screening primers\n",
              "Synonymous repair\n",
              "\n",
              f"{screening_primers_df_syn}\n",
              "\n",
              "\n",
              "Nonsynonymous primers\n"
              f"{screening_primers_df_nonsyn}",
              "\n",
              "\n",
              "Repair template primers\n",
              "Synonymous\n",
              f"{syn_repair_primers_output}\n",
              "\n",
              "Nonsynonymous\n",
              f"{nonsyn_repair_primers_output}\n",
              "\n",
              f"WT sequence (no spaces):
{WT_entire_repair_region}\n",
              f"Synonymous sequence (no spaces):
{entire_syn_repair}\n",
              f"Nonsynonymous sequence (no spaces):
{entire_nonsyn_repair}\n",
              "\n",
              "\n",
              "Alignments\n",
              "Synonymous Repair\n",
              f"Score = {syn_score}\n",
              f"{alignment_str_syn}\n",
              "\n",
              "Nonsynonymous\n",
              f"Score = {nonsyn_score}\n",
              f"{alignment_str_nonsyn}\n"

    ]

output_file.writelines(file_lines)
output_file.close()

#print confirmation message to make it clearer that it worked
print(f"\n\n\nYour repair template designs have completed
successfully. Please check your folder for a file with the name
'{job_name}.txt'\n")
print("\t.\t.\n", "\n\t\___/\n\n\n")
```

7.2.11 READING INPUT FILE

```
# -*- coding: utf-8 -*-
"""
Created on Sat Jan 27 10:16:59 2024

@author: sharl
"""

import pandas as pd
#from Bio import SeqIO
from Bio import Seq
import numpy as np
#import openpyxl
#from Bio import Align
import io




def codon_table_processor(filename):
    """Converts a text file with data copied from 
https://www.kazusa.or.jp/ codon tables into a dataframe.

    Codon tables must have selected a genetic code in the format 
options and text file does not include headers.

    Note: uracils are changed to thymines

    Arguments
    filename -- filename of the text file including extension as a 
string

    Outputs a dataframe"""


    raw_freq = open(filename, "r").read()

    headers = ["DNA", "Protein", "Fraction", "Frequency", 
"Number"]

    raw_freq_str = str(raw_freq)

    raw_freq_str_lines = raw_freq_str.replace(") ", 
")\n").replace("\n ", "\n")
    raw_freq_str_lines = raw_freq_str_lines.replace("( ", 
"").replace(")","").replace("(", "").replace("  ", " ")
    raw_freq_str_lines_Ts = raw_freq_str_lines.replace("U", "T")
    raw_freq_str_lines_Ts_tabs = raw_freq_str_lines_Ts.replace(" 
", "\t")

    df = pd.read_csv(io.StringIO(raw_freq_str_lines_Ts_tabs), 
sep="\t", header = None)
    df.columns = headers

    return df
```

7.2.12 CODON DICTIONARIES

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jan 24 08:47:49 2024

@author: ceh560
"""
from Bio import SeqIO
from Bio import Seq
import random

def ref_codon_table(amino_acid):
    """"A searchable dictionary for all possible standard triplet
codes for each amino acid.
    Arguments
    amino_acid -- single letter code for amino acid as a string

    Returns a list of possible triplet codes for that amino
acid."""


    Ala_codons = [Seq.Seq('GCT'), Seq.Seq('GCC'), Seq.Seq('GCA'),
Seq.Seq('GCG')]
    Cys_codons = [Seq.Seq('TGT'), Seq.Seq('TGC')]
    Asp_codons = [Seq.Seq('GAT'), Seq.Seq('GAC')]
    Glu_codons = [Seq.Seq('GAA'), Seq.Seq('GAG')]
    Phe_codons = [Seq.Seq('TTT'), Seq.Seq('TTC')]
    Gly_codons = [Seq.Seq('GGT'), Seq.Seq('GGC'), Seq.Seq('GGA'),
Seq.Seq('GGG')]
    His_codons = [Seq.Seq('CAT'), Seq.Seq('CAC')]
    Ile_codons = [Seq.Seq('ATT'), Seq.Seq('ATC'), Seq.Seq('ATA')]
    Lys_codons = [Seq.Seq('AAA'), Seq.Seq('AAG')]
    Leu_codons = [Seq.Seq('CTT'), Seq.Seq('CTC'), Seq.Seq('CTA'),
Seq.Seq('CTG'), Seq.Seq('TTG'), Seq.Seq('TTA')]
    Met_codons = [Seq.Seq('ATG'), Seq.Seq('ATG')]
    Asn_codons = [Seq.Seq('AAT'), Seq.Seq('AAC')]
    Pro_codons = [Seq.Seq('CCT'), Seq.Seq('CCC'), Seq.Seq('CCA'),
Seq.Seq('CCG')]
    Gln_codons = [Seq.Seq('CAA'), Seq.Seq('CAG')]
    Arg_codons = [Seq.Seq('CGC'), Seq.Seq('CGT'), Seq.Seq('CGA'),
Seq.Seq('CGG'), Seq.Seq('AGA'), Seq.Seq('AGG')]
    Ser_codons = [Seq.Seq('AGT'), Seq.Seq('AGC'), Seq.Seq('TCT'),
Seq.Seq('TCC'), Seq.Seq('TCA'), Seq.Seq('TCG')]
    Thr_codons = [Seq.Seq('ACT'), Seq.Seq('ACC'), Seq.Seq('ACA'),
Seq.Seq('ACG')]
    Val_codons = [Seq.Seq('GTT'), Seq.Seq('GTC'), Seq.Seq('GTA'),
Seq.Seq('GTG')]
    Trp_codons = [Seq.Seq('TGG'), Seq.Seq('TGG')]
    Tyr_codons = [Seq.Seq('TAT'), Seq.Seq('TAC')]
    Stop_codons = [Seq.Seq('TAA'), Seq.Seq('TAG'), Seq.Seq('TGA')]

    ref_codon_seq_all = {"A": Ala_codons,
                         "C": Cys_codons,
                         "D": Asp_codons,
                         "E": Glu_codons,
                         "F": Phe_codons,
                         "G": Gly_codons,
                         "H": His_codons,
                         "I": Ile_codons,
                         "K": Lys_codons,
                         "L": Leu_codons,
                         "M": Met_codons,
                         "N": Asn_codons,
                         "P": Pro_codons,
                         "Q": Gln_codons,
                         "R": Arg_codons,
                         "S": Ser_codons,
                         "T": Thr_codons,
                         "V": Val_codons,
                         "W": Trp_codons,
                         "Y": Tyr_codons,
                         "*": Stop_codons
        }
    return ref_codon_seq_all[amino_acid]


#convert the tupules to a dictionary with a custom function
def DictConvert(tup, dic):
    for a, b in tup:
        dic.setdefault(a, b)
    return dic


def protein_dict_maker(input_seq):
    """"Converts a DNA sequence into a dictionary of the
translated amino acids of each codon, numbered by the order of
appearance in the sequence. """


    codon_length = 3

    codon_sequences_list =
[input_seq[current_base:current_base+codon_length] for
current_base in range(0, len(input_seq), codon_length)]

    codon_no_seq_tupule = list(enumerate(codon_sequences_list))

    dict_of_AAs = {}
    DictConvert(codon_no_seq_tupule, dict_of_AAs)

    no_of_codons = int(len(input_seq)/3)
    codon_nos_all = list(range(0, no_of_codons , 1))

    for codon_no, codon_seq in dict_of_AAs.items():
        trans_codon = codon_seq.translate()
        #print(trans_codon)
        if codon_no in codon_nos_all:
            dict_of_AAs[codon_no] = trans_codon
    return dict_of_AAs
```

```
def codon_dict_maker(input_seq = None, key_format = "number"):
    """Converts a DNA sequence into a dictionary of the composite
codons.

    Keyword Arguments
    input_seq -- the DNA sequence to convert
    key_format -- 'number' gives keys as number in the sequence
(default), 'letter-number' gives the keys in the form amino acid
single letter code followed by the number in the sequence.

    Returns a dictionary with the desired format.
    """


    codon_length = 3

    codon_sequences_list =
[input_seq[current_base:current_base+codon_length] for
current_base in range(0, len(input_seq), codon_length)]

    codon_no_seq_tupule = list(enumerate(codon_sequences_list))

    #make a dictionary and convert the tupules into a dictionary
    dict_of_codons = {}

    if key_format == "number":
        DictConvert(codon_no_seq_tupule, dict_of_codons)

    if key_format == "letter-number":
        DictConvert(codon_no_seq_tupule, dict_of_codons)
        dict_of_codons2 = {}
        for codon_no, codon_seq in dict_of_codons.items():
            trans_codon = codon_seq.translate()
            trans_codon_name = str(trans_codon)
            codon_no_name = str(codon_no)
            codon_no_plus_name = trans_codon_name + codon_no_name

            dict_of_codons2[codon_no_plus_name] = codon_seq
        dict_of_codons = dict_of_codons2

    return dict_of_codons



def alt_codons(input_dict):

    """Creates a dictionary of the alternate codon sequences for
the same amino acid as the input.

    Arguments
    input_dict -- dictionary in the form {single-letter code +
number: original codon sequence}

    Outputs a dictionary in the form {single-letter code + number:
list of alternate codons}
```

```
    Note: Methionine and Tyrosine will output their only codon"""

    alt_codons_dict = {}

    for codon_name, codon_seq in input_dict.items():

        trans_codon = codon_name[0]
        #make a list of the codons for each AA minus the one that
was used in the WT
        current_AA_codon_list = list(ref_codon_table(trans_codon))
        alt_AA_codons_list = current_AA_codon_list
        alt_AA_codons_list.remove(codon_seq)
        alt_codons_dict[codon_name]= alt_AA_codons_list

    return alt_codons_dict






def Syn_random_recoder(input_dict):
    """Creates a dictionary of the a synonymous codon sequence for
the same amino acid as the input.

    Chosen codon will be randomly chosen from the alternate codons
for that amino acid.

    Arguments
    input_dict -- dictionary in the form {single-letter code +
number: list of alternate codon sequences}

    Outputs a dictionary in the form {single-letter code + number:
randomly chosen alternate codon}

    Note: Methionine and Tyrosine will output their only codon."""


    codons_for_mutated_seq = {}

    for codon_no_name, chosen_codon_seq in input_dict.items():

        chosen_AA = random.choice(chosen_codon_seq)
        codons_for_mutated_seq[codon_no_name] = chosen_AA

    return codons_for_mutated_seq
```

7.2.13 CODON DATAFRAMES

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jan 24 13:31:23 2024

@author: ceh560
"""

import pandas as pd
from Bio import SeqIO
from Bio import Seq
import numpy as np
from Bio import Align
import random
import codon_dictionaries as cdict



def ref_codon_table_freqs_excel(input_file = None):
    """Converts an excel spreadsheet of the frequency data into a
dictionary searchable by single-letter amino acid.

    Arguments
    input_file -- the filename of the reference spreadsheet with
file extension as a string

    Ouputs a dictionary in the form {single-letter code: data
frame of frequency data}"""


    codon_usage_df = pd.read_excel(input_file)

    Ala_df = codon_usage_df.query("Protein == 'A'")
    Cys_df = codon_usage_df.query("Protein == 'C'")
    Asp_df = codon_usage_df.query("Protein == 'D'")
    Glu_df = codon_usage_df.query("Protein == 'E'")
    Phe_df = codon_usage_df.query("Protein == 'F'")
    Gly_df = codon_usage_df.query("Protein == 'G'")
    His_df = codon_usage_df.query("Protein == 'H'")
    Ile_df = codon_usage_df.query("Protein == 'I'")
    Lys_df = codon_usage_df.query("Protein == 'K'")
    Leu_df = codon_usage_df.query("Protein == 'L'")
    Met_df = codon_usage_df.query("Protein == 'M'")
    Asn_df = codon_usage_df.query("Protein == 'N'")
    Pro_df = codon_usage_df.query("Protein == 'P'")
    Gln_df = codon_usage_df.query("Protein == 'Q'")
    Arg_df = codon_usage_df.query("Protein == 'R'")
    Ser_df = codon_usage_df.query("Protein == 'S'")
    Thr_df = codon_usage_df.query("Protein == 'T'")
    Val_df = codon_usage_df.query("Protein == 'V'")
    Trp_df = codon_usage_df.query("Protein == 'W'")
    Tyr_df = codon_usage_df.query("Protein == 'Y'")
    Stop_df = codon_usage_df.query("Protein == '*'")
    dict_of_AAs_dfs_all = {"A": Ala_df,
                           "C": Cys_df,
                           "D": Asp_df,
                           "E": Glu_df,
                           "F": Phe_df,
                           "G": Gly_df,
                           "H": His_df,
                           "I": Ile_df,
                           "K": Lys_df,
                           "L": Leu_df,
                           "M": Met_df,
                           "N": Asn_df,
                           "P": Pro_df,
                           "Q": Gln_df,
                           "R": Arg_df,
                           "S": Ser_df,
                           "T": Thr_df,
                           "V": Val_df,
                           "W": Trp_df,
                           "Y": Tyr_df,
                           "*": Stop_df}

    return dict_of_AAs_dfs_all


def ref_codon_table_freqs(input_df = None):
    """ Converts a dataframe of the frequency data into a 
dictionary, searchable by single-letter amino acid.

    Arguments
    input_file -- the filename of the reference spreadsheet with 
file extension as a string

    Ouputs a dictionary in the form {single-letter code: data 
frame of frequency data}"""


    codon_usage_df = input_df

    Ala_df = codon_usage_df.query("Protein == 'A'")
    Cys_df = codon_usage_df.query("Protein == 'C'")
    Asp_df = codon_usage_df.query("Protein == 'D'")
    Glu_df = codon_usage_df.query("Protein == 'E'")
    Phe_df = codon_usage_df.query("Protein == 'F'")
    Gly_df = codon_usage_df.query("Protein == 'G'")
    His_df = codon_usage_df.query("Protein == 'H'")
    Ile_df = codon_usage_df.query("Protein == 'I'")
    Lys_df = codon_usage_df.query("Protein == 'K'")
    Leu_df = codon_usage_df.query("Protein == 'L'")
    Met_df = codon_usage_df.query("Protein == 'M'")
    Asn_df = codon_usage_df.query("Protein == 'N'")
    Pro_df = codon_usage_df.query("Protein == 'P'")
    Gln_df = codon_usage_df.query("Protein == 'Q'")
    Arg_df = codon_usage_df.query("Protein == 'R'")
    Ser_df = codon_usage_df.query("Protein == 'S'")
    Thr_df = codon_usage_df.query("Protein == 'T'")
    Val_df = codon_usage_df.query("Protein == 'V'")
    Trp_df = codon_usage_df.query("Protein == 'W'")
    Tyr_df = codon_usage_df.query("Protein == 'Y'")
    Stop_df = codon_usage_df.query("Protein == '*'")



    dict_of_AAs_dfs_all = {"A": Ala_df,
                           "C": Cys_df,
                           "D": Asp_df,
                           "E": Glu_df,
                           "F": Phe_df,
                           "G": Gly_df,
                           "H": His_df,
                           "I": Ile_df,
                           "K": Lys_df,
                           "L": Leu_df,
                           "M": Met_df,
                           "N": Asn_df,
                           "P": Pro_df,
                           "Q": Gln_df,
                           "R": Arg_df,
                           "S": Ser_df,
                           "T": Thr_df,
                           "V": Val_df,
                           "W": Trp_df,
                           "Y": Tyr_df,
                           "*": Stop_df}

    return dict_of_AAs_dfs_all



def codon_frequency_collector(input_dict, reference_dict, type =
"value"):
    """Creates a dictionary with the frequencies of the codons
used in the input dictionary.

    Arguments
    input_dict -- a dictionary of the codons used in the sequence
to assess in the form (when type = dataframe or value) {codon
number: sequence} or (when type = list) {single-letter code +
number: list of alternate codons}
    reference_dict -- a dictionary of the frequency data for all
codons in the form {single-letter code: data frame of frequency
data}
    type -- choice of collection of only a single frequency (type
= value, default), (type = dataframe) the frequencies for all the
codons for the amino acid the input codes for as a dataframe, or
(type = list) the frequencies for all the codons for the amino
acid the input codes for as a list.

    Value outputs a dictionary in the form {single-letter code +
codon number: frequency value}
```

```
    Dataframe outputs a dictionary in the form {single-letter code
+ codon number: dataframe of frequencies for all codons}
    List outputs a dictionary in the form {single-letter code +
number: list of frequencies for all codons}
    """


    dict_of_codons_value = {}
    dict_of_codons_dataframe = {}
    dict_of_codons_list = {}

    if type == "value" or type == "dataframe":
        for codon_no, codon_seq in input_dict.items():
            #create the keys in the form AA single letter code +
codon number
            trans_codon = codon_seq.translate()
            trans_codon_name = str(trans_codon)
            codon_no_name = str(codon_no)
            codon_no = trans_codon_name + codon_no_name

            #find the data frame corrsponding to the relevant AA
            current_AA_df = reference_dict[trans_codon]

            #find the frequency value for the input sequence codon
            codon_seq = str(codon_seq)
            relevant_seq_freq_row =
current_AA_df.loc[current_AA_df["DNA"]==codon_seq]


            #collect only the frequency value
            relevant_seq_freq =
np.array(relevant_seq_freq_row["Fraction"])

            relevant_seq_freq = float(relevant_seq_freq)
            #relevant_seq_freq = np.vectorize(relevant_seq_freq)

            #add the frequency value to dictionary
            dict_of_codons_value[codon_no] = relevant_seq_freq


            #to a second dictionary, add the relevant data frames
(needed later)
            dict_of_codons_dataframe[codon_no] =
current_AA_df.copy()

    if type == "list":
        for codon_no_name, codon_seqs in input_dict.items():
            trans_codon = codon_no_name[0]
            current_AA_df = reference_dict[trans_codon]

            #collect only the frequency value
            relevant_seq_freq = list(current_AA_df["Fraction"])

            dict_of_codons_list[codon_no_name] = relevant_seq_freq
    if type == "dataframe":
        return dict_of_codons_dataframe

    if type == "value":
        return dict_of_codons_value

    if type == "list":
        return dict_of_codons_list




def codon_frequency_difference_calc(input_dict, ref_dict):
    """Creates a dictionary of the absolute frequency differences 
between an input codon and all other codons in a site-specific 
manner.

    Arguments
    input_codon_dict -- a dictionary of sequences to compare to in 
the form {single-letter code + number: sequence}
    ref_codon_dict -- a dictionary of dataframes of frequency 
usage data in the form {single-letter code: dataframe}

    Outputs a dictionary in the form {single-letter code + number: 
list of absolute differences in frequency"""

    alt_codon_dict = {}
    for codon_no_name, codon_seqs in input_dict.items():
        trans_codon = codon_no_name[0]
        current_AA_df = ref_dict[trans_codon]

        #collect only the frequency value
        relevant_seq_freq = list(current_AA_df["Fraction"])

        alt_codon_dict[codon_no_name] = relevant_seq_freq



    alt_codons_freqs_diff = {}
    for codon_no_name, codon_freqs in alt_codon_dict.items():
        #creating the keys as the codon number and translated 
letter

        ref_codon_seq = input_dict[codon_no_name]


        #find the dataframe for the relevant AA from the 
dictionary of dataframes
        trans_codon = ref_codon_seq.translate()
        current_AA_df = ref_dict[trans_codon]

        #find the triplet code to compare to from the previous 
dictionary with the DNA sequences from the input
        relevant_seq = input_dict[codon_no_name]
        ref_codon_freq = 
current_AA_df.loc[current_AA_df["DNA"]==relevant_seq]

        #collect only the frequency value from the data frame
        relevant_seq_freq = np.array(ref_codon_freq["Fraction"])
        #relevant_seq_freq = np.vectorize(relevant_seq_freq)
        relevant_seq_freq = float(relevant_seq_freq)

        #make a list of the values of the differences for each of 
the possible codons and add that to a dictionary which links these 
to their respective codon
        codon_freqs_diff = []

        for frequency in codon_freqs:
            frequency_diff = abs(frequency - relevant_seq_freq)

            codon_freqs_diff.append(frequency_diff)
            alt_codons_freqs_diff[codon_no_name] = 
codon_freqs_diff

    return alt_codons_freqs_diff


def codon_freq_diff_adder(dict_of_codons, dict_of_dfs, diff_dict):
    """Creates a dictionary with a modified data frame to the 
input to include absolute differences

    Arguments
   dict_of_codons -- a dictionary in the form of {single-letter 
code + number: sequence}
    dict_of_dfs -- a dictionary in the form of {single-letter code 
+ number: dataframe for all codons for that amino acid}
    diff_dict -- a dictionary in the form of {single-letter code + 
number: list of absolute differences}

    Note: values in the lists in diff_dict must be in the same 
order as the values they correspond to in the dataframe in 
input_dict

    Outputs a dictionary with the data frame from input_dict 
ammended with the values from the lists in diff_dict"""



    output_dict = dict_of_dfs
    #ensure the original codon can never be selected as the new 
one (except for Met and Tyr)
    for codon_no_name, codon_df in dict_of_dfs.items():
        #take the data frame which had the copies of each AAs info 
for the input sequence
        current_codon_df = dict_of_dfs[codon_no_name]

        #add a new column to the data frame which is the 
differences calculated in the previous dictionary
```

```
        current_codon_df["Absolute Difference"] =
diff_dict[codon_no_name]

        #pull out the original input sequence for the codon being
assessed
        #current_codon_no = int(codon_no_name[1:])
        input_codon = dict_of_codons[codon_no_name]


        #set the input codon freq to 1 for the input codon so that
it's never chosen as the lowest value except for met and tyr
        current_codon_df.loc[current_codon_df["DNA"] ==
input_codon, "Absolute Difference"] = 1

    return output_dict





def codon_freq_selector(input_dict):
    """Creates a dictionary of the codon sequences with the lowest
Absolute Difference in Frequency.

    Arguments
    input_dict -- a dictionary in the form {single-letter code +
number: dataframe} where the dataframe contains a column for DNA
sequence, Fraction and Absolute Difference.

    Output is a dictionary in the form {single-letter code +
number: chosen sequence (as a string)}"""


    codons_for_mutated_seq = {}

    for codon_no_name, chosen_codon_seq in input_dict.items():

        #find the dataframe for this codon
        current_codon_df = input_dict[codon_no_name]

        #find the value with the smallest absolute difference in
that dataframe
        smallest_diff = min(current_codon_df["Absolute
Difference"])

        corresponding_seq_to_freq =
current_codon_df.loc[current_codon_df["Absolute Difference"] ==
smallest_diff]


        #dealing with multiple equivalent differences - if found,
take the one with the bigger fraction of usage if they don't have
equal
        #pull out only the sequence of the smallest difference
        if len(corresponding_seq_to_freq) == 1:
            seq_to_use = corresponding_seq_to_freq["DNA"].item()
        else:
            highest_freq =
current_codon_df.loc[current_codon_df["Fraction"] ==
max(current_codon_df["Fraction"])]
            highest_freq_value = highest_freq["Fraction"].item()

            highest_freq_list = []
            highest_freq_list.append(highest_freq_value)


            #check if there are two values with equal abs diff and
equal fraction
            if len(highest_freq_list) == 1:
                seq_to_use = highest_freq["DNA"].item()

            #needs a better else clause
            else: print("\n\n\n ***Error: there are two or more
values with equal abs difference in frequency and fraction of
usage***\n\n\n")

        codons_for_mutated_seq[codon_no_name] = seq_to_use
    return codons_for_mutated_seq



def non_syn_mutator(target_AA, AA_num, new_AA, input_dict,
ref_dict = None, type = "random"):

    """Generates a dictionary of codons, replacing a target amino
acid with another either randomly or in a strategised manner.

    Arguments
    target_AA -- the starting amino acid residue in the WT
sequence (single-letter code)
    AA_num -- the amino acid number in the WT sequence
    new_AA -- the amino acid to replace the target with (single-
letter code)
    input_dict -- a dictionary of the WT sequence codons in the
form {single-letter code + number: sequence}
    ref_dict -- a reference dictionary of all codons. Either in
the form (type = "random") {single-letter code: list of sequences}
or (type = "highest" or "lowest") {single-letter code: dataframe
of frequency data}
    type -- determines how to pick the replacement codon. Default
= "random". Options are random, highest (highest frequency), or
lowest (lowest frequency)

    Output is a dictionary which has replaced the target codon as
specified in the form {single-letter code + number: sequence}
    """


    #use input codon information to identify which codon is going
to be mutated
```

```
    residue_to_mutate = target_AA + str(AA_num)

    #define the output residue
    mutated_target_residue = new_AA + str(AA_num)

    if type == "random":
        #determine DNA sequence for the new codon - random version

        replacement_protein_DNA = random.choice(ref_dict[new_AA])



    if type == "highest":
        mutated_res = {mutated_target_residue: "blank"}

        mutated_res_freqs = codon_frequency_collector(mutated_res,
ref_dict, type = "list")

        max_freq_mutated_res =
max(mutated_res_freqs[mutated_target_residue])

        mutated_res_df = ref_dict[new_AA]

        mutated_res_df_chosen =
mutated_res_df.loc[mutated_res_df["Fraction"] ==
max_freq_mutated_res]
        #tie breaker
        if len(mutated_res_df_chosen) > 1:
            max_number_codon = max(mutated_res_df["Number"])
            max_number_codon_seq =
mutated_res_df.loc[mutated_res_df["Number"] == max_number_codon,
"DNA"].item()
            replacement_protein_DNA = max_number_codon_seq

        else:
            replacement_protein_DNA =
mutated_res_df_chosen["DNA"].item()



    if type == "lowest":
        mutated_res = {mutated_target_residue: "blank"}

        mutated_res_freqs = codon_frequency_collector(mutated_res,
ref_dict, type = "list")

        min_freq_mutated_res =
min(mutated_res_freqs[mutated_target_residue])

        mutated_res_df = ref_dict[new_AA]

        mutated_res_df_chosen =
mutated_res_df.loc[mutated_res_df["Fraction"] ==
min_freq_mutated_res]
        #tie breaker
        if len(mutated_res_df_chosen) > 1:
            min_number_codon = min(mutated_res_df["Number"])
            min_number_codon_seq =
mutated_res_df.loc[mutated_res_df["Number"] == min_number_codon,
"DNA"].item()
            replacement_protein_DNA = min_number_codon_seq

        else:
            replacement_protein_DNA =
mutated_res_df_chosen["DNA"].item()


    #create a dictionary that has the new codon and removes the
old one

    dict_of_codons_output = {}

    for codon_name, codon_seq in input_dict.items():
        dict_of_codons_output[codon_name] = codon_seq
        dict_of_codons_output[mutated_target_residue] =
replacement_protein_DNA


    del dict_of_codons_output[residue_to_mutate]

    return dict_of_codons_output
```

7.2.14 FORMATTING FUNCTIONS

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jan 25 09:52:49 2024

@author: ceh560
"""

from Bio import Seq



def codon_spacing(sequence):
    """Adds a space every 3 bases to a sequence to visualise the 
codons more clearly.

    Arguments
    sequence -- a string or a sequence to be spaced

    Note: if the sequence length is not a multiple of 3, the 
spacing still starts from the beginning so the last codon will be 
incomplete. The sequence will also end on a space. """


    spaced_seq = Seq.Seq("")
    codon_length = 3

    for base in range(0, len(sequence), codon_length):

        spaced_seq += sequence[base:base+3]
        spaced_seq += " "

    return spaced_seq


def protein_align_codon(protein_sequence):
    """Adds space after every amino acid to align with a codon 
spaced DNA sequence.

    Arguments
    protein_sequence -- a string or a sequence to be spaced

    Note: if the sequence length is not a multiple of 3, the 
spacing still starts from the beginning so the last codon will be 
incomplete. The sequence will also end on a space. """


    spaced_seq = Seq.Seq("")

    for amino_acid in protein_sequence:
        #spaced_seq += " "
        spaced_seq += amino_acid
        spaced_seq += "   "

    return spaced_seq
```

7.2.15 STITCHING FUNCTIONS

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jan 23 10:57:41 2024

@author: ceh560
"""

from Bio import Seq


def repair_stitcher(*, ref_seq = None, recoded_seq = None, 
up_length = None, replacement_length = None, down_length = None):
    """"Takes an input sequence and replaces a central sequence 
with another specified sequence

    Keyword arguments:
        ref_seq -- a sequence/string to work from
        recoded_seq -- a sequence/string to replace part of the 
ref_seq with
        up_length -- the legnth of the first sequence from the 
start of the input sequence to keep
        replacement_length -- the length of the sequence which 
will be replaced, starting from the next base/character of the 
up_length
        down_length -- the length of the third sequence, starting 
from the next base/character of the replacement_length to keep

        Outputs a new sequence of the original starting sequence, 
followed by the replaced sequence, followed by the original end 
sequence
        """
    upstream_seq = ref_seq[:up_length]
    downstream_seq = ref_seq[(up_length + 
replacement_length):(up_length + replacement_length + 
down_length)]

    return upstream_seq + recoded_seq + downstream_seq




def sequence_splitter(*, ref_seq = None, up_length = None, 
mid_length = None, down_length = None):
    """"Takes an input sequence and breaks it into 3 constituent 
sequences

    Keyword arguments:
        ref_seq -- a sequence/string to split
        up_length -- the legnth of the first sequence from the 
start of the input sequence
        mid_length -- the length of the second sequence, starting 
from the next base/character of the up_length
        down_length -- the length of the third sequence, starting 
from the next base/character of the mid_length
```

```
        Outputs the sequences of start to up, up to mid and mid to
down either in a tupule or assigned to 3 variables if specified.
        """
    upstream_seq = ref_seq[:up_length]
    middle_seq = ref_seq[up_length:(up_length + mid_length)]
    downstream_seq = ref_seq[(up_length + mid_length):(up_length +
mid_length + down_length)]

    return upstream_seq, middle_seq, downstream_seq


def sequence_constructor(input_dict, type = "number"):

    """Constructs a sequence object from numbered sequences in a
dictionary

    Arguments
    input_dict -- a dictionary in the form {number: sequence} or
{single-letter code + number: sequence}
    type -- default = "number", alternative is type = "letter-
number"

    Output is a sequence constructed in numerical order from the
constituent sequences in the dictionary."""


    counter = 0

    mutated_seq = Seq.Seq("")

    if type == "letter-number":
        output_dict = {}

        for codon_no_name, chosen_codon_seq in input_dict.items():
            codon_no = codon_no_name[1:]
            output_dict[codon_no] = chosen_codon_seq

        for codon_no, chosen_codon_seq in output_dict.items():
            mutated_seq += output_dict[str(counter)]
            counter = counter + 1

    if type == "number":

        for codon_no_name, chosen_codon_seq in input_dict.items():


            mutated_seq += input_dict[str(counter)]
            counter = counter + 1


    return mutated_seq
```

```
def mut_seq_integrator(repair_seq, ref_seq, repair_start,
repair_end, WT_repair_seq = "No"):

    """Generates a sequence corresponding to the integration of
the inputted repair sequence into the gene sequence.

    Arguments

    repair_seq -- the sequence being integrated
    ref_seq -- the gene sequence before replacement
    repair start -- the starting base of where the repair template
integrates
    repair_end -- the ending base of where the repair template
integrates
    WT_repair_seq -- default = "No", if = Yes, it will also output
the sequence for the WT region that is being replaced

    Note: the repair_seq does not have to be the length of the
replaced region (repair_start to repair_end) but there is no
validation if this is the case."""

    #repair_start_py = repair_start - 1

    WT_template_seq = ref_seq[repair_start:repair_end]

    upstream_dna = ref_seq[:repair_start]

    downstream_dna = ref_seq[repair_end:]

    repair_total_seq = upstream_dna + repair_seq + downstream_dna

    if WT_repair_seq == "No":
        return repair_total_seq

    if WT_repair_seq == "Yes":
        return repair_total_seq, WT_template_seq
```

7.2.16 VALIDATOR

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jan 24 08:35:58 2024

@author: ceh560
"""

from Bio import Seq
from Bio import SeqIO

def triplet_checker(input_seq):
    """"Checks the input sequence is a whole number of codons
(divisible by 3). Returns True, or False + error message."""
    seq_length = len(input_seq)
    if seq_length % 3 == 0:
        return True
    else:
        return False, print("\n\n\n ***Sequence input not a
multiple of three*** \n Any result generated will likely be
erroneous.\n\n\n")




def translate_checker(input_seq, target_res_num, target_res_AA):


    if type(input_seq) == str:
        input_seq = Seq.Seq(input_seq)

    target_res_num_py = target_res_num - 1

    #determine the DNA sequence range from the codon number
    target_codon_seq =
input_seq[(target_res_num_py*3):((target_res_num_py*3)+3)]
    #return print(input_seq, target_codon_seq)

    #check the information matches up
    target_codon_seq_translated = target_codon_seq.translate()
    if target_codon_seq_translated != target_res_AA:
        return False, print(f"\n\n\n ***Error, requested residue
does not code for expected amino acid. Requested residue number
{target_res_num} codes for {target_codon_seq_translated} but was
expected to be {target_res_AA}.*** \n\n\n\n")

    if  target_codon_seq_translated == target_res_AA:
        return True




def mutation_counter(mutated_seq, WT_seq):
    """Determines the number of mutations in a DNA sequence
compared to a reference sequence
```

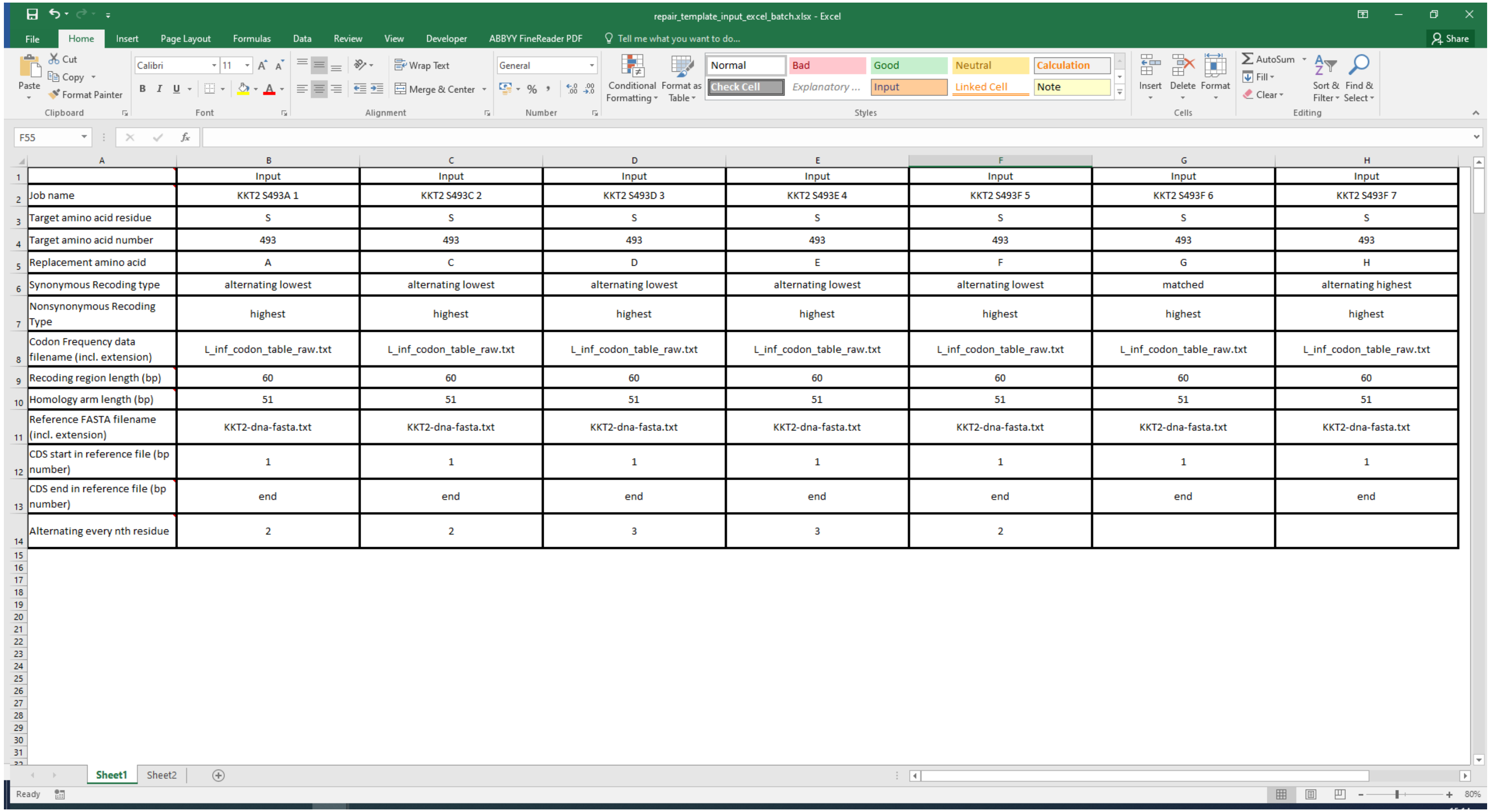
```
    Arguments
    mutated_seq -- the sequence which is expected to contain 
mutations
    WT_seq -- a reference sequence for the same region of DNA

    Outputs the number of mutations found"""



    counter = 0

    mutations = 0

    for base in WT_seq:
        if WT_seq[counter] != mutated_seq[counter]:
            mutations += 1

        counter += 1

    return mutations
```

7.2.17 PRIMER FUNCTIONS

```
# -*- coding: utf-8 -*-
"""
Created on Fri Jan 26 16:53:51 2024

@author: ceh560
"""

import primer3
import primer3.bindings
from Bio import SeqIO
from Bio import Seq
import pandas as pd



def screening_primer_designer(WT_seq, integrated_repair_seq,
repair_start, repair_end):
    """Designs screening primers to distinguish an integrated
repair sequence from the WT sequence.

    Argumets
    WT_seq -- the gene sequence of the WT gene
    integrated_repair_seq -- the gene sequence if the desired
repair template is integrated in place
    repair_start -- the base pair number that the recoding region
of the repair template starts
    repair_end -- the base pair number that the recoding region of
the repair template ends

    Outputs a dataframe of the designed primers and some useful
information"""

    repair_start_py = repair_start - 1
    WT_template_seq = WT_seq[repair_start_py:repair_end]

    upstream_dna = WT_seq[:repair_start_py]

    downstream_dna = WT_seq[repair_end:]



    WT_dna_for_primers = str(WT_seq)


    WT_primers = {'SEQUENCE_ID': "gene name",
            "SEQUENCE_TEMPLATE": WT_dna_for_primers,
            #"SEQUENCE_TARGET": [87,36], #first value = start,
second value = length, the primers must cover this entire region
            "PRIMER_TASK": "generic",
            "PRIMER_PICK_LEFT_PRIMER": 1,
            #"PRIMER_PICK_INTERNAL_OLIGO": 0,
            "PRIMER_PICK_RIGHT_PRIMER": 1,
            "PRIMER_OPT_SIZE": 18,
            "PRIMER_MIN_SIZE": 15,
            "PRIMER_MAX_SIZE": 21,
            "PRIMER_MAX_NS_ACCEPTED": 1,
            "PRIMER_PRODICT_SIZE_RANGE": [150,1500],
            "P3_FILE_FLAG": 1,
            #"SEQUENCE_INTERNAL_EXCLUDED_REGION": [37,21],
            "PRIMER_EXPLAIN_FLAG": 1,
            "SEQUENCE_PRIMER_PAIR_OK_REGION_LIST": [0,
repair_start_py, repair_start_py,(repair_end - repair_start_py)]
            #"SEQUENCE_PRIMER": "GTCACACTTTTGCGGCTCG" #allows you
to specify a left (fwd) primer only to use to design the right
(rev) primer
            }

    global_args1 = {"PRIMER_TASK": "generic",
                   "PRIMER_PICK_LEFT_PRIMER": 1,
                   "PRIMER_PICK_INTERNAL_OLIGO": 0,
                   "PRIMER_PICK_RIGHT_PRIMER": 1,
                   "PRIMER_NUM_RETURN": 3,
                   "PRIMER_MIN_3_PRIME_OVERLAP_OF_JUNCTION": 4,

"PRIMER_INTERNAL_MIN_3_PRIME_OVERLAP_OF_JUNCTION": 4,
                   "PRIMER_MIN_5_PRIME_OVERLAP_OF_JUNCTION": 7,

"PRIMER_INTERNAL_MIN_5_PRIME_OVERLAP_OF_JUNCTION": 7,
                   #"PRIMER_MUST_MATCH_FIVE_PRIME": "empty",
                   #"PRIMER_INTERNAL_MUST_MATCH_FIVE_PRIME":
"empty",
                   #"PRIMER_MUST_MATCH_THREE_PRIME": "empty",
                   #"PRIMER_INTERNAL_MUST_MATCH_THREE_PRIME":
"empty",
                   "PRIMER_PRODUCT_SIZE_RANGE": [100, 1500],
                   "PRIMER_PRODUCT_OPT_SIZE": 500,
                   "PRIMER_PAIR_WT_PRODUCT_SIZE_LT": 0.0,
                   "PRIMER_PAIR_WT_PRODUCT_SIZE_GT": 0.0,
                   "PRIMER_MIN_SIZE": 18,
                   "PRIMER_INTERNAL_MIN_SIZE": 18,
                   "PRIMER_OPT_SIZE": 20,
                   "PRIMER_INTERNAL_OPT_SIZE": 20,
                   "PRIMER_MAX_SIZE": 27,
                   "PRIMER_INTERNAL_MAX_SIZE": 27,
                   "PRIMER_WT_SIZE_LT": 1.0,
                   "PRIMER_INTERNAL_WT_SIZE_LT": 1.0,
                   "PRIMER_WT_SIZE_GT": 1.0,
                   "PRIMER_INTERNAL_WT_SIZE_GT": 1.0,
                   "PRIMER_MIN_GC": 20.0,
                   "PRIMER_INTERNAL_MIN_GC": 20.0,
                   "PRIMER_OPT_GC_PERCENT": 50.0,
                   "PRIMER_INTERNAL_OPT_GC_PERCENT": 50.0,
                   "PRIMER_MAX_GC": 80.0,
                   "PRIMER_INTERNAL_MAX_GC": 80.0,
                   "PRIMER_WT_GC_PERCENT_LT": 0.0,
                   "PRIMER_INTERNAL_WT_GC_PERCENT_LT": 0.0,
                   "PRIMER_WT_GC_PERCENT_GT": 0.0,
                   "PRIMER_INTERNAL_WT_GC_PERCENT_GT": 0.0,
                   "PRIMER_GC_CLAMP": 0,
                   "PRIMER_MAX_END_GC": 5,
                   "PRIMER_MIN_TM": 56.0,
                   "PRIMER_INTERNAL_MIN_TM": 56.0,
                   "PRIMER_OPT_TM": 60.0,
                   "PRIMER_INTERNAL_OPT_TM": 60.0,
                   "PRIMER_MAX_TM": 63.0,
                   "PRIMER_INTERNAL_MAX_TM": 63.0,
                   "PRIMER_PAIR_MAX_DIFF_TM": 63.0,
                   "PRIMER_WT_TM_LT": 1.0,
                   "PRIMER_INTERNAL_WT_TM_LT": 1.0,
                   "PRIMER_WT_TM_GT": 1.0,
                   "PRIMER_INTERNAL_WT_TM_GT": 1.0,
                   "PRIMER_PAIR_WT_DIFF_TM": 0.0,
                   "PRIMER_PRODUCT_MIN_TM": -1000000.0,
                   "PRIMER_PRODUCT_OPT_TM": 0.0,
                   "PRIMER_PRODUCT_MAX_TM": 1000000.0,
                   "PRIMER_PAIR_WT_PRODUCT_TM_LT": 0.0,
                   "PRIMER_PAIR_WT_PRODUCT_TM_GT": 0.0,
                   "PRIMER_TM_FORMULA": 1,
                   "PRIMER_SALT_MONOVALENT": 50.0,
                   "PRIMER_INTERNAL_SALT_MONOVALENT": 50.0,
                   "PRIMER_SALT_DIVALENT": 1.5,
                   "PRIMER_INTERNAL_SALT_DIVALENT": 0.0,
                   "PRIMER_DNTP_CONC": 0.6,
                   "PRIMER_INTERNAL_DNTP_CONC": 0.0,
                   "PRIMER_SALT_CORRECTIONS": 1,
                   "PRIMER_DNA_CONC": 50.0,
                   "PRIMER_INTERNAL_DNA_CONC": 50.0,
                   "PRIMER_DMSO_CONC": 0.0,
                   "PRIMER_INTERNAL_DMSO_CONC": 0.0,
                   "PRIMER_DMSO_FACTOR": 0.6,
                   "PRIMER_INTERNAL_DMSO_FACTOR": 0.6,
                   "PRIMER_FORMAMIDE_CONC": 0.0,
                   "PRIMER_INTERNAL_FORMAMIDE_CONC": 0.0,
                   "PRIMER_THERMODYNAMIC_OLIGO_ALIGNMENT": 1,
                   "PRIMER_THERMODYNAMIC_TEMPLATE_ALIGNMENT": 0,
                   "PRIMER_SECONDARY_STRUCTURE_ALIGNMENT": 0,
                   "PRIMER_THERMODYNAMIC_PARAMETERS_PATH":
"./primer3_config",
                   "PRIMER_ANNEALING_TEMP": -10.0,
                   "PRIMER_MIN_BOUND": -10.0,
                   "PRIMER_INTERNAL_MIN_BOUND": -10.0,
                   "PRIMER_OPT_BOUND": 97.0,
                   "PRIMER_INTERNAL_OPT_BOUND": 97.0,
                   "PRIMER_MAX_BOUND": 110.0,
                   "PRIMER_INTERNAL_MAX_BOUND": 110.0,
                   "PRIMER_WT_BOUND_LT": 0.0,
                   "PRIMER_INTERNAL_WT_BOUND_LT": 0.0,
                   "PRIMER_WT_BOUND_GT": 0.0,
                   "PRIMER_INTERNAL_WT_BOUND_GT": 0.0,
                   "PRIMER_MAX_SELF_ANY": 8.00,
                   "PRIMER_MAX_SELF_ANY_TH": 47.00,
                   "PRIMER_INTERNAL_MAX_SELF_ANY": 12.00,
                   "PRIMER_INTERNAL_MAX_SELF_ANY_TH": 47.00,
                   "PRIMER_PAIR_MAX_COMPL_ANY": 8.00,
                   "PRIMER_PAIR_MAX_COMPL_ANY_TH": 47.00,
                   "PRIMER_WT_SELF_ANY": 0.0,
                   "PRIMER_WT_SELF_ANY_TH": 0.0,
                   "PRIMER_INTERNAL_WT_SELF_ANY": 0.0,
                   "PRIMER_INTERNAL_WT_SELF_ANY_TH": 0.0,
                   "PRIMER_PAIR_WT_COMPL_ANY": 0.0,
                   "PRIMER_PAIR_WT_COMPL_ANY_TH": 0.0,
                   "PRIMER_MAX_SELF_END": 3.00,
                   "PRIMER_MAX_SELF_END_TH": 47.00,
                   "PRIMER_INTERNAL_MAX_SELF_END": 12.00,
                   "PRIMER_INTERNAL_MAX_SELF_END_TH": 47.00,
                   "PRIMER_PAIR_MAX_COMPL_END": 3.00,
                   "PRIMER_PAIR_MAX_COMPL_END_TH": 47.00,
                   "PRIMER_WT_SELF_END": 0.0,
                   "PRIMER_WT_SELF_END_TH": 0.0,
                   "PRIMER_INTERNAL_WT_SELF_END": 0.0,
                   "PRIMER_INTERNAL_WT_SELF_END_TH": 0.0,
                   "PRIMER_PAIR_WT_COMPL_END": 0.0,
                   "PRIMER_PAIR_WT_COMPL_END_TH": 0.0,
                   "PRIMER_MAX_HAIRPIN_TH": 47.0,
                   "PRIMER_INTERNAL_MAX_HAIRPIN_TH": 47.0,
                   "PRIMER_WT_HAIRPIN_TH": 0.0,
                   "PRIMER_INTERNAL_WT_HAIRPIN_TH": 0.0,
                   "PRIMER_MAX_END_STABILITY": 100.0,
                   "PRIMER_WT_END_STABILITY": 0.0,
                   "PRIMER_MAX_NS_ACCEPTED": 0,
                   "PRIMER_INTERNAL_MAX_NS_ACCEPTED": 0,
                   "PRIMER_WT_NUM_NS": 0.0,
                   "PRIMER_INTERNAL_WT_NUM_NS": 0.0,
                   "PRIMER_MAX_POLY_X": 5,
                   "PRIMER_INTERNAL_MAX_POLY_X": 5,
                   #"PRIMER_MIN_LEFT_THREE_PRIME_DISTANCE": -1,
                   #"PRIMER_INTERNAL_MIN_THREE_PRIME_DISTANCE": -
1,

                   #"PRIMER_MIN_RIGHT_THREE_PRIME_DISTANCE": -1,
                   "PRIMER_MIN_THREE_PRIME_DISTANCE": -1,
                   "PRIMER_PICK_ANYWAY": 0,
                   "PRIMER_LOWERCASE_MASKING": 0,
                   "PRIMER_EXPLAIN_FLAG": 0,
                   "PRIMER_LIBERAL_BASE": 0,
                   "PRIMER_FIRST_BASE_INDEX": 0,
                   "PRIMER_MAX_TEMPLATE_MISPRIMING": -1.00,
                   "PRIMER_MAX_TEMPLATE_MISPRIMING_TH": -1.00,
                   "PRIMER_PAIR_MAX_TEMPLATE_MISPRIMING": -1.00,
                   "PRIMER_PAIR_MAX_TEMPLATE_MISPRIMING_TH": -
1.00,

                   "PRIMER_WT_TEMPLATE_MISPRIMING": 0.0,
                   "PRIMER_WT_TEMPLATE_MISPRIMING_TH": 0.0,
                   "PRIMER_PAIR_WT_TEMPLATE_MISPRIMING": 0.0,
                   "PRIMER_PAIR_WT_TEMPLATE_MISPRIMING_TH": 0.0,
                   "PRIMER_MISPRIMING_LIBRARY": "",
                   "PRIMER_INTERNAL_MISHYB_LIBRARY": "",
                   "PRIMER_LIB_AMBIGUITY_CODES_CONSENSUS": 0,
                   "PRIMER_MAX_LIBRARY_MISPRIMING": 12.00,
                   "PRIMER_INTERNAL_MAX_LIBRARY_MISHYB": 12.00,
                   "PRIMER_PAIR_MAX_LIBRARY_MISPRIMING": 24.00,
                   "PRIMER_WT_LIBRARY_MISPRIMING": 0.0,
                   "PRIMER_INTERNAL_WT_LIBRARY_MISHYB": 0.0,
                   "PRIMER_PAIR_WT_LIBRARY_MISPRIMING": 0.0,
                   "PRIMER_MASK_TEMPLATE": 0,
                   "PRIMER_MASK_FAILURE_RATE": 0.1,
                   "PRIMER_WT_MASK_FAILURE_RATE": 0.0,
                   "PRIMER_MASK_5P_DIRECTION": 1,
                   "PRIMER_MASK_3P_DIRECTION": 0,
                   #"PRIMER_MASK_KMERLIST_PATH": "../kmer_lists/",
                   "PRIMER_MASK_KMERLIST_PREFIX": "homo_sapiens",
                   "PRIMER_MIN_QUALITY": 0,
                   "PRIMER_INTERNAL_MIN_QUALITY": 0,
                   "PRIMER_MIN_END_QUALITY": 0,
                   "PRIMER_QUALITY_RANGE_MIN": 0,
                   "PRIMER_QUALITY_RANGE_MAX": 100,
                   "PRIMER_WT_SEQ_QUAL": 0.0,
                   "PRIMER_INTERNAL_WT_SEQ_QUAL": 0.0,
                   "PRIMER_PAIR_WT_PR_PENALTY": 1.0,
                   "PRIMER_PAIR_WT_IO_PENALTY": 0.0,
                   "PRIMER_INSIDE_PENALTY": -1.0,
                   "PRIMER_OUTSIDE_PENALTY": 0.0,
                   "PRIMER_WT_POS_PENALTY": 1.0,
                   "PRIMER_SEQUENCING_LEAD": 50,
                   "PRIMER_SEQUENCING_SPACING": 500,
                   "PRIMER_SEQUENCING_INTERVAL": 250,
                   "PRIMER_SEQUENCING_ACCURACY": 20,
                   "PRIMER_WT_END_QUAL": 0.0,
                   "PRIMER_INTERNAL_WT_END_QUAL": 0.0
                   }

    WT_primers_results = primer3.design_primers(seq_args =
WT_primers, global_args = global_args1)



    upstream_fwd = WT_primers_results["PRIMER_LEFT_0_SEQUENCE"]
    WT_rev = WT_primers_results["PRIMER_RIGHT_0_SEQUENCE"]
    WT_PCR_product_size =
WT_primers_results["PRIMER_PAIR_0_PRODUCT_SIZE"]
    upstream_fwd_coords = WT_primers_results["PRIMER_LEFT_0"]
    WT_rev_coords = WT_primers_results["PRIMER_RIGHT_0"]

    #sticking some useful info into a smaller dictionary in case
it comes in handy later
    WT_primers_useful_results = {
        "Forward primer sequence": upstream_fwd,
        "Reverse primer sequence": WT_rev,
        "PCR product size (bp)": WT_PCR_product_size,
        "Forward GC content (%)":
WT_primers_results["PRIMER_LEFT_0_GC_PERCENT"],
        "Reverse GC content (%)":
WT_primers_results["PRIMER_RIGHT_0_GC_PERCENT"],
        "Forward Tm ('C)": WT_primers_results["PRIMER_LEFT_0_TM"],
        "Reverse Tm ('C)": WT_primers_results["PRIMER_RIGHT_0_TM"]
        #"WT PCR product sequence (5'->3')":
WT_dna_for_primers[upstream_fwd_coords[0]:(WT_rev_coords[0] + 1)]
        }
    #part 4 - design the mutant primers

    recoded_dna_for_primers = str(integrated_repair_seq)

    recoded_primers = {'SEQUENCE_ID': "gene name",
            "SEQUENCE_TEMPLATE": recoded_dna_for_primers,
            #"SEQUENCE_TARGET": [87,36], #first value = start,
second value = length, the primers must cover this entire region
            "PRIMER_TASK": "generic",
            "PRIMER_PICK_LEFT_PRIMER": 1,
            #"PRIMER_PICK_INTERNAL_OLIGO": 0,
            "PRIMER_PICK_RIGHT_PRIMER": 1,
            "PRIMER_OPT_SIZE": 18,
            "PRIMER_MIN_SIZE": 15,
            "PRIMER_MAX_SIZE": 22,
            "PRIMER_MAX_NS_ACCEPTED": 1,
            "PRIMER_PRODICT_SIZE_RANGE": [150,1500],
            "P3_FILE_FLAG": 1,
            #"SEQUENCE_INTERNAL_EXCLUDED_REGION": [37,21],
            "PRIMER_EXPLAIN_FLAG": 1,
            "SEQUENCE_PRIMER_PAIR_OK_REGION_LIST": [0,
repair_start_py, repair_start_py,(repair_end - repair_start_py)],
            "SEQUENCE_PRIMER": upstream_fwd #allows you to specify
a left (fwd) primer only to use to design the right (rev) primer
            }

    recoded_primers_results = primer3.design_primers(seq_args =
recoded_primers, global_args = global_args1)


    recoded_rev =
recoded_primers_results["PRIMER_RIGHT_0_SEQUENCE"]
    recoded_PCR_product_size =
recoded_primers_results["PRIMER_PAIR_0_PRODUCT_SIZE"]
    recoded_rev_coords = recoded_primers_results["PRIMER_RIGHT_0"]

    recoded_primers_useful_results = {
        "Forward primer sequence": upstream_fwd,
        "Reverse primer sequence": recoded_rev,
        "PCR product size (bp)": recoded_PCR_product_size,
        "Forward GC content (%)":
recoded_primers_results["PRIMER_LEFT_0_GC_PERCENT"],
        "Reverse GC content (%)":
recoded_primers_results["PRIMER_RIGHT_0_GC_PERCENT"],
        "Forward Tm ('C)":
recoded_primers_results["PRIMER_LEFT_0_TM"],
        "Reverse Tm ('C)":
recoded_primers_results["PRIMER_RIGHT_0_TM"],
        #"Recoded PCR product sequence (5'->3')":
recoded_dna_for_primers[upstream_fwd_coords[0]:(recoded_rev_coords
[0] + 1)]
        }
    primers_to_add = [WT_primers_useful_results,
recoded_primers_useful_results]
    primer_details = pd.DataFrame(primers_to_add)

    row_names = ["WT primers", "Repair primers"]
    primer_details.index = row_names

    return primer_details



def repair_primer_designer(repair_seq, hom_arm_length,
downstream_seq):
    """Designs primers to produce the repair template sequence put
in.

    Arguments
    repair_seq -- a DNA sequence of the entire repair template.
Must be less than/equal to 220 bp.
    hom_arm_length -- the length of the homology arms in the
repair template
    downstream_dna -- a DNA sequence, essentially a dummy but
ideally sequence from the same organism. Needs to be larger than
the repair sequence length.

    Outputs a dictionary of the necessary primer sequences, and
some other useful information."""



    recoding_start_base = hom_arm_length
    recoding_end_base = len(repair_seq) - hom_arm_length


    recoding_start_base_py = recoding_start_base - 1


    repair_length_total = len(repair_seq)
    if repair_length_total > 220:
        return print("\n\n\n***ERROR: repair length is too long to
deisgn primers for***\n\n\n")

    if recoding_end_base > 120:
        annealing_region_end = 119

        recoding_adjustment = recoding_end_base -
annealing_region_end

        left_primer_length_from_start = annealing_region_end -
hom_arm_length - recoding_adjustment
        recoding_start_base_py = recoding_start_base_py +
recoding_adjustment

    else:
        annealing_region_end = recoding_end_base
        left_primer_length_from_start = annealing_region_end -
hom_arm_length



    repair_dna_for_primers = str(repair_seq)+str(downstream_seq)
    primers_end = len(downstream_seq)

    annealing = {'SEQUENCE_ID': "gene name",
            "SEQUENCE_TEMPLATE": repair_dna_for_primers,
            #"SEQUENCE_TARGET": [87,36], #first value = start,
second value = length, the primers must cover this entire region
            "PRIMER_TASK": "generic",
            "PRIMER_PICK_LEFT_PRIMER": 3,
            #"PRIMER_PICK_INTERNAL_OLIGO": 0,
            "PRIMER_PICK_RIGHT_PRIMER": 3,
            "PRIMER_OPT_SIZE": 18,
            "PRIMER_MIN_SIZE": 15,
            "PRIMER_MAX_SIZE": 24,
            "PRIMER_MAX_NS_ACCEPTED": 1,
            "PRIMER_PRODICT_SIZE_RANGE": [100,1000],
            "P3_FILE_FLAG": 1,
            #"SEQUENCE_INTERNAL_EXCLUDED_REGION": [37,21],
            "PRIMER_EXPLAIN_FLAG": 1,
            "SEQUENCE_PRIMER_PAIR_OK_REGION_LIST":
[recoding_start_base_py, left_primer_length_from_start,
annealing_region_end, primers_end]
            #"SEQUENCE_PRIMER": "GTCACACTTTTGCGGCTCG" #allows you
to specify a left (fwd) primer only to use to design the right
(rev) primer
            }

    global_args1 = {"PRIMER_TASK": "generic",
                   "PRIMER_PICK_LEFT_PRIMER": 3,
                   "PRIMER_PICK_INTERNAL_OLIGO": 0,
                   "PRIMER_PICK_RIGHT_PRIMER": 3,
                   "PRIMER_NUM_RETURN": 3,
                   "PRIMER_MIN_3_PRIME_OVERLAP_OF_JUNCTION": 4,

"PRIMER_INTERNAL_MIN_3_PRIME_OVERLAP_OF_JUNCTION": 4,
                   "PRIMER_MIN_5_PRIME_OVERLAP_OF_JUNCTION": 7,

"PRIMER_INTERNAL_MIN_5_PRIME_OVERLAP_OF_JUNCTION": 7,
                   #"PRIMER_MUST_MATCH_FIVE_PRIME": "empty",
                   #"PRIMER_INTERNAL_MUST_MATCH_FIVE_PRIME":
"empty",
                   #"PRIMER_MUST_MATCH_THREE_PRIME": "empty",
                   #"PRIMER_INTERNAL_MUST_MATCH_THREE_PRIME":
"empty",
                   "PRIMER_PRODUCT_SIZE_RANGE": [100, 1000],
                   "PRIMER_PRODUCT_OPT_SIZE": 0,
                   "PRIMER_PAIR_WT_PRODUCT_SIZE_LT": 0.0,
                   "PRIMER_PAIR_WT_PRODUCT_SIZE_GT": 0.0,
                   "PRIMER_MIN_SIZE": 16,
                   "PRIMER_INTERNAL_MIN_SIZE": 16,
                   "PRIMER_OPT_SIZE": 20,
"PRIMER_INTERNAL_OPT_SIZE": 20,
"PRIMER_MAX_SIZE": 27,
"PRIMER_INTERNAL_MAX_SIZE": 27,
"PRIMER_WT_SIZE_LT": 1.0,
"PRIMER_INTERNAL_WT_SIZE_LT": 1.0,
"PRIMER_WT_SIZE_GT": 1.0,
"PRIMER_INTERNAL_WT_SIZE_GT": 1.0,
"PRIMER_MIN_GC": 20.0,
"PRIMER_INTERNAL_MIN_GC": 20.0,
"PRIMER_OPT_GC_PERCENT": 50.0,
"PRIMER_INTERNAL_OPT_GC_PERCENT": 50.0,
"PRIMER_MAX_GC": 80.0,
"PRIMER_INTERNAL_MAX_GC": 80.0,
"PRIMER_WT_GC_PERCENT_LT": 0.0,
"PRIMER_INTERNAL_WT_GC_PERCENT_LT": 0.0,
"PRIMER_WT_GC_PERCENT_GT": 0.0,
"PRIMER_INTERNAL_WT_GC_PERCENT_GT": 0.0,
"PRIMER_GC_CLAMP": 0,
"PRIMER_MAX_END_GC": 5,
"PRIMER_MIN_TM": 55.0,
"PRIMER_INTERNAL_MIN_TM": 55.0,
"PRIMER_OPT_TM": 60.0,
"PRIMER_INTERNAL_OPT_TM": 60.0,
"PRIMER_MAX_TM": 67.0,
"PRIMER_INTERNAL_MAX_TM": 67.0,
"PRIMER_PAIR_MAX_DIFF_TM": 67.0,
"PRIMER_WT_TM_LT": 1.0,
"PRIMER_INTERNAL_WT_TM_LT": 1.0,
"PRIMER_WT_TM_GT": 1.0,
"PRIMER_INTERNAL_WT_TM_GT": 1.0,
"PRIMER_PAIR_WT_DIFF_TM": 0.0,
"PRIMER_PRODUCT_MIN_TM": -1000000.0,
"PRIMER_PRODUCT_OPT_TM": 0.0,
"PRIMER_PRODUCT_MAX_TM": 1000000.0,
"PRIMER_PAIR_WT_PRODUCT_TM_LT": 0.0,
"PRIMER_PAIR_WT_PRODUCT_TM_GT": 0.0,
"PRIMER_TM_FORMULA": 1,
"PRIMER_SALT_MONOVALENT": 50.0,
"PRIMER_INTERNAL_SALT_MONOVALENT": 50.0,
"PRIMER_SALT_DIVALENT": 1.5,
"PRIMER_INTERNAL_SALT_DIVALENT": 0.0,
"PRIMER_DNTP_CONC": 0.6,
"PRIMER_INTERNAL_DNTP_CONC": 0.0,
"PRIMER_SALT_CORRECTIONS": 1,
"PRIMER_DNA_CONC": 50.0,
"PRIMER_INTERNAL_DNA_CONC": 50.0,
"PRIMER_DMSO_CONC": 0.0,
"PRIMER_INTERNAL_DMSO_CONC": 0.0,
"PRIMER_DMSO_FACTOR": 0.6,
"PRIMER_INTERNAL_DMSO_FACTOR": 0.6,
"PRIMER_FORMAMIDE_CONC": 0.0,
"PRIMER_INTERNAL_FORMAMIDE_CONC": 0.0,
"PRIMER_THERMODYNAMIC_OLIGO_ALIGNMENT": 1,
"PRIMER_THERMODYNAMIC_TEMPLATE_ALIGNMENT": 0,
"PRIMER_SECONDARY_STRUCTURE_ALIGNMENT": 0,
                   "PRIMER_THERMODYNAMIC_PARAMETERS_PATH":
"./primer3_config",
                   "PRIMER_ANNEALING_TEMP": -10.0,
                   "PRIMER_MIN_BOUND": -10.0,
                   "PRIMER_INTERNAL_MIN_BOUND": -10.0,
                   "PRIMER_OPT_BOUND": 97.0,
                   "PRIMER_INTERNAL_OPT_BOUND": 97.0,
                   "PRIMER_MAX_BOUND": 110.0,
                   "PRIMER_INTERNAL_MAX_BOUND": 110.0,
                   "PRIMER_WT_BOUND_LT": 0.0,
                   "PRIMER_INTERNAL_WT_BOUND_LT": 0.0,
                   "PRIMER_WT_BOUND_GT": 0.0,
                   "PRIMER_INTERNAL_WT_BOUND_GT": 0.0,
                   "PRIMER_MAX_SELF_ANY": 8.00,
                   "PRIMER_MAX_SELF_ANY_TH": 47.00,
                   "PRIMER_INTERNAL_MAX_SELF_ANY": 12.00,
                   "PRIMER_INTERNAL_MAX_SELF_ANY_TH": 47.00,
                   "PRIMER_PAIR_MAX_COMPL_ANY": 8.00,
                   "PRIMER_PAIR_MAX_COMPL_ANY_TH": 47.00,
                   "PRIMER_WT_SELF_ANY": 0.0,
                   "PRIMER_WT_SELF_ANY_TH": 0.0,
                   "PRIMER_INTERNAL_WT_SELF_ANY": 0.0,
                   "PRIMER_INTERNAL_WT_SELF_ANY_TH": 0.0,
                   "PRIMER_PAIR_WT_COMPL_ANY": 0.0,
                   "PRIMER_PAIR_WT_COMPL_ANY_TH": 0.0,
                   "PRIMER_MAX_SELF_END": 3.00,
                   "PRIMER_MAX_SELF_END_TH": 47.00,
                   "PRIMER_INTERNAL_MAX_SELF_END": 12.00,
                   "PRIMER_INTERNAL_MAX_SELF_END_TH": 47.00,
                   "PRIMER_PAIR_MAX_COMPL_END": 3.00,
                   "PRIMER_PAIR_MAX_COMPL_END_TH": 47.00,
                   "PRIMER_WT_SELF_END": 0.0,
                   "PRIMER_WT_SELF_END_TH": 0.0,
                   "PRIMER_INTERNAL_WT_SELF_END": 0.0,
                   "PRIMER_INTERNAL_WT_SELF_END_TH": 0.0,
                   "PRIMER_PAIR_WT_COMPL_END": 0.0,
                   "PRIMER_PAIR_WT_COMPL_END_TH": 0.0,
                   "PRIMER_MAX_HAIRPIN_TH": 47.0,
                   "PRIMER_INTERNAL_MAX_HAIRPIN_TH": 47.0,
                   "PRIMER_WT_HAIRPIN_TH": 0.0,
                   "PRIMER_INTERNAL_WT_HAIRPIN_TH": 0.0,
                   "PRIMER_MAX_END_STABILITY": 100.0,
                   "PRIMER_WT_END_STABILITY": 0.0,
                   "PRIMER_MAX_NS_ACCEPTED": 0,
                   "PRIMER_INTERNAL_MAX_NS_ACCEPTED": 0,
                   "PRIMER_WT_NUM_NS": 0.0,
                   "PRIMER_INTERNAL_WT_NUM_NS": 0.0,
                   "PRIMER_MAX_POLY_X": 5,
                   "PRIMER_INTERNAL_MAX_POLY_X": 5,
                   #"PRIMER_MIN_LEFT_THREE_PRIME_DISTANCE": -1,
                   #"PRIMER_INTERNAL_MIN_THREE_PRIME_DISTANCE": -
1,
                   #"PRIMER_MIN_RIGHT_THREE_PRIME_DISTANCE": -1,
                   "PRIMER_MIN_THREE_PRIME_DISTANCE": -1,
                   "PRIMER_PICK_ANYWAY": 0,
                   "PRIMER_LOWERCASE_MASKING": 0,
                   "PRIMER_EXPLAIN_FLAG": 0,
                   "PRIMER_LIBERAL_BASE": 0,
                   "PRIMER_FIRST_BASE_INDEX": 0,
                   "PRIMER_MAX_TEMPLATE_MISPRIMING": -1.00,
                   "PRIMER_MAX_TEMPLATE_MISPRIMING_TH": -1.00,
                   "PRIMER_PAIR_MAX_TEMPLATE_MISPRIMING": -1.00,
                   "PRIMER_PAIR_MAX_TEMPLATE_MISPRIMING_TH": -
1.00,
                   "PRIMER_WT_TEMPLATE_MISPRIMING": 0.0,
                   "PRIMER_WT_TEMPLATE_MISPRIMING_TH": 0.0,
                   "PRIMER_PAIR_WT_TEMPLATE_MISPRIMING": 0.0,
                   "PRIMER_PAIR_WT_TEMPLATE_MISPRIMING_TH": 0.0,
                   "PRIMER_MISPRIMING_LIBRARY": "",
                   "PRIMER_INTERNAL_MISHYB_LIBRARY": "",
                   "PRIMER_LIB_AMBIGUITY_CODES_CONSENSUS": 0,
                   "PRIMER_MAX_LIBRARY_MISPRIMING": 12.00,
                   "PRIMER_INTERNAL_MAX_LIBRARY_MISHYB": 12.00,
                   "PRIMER_PAIR_MAX_LIBRARY_MISPRIMING": 24.00,
                   "PRIMER_WT_LIBRARY_MISPRIMING": 0.0,
                   "PRIMER_INTERNAL_WT_LIBRARY_MISHYB": 0.0,
                   "PRIMER_PAIR_WT_LIBRARY_MISPRIMING": 0.0,
                   "PRIMER_MASK_TEMPLATE": 0,
                   "PRIMER_MASK_FAILURE_RATE": 0.1,
                   "PRIMER_WT_MASK_FAILURE_RATE": 0.0,
                   "PRIMER_MASK_5P_DIRECTION": 1,
                   "PRIMER_MASK_3P_DIRECTION": 0,
                   #"PRIMER_MASK_KMERLIST_PATH": "../kmer_lists/",
                   "PRIMER_MASK_KMERLIST_PREFIX": "homo_sapiens",
                   "PRIMER_MIN_QUALITY": 0,
                   "PRIMER_INTERNAL_MIN_QUALITY": 0,
                   "PRIMER_MIN_END_QUALITY": 0,
                   "PRIMER_QUALITY_RANGE_MIN": 0,
                   "PRIMER_QUALITY_RANGE_MAX": 100,
                   "PRIMER_WT_SEQ_QUAL": 0.0,
                   "PRIMER_INTERNAL_WT_SEQ_QUAL": 0.0,
                   "PRIMER_PAIR_WT_PR_PENALTY": 1.0,
                   "PRIMER_PAIR_WT_IO_PENALTY": 0.0,
                   "PRIMER_INSIDE_PENALTY": -1.0,
                   "PRIMER_OUTSIDE_PENALTY": 0.0,
                   "PRIMER_WT_POS_PENALTY": 1.0,
                   "PRIMER_SEQUENCING_LEAD": 50,
                   "PRIMER_SEQUENCING_SPACING": 500,
                   "PRIMER_SEQUENCING_INTERVAL": 250,
                   "PRIMER_SEQUENCING_ACCURACY": 20,
                   "PRIMER_WT_END_QUAL": 0.0,
                   "PRIMER_INTERNAL_WT_END_QUAL": 0.0
                   }
    repair_primers = primer3.design_primers(seq_args = annealing,
global_args = global_args1)

    annealing_seq = repair_primers["PRIMER_LEFT_0_SEQUENCE"]
    annealing_tm = repair_primers["PRIMER_LEFT_0_TM"]
    annealing_coords = repair_primers["PRIMER_LEFT_0"]
    forward_primer = str(repair_seq[:(annealing_coords[0] +
annealing_coords[1])])

    repair_as_seq = Seq.Seq(repair_seq)
    repair_rc = repair_as_seq.reverse_complement()
    repair_rc_str = str(repair_rc)
    reverse_primer = repair_rc_str[0:(len(repair_seq) -
annealing_coords[0])]


    if len(forward_primer) > 120 or len(reverse_primer) > 120:
        annealing_seq = repair_primers["PRIMER_LEFT_1_SEQUENCE"]
        annealing_tm = repair_primers["PRIMER_LEFT_1_TM"]
        annealing_coords = repair_primers["PRIMER_LEFT_1"]


        forward_primer = str(repair_seq[:(annealing_coords[0] +
annealing_coords[1])])

        repair_as_seq = Seq.Seq(repair_seq)
        repair_rc = repair_as_seq.reverse_complement()
        repair_rc_str = str(repair_rc)
        reverse_primer = repair_rc_str[0:(len(repair_seq) -
annealing_coords[0])]

    if len(forward_primer) > 120 or len(reverse_primer) > 120:
        annealing_seq = repair_primers["PRIMER_LEFT_2_SEQUENCE"]
        annealing_tm = repair_primers["PRIMER_LEFT_2_TM"]
        annealing_coords = repair_primers["PRIMER_LEFT_2"]


        forward_primer = str(repair_seq[:(annealing_coords[0] +
annealing_coords[1])])

        repair_as_seq = Seq.Seq(repair_seq)
        repair_rc = repair_as_seq.reverse_complement()
        repair_rc_str = str(repair_rc)
        reverse_primer = repair_rc_str[0:(len(repair_seq) -
annealing_coords[0])]


    output_dict = {"Forward primer (5'-)": forward_primer,
                   "Reverse primer (5'-)": reverse_primer,
                   "Annealing sequence (5'-)": annealing_seq,
                   "Tm ('C)": annealing_tm}

    return output_dict
```

7.2.18 MAIN CODE BATCH VERSION

7.2.18.1 Modified Configuration Spreadsheet



	A	B	C	D	E	F	G	H
1		Input	Input	Input	Input	Input	Input	Input
2	Job name	KKT2 S493A 1	KKT2 S493C 2	KKT2 S493D 3	KKT2 S493E 4	KKT2 S493F 5	KKT2 S493F 6	KKT2 S493F 7
3	Target amino acid residue	S	S	S	S	S	S	S
4	Target amino acid number	493	493	493	493	493	493	493
5	Replacement amino acid	A	C	D	E	F	G	H
6	Synonymous Recoding type	alternating lowest	alternating lowest	alternating lowest	alternating lowest	alternating lowest	matched	alternating highest
7	Nonsynonymous Recoding Type	highest	highest	highest	highest	highest	highest	highest
8	Codon Frequency data filename (incl. extension)	L_inf_codon_table_raw.txt	L_inf_codon_table_raw.txt	L_inf_codon_table_raw.txt	L_inf_codon_table_raw.txt	L_inf_codon_table_raw.txt	L_inf_codon_table_raw.txt	L_inf_codon_table_raw.txt
9	Recoding region length (bp)	60	60	60	60	60	60	60
10	Homology arm length (bp)	51	51	51	51	51	51	51
11	Reference FASTA filename (incl. extension)	KKT2-dna-fasta.txt	KKT2-dna-fasta.txt	KKT2-dna-fasta.txt	KKT2-dna-fasta.txt	KKT2-dna-fasta.txt	KKT2-dna-fasta.txt	KKT2-dna-fasta.txt
12	CDS start in reference file (bp number)	1	1	1	1	1	1	1
13	CDS end in reference file (bp number)	end	end	end	end	end	end	end
14	Alternating every nth residue	2	2	3	3	2		

7.2.18.2 Code

```
# -*- coding: utf-8 -*-
"""
Created on Thu May  9 13:06:02 2024

@author: ceh560
"""

#packages used in this file and/or the feeder files
import pandas as pd
from Bio import SeqIO
from Bio import Seq
from Bio import Align
import numpy as np
import io
import random
import primer3
import primer3.bindings

#custom files to import
import codon_dataframes as cdf
import codon_dictionaries as cdict
import formatting_functions as formats
import primer_functions as primers
import reading_input_file as rif
import validator as val
import stitching_functions as stitch

#read input files

input_data =
pd.read_excel("repair_template_input_excel_batch.xlsx", index_col
= 0, header = 0)


pd.set_option('display.max_columns', 20)
pd.set_option('display.max_rows', None)
pd.set_option("display.width", 1000)
pd.options.display.float_format = "{:,.2f}".format


#check for missing values in each column before proceeding
column_keys = list(input_data.keys())

complete_columns_keys = []

for column in column_keys:
    if input_data[column].notna().all() == True:
        complete_columns_keys.append(column)

    if input_data[column].notna().all() == False:
        if (input_data.isna().at["Alternating every nth residue",
column] == True) and (input_data[column]["Synonymous Recoding
type"] == "lowest" or input_data[column]["Synonymous Recoding
type"] == "highest" or input_data[column]["Synonymous Recoding
type"] ==  "matched" or input_data[column]["Synonymous Recoding
type"] == "random"):
            complete_columns_keys.append(column)

        if (input_data.isna().at["Alternating every nth residue",
column] == True) and (input_data[column]["Synonymous Recoding
type"] == "alternating lowest" or input_data[column]["Synonymous
Recoding type"] == "alternating highest" or
input_data[column]["Synonymous Recoding type"] == "alternating
matched") and (input_data[column][input_data.index != "Alternating
every nth residue"].notna().all() == True):
            complete_columns_keys.append(column)

if len(complete_columns_keys) != len(column_keys):
    print("\n\n\n***Warning, one or more columns have not been
included due to missing values.***\n\n\n")


#check for duplicate filenames to prevent overwriting

job_names = input_data.loc["Job name"].copy().transpose()

#duplicated_names = []


if job_names.duplicated().any() == True:
    duplicated_names = job_names.where(job_names.duplicated(keep =
False) == True)

    dup_names1 = dict(duplicated_names)
    dup_names2 = dict(duplicated_names.isna())

    dup_names1_df = pd.DataFrame(dict(duplicated_names), index =
["Duplicate vales"])
    dup_names2_df = pd.DataFrame(dict(duplicated_names.isna()),
index = ["True/False"])

    #duplicated_names_df = pd.DataFrame(dup_names1, index =
["Duplicate value"])
    duplicated_names_df = pd.concat([dup_names1_df,
dup_names2_df.astype(bool)], ignore_index = True)

    counter = 1

    for columns in duplicated_names_df.columns.values.tolist():
        job_name = duplicated_names_df[columns][0]
        unique_status = duplicated_names_df[columns][1]

        if unique_status == False:
            old_job_name = input_data.at["Job name", columns]
            input_data.at["Job name", columns] = old_job_name +
"(" + str(counter) + ")"

            counter += 1
```

```
for column in complete_columns_keys:
    job_name = input_data.loc["Job name"][column]
    target_AA = input_data.loc["Target amino acid
residue"][column]
    target_res_num = input_data.loc["Target amino acid
number"][column]
    output_AA = input_data.loc["Replacement amino acid"][column]
    syn_recode_type = input_data.loc["Synonymous Recoding
type"][column]
    nonsyn_recode_type = input_data.loc["Nonsynonymous Recoding
Type"][column]
    codon_freq_input_file = input_data.loc["Codon Frequency data
filename (incl. extension)"][column]
    recode_region_length = input_data.loc["Recoding region length
(bp)"][column]
    hom_arm_length = input_data.loc["Homology arm length
(bp)"][column]
    ref_file_name = input_data.loc["Reference FASTA filename
(incl. extension)"][column]
    CDS_start = input_data.loc["CDS start in reference file (bp
number)"][column]
    CDS_end = input_data.loc["CDS end in reference file (bp
number)"][column]
    alternating_repeat = input_data.loc["Alternating every nth
residue"][column]

    print("\n--------------------------------\n\n")
    print(f"Start of {job_name}, mutation:
{target_AA}{target_res_num}{output_AA}\n\n")



    #read input fasta file and process as necessary
    gene_name = job_name

    target_res_base_nums = [((target_res_num-1)*3),
(target_res_num*3)]


    num_of_codons_to_recode = recode_region_length / 3
    target_codon_no = int(num_of_codons_to_recode/2)

    if recode_region_length % 2 == 0:
        recode_start = int(target_res_base_nums[0] -
(recode_region_length/2))

    else:
        half_codon_percent = target_codon_no /
num_of_codons_to_recode
        back_bases = recode_region_length * half_codon_percent
        recode_start = int(target_res_base_nums[0] - back_bases)
    recode_end = recode_start + recode_region_length

    #need some special cases for close to the start or end of the
CDS
    #near the start special case

    if num_of_codons_to_recode > target_res_num:
        recode_start = 0
        recode_end = recode_region_length
        target_codon_no = target_res_num - 1



    for gene_name in SeqIO.parse(ref_file_name,"fasta"):
        #print(gene_name.id)
        print(gene_name.description)
        print(repr(gene_name.seq))
        print("Gene sequence length: ", len(gene_name), "bp")
        print("\n")

    if CDS_end == "end":
        CDS_end = len(gene_name.seq)
    else:
        CDS_end = CDS_end


    if CDS_start > 1:
        CDS_start = CDS_start - 1
        WT_CDS_seq = gene_name.seq[(CDS_start):CDS_end]
        recode_start_whole = recode_start + CDS_start
        recode_end_whole = recode_end + CDS_start

    else:
        WT_CDS_seq = gene_name.seq[:CDS_end]
        recode_start_whole = recode_start
        recode_end_whole = recode_end



    #check input is a length divisible by 3
    val.triplet_checker(WT_CDS_seq)

    #check that the input given is correct and that the target
codes for the expected residue
    val.translate_checker(WT_CDS_seq, target_res_num, target_AA)




    #near the end special case
    total_num_AAs = len(WT_CDS_seq.translate())

    if target_res_num > (total_num_AAs - num_of_codons_to_recode):
        recode_end = len(WT_CDS_seq)
        recode_start = len(WT_CDS_seq) - recode_region_length

        if CDS_start > 1:
            recode_end_whole = recode_end + CDS_start
            recode_start_whole = recode_start + CDS_start
        else:
            recode_end_whole = recode_end
            recode_start_whole = recode_start

        num_of_codons_to_recode = int((recode_end - recode_start +
1) / 3)

        target_codon_no = num_of_codons_to_recode - (total_num_AAs
- target_res_num) - 1




    #establish the sequence to replace, and sequences before and
after to stay the same
    WT_template_seq =
gene_name.seq[recode_start_whole:recode_end_whole]
    upstream_dna = gene_name.seq[:recode_start_whole]
    downstream_dna = gene_name.seq[recode_end_whole:]






    #make dictionary of codons with number keys and one with
numbers and amino acids

    codons_to_recode = cdict.codon_dict_maker(WT_template_seq,
key_format= "number")
    codons_to_recode_let_num =
cdict.codon_dict_maker(WT_template_seq, key_format= "letter-
number")


    #make reference dictionaries for all the amino acids
    ref_codon_table_df =
rif.codon_table_processor(codon_freq_input_file)

    ref_codons = cdf.ref_codon_table_freqs(ref_codon_table_df)

    if syn_recode_type == "matched":

        #use that dictionary to create a new one with the specific
frequency values
        codons_to_recode_freqs =
cdf.codon_frequency_collector(input_dict = codons_to_recode,
reference_dict = ref_codons, type = "value")

        #create a dictionary with all the frequencies for the
amino acids in this sequence for each codon
```

```
        codons_to_recode_all_freqs = 
cdf.codon_frequency_collector(input_dict = codons_to_recode, 
reference_dict = ref_codons, type = "dataframe")


        #calculate the differences for each possible codon to the 
original
        recode_freq_diffs = 
cdf.codon_frequency_difference_calc(codons_to_recode_let_num, 
ref_codons)

        #add the differences in frequency to "the" dataframe
        codons_to_recode_abs_diffs = 
cdf.codon_freq_diff_adder(codons_to_recode_let_num 
,codons_to_recode_all_freqs, recode_freq_diffs)

        #choose which codons to use for synonymous recoding
        codons_to_use_syn = 
cdf.codon_freq_selector(codons_to_recode_abs_diffs)


    if syn_recode_type == "highest" or syn_recode_type == 
"lowest":

        #use that dictionary to create a new one with the specific 
frequency values
        codons_to_recode_freqs = 
cdf.codon_frequency_collector(input_dict = codons_to_recode, 
reference_dict = ref_codons, type = "value")

        #create a dictionary with all the frequencies for the 
amino acids in this sequence for each codon
        codons_to_recode_all_freqs = 
cdf.codon_frequency_collector(input_dict = codons_to_recode, 
reference_dict = ref_codons, type = "dataframe")

        codons_to_recode_choices_freqs = {}
        #remove input codon from list
        for let_num, seq in codons_to_recode_let_num.items():
            if seq == Seq.Seq("ATG") or seq == Seq.Seq("TGG"):

                codons_to_recode_choices_freqs[let_num] = 
ref_codon_table_df.loc[ref_codon_table_df["DNA"] == str(seq)]
            else:
                current_df = codons_to_recode_all_freqs[let_num]
                codons_to_recode_choices_freqs[let_num] = 
current_df.loc[current_df["DNA"] != str(seq)]

    #make the list of codons to use depending on recoding type
        codons_to_use_syn = {}

        if syn_recode_type == "highest":

            for codon_num_let, seq_df in 
codons_to_recode_choices_freqs.items():
                max_freq_codon = max(seq_df["Fraction"])
                mutated_res_df_chosen =
seq_df.loc[seq_df["Fraction"] == max_freq_codon, "DNA"]

                #tie breaker for instances with same fraction
usage - hopefully number won't ever have duplicate values
                if len(mutated_res_df_chosen) > 1:
                    max_number_codon = max(seq_df["Number"])
                    max_number_codon_seq =
seq_df.loc[seq_df["Number"] == max_number_codon, "DNA"].item()
                    codons_to_use_syn[codon_num_let] =
max_number_codon_seq

                else:
                    codons_to_use_syn[codon_num_let] =
seq_df.loc[seq_df["Fraction"] == max_freq_codon, "DNA"].item()

        if syn_recode_type == "lowest":

            for codon_num_let, seq_df in
codons_to_recode_choices_freqs.items():
                min_freq_codon = min(seq_df["Fraction"])
                mutated_res_df_chosen =
seq_df.loc[seq_df["Fraction"] == min_freq_codon, "DNA"]

                #tie breaker
                if len(mutated_res_df_chosen) > 1:
                    min_number_codon = max(seq_df["Number"])
                    min_number_codon_seq =
seq_df.loc[seq_df["Number"] == min_number_codon, "DNA"].item()
                    codons_to_use_syn[codon_num_let] =
min_number_codon_seq

                else:
                    codons_to_use_syn[codon_num_let] =
seq_df.loc[seq_df["Fraction"] == min_freq_codon, "DNA"].item()


    if syn_recode_type == "alternating matched" or syn_recode_type
== "alternating random" or syn_recode_type == "alternating
highest" or syn_recode_type == "alternating lowest":
        #check input has been given suitably
        if alternating_repeat == "N/A" or alternating_repeat <= 0
or pd.isna(alternating_repeat) == True:
            print("\n\n\n***ERROR: No value or an invalid value
was set for the alternating pattern of the codons to
recode.***\n\n\n")
            alternating_repeat = int(input("Please enter a
positive integrer for the alternating repeat value: "))

            input_data.at["Alternating every nth residue", column]
= alternating_repeat

        if alternating_repeat > (0.5 * num_of_codons_to_recode):
            proceed_alt = input("The chosen repeat value is
greater than half of the total number of codons being recoded so
```

```
only 2 or fewer codons will be mutated.\n\nDo you wish to proceed?
Y/N \n")

            if proceed_alt == "N" or proceed_alt == "n" or
proceed_alt == "NO" or proceed_alt == "No" or proceed_alt == "no":
                alternating_repeat = int(input("Please enter a
positive integer for the alternating repeat value: "))

                input_data.at["Alternating every nth residue",
column] = alternating_repeat

            elif proceed_alt == "Y" or proceed_alt =="y" or
proceed_alt =="YES" or proceed_alt == "Yes" or proceed_alt ==
"yes":
               pass
            else:
                proceed_alt = input("\n\nThe input given is not
valid. Please try again.\n\nThe chosen repeat value is greater
than half of the total number of codons being recoded so only 2 or
fewer codons will be mutated.\n\nDo you wish to proceed? Y/N \n")
                if proceed_alt == "N" or proceed_alt == "n" or
proceed_alt == "NO" or proceed_alt == "No" or proceed_alt == "no":
                    alternating_repeat = int(input("\nPlease enter
a positive integer for the alternating repeat value: "))

                    input_data.at["Alternating every nth residue",
column] = alternating_repeat

                elif proceed_alt == "Y" or proceed_alt =="y" or
proceed_alt =="YES" or proceed_alt == "Yes" or proceed_alt ==
"yes":
                   pass
                else:
                    proceed_alt = input("\n\nThe input given is
not valid. Please try again.\n\nThe chosen repeat value is greater
than half of the total number of codons being recoded so only 2 or
fewer codons will be mutated.\n\nDo you wish to proceed? Y/N \n")
                    if proceed_alt == "N" or proceed_alt == "n" or
proceed_alt == "NO" or proceed_alt == "No" or proceed_alt == "no":
                        alternating_repeat = int(input("\nPlease
enter a positive integer for the alternating repeat value: "))

                        input_data.at["Alternating every nth
residue", column] = alternating_repeat

                    elif proceed_alt == "Y" or proceed_alt =="y"
or proceed_alt =="YES" or proceed_alt == "Yes" or proceed_alt ==
"yes":
                       pass
                    else:
                        print("\n---------------------------------
---------------------------")
                        print(f"\n***Warning, your input was
invalid so the code will continue with the value given. Your
repair template will recode every {alternating_repeat} codons. If
```

```
you do not want this, modify the input spreadsheet and rerun the
programme.***")


        if alternating_repeat == "N/A" or alternating_repeat <= 0
or pd.isna(alternating_repeat) == True:
            print("\n\n\n***ERROR: An invalid value was set for
the alternating pattern of the codons to recode.***\n\n\n")
            alternating_repeat = int(input("Please enter a
positive integer for the alternating repeat value: "))

        if alternating_repeat == "N/A" or alternating_repeat <= 0
or pd.isna(alternating_repeat) == True:
            print("\n\n\n***ERROR: An invalid value was set for
the alternating pattern of the codons to recode.***\n\n\n")
            alternating_repeat = int(input("Last chance - please
enter a positive integer for the alternating repeat value: "))

        if alternating_repeat == "N/A" or alternating_repeat <= 0
or pd.isna(alternating_repeat) == True:
            print("\n\n\n\nYou failed to provide an appropriate
input so the programme will be cancelled.\n\nIf you wish to try
again, either modify the input spreadsheet or provide a suitable
value when prompted in the console.\n")
            raise SystemExit


    if syn_recode_type == "alternating matched" or syn_recode_type
== "alternating random":


        #determine which codon numbers in range are to be mutated
and which are not
        num_of_codons_to_mutate = int(num_of_codons_to_recode /
alternating_repeat)
        n_terms = list(range(num_of_codons_to_mutate))
        codon_nums_to_recode = []

        for n in n_terms:
            codon_num = n * alternating_repeat
            codon_nums_to_recode.append(codon_num)

        #ensure that target codon is always recoded even if it
doesn't fit the alternating pattern
        if target_codon_no not in codon_nums_to_recode:
            codon_nums_to_recode.append(target_codon_no)


        codon_nums_all = list(codons_to_recode.keys())

        #split the codons to be mutated into a separate dictionary
from the ones to stay the same
        codons_to_keep_WT = {}
        specific_codons_to_recode = {}

        for numbers in codon_nums_all:
            if numbers not in codon_nums_to_recode:
                codons_to_keep_WT[numbers] =
codons_to_recode[numbers]

            if numbers in codon_nums_to_recode:
                specific_codons_to_recode[numbers] =
codons_to_recode[numbers]


        for numbers in codon_nums_to_recode:
            if numbers not in codon_nums_to_recode:
                codons_to_keep_WT = codons_to_recode[numbers]

        if syn_recode_type == "alternating matched":
            #on only the codons to recode
            #use that dictionary to create a new one with the
specific frequency values
            codons_to_recode_freqs =
cdf.codon_frequency_collector(input_dict = codons_to_recode,
reference_dict = ref_codons, type = "value")

            #create a dictionary with all the frequencies for the
amino acids in this sequence for each codon
            codons_to_recode_all_freqs =
cdf.codon_frequency_collector(input_dict = codons_to_recode,
reference_dict = ref_codons, type = "dataframe")


            #calculate the differences for each possible codon to
the original
            recode_freq_diffs =
cdf.codon_frequency_difference_calc(codons_to_recode_let_num,
ref_codons)

            #add the differences in frequency to "the" dataframe
            codons_to_recode_abs_diffs =
cdf.codon_freq_diff_adder(codons_to_recode_let_num
,codons_to_recode_all_freqs, recode_freq_diffs)

            #choose which codons to use for synonymous recoding
            codons_to_use_syn =
cdf.codon_freq_selector(codons_to_recode_abs_diffs)


        if syn_recode_type == "alternating random":

            #add letters to dictionary
            specific_codons_to_recode_let_num = {}

            for keys, seq in specific_codons_to_recode.items():
                let_num = str(seq.translate()) + str(keys)
                specific_codons_to_recode_let_num[let_num] = seq


            #make a dictionary of the alternate codons to the
input sequence
```

```
            alt_codons_to_recode = 
cdict.alt_codons(specific_codons_to_recode_let_num)

            #randomly select which of these to use for each codon
            codons_to_use_syn = 
cdict.Syn_random_recoder(alt_codons_to_recode)


        #combine the unchanged codons with the changed codons

        codons_to_keep_WT_let_num = {}

        for codon_num, seq in codons_to_keep_WT.items():
            translation = seq.translate()
            codon_num_let = str(translation) + str(codon_num)

            codons_to_keep_WT_let_num[codon_num_let] = seq

        codons_to_use_syn.update(codons_to_keep_WT_let_num)


    if syn_recode_type == "alternating highest" or syn_recode_type 
== "alternating lowest":

        num_of_codons_to_mutate = int(num_of_codons_to_recode / 
alternating_repeat)
        n_terms = list(range(num_of_codons_to_mutate))
        codon_nums_to_recode = []

        for n in n_terms:
            codon_num = n * alternating_repeat
            codon_nums_to_recode.append(codon_num)

        if target_codon_no not in codon_nums_to_recode:
            codon_nums_to_recode.append(target_codon_no)


        codon_nums_all = list(codons_to_recode.keys())

        codons_to_keep_WT = {}
        specific_codons_to_recode = {}

        for numbers in codon_nums_all:
            if numbers not in codon_nums_to_recode:
                translate = codons_to_recode[numbers].translate()
                let_num = str(translate) + str(numbers)
                codons_to_keep_WT[let_num] = 
codons_to_recode[numbers]

            if numbers in codon_nums_to_recode:
                #translate = codons_to_recode[numbers].translate()
                #let_num = str(translate) + str(numbers)
                specific_codons_to_recode[numbers] = 
codons_to_recode[numbers]
```

```
        for numbers in codon_nums_to_recode:
            if numbers not in codon_nums_to_recode:
                codons_to_keep_WT = codons_to_recode[numbers]

        #use that dictionary to create a new one with the specific
frequency values
        codons_to_recode_freqs =
cdf.codon_frequency_collector(input_dict =
specific_codons_to_recode, reference_dict = ref_codons, type =
"value")

        #create a dictionary with all the frequencies for the
amino acids in this sequence for each codon
        codons_to_recode_all_freqs =
cdf.codon_frequency_collector(input_dict =
specific_codons_to_recode, reference_dict = ref_codons, type =
"dataframe")

        codons_to_recode_choices_freqs = {}
        #remove input codon from list unless it's Met or Trp
        for let_num, df in codons_to_recode_all_freqs.items():
            input_codon = codons_to_recode_let_num[let_num]
            if input_codon == Seq.Seq("ATG") or input_codon ==
Seq.Seq("TGG"):

                codons_to_recode_choices_freqs[let_num] =
ref_codon_table_df.loc[ref_codon_table_df["DNA"] ==
str(input_codon)]
            else:
                current_df = codons_to_recode_all_freqs[let_num]
                codons_to_recode_choices_freqs[let_num] =
current_df.loc[current_df["DNA"] != str(input_codon)]

        #recode based on input type
        codons_to_use_syn = {}

        if syn_recode_type == "alternating highest":

            for codon_num_let, seq_df in
codons_to_recode_choices_freqs.items():
                max_freq_codon = max(seq_df["Fraction"])
                max_freq_codon_seq = seq_df.loc[seq_df["Fraction"]
== max_freq_codon, "DNA"]
                if len(max_freq_codon_seq) > 1:
                    max_number_codon = max(seq_df["Number"])
                    max_freq_codon_seq =
seq_df.loc[seq_df["Number"] == max_number_codon, "DNA"].item()
                    codons_to_use_syn[codon_num_let] =
max_freq_codon_seq

                else:
                 codons_to_use_syn[codon_num_let] =
max_freq_codon_seq.item()
```

```
        if syn_recode_type == "alternating lowest":

            for codon_num_let, seq_df in
codons_to_recode_choices_freqs.items():
                min_freq_codon = min(seq_df["Fraction"])
                min_freq_codon_seq = seq_df.loc[seq_df["Fraction"]
== min_freq_codon, "DNA"]
                if len(min_freq_codon_seq) > 1:
                    min_number_codon = min(seq_df["Number"])
                    min_freq_codon_seq =
seq_df.loc[seq_df["Number"] == min_number_codon, "DNA"].item()
                    codons_to_use_syn[codon_num_let] =
min_freq_codon_seq

                else:
                 codons_to_use_syn[codon_num_let] =
min_freq_codon_seq.item()



        #combine the unchanged codons with the changed codons

        codons_to_keep_WT_let_num = {}

        for codon_num, seq in codons_to_keep_WT.items():
            codons_to_keep_WT_let_num[codon_num] = seq

        codons_to_use_syn.update(codons_to_keep_WT_let_num)


    if syn_recode_type == "random":
        #make a dictionary of the alternate codons to the input
sequence
        alt_codons_to_recode =
cdict.alt_codons(codons_to_recode_let_num)

        #randomly select which of these to use for each codon
        codons_to_use_syn =
cdict.Syn_random_recoder(alt_codons_to_recode)


    #add in the nonsynonymous mutation

    if nonsyn_recode_type == "highest" or nonsyn_recode_type ==
"lowest":

        nonsyn_ref_dict = ref_codons

    if nonsyn_recode_type == "random":
        nonsyn_ref_dict =
cdict.alt_codons(codons_to_recode_let_num)
        nonsyn_ref_dict = {output_AA :
cdict.ref_codon_table(output_AA)}
```

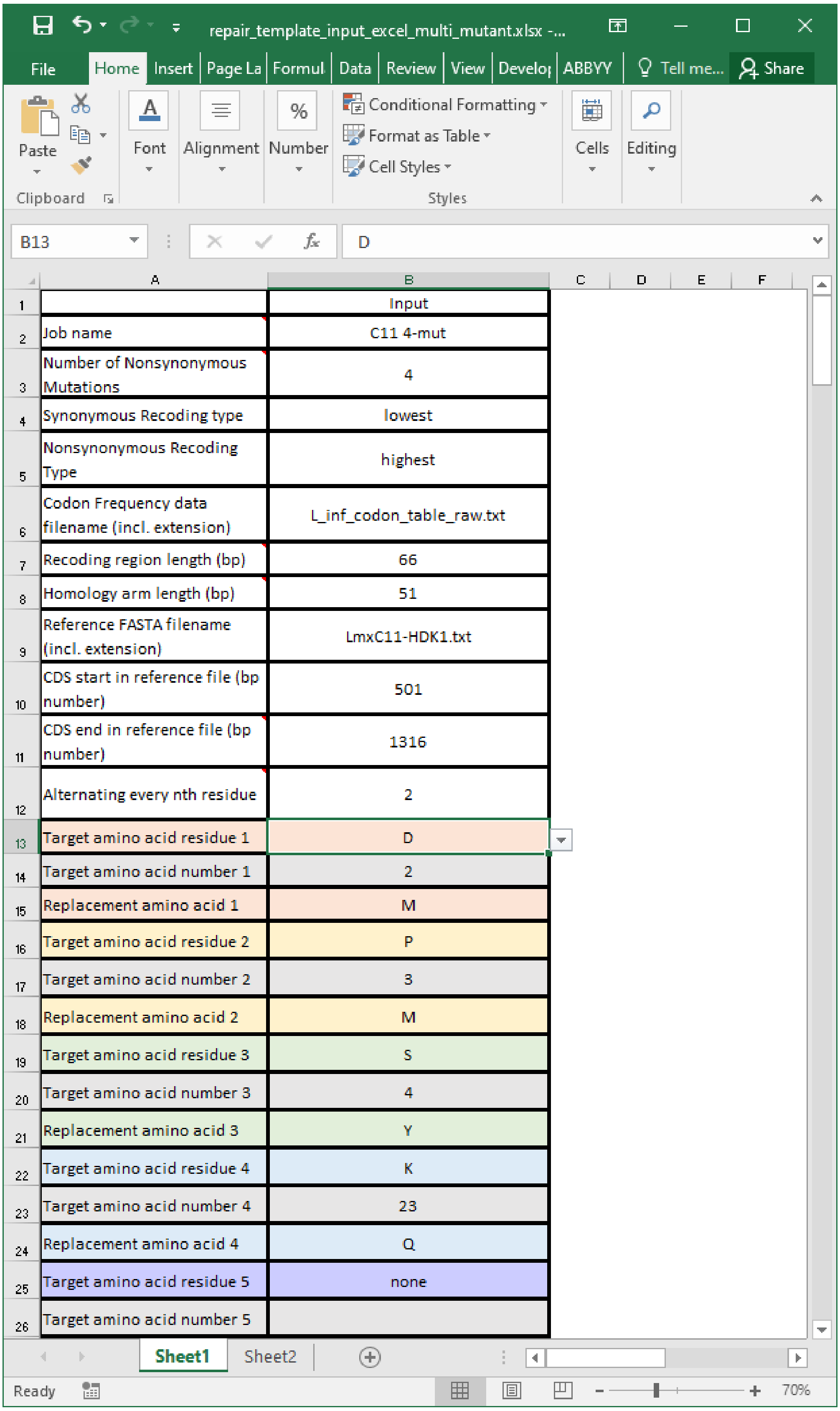
```
    codons_to_use_nonsyn = cdf.non_syn_mutator(target_AA,
target_codon_no, new_AA = output_AA, input_dict =
codons_to_use_syn, type = nonsyn_recode_type, ref_dict =
nonsyn_ref_dict )



    #construct the final recoded sequences

    synonymous_repair =
stitch.sequence_constructor(codons_to_use_syn, type = "letter-
number")
    nonsynonymous_repair =
stitch.sequence_constructor(codons_to_use_nonsyn, type = "letter-
number")

    #check all the modifications were as expected
    #adjust target codon number to what it would be by normal
counting rather than python counting
    target_codon_no_not_py = target_codon_no + 1


    val.translate_checker(synonymous_repair,
target_codon_no_not_py, target_AA)

    val.translate_checker(nonsynonymous_repair,
target_codon_no_not_py, output_AA)



    #create the final repair sequence including the homology arms

    upstream_hom_arm = gene_name.seq[(recode_start_whole -
hom_arm_length):recode_start_whole]
    downstream_hom_arm = gene_name.seq[recode_end_whole:
(recode_end_whole + hom_arm_length)]

    WT_entire_repair_region = upstream_hom_arm + WT_template_seq +
downstream_hom_arm
    entire_syn_repair = upstream_hom_arm + synonymous_repair +
downstream_hom_arm
    entire_nonsyn_repair = upstream_hom_arm + nonsynonymous_repair
+ downstream_hom_arm



    #construct "gene" sequences for primer design
    integrated_synonymous, WT_recode_region =
stitch.mut_seq_integrator(repair_seq = synonymous_repair, ref_seq
= gene_name.seq, repair_start = recode_start_whole, repair_end =
recode_end_whole, WT_repair_seq= "Yes")
    integrated_nonsynonymous =
stitch.mut_seq_integrator(repair_seq = nonsynonymous_repair,
ref_seq = gene_name.seq, repair_start = recode_start_whole,
repair_end = recode_end_whole, WT_repair_seq= "No")
```

```
    #design screening primers
    screening_primers_df_syn  =
primers.screening_primer_designer(gene_name.seq,
integrated_synonymous, recode_start_whole, recode_end_whole)
    screening_primers_df_nonsyn  =
primers.screening_primer_designer(gene_name.seq,
integrated_nonsynonymous, recode_start_whole, recode_end_whole)

    #design primers to generate the repair template
    syn_repair_template_primers =
primers.repair_primer_designer(entire_syn_repair, hom_arm_length,
downstream_dna)
    nonsyn_repair_template_primers =
primers.repair_primer_designer(entire_nonsyn_repair,
hom_arm_length, downstream_dna)

    #repair_template_primers = [syn_repair_template_primers,
nonsyn_repair_template_primers]

    #repair_template_primers_df =
pd.DataFrame(repair_template_primers)
    #repair_template_primers_df.index = ["Synonymous repair",
"Nonsynonymous repair"]

    #do an alignment

    #create a pariwise alignment object
    aligner = Align.PairwiseAligner(target_internal_open_gap_score
= -10.0, query_internal_open_gap_score = -10.0)



    syn_alignment = aligner.align(WT_entire_repair_region,
entire_syn_repair)
    for alignment1 in sorted(syn_alignment):
        #print("Score = %.1f:" % alignment1.score)
        #print(alignment1)
        syn_score = alignment1.score
    alignment_str_syn = str(alignment1)
    alignment_str_syn = alignment_str_syn.replace("target", "WT
sequence").replace("query", "Syn. repair").replace("\n
", "\n                ")
    alignment_str_syn = alignment_str_syn.replace("Syn. repair
", "Syn. repair          ")
    #print(alignment_str_syn)

    nonsyn_alignment = aligner.align(WT_entire_repair_region,
entire_nonsyn_repair)
    for alignment2 in sorted(syn_alignment):
        #print("Score = %.1f:" % alignment2.score)
        nonsyn_score = alignment2.score
    alignment_str_nonsyn = str(alignment2)
    alignment_str_nonsyn = alignment_str_nonsyn.replace("target",
"WT sequence").replace("query", "Nonsyn. repair").replace("\n
", "\n                ")
```

```
    alignment_str_nonsyn = alignment_str_nonsyn.replace("Nonsyn.
repair           ", "Nonsyn. repair        ")
    #print(alignment_str_nonsyn)




    #format some outputs

    WT_repair_seq_spaced =
formats.codon_spacing(WT_entire_repair_region)
    syn_repair_spaced = formats.codon_spacing(entire_syn_repair)
    nonsyn_repair_spaced =
formats.codon_spacing(entire_nonsyn_repair)

    WT_repair_translate = WT_entire_repair_region.translate()
    syn_repair_translate = entire_syn_repair.translate()
    nonsyn_repair_translate = entire_nonsyn_repair.translate()

    WT_repair_translate_spaced =
formats.protein_align_codon(WT_repair_translate)
    syn_repair_translate_spaced =
formats.protein_align_codon(syn_repair_translate)
    nonsyn_repair_translate_spaced =
formats.protein_align_codon(nonsyn_repair_translate)

    syn_repair_mutations_count =
val.mutation_counter(entire_syn_repair, WT_entire_repair_region)
    nonsyn_repair_mutations_count =
val.mutation_counter(entire_nonsyn_repair,
WT_entire_repair_region)

    syn_repair_primers_output = ""

    for category, item in syn_repair_template_primers.items():
        if type(item) == float:
            item = '{:.1f}'.format(item)
        syn_repair_primers_output += category
        syn_repair_primers_output += ": "
        syn_repair_primers_output += str(item)
        syn_repair_primers_output += "\n"

    nonsyn_repair_primers_output = ""

    for category, item in nonsyn_repair_template_primers.items():
        if type(item) == float:
            item = '{:.1f}'.format(item)
        nonsyn_repair_primers_output += category
        nonsyn_repair_primers_output += ": "
        nonsyn_repair_primers_output += str(item)
        nonsyn_repair_primers_output += "\n"

    if syn_recode_type == "alternating matched" or syn_recode_type
== "alternating highest" or syn_recode_type == "alternating
lowest" or syn_recode_type == "alternating random":
        alternating_info = f"Alternating recoding every
{alternating_repeat} codons"
    else:
        alternating_info = ""


    output_file = open(f"{job_name}.txt", "w")

    file_lines = ["Job request details\n",
                  f"Job name: {job_name}\n",
                  f"Target amino acid:
{target_AA}{target_res_num}{output_AA}\n",
                  f"Synonymous recoding type:
{syn_recode_type}\n",
                  f"Nonsynonymous recode type:
{nonsyn_recode_type}\n",
                  f"Homology arm length (bp): {hom_arm_length}\n",
                  f"Recoding region length (bp):
{recode_region_length}\n",
                  f"Total repair length (bp): {(2*hom_arm_length)
+ recode_region_length}\n",
                  f"{alternating_info}\n",
                  "\n",
                  "\n",
                  "Repair templates\n",
                  f"WT repair region sequence:
\t\t{WT_repair_seq_spaced}\n",
                  f"WT translation:
\t\t\t{WT_repair_translate_spaced}\n",
                  f"Synonymous repair region sequence:
\t{syn_repair_spaced}\n",
                  f"Synonymous repair translation:
\t\t{syn_repair_translate_spaced}\n",
                  f"Nonsynonymous repair region sequence:
\t{nonsyn_repair_spaced}\n",
                  f"Nonsynonymous repair translation:
\t{nonsyn_repair_translate_spaced}\n",
                  "\n",
                  f"Number of mutations in the synonymous repair
template: {syn_repair_mutations_count}\n",
                  f"Number of mutations in the nonsynonymous
repair template: {nonsyn_repair_mutations_count}\n",
                  "\n",
                  "\n",
                  "Screening primers\n",
                  "Synonymous repair\n",
                  "\n",
                  f"{screening_primers_df_syn}\n",
                  "\n",
                  "\n",
                  "Nonsynonymous primers\n"
                  f"{screening_primers_df_nonsyn}",
                  "\n",
                  "\n",
                  "Repair template primers\n",
                  "Synonymous\n",
                  f"{syn_repair_primers_output}\n",
                  "\n",
                  "Nonsynonymous\n",
                  f"{nonsyn_repair_primers_output}\n",
                  "\n",
                  f"WT sequence (no spaces):
{WT_entire_repair_region}\n",
                  f"Synonymous sequence (no spaces):
{entire_syn_repair}\n",
                  f"Nonsynonymous sequence (no spaces):
{entire_nonsyn_repair}\n",
                  "\n",
                  "\n",
                  "Alignments\n",
                  "Synonymous Repair\n",
                  f"Score = {syn_score}\n",
                  f"{alignment_str_syn}\n",
                  "\n",
                  "Nonsynonymous\n",
                  f"Score = {nonsyn_score}\n",
                  f"{alignment_str_nonsyn}\n"

        ]

    output_file.writelines(file_lines)
    output_file.close()

    #print confirmation message to make it clearer that it worked
    print(f"\n\n\nYour repair template designs have completed
successfully. Please check your folder for a file with the name
'{job_name}.txt'\n")
    print("\t.\t.\n", "\n\t\___/\n\n\n")


print("\n--------------------------------\n")



print("Jobs that were completed:\n")


for column in complete_columns_keys:

    print(input_data[column]["Job name":"Nonsynonymous Recoding
Type"])
    print("\n")


if job_names.duplicated().any() == True:
    print("\n***Warning: duplicate job names (file names)
detected. Some files will be renamed to avoid
overwriting.***\n\t\t\t\t***Please check the completed jobs above
for details.***\n")
```

7.2.19 MAIN CODE MULTI MUTANT VERSION

7.2.19.1 Modified Configuration Spreadsheet



	A	B
1		Input
2	Job name	C11 4-mut
3	Number of Nonsynonymous Mutations	4
4	Synonymous Recoding type	lowest
5	Nonsynonymous Recoding Type	highest
6	Codon Frequency data filename (incl. extension)	L_inf_codon_table_raw.txt
7	Recoding region length (bp)	66
8	Homology arm length (bp)	51
9	Reference FASTA filename (incl. extension)	LmxC11-HDK1.txt
10	CDS start in reference file (bp number)	501
11	CDS end in reference file (bp number)	1316
12	Alternating every nth residue	2
13	Target amino acid residue 1	D
14	Target amino acid number 1	2
15	Replacement amino acid 1	M
16	Target amino acid residue 2	P
17	Target amino acid number 2	3
18	Replacement amino acid 2	M
19	Target amino acid residue 3	S
20	Target amino acid number 3	4
21	Replacement amino acid 3	Y
22	Target amino acid residue 4	K
23	Target amino acid number 4	23
24	Replacement amino acid 4	Q
25	Target amino acid residue 5	none
26	Target amino acid number 5	

7.2.19.2 Code

```
# -*- coding: utf-8 -*-
"""
Created on Fri Jun  7 14:26:56 2024

@author: ceh560
"""

#packages used in this file and/or the feeder files
import pandas as pd
from Bio import SeqIO
from Bio import Seq
from Bio import Align
import numpy as np
import io
import random
import primer3
import primer3.bindings
import statistics

#custom files to import
import codon_dataframes as cdf
import codon_dictionaries as cdict
import formatting_functions as formats
import primer_functions as primers
import reading_input_file as rif
import validator as val
import stitching_functions as stitch




#read input files

input_data =
pd.read_excel("repair_template_input_excel_multi_mutant.xlsx",
index_col = 0, header = 0)


pd.set_option('display.max_columns', 20)
pd.set_option('display.max_rows', None)
pd.set_option("display.width", 1000)
pd.options.display.float_format = "{:,.2f}".format


job_name = input_data.loc["Job name"][0]
syn_recode_type = input_data.loc["Synonymous Recoding type"][0]
nonsyn_recode_type = input_data.loc["Nonsynonymous Recoding
Type"][0]
codon_freq_input_file = input_data.loc["Codon Frequency data
filename (incl. extension)"][0]
recode_region_length = input_data.loc["Recoding region length
(bp)"][0]
hom_arm_length = input_data.loc["Homology arm length (bp)"][0]
```

```
ref_file_name = input_data.loc["Reference FASTA filename (incl.
extension)"][0]
CDS_start = input_data.loc["CDS start in reference file (bp
number)"][0]
CDS_end = input_data.loc["CDS end in reference file (bp
number)"][0]
alternating_repeat = input_data.loc["Alternating every nth
residue"][0]


num_of_mutations = input_data.loc["Number of Nonsynonymous
Mutations"][0]

#mutation 1
target_AA_1 = input_data.loc["Target amino acid residue 1"][0]
target_res_num_1 = input_data.loc["Target amino acid number 1"][0]
output_AA_1 = input_data.loc["Replacement amino acid 1"][0]

#mutation 2
target_AA_2 = input_data.loc["Target amino acid residue 2"][0]
target_res_num_2 = input_data.loc["Target amino acid number 2"][0]
output_AA_2 = input_data.loc["Replacement amino acid 2"][0]


#mutation 3
target_AA_3 = input_data.loc["Target amino acid residue 3"][0]
target_res_num_3 = input_data.loc["Target amino acid number 3"][0]
output_AA_3 = input_data.loc["Replacement amino acid 3"][0]


#mutation 4
target_AA_4 = input_data.loc["Target amino acid residue 4"][0]
target_res_num_4 = input_data.loc["Target amino acid number 4"][0]
output_AA_4 = input_data.loc["Replacement amino acid 4"][0]


#mutation 5
target_AA_5 = input_data.loc["Target amino acid residue 5"][0]
target_res_num_5 = input_data.loc["Target amino acid number 5"][0]
output_AA_5 = input_data.loc["Replacement amino acid 5"][0]


#put all the mutants in a dataframe
mut_details = {"Mutation number": [1,2,3,4,5],
    "Target AA": [target_AA_1, target_AA_2, target_AA_3,
target_AA_4, target_AA_5],
    "Target residue number": [target_res_num_1, target_res_num_2,
target_res_num_3, target_res_num_4, target_res_num_5],
    "Replacement AA": [output_AA_1, output_AA_2, output_AA_3,
output_AA_4, output_AA_5]}

mut_details_df = pd.DataFrame(mut_details)


#target_res_base_nums = [((target_res_num-1)*3),
(target_res_num*3)]
```

```
#add residue numbers to dataframe

for row in mut_details_df.index:
    residue_no = mut_details_df.at[row, "Target residue number"]
    residue_start = (residue_no - 1)*3
    residue_end = residue_no * 3
    mut_details_df.at[row, "Residue Start Base"] = residue_start
    mut_details_df.at[row, "Residue End Base"] = residue_end


#remove rows not needed for less than 5 mutations
if num_of_mutations < 5:
    mut_details_df = mut_details_df.iloc[:num_of_mutations]

#when n/a's are present, they cause the other numbers to be
floats, so to ensure that doesn't happen, after removing them,
convert to integers

mut_details_df["Target residue number"] = mut_details_df["Target
residue number"].astype(int)



#read input fasta file and process as necessary
gene_name = job_name


num_of_codons_to_recode = recode_region_length / 3

lowest_target = min(mut_details_df["Target residue number"])
centre_target = statistics.median(mut_details_df["Target residue
number"])
highest_target = max(mut_details_df["Target residue number"])

#if the median does not exist as a target e.g. the median of two
values is halfway between them
if mut_details_df.isin([centre_target]).any().all() == True:

    #define codon numbers for each target
    central_target_codon_no = int(num_of_codons_to_recode/2)
    central_target_index =
mut_details_df.loc[mut_details_df["Target residue
number"]==centre_target].index.item()

    mut_details_df["Codon no"] = None
    mut_details_df.at[central_target_index, "Codon no"] =
central_target_codon_no

    #add target codon numbers to dataframe
    for row in mut_details_df.index:
        if pd.isna(mut_details_df.at[row, "Codon no"]) == False:
            pass
        target_res_num = mut_details_df.at[row, "Target residue
number"]
        diff = centre_target - target_res_num
        codon_no = central_target_codon_no + diff
        mut_details_df.at[row, "Codon no"] = codon_no

    centre_target_base_start = mut_details_df["Residue Start
Base"].loc[(mut_details_df["Target residue number"] ==
centre_target)]


    if recode_region_length % 2 == 0:
        recode_start = int(centre_target_base_start -
(recode_region_length/2))

    else:
        half_codon_percent = central_target_codon_no /
num_of_codons_to_recode
        back_bases = recode_region_length * half_codon_percent
        recode_start = int(centre_target_base_start - back_bases)




else:


    central_target_codon_no = int(num_of_codons_to_recode/2)
    central_target_codon_start = int(centre_target * 3)
    central_target_codon_end = central_target_codon_start + 3

    mut_details_df["Codon no"] = None

    for row in mut_details_df.index:
        target_res_num = mut_details_df.at[row, "Target residue
number"]
        diff = centre_target - target_res_num
        codon_no = central_target_codon_no - diff
        mut_details_df.at[row, "Codon no"] = codon_no

    if recode_region_length % 2 == 0:
        recode_start = int(central_target_codon_start -
(recode_region_length/2) - 1)

    else:
         half_codon_percent = central_target_codon_no /
num_of_codons_to_recode
         back_bases = recode_region_length * half_codon_percent
         recode_start = int(central_target_codon_start -
back_bases)

    if max(mut_details_df["Codon no"]) > num_of_codons_to_recode:
        min_recode_start = min(mut_details_df["Residue Start
Base"])
        min_recode_end = max(mut_details_df["Residue End Base"])

        min_recoding_region = min_recode_end - min_recode_start

        if min_recoding_region == recode_region_length:
            recode_start = min_recode_start
            recode_end = min_recode_end

            lowest_target_index =
mut_details_df.loc[mut_details_df["Target residue
number"]==lowest_target].index.item()
            mut_details_df.at[lowest_target_index, "Codon no"] = 0
            lowest_target_codon_no =
mut_details_df.at[lowest_target_index, "Codon no"]

            for row in mut_details_df.index:
                if pd.isna(mut_details_df.at[row, "Codon no"]) ==
False:
                    pass
                target_res_num = mut_details_df.at[row, "Target
residue number"]
                diff =  target_res_num - lowest_target
                codon_no = lowest_target_codon_no + diff
                mut_details_df.at[row, "Codon no"] = codon_no

        else:
            extra_bases = recode_region_length -
min_recoding_region
            extra_codons = extra_bases/3

            codons_to_start = lowest_target - 1

    #accounting for times where an uneven distribution of targets
causes an inappropriate centre
    if min(mut_details_df["Codon no"]) < 0:

        targets_range = max(mut_details_df["Codon no"]) -
min(mut_details_df["Codon no"])

        #if the number of codons to recode is the same distance as
the range of the target sites
        if targets_range == num_of_codons_to_recode:
            recode_start = min_recode_start
            recode_end = min_recode_end

            lowest_target_index =
mut_details_df.loc[mut_details_df["Target residue
number"]==lowest_target].index.item()
            mut_details_df.at[lowest_target_index, "Codon no"] = 0
            lowest_target_codon_no =
mut_details_df.at[lowest_target_index, "Codon no"]


        #if the number of codons to recode is (larger) than the
recoding region covered by the targets
        else:
            extra_codons = num_of_codons_to_recode - targets_range

            lowest_target_index =
mut_details_df.loc[mut_details_df["Target residue
number"]==lowest_target].index.item()
```

```
            min_recode_start =
mut_details_df.at[lowest_target_index, "Residue Start Base"]

            #for an even number of spare codons, split equally at
each end
            if extra_codons % 2 == 0:
                half_extra_codons = extra_codons / 2


                mut_details_df.at[lowest_target_index, "Codon no"]
= half_extra_codons

                recode_start = min_recode_start -
(half_extra_codons * 3)

            #for an odd number of spare codons, put +1 codon
upstream than downstream
            else:
                downstream_codons = (extra_codons - 1) /2
                upstream_codons = extra_codons - downstream_codons

                mut_details_df.at[lowest_target_index, "Codon no"]
= upstream_codons

                recode_start = min_recode_start - (upstream_codons
* 3)

            lowest_target_codon_no =
mut_details_df.at[lowest_target_index, "Codon no"]


        for row in mut_details_df.index:
            if pd.isna(mut_details_df.at[row, "Codon no"]) ==
False:
                pass
            target_res_num = mut_details_df.at[row, "Target
residue number"]
            diff =  target_res_num - lowest_target
            codon_no = lowest_target_codon_no + diff
            mut_details_df.at[row, "Codon no"] = codon_no

    if max(mut_details_df["Codon no"]) > num_of_codons_to_recode:
        targets_range = max(mut_details_df["Codon no"]) -
min(mut_details_df["Codon no"])

        #if the number of codons to recode is the same distance as
the range of the target sites
        if targets_range == num_of_codons_to_recode:
            recode_start = min_recode_start
            recode_end = min_recode_end

            lowest_target_index =
mut_details_df.loc[mut_details_df["Target residue
number"]==lowest_target].index.item()
            mut_details_df.at[lowest_target_index, "Codon no"] = 0
            lowest_target_codon_no = 
mut_details_df.at[lowest_target_index, "Codon no"]

        #if the number of codons to recode is (larger) than the 
recoding region covered by the targets
        else:
            extra_codons = num_of_codons_to_recode - targets_range

            highest_target_index = 
mut_details_df.loc[mut_details_df["Target residue 
number"]==highest_target].index.item()
            min_recode_end = 
mut_details_df.at[highest_target_index, "Residue End Base"]

            #for an even number of spare codons, split equally at 
each end
            if extra_codons % 2 == 0:
                half_extra_codons = extra_codons / 2


                mut_details_df.at[highest_target_index, "Codon 
no"] = num_of_codons_to_recode - half_extra_codons

                recode_start = min_recode_start - 
(half_extra_codons * 3)

            #for an odd number of spare codons, put +1 codon 
upstream than downstream
            else:
                downstream_codons = (extra_codons - 1) /2
                upstream_codons = extra_codons - downstream_codons

                mut_details_df.at[highest_target_index, "Codon 
no"] = downstream_codons

                recode_start = min_recode_start - (upstream_codons 
* 3)

            highest_target_codon_no = 
mut_details_df.at[highest_target_index, "Codon no"]

        for row in mut_details_df.index:
            if pd.isna(mut_details_df.at[row, "Codon no"]) == 
False:
                pass
            target_res_num = mut_details_df.at[row, "Target 
residue number"]
            diff =  highest_target - target_res_num
            codon_no = highest_target_codon_no - diff
            mut_details_df.at[row, "Codon no"] = codon_no

recode_end = recode_start + recode_region_length
#need some special cases for close to the start or end of the CDS

#near the start special case
```

```
#all within the recoding regin number of amino acids
if num_of_codons_to_recode > lowest_target and
num_of_codons_to_recode > highest_target:
    recode_start = 0
    recode_end = recode_region_length
    #target_codon_no = lowest_target - 1

    for row in mut_details_df.index:
        target_res_num = mut_details_df.at[row, "Target residue
number"]
        mut_details_df.at[row, "Codon no"] = int(target_res_num -
1)

#for when the highet target residue is outisde the recoding range
if it started at the beginning of the gene

if num_of_codons_to_recode >= lowest_target and
num_of_codons_to_recode <= highest_target:
    min_recode_start = min(mut_details_df["Residue Start Base"])
    min_recode_end = max(mut_details_df["Residue End Base"])

    min_recoding_region = min_recode_end - min_recode_start

    if min_recoding_region == recode_region_length:
        recode_start = min_recode_start
        recode_end = min_recode_end

        lowest_target_index =
mut_details_df.loc[mut_details_df["Target residue
number"]==lowest_target].index.item()
        mut_details_df.at[lowest_target_index, "Codon no"] = 0
        lowest_target_codon_no =
mut_details_df.at[lowest_target_index, "Codon no"]

        for row in mut_details_df.index:
            if pd.isna(mut_details_df.at[row, "Codon no"]) ==
False:
                pass
            target_res_num = mut_details_df.at[row, "Target
residue number"]
            diff =  target_res_num - lowest_target
            codon_no = lowest_target_codon_no + diff
            mut_details_df.at[row, "Codon no"] = codon_no

    else:
        extra_bases = recode_region_length - min_recoding_region
        extra_codons = extra_bases/3

        codons_to_start = lowest_target - 1

        if codons_to_start >= extra_codons:

            if extra_codons % 2 == 0:
                recode_start = min_recode_start - (0.5 *
extra_bases)
                recode_end = min_recode_end + (0.5 * extra_bases)
            else:
                half_extra_codons_down = int(extra_codons/2)
                half_extra_codons_up = extra_codons -
half_extra_codons_down
                recode_start = int(min_recode_start -
(half_extra_codons_up * 3))
                recode_end = int(min_recode_end +
(half_extra_codons_down * 3))


            mut_details_df["Codon no"] = None
            lowest_target_start = min(mut_details_df["Residue
Start Base"])
            codons_before_lowest = (min_recode_start -
recode_start)/3
            lowest_target_index =
mut_details_df.loc[mut_details_df["Target residue
number"]==lowest_target].index.item()
            mut_details_df.at[lowest_target_index, "Codon no"] =
codons_before_lowest
            lowest_target_codon_no =
mut_details_df.at[lowest_target_index, "Codon no"]

            for row in mut_details_df.index:
                if pd.isna(mut_details_df.at[row, "Codon no"]) ==
False:
                    pass
                target_res_num = mut_details_df.at[row, "Target
residue number"]
                diff =  target_res_num - lowest_target
                codon_no = lowest_target_codon_no + diff
                mut_details_df.at[row, "Codon no"] = codon_no




        if codons_to_start < extra_codons:
            recode_start = 0
            extra_end_codons = extra_codons - codons_to_start
            recode_end = min_recode_end + (3 * extra_end_codons)

            for row in mut_details_df.index:
                target_res_num = mut_details_df.at[row, "Target
residue number"]
                mut_details_df.at[row, "Codon no"] =
int(target_res_num - 1)

mut_details_df["Codon no"] = mut_details_df["Codon
no"].astype(int)


#read gene fasta file and define the CDS
for gene_name in SeqIO.parse(ref_file_name,"fasta"):
    #print(gene_name.id)
    print(gene_name.description)
    print(repr(gene_name.seq))
    print("Gene sequence length: ", len(gene_name), "bp")
    print("\n")

if CDS_end == "end":
    CDS_end = len(gene_name.seq)
else:
    CDS_end = CDS_end


if CDS_start > 1:
    CDS_start = CDS_start - 1
    WT_CDS_seq = gene_name.seq[(CDS_start):CDS_end]
    recode_start_whole = int(recode_start + CDS_start)
    recode_end_whole = int(recode_end + CDS_start)

else:
    WT_CDS_seq = gene_name.seq[:CDS_end]
    recode_start_whole = int(recode_start)
    recode_end_whole = int(recode_end)



#check input is a length divisible by 3
val.triplet_checker(WT_CDS_seq)

#near the end special case
total_num_AAs = len(WT_CDS_seq.translate())

if highest_target > (total_num_AAs - num_of_codons_to_recode):

    if lowest_target > (total_num_AAs - num_of_codons_to_recode):

        recode_end = len(WT_CDS_seq)
        recode_start = len(WT_CDS_seq) - recode_region_length

        num_of_codons_to_recode = int((recode_end - recode_start +
1) / 3)
        highest_target_codon_no = num_of_codons_to_recode -
(total_num_AAs - highest_target) - 1

        mut_details_df["Codon no"] = None

        highest_target_index =
mut_details_df.loc[mut_details_df["Target residue
number"]==highest_target].index.item()
        mut_details_df.at[highest_target_index, "Codon no"] =
highest_target_codon_no


        for row in mut_details_df.index:
            if pd.isna(mut_details_df.at[row, "Codon no"]) ==
False:
                pass
            target_res_num = mut_details_df.at[row, "Target
residue number"]
            diff = highest_target - target_res_num
            codon_no = highest_target_codon_no - diff
            mut_details_df.at[row, "Codon no"] = codon_no

    else:
        min_recode_start = min(mut_details_df["Residue Start
Base"])
        min_recode_end = max(mut_details_df["Residue End Base"])

        min_recoding_region = min_recode_end - min_recode_start

        codons_to_end = total_num_AAs - highest_target

        extra_bases = recode_region_length - min_recoding_region
        extra_codons = extra_bases/3

        if min_recoding_region == recode_region_length:
            recode_start = min_recode_start
            recode_end = min_recode_end


            highest_target_index =
mut_details_df.loc[mut_details_df["Target residue
number"]==highest_target].index.item()
            mut_details_df.at[highest_target_index, "Codon no"] =
num_of_codons_to_recode - 1
            highest_target_codon_no =
mut_details_df.at[highest_target_index, "Codon no"]

            for row in mut_details_df.index:
                if pd.isna(mut_details_df.at[row, "Codon no"]) ==
False:
                    pass
                target_res_num = mut_details_df.at[row, "Target
residue number"]
                diff =  highest_target - target_res_num
                codon_no = highest_target_codon_no - diff
                mut_details_df.at[row, "Codon no"] = codon_no



        else:
            if codons_to_end >= extra_codons:

                if extra_codons % 2 == 0:
                    recode_start = min_recode_start - (0.5 *
extra_bases)
                    recode_end = min_recode_end + (0.5 *
extra_bases)


                else:
                    half_extra_codons_down = int(extra_codons/2)
                    half_extra_codons_up = extra_codons - 
half_extra_codons_down
                    recode_start = int(min_recode_start - 
(half_extra_codons_up * 3))
                    recode_end = int(min_recode_end + 
(half_extra_codons_down * 3))


                mut_details_df["Codon no"] = None
                highest_target_end = max(mut_details_df["Residue 
End Base"])
                codons_after_highest = (min_recode_end - 
recode_end)/3
                highest_target_index = 
mut_details_df.loc[mut_details_df["Target residue 
number"]==highest_target].index.item()
                mut_details_df.at[lowest_target_index, "Codon no"] 
= codons_after_highest
                highest_target_codon_no = 
mut_details_df.at[highest_target_index, "Codon no"]

                for row in mut_details_df.index:
                    if pd.isna(mut_details_df.at[row, "Codon no"]) 
== False:
                        pass
                    target_res_num = mut_details_df.at[row, 
"Target residue number"]
                    diff =  highest_target - target_res_num
                    codon_no = highest_target_codon_no - diff
                    mut_details_df.at[row, "Codon no"] = codon_no




            if codons_to_start < extra_codons:
                recode_start = 0
                extra_end_codons = extra_codons - codons_to_start
                recode_end = min_recode_end + (3 * 
extra_end_codons)

                for row in mut_details_df.index:
                    target_res_num = mut_details_df.at[row, 
"Target residue number"]
                    mut_details_df.at[row, "Codon no"] = 
int(target_res_num - 1)


    if CDS_start > 1:
        recode_end_whole = int(recode_end + CDS_start)
        recode_start_whole = int(recode_start + CDS_start)
    else:
        recode_end_whole = int(recode_end)
        recode_start_whole = int(recode_start)

    mut_details_df["Codon no"] = mut_details_df["Codon 
no"].astype(int)
```

```
mut_details_df["Codon no"] = mut_details_df["Codon
no"].astype(int)


#check that the input given is correct and that the target codes
for the expected residue

for row in mut_details_df.index:
    residue_no = int(mut_details_df.at[row, "Target residue
number"])
    target_AA = mut_details_df.at[row, "Target AA"]
    mut_details_df["Input AA Correct"] =
val.translate_checker(WT_CDS_seq, residue_no, target_AA)

#cancel the code if some incorrect starting amino acids given
if mut_details_df["Input AA Correct"].any() == False:
    print("\n\n\n***WARNING: One or more incorrect starting amino
acids. Please review your inputs.\nThe code will now
abort.***\n\n\n")
    print("Your inputs:")
    print(mut_details_df.loc[:, ["Mutation number", "Target AA",
"Target residue number"]])
    raise SystemExit


#check that the range of amino acids to mutate is not larger than
the recoding range
mutation_distance = max(mut_details_df["Residue End Base"]) -
min(mut_details_df["Residue Start Base"])

if mutation_distance > recode_region_length:
    print("\n\n\n***Warning: The distance between the target sites
is greater than the recoding region length. Please ensure your
recoding region length covers all target mutations.\nThe code will
now abort.***\n")
    print(f"You asked for a recoding region of
{recode_region_length} bp, but the needed recoding region length
is at least {int(mutation_distance)} bp.\n\n")

    raise SystemExit




#establish the sequence to replace, and sequences before and after
to stay the same
WT_template_seq =
gene_name.seq[recode_start_whole:recode_end_whole]
upstream_dna = gene_name.seq[:recode_start_whole]
downstream_dna = gene_name.seq[recode_end_whole:]



#make dictionary of codons with number keys and one with numbers
and amino acids
```

```
codons_to_recode = cdict.codon_dict_maker(WT_template_seq, 
key_format= "number")
codons_to_recode_let_num = cdict.codon_dict_maker(WT_template_seq, 
key_format= "letter-number")


#make reference dictionaries for all the amino acids
ref_codon_table_df = 
rif.codon_table_processor(codon_freq_input_file)

ref_codons = cdf.ref_codon_table_freqs(ref_codon_table_df)


#synonymous recoding - irrelevant to additional mutations
if syn_recode_type == "matched":

    #use that dictionary to create a new one with the specific 
frequency values
    codons_to_recode_freqs = 
cdf.codon_frequency_collector(input_dict = codons_to_recode, 
reference_dict = ref_codons, type = "value")

    #create a dictionary with all the frequencies for the amino 
acids in this sequence for each codon
    codons_to_recode_all_freqs = 
cdf.codon_frequency_collector(input_dict = codons_to_recode, 
reference_dict = ref_codons, type = "dataframe")


    #calculate the differences for each possible codon to the 
original
    recode_freq_diffs = 
cdf.codon_frequency_difference_calc(codons_to_recode_let_num, 
ref_codons)

    #add the differences in frequency to "the" dataframe
    codons_to_recode_abs_diffs = 
cdf.codon_freq_diff_adder(codons_to_recode_let_num 
,codons_to_recode_all_freqs, recode_freq_diffs)

    #choose which codons to use for synonymous recoding
    codons_to_use_syn = 
cdf.codon_freq_selector(codons_to_recode_abs_diffs)


if syn_recode_type == "highest" or syn_recode_type == "lowest":

    #use that dictionary to create a new one with the specific 
frequency values
    codons_to_recode_freqs = 
cdf.codon_frequency_collector(input_dict = codons_to_recode, 
reference_dict = ref_codons, type = "value")

    #create a dictionary with all the frequencies for the amino 
acids in this sequence for each codon
```

```
    codons_to_recode_all_freqs =
cdf.codon_frequency_collector(input_dict = codons_to_recode,
reference_dict = ref_codons, type = "dataframe")

    codons_to_recode_choices_freqs = {}
    #remove input codon from list
    for let_num, seq in codons_to_recode_let_num.items():
        if seq == Seq.Seq("ATG") or seq == Seq.Seq("TGG"):

            codons_to_recode_choices_freqs[let_num] =
ref_codon_table_df.loc[ref_codon_table_df["DNA"] == str(seq)]
        else:
            current_df = codons_to_recode_all_freqs[let_num]
            codons_to_recode_choices_freqs[let_num] =
current_df.loc[current_df["DNA"] != str(seq)]

#make the list of codons to use depending on recoding type
    codons_to_use_syn = {}

    if syn_recode_type == "highest":

        for codon_num_let, seq_df in
codons_to_recode_choices_freqs.items():
            max_freq_codon = max(seq_df["Fraction"])
            mutated_res_df_chosen = seq_df.loc[seq_df["Fraction"]
== max_freq_codon, "DNA"]

            #tie breaker for instances with same fraction usage -
hopefully number won't ever have duplicate values
            if len(mutated_res_df_chosen) > 1:
                max_number_codon = max(seq_df["Number"])
                max_number_codon_seq = seq_df.loc[seq_df["Number"]
== max_number_codon, "DNA"].item()
                codons_to_use_syn[codon_num_let] =
max_number_codon_seq

            else:
                codons_to_use_syn[codon_num_let] =
seq_df.loc[seq_df["Fraction"] == max_freq_codon, "DNA"].item()

    if syn_recode_type == "lowest":

        for codon_num_let, seq_df in
codons_to_recode_choices_freqs.items():
            min_freq_codon = min(seq_df["Fraction"])
            mutated_res_df_chosen = seq_df.loc[seq_df["Fraction"]
== min_freq_codon, "DNA"]

            #tie breaker
            if len(mutated_res_df_chosen) > 1:
                min_number_codon = max(seq_df["Number"])
                min_number_codon_seq = seq_df.loc[seq_df["Number"]
== min_number_codon, "DNA"].item()
                codons_to_use_syn[codon_num_let] =
min_number_codon_seq
```

```
            else:
                codons_to_use_syn[codon_num_let] =
seq_df.loc[seq_df["Fraction"] == min_freq_codon, "DNA"].item()


if syn_recode_type == "alternating matched" or syn_recode_type ==
"alternating random" or syn_recode_type == "alternating highest"
or syn_recode_type == "alternating lowest":
    #check input has been given suitably
    if alternating_repeat == "N/A" or alternating_repeat <= 0 or
pd.isna(alternating_repeat) == True:
        print("\n\n\n***ERROR: No value or an invalid value was
set for the alternating pattern of the codons to
recode.***\n\n\n")
        alternating_repeat = int(input("Please enter a positive
integrer for the alternating repeat value: "))

    if alternating_repeat > (0.5 * num_of_codons_to_recode):
        proceed_alt = input("The chosen repeat value is greater
than half of the total number of codons being recoded so only 2 or
fewer codons will be mutated.\n\nDo you wish to proceed? Y/N \n")

        if proceed_alt == "N" or proceed_alt == "n" or proceed_alt
== "NO" or proceed_alt == "No" or proceed_alt == "no":
            alternating_repeat = int(input("Please enter a
positive integer for the alternating repeat value: "))

        elif proceed_alt == "Y" or proceed_alt =="y" or
proceed_alt =="YES" or proceed_alt == "Yes" or proceed_alt ==
"yes":
           pass
        else:
            proceed_alt = input("\n\nThe input given is not valid.
Please try again.\n\nThe chosen repeat value is greater than half
of the total number of codons being recoded so only 2 or fewer
codons will be mutated.\n\nDo you wish to proceed? Y/N \n")
            if proceed_alt == "N" or proceed_alt == "n" or
proceed_alt == "NO" or proceed_alt == "No" or proceed_alt == "no":
                alternating_repeat = int(input("\nPlease enter a
positive integer for the alternating repeat value: "))
            elif proceed_alt == "Y" or proceed_alt =="y" or
proceed_alt =="YES" or proceed_alt == "Yes" or proceed_alt ==
"yes":
               pass
            else:
                proceed_alt = input("\n\nThe input given is not
valid. Please try again.\n\nThe chosen repeat value is greater
than half of the total number of codons being recoded so only 2 or
fewer codons will be mutated.\n\nDo you wish to proceed? Y/N \n")
                if proceed_alt == "N" or proceed_alt == "n" or
proceed_alt == "NO" or proceed_alt == "No" or proceed_alt == "no":
                    alternating_repeat = int(input("\nPlease enter
a positive integer for the alternating repeat value: "))
                elif proceed_alt == "Y" or proceed_alt =="y" or
proceed_alt =="YES" or proceed_alt == "Yes" or proceed_alt ==
"yes":
```

```
                   pass
                else:
                    print("\n-------------------------------------
-----------------------")
                    print(f"\n***Warning, your input was invalid
so the code will continue with the value given. Your repair
template will recode every {alternating_repeat} codons. If you do
not want this, modify the input spreadsheet and rerun the
programme.***")


    if alternating_repeat == "N/A" or alternating_repeat <= 0 or
pd.isna(alternating_repeat) == True:
        print("\n\n\n***ERROR: An invalid value was set for the
alternating pattern of the codons to recode.***\n\n\n")
        alternating_repeat = int(input("Please enter a positive
integer for the alternating repeat value: "))

    if alternating_repeat == "N/A" or alternating_repeat <= 0 or
pd.isna(alternating_repeat) == True:
        print("\n\n\n***ERROR: An invalid value was set for the
alternating pattern of the codons to recode.***\n\n\n")
        alternating_repeat = int(input("Last chance - please enter
a positive integer for the alternating repeat value: "))

    if alternating_repeat == "N/A" or alternating_repeat <= 0 or
pd.isna(alternating_repeat) == True:
        print("\n\n\n\nYou failed to provide an appropriate input
so the programme will be cancelled.\n\nIf you wish to try again,
either modify the input spreadsheet or provide a suitable value
when prompted in the console.\n")
        raise SystemExit


if syn_recode_type == "alternating matched" or syn_recode_type ==
"alternating random":


    #determine which codon numbers in range are to be mutated and
which are not
    num_of_codons_to_mutate = int(num_of_codons_to_recode /
alternating_repeat)
    n_terms = list(range(num_of_codons_to_mutate))
    codon_nums_to_recode = []

    for n in n_terms:
        codon_num = n * alternating_repeat
        codon_nums_to_recode.append(codon_num)

    #ensure that target codons are always recoded even if they
don't fit the alternating pattern

    target_codons_nos = list(mut_details_df["Codon no"])

    for codon_no in target_codons_nos:
        if codon_no not in codon_nums_to_recode:
            codon_nums_to_recode.append(codon_no)


    codon_nums_all = list(codons_to_recode.keys())

    #split the codons to be mutated into a separate dictionary
from the ones to stay the same
    codons_to_keep_WT = {}
    specific_codons_to_recode = {}

    for numbers in codon_nums_all:
        if numbers not in codon_nums_to_recode:
            codons_to_keep_WT[numbers] = codons_to_recode[numbers]

        if numbers in codon_nums_to_recode:
            specific_codons_to_recode[numbers] =
codons_to_recode[numbers]


    for numbers in codon_nums_to_recode:
        if numbers not in codon_nums_to_recode:
            codons_to_keep_WT = codons_to_recode[numbers]

    if syn_recode_type == "alternating matched":
        #on only the codons to recode
        #use that dictionary to create a new one with the specific
frequency values
        codons_to_recode_freqs =
cdf.codon_frequency_collector(input_dict = codons_to_recode,
reference_dict = ref_codons, type = "value")

        #create a dictionary with all the frequencies for the
amino acids in this sequence for each codon
        codons_to_recode_all_freqs =
cdf.codon_frequency_collector(input_dict = codons_to_recode,
reference_dict = ref_codons, type = "dataframe")


        #calculate the differences for each possible codon to the
original
        recode_freq_diffs =
cdf.codon_frequency_difference_calc(codons_to_recode_let_num,
ref_codons)

        #add the differences in frequency to "the" dataframe
        codons_to_recode_abs_diffs =
cdf.codon_freq_diff_adder(codons_to_recode_let_num
,codons_to_recode_all_freqs, recode_freq_diffs)

        #choose which codons to use for synonymous recoding
        codons_to_use_syn =
cdf.codon_freq_selector(codons_to_recode_abs_diffs)


    if syn_recode_type == "alternating random":
        #add letters to dictionary
        specific_codons_to_recode_let_num = {}

        for keys, seq in specific_codons_to_recode.items():
            let_num = str(seq.translate()) + str(keys)
            specific_codons_to_recode_let_num[let_num] = seq


        #make a dictionary of the alternate codons to the input
sequence
        alt_codons_to_recode =
cdict.alt_codons(specific_codons_to_recode_let_num)

        #randomly select which of these to use for each codon
        codons_to_use_syn =
cdict.Syn_random_recoder(alt_codons_to_recode)


    #combine the unchanged codons with the changed codons

    codons_to_keep_WT_let_num = {}

    for codon_num, seq in codons_to_keep_WT.items():
        translation = seq.translate()
        codon_num_let = str(translation) + str(codon_num)

        codons_to_keep_WT_let_num[codon_num_let] = seq

    codons_to_use_syn.update(codons_to_keep_WT_let_num)


if syn_recode_type == "alternating highest" or syn_recode_type ==
"alternating lowest":

    num_of_codons_to_mutate = int(num_of_codons_to_recode /
alternating_repeat)
    n_terms = list(range(num_of_codons_to_mutate))
    codon_nums_to_recode = []

    for n in n_terms:
        codon_num = n * alternating_repeat
        codon_nums_to_recode.append(codon_num)

    #ensure that target codons are always recoded even if they
don't fit the alternating pattern

    target_codons_nos = list(mut_details_df["Codon no"])

    for codon_no in target_codons_nos:

        if codon_no not in codon_nums_to_recode:
            codon_nums_to_recode.append(codon_no)


    codon_nums_all = list(codons_to_recode.keys())
    codons_to_keep_WT = {}
    specific_codons_to_recode = {}

    for numbers in codon_nums_all:
        if numbers not in codon_nums_to_recode:
            translate = codons_to_recode[numbers].translate()
            let_num = str(translate) + str(numbers)
            codons_to_keep_WT[let_num] = codons_to_recode[numbers]

        if numbers in codon_nums_to_recode:
            #translate = codons_to_recode[numbers].translate()
            #let_num = str(translate) + str(numbers)
            specific_codons_to_recode[numbers] =
codons_to_recode[numbers]



    for numbers in codon_nums_to_recode:
        if numbers not in codon_nums_to_recode:
            codons_to_keep_WT = codons_to_recode[numbers]

    #use that dictionary to create a new one with the specific
frequency values
    codons_to_recode_freqs =
cdf.codon_frequency_collector(input_dict =
specific_codons_to_recode, reference_dict = ref_codons, type =
"value")

    #create a dictionary with all the frequencies for the amino
acids in this sequence for each codon
    codons_to_recode_all_freqs =
cdf.codon_frequency_collector(input_dict =
specific_codons_to_recode, reference_dict = ref_codons, type =
"dataframe")

    codons_to_recode_choices_freqs = {}
    #remove input codon from list unless it's Met or Trp
    for let_num, df in codons_to_recode_all_freqs.items():
        input_codon = codons_to_recode_let_num[let_num]
        if input_codon == Seq.Seq("ATG") or input_codon ==
Seq.Seq("TGG"):

            codons_to_recode_choices_freqs[let_num] =
ref_codon_table_df.loc[ref_codon_table_df["DNA"] ==
str(input_codon)]
        else:
            current_df = codons_to_recode_all_freqs[let_num]
            codons_to_recode_choices_freqs[let_num] =
current_df.loc[current_df["DNA"] != str(input_codon)]

    #recode based on input type
    codons_to_use_syn = {}

    if syn_recode_type == "alternating highest":
        for codon_num_let, seq_df in
codons_to_recode_choices_freqs.items():
            max_freq_codon = max(seq_df["Fraction"])
            max_freq_codon_seq = seq_df.loc[seq_df["Fraction"] ==
max_freq_codon, "DNA"]
            if len(max_freq_codon_seq) > 1:
                max_number_codon = max(seq_df["Number"])
                max_freq_codon_seq = seq_df.loc[seq_df["Number"]
== max_number_codon, "DNA"].item()
                codons_to_use_syn[codon_num_let] =
max_freq_codon_seq

            else:
             codons_to_use_syn[codon_num_let] =
max_freq_codon_seq.item()



    if syn_recode_type == "alternating lowest":

        for codon_num_let, seq_df in
codons_to_recode_choices_freqs.items():
            min_freq_codon = min(seq_df["Fraction"])
            min_freq_codon_seq = seq_df.loc[seq_df["Fraction"] ==
min_freq_codon, "DNA"]
            if len(min_freq_codon_seq) > 1:
                min_number_codon = min(seq_df["Number"])
                min_freq_codon_seq = seq_df.loc[seq_df["Number"]
== min_number_codon, "DNA"].item()
                codons_to_use_syn[codon_num_let] =
min_freq_codon_seq

            else:
             codons_to_use_syn[codon_num_let] =
min_freq_codon_seq.item()




    #combine the unchanged codons with the changed codons

    codons_to_keep_WT_let_num = {}

    for codon_num, seq in codons_to_keep_WT.items():
        codons_to_keep_WT_let_num[codon_num] = seq

    codons_to_use_syn.update(codons_to_keep_WT_let_num)


if syn_recode_type == "random":
    #make a dictionary of the alternate codons to the input
sequence
    alt_codons_to_recode =
cdict.alt_codons(codons_to_recode_let_num)

    #randomly select which of these to use for each codon
    codons_to_use_syn =
cdict.Syn_random_recoder(alt_codons_to_recode)


#add in the nonsynonymous mutations

if nonsyn_recode_type == "highest" or nonsyn_recode_type ==
"lowest":

    nonsyn_ref_dict = ref_codons

if nonsyn_recode_type == "random":
    nonsyn_ref_dict = cdict.alt_codons(codons_to_recode_let_num)
    nonsyn_ref_dict = {}

    for row in mut_details_df.index:
        output_AA = mut_details_df.at[row, "Replacement AA"]
        nonsyn_ref_dict[output_AA] =
cdict.ref_codon_table(output_AA)


#codons_to_use_nonsyn = cdf.non_syn_mutator(target_AA,
target_codon_no, new_AA = output_AA, input_dict =
codons_to_use_syn, type = nonsyn_recode_type, ref_dict =
nonsyn_ref_dict )

codons_to_use_nonsyn = codons_to_use_syn


for row in mut_details_df.index:
    target_AA = mut_details_df.at[row, "Target AA"]
    target_codon_no = mut_details_df.at[row, "Codon no"]
    output_AA = mut_details_df.at[row,"Replacement AA"]

    current_mut_nonsyn_codon = cdf.non_syn_mutator(target_AA,
target_codon_no, new_AA = output_AA, input_dict =
codons_to_use_nonsyn, type = nonsyn_recode_type, ref_dict =
nonsyn_ref_dict)

    codons_to_use_nonsyn = current_mut_nonsyn_codon

    target_key = str(output_AA) + str(target_codon_no)



#construct the final recoded sequences

synonymous_repair = stitch.sequence_constructor(codons_to_use_syn,
type = "letter-number")
nonsynonymous_repair =
stitch.sequence_constructor(codons_to_use_nonsyn, type = "letter-
number")

#check all the modifications were as expected
#adjust target codon number to what it would be by normal counting
rather than python counting
```

```
target_codon_no_not_py = target_codon_no + 1


#check all is as expected

for row in mut_details_df.index:
    target_AA = mut_details_df.at[row, "Target AA"]
    output_AA = mut_details_df.at[row, "Replacement AA"]
    codon_no_py = mut_details_df.at[row, "Codon no"] + 1

    mut_details_df.at[row, "Syn mutation correct"] =
val.translate_checker(synonymous_repair, codon_no_py, target_AA)

    mut_details_df.at[row, "Nonsyn mutation correct"] =
val.translate_checker(nonsynonymous_repair, codon_no_py,
output_AA)


#error if some of these fail

if mut_details_df["Syn mutation correct"].any() == False or
mut_details_df["Nonsyn mutation correct"].any() == False:
    print("\n\n\n***WARNING - Errors in recoding or mutating
detected***\n\n\n")



#create the final repair sequence including the homology arms

upstream_hom_arm = gene_name.seq[(recode_start_whole -
hom_arm_length):recode_start_whole]
downstream_hom_arm = gene_name.seq[recode_end_whole:
(recode_end_whole + hom_arm_length)]

WT_entire_repair_region = upstream_hom_arm + WT_template_seq +
downstream_hom_arm
entire_syn_repair = upstream_hom_arm + synonymous_repair +
downstream_hom_arm
entire_nonsyn_repair = upstream_hom_arm + nonsynonymous_repair +
downstream_hom_arm



#construct "gene" sequences for primer design
integrated_synonymous, WT_recode_region =
stitch.mut_seq_integrator(repair_seq = synonymous_repair, ref_seq
= gene_name.seq, repair_start = recode_start_whole, repair_end =
recode_end_whole, WT_repair_seq= "Yes")
integrated_nonsynonymous = stitch.mut_seq_integrator(repair_seq =
nonsynonymous_repair, ref_seq = gene_name.seq, repair_start =
recode_start_whole, repair_end = recode_end_whole, WT_repair_seq=
"No")

#design screening primers
```

```
screening_primers_df_syn  =
primers.screening_primer_designer(gene_name.seq,
integrated_synonymous, recode_start_whole, recode_end_whole)
screening_primers_df_nonsyn  =
primers.screening_primer_designer(gene_name.seq,
integrated_nonsynonymous, recode_start_whole, recode_end_whole)

#design primers to generate the repair template
syn_repair_template_primers =
primers.repair_primer_designer(entire_syn_repair, hom_arm_length,
downstream_dna)
nonsyn_repair_template_primers =
primers.repair_primer_designer(entire_nonsyn_repair,
hom_arm_length, downstream_dna)

#repair_template_primers = [syn_repair_template_primers,
nonsyn_repair_template_primers]

#repair_template_primers_df =
pd.DataFrame(repair_template_primers)
#repair_template_primers_df.index = ["Synonymous repair",
"Nonsynonymous repair"]

#do an alignment

#create a pariwise alignment object
aligner = Align.PairwiseAligner(target_internal_open_gap_score = -
10.0, query_internal_open_gap_score = -10.0)



syn_alignment = aligner.align(WT_entire_repair_region,
entire_syn_repair)
for alignment1 in sorted(syn_alignment):
    #print("Score = %.1f:" % alignment1.score)
    #print(alignment1)
    syn_score = alignment1.score
alignment_str_syn = str(alignment1)
alignment_str_syn = alignment_str_syn.replace("target", "WT
sequence").replace("query", "Syn. repair").replace("\n
", "\n                ")
alignment_str_syn = alignment_str_syn.replace("Syn. repair
", "Syn. repair          ")
#print(alignment_str_syn)

nonsyn_alignment = aligner.align(WT_entire_repair_region,
entire_nonsyn_repair)
for alignment2 in sorted(nonsyn_alignment):
    #print("Score = %.1f:" % alignment2.score)
    nonsyn_score = alignment2.score
alignment_str_nonsyn = str(alignment2)
alignment_str_nonsyn = alignment_str_nonsyn.replace("target", "WT
sequence").replace("query", "Nonsyn. repair").replace("\n
", "\n                ")
alignment_str_nonsyn = alignment_str_nonsyn.replace("Nonsyn.
repair          ", "Nonsyn. repair       ")
```

```
#print(alignment_str_nonsyn)




#format some outputs

WT_repair_seq_spaced =
formats.codon_spacing(WT_entire_repair_region)
syn_repair_spaced = formats.codon_spacing(entire_syn_repair)
nonsyn_repair_spaced = formats.codon_spacing(entire_nonsyn_repair)

WT_repair_translate = WT_entire_repair_region.translate()
syn_repair_translate = entire_syn_repair.translate()
nonsyn_repair_translate = entire_nonsyn_repair.translate()

WT_repair_translate_spaced =
formats.protein_align_codon(WT_repair_translate)
syn_repair_translate_spaced =
formats.protein_align_codon(syn_repair_translate)
nonsyn_repair_translate_spaced =
formats.protein_align_codon(nonsyn_repair_translate)

syn_repair_mutations_count =
val.mutation_counter(entire_syn_repair, WT_entire_repair_region)
nonsyn_repair_mutations_count =
val.mutation_counter(entire_nonsyn_repair,
WT_entire_repair_region)

syn_repair_primers_output = ""

for category, item in syn_repair_template_primers.items():
    if type(item) == float:
        item = '{:.1f}'.format(item)
    syn_repair_primers_output += category
    syn_repair_primers_output += ": "
    syn_repair_primers_output += str(item)
    syn_repair_primers_output += "\n"

nonsyn_repair_primers_output = ""

for category, item in nonsyn_repair_template_primers.items():
    if type(item) == float:
        item = '{:.1f}'.format(item)
    nonsyn_repair_primers_output += category
    nonsyn_repair_primers_output += ": "
    nonsyn_repair_primers_output += str(item)
    nonsyn_repair_primers_output += "\n"

if syn_recode_type == "alternating matched" or syn_recode_type ==
"alternating highest" or syn_recode_type == "alternating lowest"
or syn_recode_type == "alternating random":
    alternating_info = f"Alternating recoding every
{alternating_repeat} codons"
else:
    alternating_info = ""
```

```
mut_details_df.sort_values("Target residue number", inplace =
True)

mutations = []

for row in mut_details_df.index:
    target_AA = mut_details_df.at[row, "Target AA"]
    target_res_num = mut_details_df.at[row, "Target residue
number"]
    output_AA = mut_details_df.at[row, "Replacement AA"]
    mutation = str(target_AA) + str(target_res_num) +
str(output_AA)
    mutations.append(mutation)

mutations_text = str(mutations).replace("[", "").replace("]",
"").replace("'", "")


output_file = open(f"{job_name}.txt", "w")

file_lines = ["Job request details\n",
              f"Job name: {job_name}\n",
              f"Number of Nonsynonymous mutations:
{num_of_mutations}\n"
              f"Mutations: {mutations_text}\n",
              f"Synonymous recoding type: {syn_recode_type}\n",
              f"Nonsynonymous recode type:
{nonsyn_recode_type}\n",
              f"Homology arm length (bp): {hom_arm_length}\n",
              f"Recoding region length (bp):
{recode_region_length}\n",
              f"Total repair length (bp): {(2*hom_arm_length) +
recode_region_length}\n",
              f"{alternating_info}\n",
              "\n",
              "\n",
              "Repair templates\n",
              f"WT repair region sequence:
\t\t{WT_repair_seq_spaced}\n",
              f"WT translation:
\t\t\t{WT_repair_translate_spaced}\n",
              f"Synonymous repair region sequence:
\t{syn_repair_spaced}\n",
              f"Synonymous repair translation:
\t\t{syn_repair_translate_spaced}\n",
              f"Nonsynonymous repair region sequence:
\t{nonsyn_repair_spaced}\n",
              f"Nonsynonymous repair translation:
\t{nonsyn_repair_translate_spaced}\n",
              "\n",
              f"Number of mutations in the synonymous repair
template: {syn_repair_mutations_count}\n",
              f"Number of mutations in the nonsynonymous repair
template: {nonsyn_repair_mutations_count}\n",
              "\n",
              "\n",
              "Screening primers\n",
              "Synonymous repair\n",
              "\n",
              f"{screening_primers_df_syn}\n",
              "\n",
              "\n",
              "Nonsynonymous primers\n"
              f"{screening_primers_df_nonsyn}",
              "\n",
              "\n",
              "Repair template primers\n",
              "Synonymous\n",
              f"{syn_repair_primers_output}\n",
              "\n",
              "Nonsynonymous\n",
              f"{nonsyn_repair_primers_output}\n",
              "\n",
              f"WT sequence (no spaces):
{WT_entire_repair_region}\n",
              f"Synonymous sequence (no spaces):
{entire_syn_repair}\n",
              f"Nonsynonymous sequence (no spaces):
{entire_nonsyn_repair}\n",
              "\n",
              "\n",
              "Alignments\n",
              "Synonymous Repair\n",
              f"Score = {syn_score}\n",
              f"{alignment_str_syn}\n",
              "\n",
              "Nonsynonymous\n",
              f"Score = {nonsyn_score}\n",
              f"{alignment_str_nonsyn}\n"

    ]

output_file.writelines(file_lines)
output_file.close()

#print confirmation message to make it clearer that it worked
print(f"\n\n\nYour repair template designs have completed
successfully. Please check your folder for a file with the name
'{job_name}.txt'\n")
print("\t.\t.\n", "\n\t\___/\n\n\n")
```