SUPPLEMENTAL CODE

BIOSURFER FOR SYSTEMATIC TRACKING OF REGULATORY MECHANISMS LEADING TO PROTEIN ISOFORM DIVERSITY

This supplementary file contains the source code for **Biosurfer (main)** and **Biosurfer-Analysis**, which accompany the manuscript.

- **Biosurfer (main)** is the core codebase of the tool and is also available on GitHub: https://github.com/sheynkman-lab/biosurfer
- **Biosurfer-Analysis** includes analysis scripts utilized in the accompanying manuscript to demonstrate the application of Biosurfer. Can be found on GitHub: https://github.com/sheynkman-lab/biosurfer_analysis
- All the input, intermediate, and final output files can be found on Zenodo: https://zenodo.org/records/13243233

Biosurfer (main) codebase

The directory structure of the **Biosurfer** codebase is outlined below:

```
biosurfer/
├── analysis
│   ├── genome_wide_analysis.py
│   ├── load_gencode_database.py
│   └── plot_biosurfer.py
├── core
│   ├── alignments.py
│   ├── constants.py
│   ├── database.py
│   ├── helpers.py
│   ├── splice_events.py
│   └── models
│       ├── base.py
│       ├── biomolecules.py
│       ├── features.py
│       └── nonpersistent.py
└── plots
    └── plotting.py
```

genome_wide_alignment_analysis.py

```
# %%
import multiprocessing as mp
import sys
from itertools import chain, starmap
from operator import attrgetter
from pathlib import Path

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from biosurfer.core.alignments import (CodonAlignment, ProteinAlignment,
                                       SeqAlignCat, TranscriptAlignment)
from biosurfer.core.constants import APPRIS, CTerminalChange, NTerminalChange
from biosurfer.core.database import Database
from biosurfer.core.helpers import ExceptionLogger
from biosurfer.core.models.biomolecules import (ORF, GencodeTranscript, Gene,
PacBioTranscript,
                                                Protein,
                                                Transcript)
from biosurfer.core.splice_events import (BasicTranscriptEvent,
                                          CompoundTranscriptEvent, SpliceEvent,
                                          get_event_code)
from IPython.display import display
from more_itertools import first, one, only
from sqlalchemy import func, select, and_
from tqdm import tqdm
import os
import pickle



def run_hybrid_alignment_for_all_genes(db_name, output_dir: 'Path', gencode: bool =
False, gene_to_anchor_tx: dict[str, str] = None):
    """ Runs hybrid alignment on input database and saves tables of p-blocks and c-blocks.
    Args:
        db_name: User input database name
        output_path: Directory to write output to.

    Returns:
      Nothing
    """

    # plt.rcParams['svg.fonttype'] = 'none'
```

```
  # sns.set_style('whitegrid')

  cblock_dir = output_dir/'cblock-tables'
  log_dir = output_dir/'alignment-errors'
  for dir in (cblock_dir, log_dir):
    dir.mkdir(exist_ok=True)

  tqdm.write('Retrieving cblock tables')
  cblock_df = get_cblocks(db_name, cblock_dir, log_dir, gencode, gene_to_anchor_tx)
  tqdm.write('Assembling pblock table')
  get_pblocks(cblock_df, output_dir)


def process_chr(chr: str, db_name: str, log_file: 'Path', gencode: bool, gene_to_anchor_tx:
dict[str, str]):
  db = Database(db_name)
  gene_to_gc_transcripts: dict[str, list[str]] = dict()
  gene_to_pb_transcripts: dict[str, list[str]] = dict()
  cblock_records = []

  with open(log_file, 'w') as flog:
    def process_gene(gene_name: str):
      out = []
      with db.get_session() as session:
        gc_transcripts: dict[str, 'GencodeTranscript'] =
GencodeTranscript.from_names(session, gene_to_gc_transcripts[gene_name])
        pb_transcripts: dict[str, 'PacBioTranscript'] = {tx.accession: tx for tx in
PacBioTranscript.from_accessions(session, gene_to_pb_transcripts[gene_name]).values()}

        # choose anchor isoform for gene
        if gene_to_anchor_tx:
          anchor_id = gene_to_anchor_tx.get(gene_name, None)
          if anchor_id in gc_transcripts:
            anchor = gc_transcripts[anchor_id]
          elif anchor_id in pb_transcripts:
            anchor = pb_transcripts[anchor_id]
          else:
            anchor = None
        elif gc_transcripts:  # by default, use APPRIS principal as anchor
          anchor = max(gc_transcripts.values(), key=attrgetter('appris'))
          if anchor.appris is not APPRIS.PRINCIPAL:
            anchor = None
        else:
          anchor = None
```

```
        if not anchor or not anchor.protein or not anchor.sequence:
            return out

        principal_length = anchor.protein.length
        anchor_start_codon =
anchor.get_genome_coord_from_transcript_coord(anchor.primary_orf.transcript_start - 1)
        anchor_stop_codon =
anchor.get_genome_coord_from_transcript_coord(anchor.primary_orf.transcript_stop - 1)

        if gencode:
            alt_transcripts = [tx for tx in chain(gc_transcripts.values(), pb_transcripts.values())
if tx is not anchor]
        else:
            alt_transcripts = [tx for tx in pb_transcripts.values() if tx is not anchor]

        for alternative in alt_transcripts:
            pblocks = ()
            with ExceptionLogger(info=f'{anchor}, {alternative}', output=flog):
                other_start_codon =
alternative.get_genome_coord_from_transcript_coord(alternative.primary_orf.transcript_st
art - 1)
                other_stop_codon =
alternative.get_genome_coord_from_transcript_coord(alternative.primary_orf.transcript_st
op - 1)
                anchor_starts_upstream = anchor_start_codon <= other_start_codon
                anchor_stops_upstream = anchor_stop_codon <= other_stop_codon

                alternative_length = alternative.protein.length
                tx_aln = TranscriptAlignment.from_transcripts(anchor, alternative)
                cd_aln = CodonAlignment.from_proteins(anchor.protein, alternative.protein)
                pr_aln = ProteinAlignment.from_proteins(anchor.protein, alternative.protein)
                pblocks = pr_aln.blocks
                anchor_start_cblock = one(cd_aln.anchor_blocks.at(0)).data
                other_start_cblock = one(cd_aln.other_blocks.at(0)).data
                anchor_stop_cblock = one(cd_aln.anchor_blocks.at(principal_length - 1)).data
                other_stop_cblock = one(cd_aln.other_blocks.at(alternative_length - 1)).data
            for p, pblock in enumerate(pblock for pblock in pblocks if pblock.category is not
SeqAlignCat.MATCH):
                for c, cblock in enumerate(pr_aln.pblock_to_cblocks[pblock]):
                    tblock = cd_aln.cblock_to_tblock.get(cblock)
                    events = tx_aln.block_to_events.get(tblock, ())
                    row = {
                        'anchor': anchor.name,
                        'other': alternative.name,
```

```
                    'pblock_number': p,
                    'pblock_category': pblock.category.name,
                    'pblock_anchor_start': pblock.anchor_range.start,
                    'pblock_anchor_stop': pblock.anchor_range.stop,
                    'pblock_other_start': pblock.other_range.start,
                    'pblock_other_stop': pblock.other_range.stop,
                    'cblock_number': c,
                    'cblock_category': cblock.category.name,
                    'cblock_anchor_start': cblock.anchor_range.start,
                    'cblock_anchor_stop': cblock.anchor_range.stop,
                    'cblock_other_start': cblock.other_range.start,
                    'cblock_other_stop': cblock.other_range.stop,
                    'tblock_category': tblock.category.name if tblock else '',
                    'tblock_anchor_start': tblock.anchor_range.start if tblock else '',
                    'tblock_anchor_stop': tblock.anchor_range.stop if tblock else '',
                    'tblock_other_start': tblock.other_range.start if tblock else '',
                    'tblock_other_stop': tblock.other_range.stop if tblock else '',
                    'events': get_event_code(events),
                    'compound_splicing':
any(set(events).intersection(compound_event.members) for compound_event in
tx_aln.events if isinstance(compound_event, SpliceEvent) and
len(compound_event.members) > 1),
                    'affects_up_start': anchor_starts_upstream and cblock is
anchor_start_cblock or not anchor_starts_upstream and cblock is other_start_cblock,
                    'affects_down_start': anchor_starts_upstream and cblock is
other_start_cblock or not anchor_starts_upstream and cblock is anchor_start_cblock,
                    'affects_up_stop': anchor_stops_upstream and cblock is
anchor_stop_cblock or not anchor_stops_upstream and cblock is other_stop_cblock,
                    'affects_down_stop': anchor_stops_upstream and cblock is
other_stop_cblock or not anchor_stops_upstream and cblock is anchor_stop_cblock,
                    'up_start_events': '',
                    'down_start_events': '',
                    'up_stop_events': '',
                    'down_stop_events': '',
                    'anchor_length': principal_length,
                    'other_length': alternative_length,
                    'cblock_anchor_seq':
anchor.protein.sequence[cblock.anchor_range.start:cblock.anchor_range.stop],
                    'cblock_other_seq':
alternative.protein.sequence[cblock.other_range.start:cblock.other_range.stop],
                }
                if cblock is anchor_start_cblock:
```

```
                    start_events = get_event_code(i.data for i in
tx_aln.anchor_events.overlap(anchor.primary_orf.transcript_start - 1,
anchor.primary_orf.transcript_start + 2) if isinstance(i.data, BasicTranscriptEvent))
                    if anchor_starts_upstream:
                      row['up_start_events'] = start_events
                    else:
                      row['down_start_events'] = start_events
                  elif cblock is other_start_cblock:
                    start_events = get_event_code(i.data for i in
tx_aln.other_events.overlap(alternative.primary_orf.transcript_start - 1,
alternative.primary_orf.transcript_start + 2) if isinstance(i.data, BasicTranscriptEvent))
                    if anchor_starts_upstream:
                      row['down_start_events'] = start_events
                    else:
                      row['up_start_events'] = start_events
                  if cblock is anchor_stop_cblock:
                    stop_events = get_event_code(i.data for i in
tx_aln.anchor_events.overlap(anchor.primary_orf.transcript_stop - 3,
anchor.primary_orf.transcript_stop) if isinstance(i.data, BasicTranscriptEvent))
                    if anchor_stops_upstream:
                      row['up_stop_events'] = stop_events
                    else:
                      row['down_stop_events'] = stop_events
                  elif cblock is other_stop_cblock:
                    stop_events = get_event_code(i.data for i in
tx_aln.other_events.overlap(alternative.primary_orf.transcript_stop - 3,
alternative.primary_orf.transcript_stop) if isinstance(i.data, BasicTranscriptEvent))
                    if anchor_stops_upstream:
                      row['down_stop_events'] = stop_events
                    else:
                      row['up_stop_events'] = stop_events
                  out.append(row)
        return out

    tqdm.write(f'Loading gene and transcript names for {chr}...')
    with db.get_session() as session:
      if gene_to_anchor_tx:
        condition = and_((Gene.chromosome_id == chr),
Gene.name.in_(gene_to_anchor_tx))
      else:
        condition = (Gene.chromosome_id == chr)
      rows = session.execute(
          select(Gene.name, Transcript.name, Transcript.accession, Transcript.type).
          select_from(Protein).
```

```
            join(Protein.orf).
            join(ORF.transcript).
            join(Transcript.gene).
            where(condition)
        ).all()
        for gene_name, tx_name, tx_acc, tx_type in rows:
          gc_txs = gene_to_gc_transcripts.setdefault(gene_name, [])
          pb_txs = gene_to_pb_transcripts.setdefault(gene_name, [])
          if tx_type == 'gencodetranscript':
            gc_txs.append(tx_name)
          elif tx_type == 'pacbiotranscript':
            pb_txs.append(tx_acc)

    t = tqdm(desc='Processing genes', total=len(gene_to_gc_transcripts), unit='gene',
file=sys.stdout)
    for result in map(process_gene, gene_to_gc_transcripts.keys()):
      cblock_records.extend(result)
      t.update()
  chr_df = pd.DataFrame.from_records(cblock_records)
  return chr_df


def get_cblocks(db_name: str, output_dir: 'Path', log_dir: 'Path', gencode: bool,
gene_to_anchor_tx: dict[str, str]):
  chrs = [f'chr{i}' for i in list(range(1, 23)) + ['X']]
  dfs: dict[str, pd.DataFrame] = dict()

  # FIXME: should discard old runs if options are different
  for chr in chrs:
    df_file = output_dir/f'cblocks-{chr}.tsv'
    try:
      dfs[chr] = pd.read_csv(df_file, sep='\t')
    except:
      log_file = log_dir/f'{chr}.txt'
      dfs[chr] = process_chr(chr, db_name, log_file, gencode, gene_to_anchor_tx)
      dfs[chr].to_csv(df_file, sep='\t', index=False)

  cblock_df = pd.concat(dfs.values(), keys=dfs.keys(), names=['chr',
'row']).fillna(value='').reset_index().drop(columns='row')
  # cblock_df['other_accession'] = cblock_df['other'].str.split('|').str.get(1)
  return cblock_df


def get_pblocks(cblock_df: pd.DataFrame, output_dir: 'Path'):
```

```
    pblock_attrs = ['anchor', 'other', 'pblock_number']
    pblock_groups = cblock_df.groupby(pblock_attrs)
    pblocks = pblock_groups[['pblock_category', 'pblock_anchor_start',
'pblock_anchor_stop', 'pblock_other_start', 'pblock_other_stop']].first()
    pblocks['pblock_category'] = pblocks['pblock_category'].astype('category')
    pblocks['aa_loss'] = pblocks['pblock_anchor_stop'] - pblocks['pblock_anchor_start']
    pblocks['aa_gain'] = pblocks['pblock_other_stop'] - pblocks['pblock_other_start']
    pblocks['length_change'] = pblocks['aa_gain'] - pblocks['aa_loss']
    pblocks[['anchor_length', 'other_length']] = pblock_groups[['anchor_length',
'other_length']].first()
    pblocks['anchor_relative_length_change'] = pblocks['length_change'] /
pblocks['anchor_length']

    for col in ('up_start', 'down_start', 'up_stop', 'down_stop'):
        indices = pblock_groups['affects_' + col].idxmax()
        pblocks[col + '_cblock_category'] =
cblock_df['cblock_category'][indices].where(pblock_groups['affects_' + col].any().array,
other='-').array
        pblocks[col + '_cblock_events'] =
cblock_df['events'][indices].where(pblock_groups['affects_' + col].any().array,
other='').array
        pblocks[col + '_events'] = pblock_groups[col + '_events'].max()

    for col in ('up_start_events', 'down_start_events', 'up_stop_events', 'down_stop_events'):
        pblocks[col] = pblock_groups[col].max()

    nterm_cat = pd.CategoricalDtype((m.name for m in NTerminalChange), ordered=True)
    cterm_cat = pd.CategoricalDtype((m.name for m in CTerminalChange), ordered=True)

    def classify_nterm(upcat, downcat):
        if downcat in {'UNTRANSLATED', 'TRANSLATED'}:
            return NTerminalChange.ALTERNATIVE_ORF.name
        if upcat in {'DELETION', 'INSERTION'}:
            return NTerminalChange.MUTUALLY_EXCLUSIVE.name if downcat in {'DELETION',
'INSERTION'} else NTerminalChange.DOWNSTREAM_SHARED.name
        elif upcat in {'UNTRANSLATED', 'TRANSLATED'}:
            return NTerminalChange.UPSTREAM_SHARED.name if downcat in {'DELETION',
'INSERTION'} else NTerminalChange.MUTUALLY_SHARED.name
        elif upcat == '-':
            return None
        else:
            return NTerminalChange.UNKNOWN.name

    def classify_cterm(upcat, downcat):
```

```
        if upcat in {'DELETION', 'INSERTION'}:
            return CTerminalChange.SPLICING.name
        elif upcat in {'FRAME_AHEAD', 'FRAME_BEHIND'}:
            return CTerminalChange.FRAMESHIFT.name
        elif upcat in {'UNTRANSLATED', 'TRANSLATED'}:
            return CTerminalChange.ALTERNATIVE_ORF.name
        elif downcat == '-':
            return None
        else:
            return CTerminalChange.UNKNOWN.name

    pblocks['nterm'] = list(starmap(classify_nterm, zip(pblocks['up_start_cblock_category'],
pblocks['down_start_cblock_category'])))
    pblocks['cterm'] = list(starmap(classify_cterm, zip(pblocks['up_stop_cblock_category'],
pblocks['down_stop_cblock_category'])))
    pblocks['nterm'] = pblocks['nterm'].astype(nterm_cat)
    pblocks['cterm'] = pblocks['cterm'].astype(cterm_cat)
    pblocks['internal'] = pblocks['nterm'].isna() & pblocks['cterm'].isna()

    pblocks['cblocks'] = pblock_groups['cblock_category'].apply(tuple)
    pblocks['tblocks'] = pblock_groups['tblock_category'].unique().apply(lambda x:
tuple(filter(None, x)))
    pblocks['tblock_events'] = pblock_groups['events'].unique().apply(lambda x:
tuple(filter(None, x)))
    pblocks['events'] = pblocks['tblock_events'].apply(lambda x:
frozenset(chain.from_iterable(x)))

    pblocks['compound_splicing'] = pblock_groups['compound_splicing'].agg(any)
    pblocks['frameshift'] = pblock_groups['cblock_category'].apply(lambda cblocks:
any(cblock in {'FRAME_AHEAD', 'FRAME_BEHIND'} for cblock in cblocks))
    pblocks['split_codons'] = pblock_groups['cblock_category'].apply(lambda cblocks:
any(cblock in {'EDGE', 'COMPLEX'} for cblock in cblocks))

    for col in ('anchor_seq', 'other_seq'):
        pblocks[col] = pblock_groups['cblock_' + col].agg(''.join)

    pblocks.to_csv(output_dir/'pblocks.tsv', sep='\t')
    return pblocks

# %%
load_gencode_database.py
#%%
from biosurfer.core.database import Database
from biosurfer.core.helpers import get_ids_from_gencode_fasta, skip_par_y
```

```
import os

# #%%
# def check_database(gencode_gtf, gencode_tx, gencode_tl, gencode_doms,
pfam_dom_info, prosite_patterns, db_name):
#    """Sanity check for SQLite3 database whether it already exists.
#    Args:
#       gencode_gtf: Gene annotation file (GTF)
#       gencode_tx: Transcript reference sequence file (FASTA)
#       gencode_tl: Translation reference sequence file (FASTA)
#       gencode_doms: grch38 protein feature file (TSV)
#       pfam_dom_info: Protein Family mapping file (TSV)
#       prosite_patterns: PROSITE pattern data file
#       db_name: User input database name

#    Returns:
#     Nothing
#    """
#    path = os.getcwd()
#    db_path = os.path.join(path, "databases")
#    db_path_list = os.listdir(db_path)

#    if (db_name + '.sqlite3') in db_path_list:
#       print('\n Database already exists. Loading ' + db_name + ' ... \n')
#       load_gencode(db_name)
#    else:
#       print('\n Creating a new database ' + db_name + ' ...\n')
#       create_gencode(gencode_gtf, gencode_tx, gencode_tl, gencode_doms,
pfam_dom_info, prosite_patterns, db_name)

#%%
def create_gencode(gencode_gtf, gencode_tx, gencode_tl, gencode_doms,
pfam_dom_info, prosite_patterns, db_name):
   """ Creating new SQLite3 gencode database
   Args:
      gencode_gtf: Gene annotation file (GTF)
      gencode_tx: Transcript reference sequence file (FASTA)
      gencode_tl: Translation reference sequence file (FASTA)
      gencode_doms: grch38 protein feature file (TSV)
      pfam_dom_info: Protein Family mapping file (TSV)
      prosite_patterns: PROSITE pattern data file
      db_name: User input database name

   Returns:
```

```
    Nothing
    """
    db = Database(db_name)
    db.recreate_tables()
    db.load_gencode_gtf(os.path.abspath(gencode_gtf), overwrite=True)
    db.load_transcript_fasta(os.path.abspath(gencode_tx), get_ids_from_gencode_fasta,
skip_par_y)
    db.load_translation_fasta(os.path.abspath(gencode_tl), get_ids_from_gencode_fasta,
skip_par_y, overwrite=True)
    db.load_domains(os.path.abspath(pfam_dom_info))
    db.load_patterns(os.path.abspath(prosite_patterns))
    db.load_feature_mappings(os.path.abspath(gencode_doms), overwrite=False)

illustrate_figs.py
# %%
from pathlib import Path
import colorsys
import matplotlib as mpl
import matplotlib.colors as mc
import matplotlib.font_manager as fm
import pandas as pd
import seaborn as sns
from scipy.stats import chi2_contingency
from itertools import combinations
from seaborn import color_palette
from re import M
import scipy.stats as stats
import matplotlib.pyplot as plt
import numpy as np
import csv
from matplotlib.patches import Patch

def run_illustrate_analysis(pblock_table:Path, output: Path):
    """ Main plot function to invoke plotting for different pipelines/scripts.
    Args:
      pblock_table (Path): Path to the pblocks.tsv file.
      output (Path): Directory to save the illustrations.
    Returns:
      Nothing
    """

    ######################################################
    ## Setting configurations for illustrating figures ##
    ######################################################
```

```
  font = {
    'family': 'sans-serif',
    'sans-serif': ['Arial'],
    'weight': 'normal',
    'size': 16
  }
  mpl.rc('font', **font)

  # from https://stackoverflow.com/a/49601444
  def adjust_lightness(color, amount=0.5):
    try:
      c = mc.cnames[color]
    except:
      c = color
    c = colorsys.rgb_to_hls(*mc.to_rgb(c))
    cnew = colorsys.hls_to_rgb(c[0], max(0, min(1, amount * c[1])), c[2])
    return mc.to_hex(cnew)

  PBLOCK_COLORS = {
    'DELETION': '#f800c0',
    'INSERTION': '#00c0f8',
    'SUBSTITUTION': '#f8c000',
  }

  PBLOCK_COLORS['SUBSTITUTION (reference)'] =
adjust_lightness(PBLOCK_COLORS['SUBSTITUTION'], 1)
  PBLOCK_COLORS['SUBSTITUTION (alternative)'] =
adjust_lightness(PBLOCK_COLORS['SUBSTITUTION'], 1)

  SECTION_COLORS = {
    'N-terminal': color_palette('pastel')[2],
    'Internal': color_palette('pastel')[7],
    'C-terminal': color_palette('pastel')[3],
    'Full-length': 'none',
  }

  NTERM_CLASSES = {
    'MUTUALLY_EXCLUSIVE': 'Mutually exclusive starts (MSX)',
    'DOWNSTREAM_SHARED': 'Shared downstream start (SDS)',
    'UPSTREAM_SHARED': 'Shared upstream start (SUS)',
    'MUTUALLY_SHARED': 'Mutually shared starts (MSS)'
  }
  NTERM_COLORS = dict(zip(
```

```
    NTERM_CLASSES.values(),
    color_palette('viridis_r', n_colors=len(NTERM_CLASSES)+1)[:-1]
))

SPLICE_EVENT_COLORS = {
    'Intron': '#EBA85F',
    'Single exon': '#649FD2',
    'Alt. donor': '#86BB6F',
    'Alt. acceptor': '#A26FBB',
    'Compound': '#888888',
    'Frameshift': '#F7D76E',
}

CTERM_CLASSES = {
    'SPLICING': 'Splice-driven',
    'FRAMESHIFT': 'Frameshift-driven',
}
cterm_splice_palette = color_palette('RdPu_r', n_colors=6)
cterm_frameshift_palette = color_palette('YlOrRd_r', n_colors=5)
CTERM_PALETTE = [cterm_splice_palette[0], cterm_frameshift_palette[0]]

pblocks = pd.read_csv(pblock_table, sep='\t')

########################################
## Genome-wide summary illustrations ##
########################################
gw_output = output / 'gw_summary_plots'
gw_output.mkdir(exist_ok=True)
# %% Fig2 panel A: Number of altered isoforms per gene vs number of genes
fig = plt.figure(figsize=(4, 2.4))
bins = list(range(1, 11)) + [100]
ax = sns.histplot(
    x=pd.cut(
        pblocks.groupby('anchor')['other'].nunique(),
        bins=bins,
        right=False,
        labels=[str(x) for x in bins[:-2]] + [f'{bins[-2]}+'],
    ),
    shrink=0.75,
    color='#888888',
    edgecolor='k',
    alpha=1,
)
ax.set_xlabel('Number of alternative isoforms\nper gene')
```

```
  ax.set_ylabel('Number of genes')
  ax.set_ylim(0, 5000)
  ### output
  fig.savefig(gw_output / 'alternative-isoforms-per-gene.png', dpi=500, facecolor=None,
bbox_inches='tight')
  # Output source data
  pblocks.groupby('anchor')['other'].nunique().to_frame(name='count').to_csv(gw_output /
'alternative-isoforms-per-gene-table.tsv', sep='\t')
  # %% Fig2 panel B: Number of observed pblocks per alternative protein isoforms
  fig = plt.figure(figsize=(4, 2.4))
  ax = sns.histplot(
    x=pd.cut(
      pblocks.groupby(['anchor', 'other']).size(),
      bins=[1, 2, 3, 4, 5, 14],
      right=False,
      labels=['1', '2', '3', '4', '5+']
    ),
    shrink=0.75,
    color='#888888',
    edgecolor='k',
    alpha=1,
  )
  ax.set_xlabel('Number of altered regions\nper isoform')
  ax.set_ylabel('Number of alternative\nprotein isoforms')

  ### output
  fig.savefig(gw_output / 'altered-regions-per-isoform.png', dpi=500, facecolor=None,
bbox_inches='tight')
  # Output source data
  pblocks.groupby(['anchor',
'other']).size().to_frame(name='num_alt_regions').to_csv(gw_output / 'altered-regions-per-
isoform-table.tsv', sep='\t')
  # %% Fig2 panel C: Distribution of lengths of the insertion, deletion and substitution
affected regions for proteins
  aa_loss = pblocks[pblocks['pblock_category'].isin({'DELETION',
'SUBSTITUTION'})].reset_index()[['anchor', 'other', 'pblock_category', 'aa_loss']]
  aa_loss['pblock_category'].replace('SUBSTITUTION', 'SUBSTITUTION (reference)',
inplace=True)
  aa_loss.rename(columns={'aa_loss': 'length'}, inplace=True)
  aa_gain = pblocks[pblocks['pblock_category'].isin({'INSERTION',
'SUBSTITUTION'})].reset_index()[['anchor', 'other', 'pblock_category', 'aa_gain']]
  aa_gain['pblock_category'].replace('SUBSTITUTION', 'SUBSTITUTION (alternative)',
inplace=True)
  aa_gain.rename(columns={'aa_gain': 'length'}, inplace=True)
```

```
    affected_lengths = pd.concat([aa_loss, aa_gain])

    binwidth = 50
    xmax = 600
    xtick = 200

    fig = plt.figure(figsize=(5, 2))
    data = affected_lengths[affected_lengths['pblock_category'] != 'SUBSTITUTION
(alternative)']
    ax = sns.histplot(
        data=data,
        x='length',
        binwidth=binwidth,
        binrange=(0, xmax),
        stat='count',
        color='#808080',
        alpha=1,
    )
    ax.set_xlabel('Length of altered region (amino acids)')
    ax.set_ylabel('Number of\naltered regions')
    ax.ticklabel_format(axis='y', style='sci', scilimits=(-1, 1))
    ax.vlines(data['length'].median(), *ax.get_ylim(), color='#b0b0b0', linestyle='-',
linewidth=1)

    ### output
    fig.savefig(gw_output / 'altered-region-affected-lengths.png', dpi=500, facecolor=None,
bbox_inches='tight')
    # Output source data
    affected_lengths[affected_lengths['pblock_category'] != 'SUBSTITUTION
(alternative)'].to_csv(gw_output / 'altered-region-affected-lengths-table.tsv', sep='\t')
    # %% Fig2 panel D: Distribution of the length of altered protein regions across the
annotated proteome
    facets = sns.displot(
        data=affected_lengths,
        x='length',
        binwidth=binwidth,
        binrange=(0, xmax),
        stat='count',
        row='pblock_category',
        hue='pblock_category',
        palette=PBLOCK_COLORS,
        row_order=('DELETION', 'INSERTION', 'SUBSTITUTION (reference)', 'SUBSTITUTION
(alternative)'),
        legend=False,
```

```
    alpha=1,
    height=2,
    aspect=2.5
  )
  facets.set_xlabels('Length of altered region (amino acids)')
  facets.set_ylabels('Number of\naltered regions')
  for category, ax in facets.axes_dict.items():
    ax.set_title(category.capitalize())
    ax.set_xticks(range(0, xmax + 1, xtick))
    ax.ticklabel_format(axis='y', style='sci', scilimits=(-1, 1))
    ax.vlines(affected_lengths[affected_lengths['pblock_category'] ==
category]['length'].median(), *ax.get_ylim(), color='#808080', linestyle='-', linewidth=1)

  ### output
  facets.fig.savefig(gw_output / 'altered-region-affected-lengths-categories.png', dpi=500,
facecolor=None, bbox_inches='tight')
  # Output source data
  affected_lengths.to_csv(gw_output / 'altered-region-affected-lengths-categories-
table.tsv', sep='\t')
  # %% Fig2 panel I =: Substitution scatter plot
  plt.figure(figsize=(4.8, 3.6))
  ax = sns.histplot(
    data=pblocks[pblocks['pblock_category'] == 'SUBSTITUTION'],
    x='aa_gain',
    y='aa_loss',
    binwidth=binwidth / 2,
    stat='count',
    color=PBLOCK_COLORS['SUBSTITUTION'],
    legend=False,
    cbar=True,
    cbar_kws={
      'label': 'Number of regions',
    },
    alpha=1,
  )
  ax.set_xlim(0, xmax)
  ax.set_ylim(0, xmax)
  ax.set_xticks(range(0, xmax + 1, xtick))
  ax.set_yticks(range(0, xmax + 1, xtick))
  ax.set_xlabel('Length of substitution region \nin alternative isoform (AA)')
  ax.set_ylabel('Length of substitution region \nin reference isoform (AA)')
  ### output
  plt.savefig(gw_output / 'substitution-reference-alternative-lengths.png', dpi=500,
facecolor=None, bbox_inches='tight')
```

```
  # Output source data
  pblocks.query("pblock_category == 'SUBSTITUTION'")[['anchor', 'other', 'pblock_category',
'aa_gain', 'aa_loss']].to_csv(gw_output / 'substitution-reference-alternative-lengths-
table.tsv', sep='\t')
  # %% Fig2 panel D: Pie chart
  category_counts = pblocks['pblock_category'].value_counts()
  total_pblocks = category_counts.sum()
  fig, ax = plt.subplots()
  wedges, texts, autotexts = plt.pie(
    category_counts,
    colors=category_counts.index.map(PBLOCK_COLORS),
    wedgeprops={'width': 0.4},
    startangle=180,
    counterclock=False,
    autopct=lambda x: f'{np.round(total_pblocks * x / 100):.0f}\n({x:.0f}%)',
    pctdistance=1.3,
  )
  for i, wedge in enumerate(wedges):
    wedge.set_edgecolor('k')
  ### output
  fig.savefig(gw_output / 'altered-region-category-donut.png', dpi=500, facecolor=None,
bbox_inches='tight')
  # Output source data
  pblocks['pblock_category'].value_counts().to_csv(gw_output / 'altered-region-category-
donut-table.tsv', sep='\t')
  # %%
  def get_section(nterm, cterm):
    if nterm and cterm:
      return 'Full-length'
    elif nterm:
      return 'N-terminal'
    elif cterm:
      return 'C-terminal'
    else:
      return 'Internal'

  pblocks['protein_section'] = list(map(get_section, ~pblocks['nterm'].isna(),
~pblocks['cterm'].isna()))
  pblock_sections = pblocks['protein_section'].value_counts()

  fig, ax = plt.subplots(figsize=(6, 1))
  left = 0
  for section, color in SECTION_COLORS.items():
    val = pblock_sections[section]
```

```
      label = f'{val:g}\n({100 * val / pblock_sections.sum():0.1f}%)'
      if section == 'Full-length':
        left += 5000
        label_type = 'edge'
        padding = 5
      else:
        label_type = 'center'
        padding = 0
      bar = plt.barh(
        [0],
        val,
        left=left,
        color=color,
        edgecolor='k',
        label=section,
      )
      plt.bar_label(bar, labels=[label], label_type=label_type, padding=padding)
      left = left + pblock_sections[section]
  ax.legend(loc='upper left', bbox_to_anchor=(0, 0, 1, -0.1), ncols=2, frameon=False)
  plt.axis('off')
  ### output
  fig.savefig(gw_output / 'protein-section-counts.png', dpi=500, facecolor=None,
bbox_inches='tight')
  # Output source data
  with open(gw_output / 'protein-section-counts-table.tsv', 'w', newline='') as file:
    writer = csv.DictWriter(file, fieldnames=SECTION_COLORS.keys(), delimiter='\t')
    writer.writeheader()
    writer.writerow(SECTION_COLORS)
  # %%
  ###################################
  ## N-term summary illustrations ##
  ###################################
  nterm_output = output / 'nterm_summary_plots'
  nterm_output.mkdir(exist_ok=True)
  nterm_pblocks = pblocks[~pblocks['nterm'].isna() & (pblocks['nterm'] !=
'ALTERNATIVE_ORF') & (pblocks['cterm'].isna())].copy()
  nterm_pblocks['nterm'].replace(NTERM_CLASSES, inplace=True)
  nterm_pblocks['altTSS'] = nterm_pblocks['events'].apply(lambda x:
eval(x).intersection('BbPp')).astype(bool)
  # %% Fig3 panel A (both Alt TSS and 5' UTR AS)
  tss_fig = plt.figure(figsize=(5, 4))
  ax = sns.countplot(
    data=nterm_pblocks,
    y='nterm',
```

```
    order=NTERM_COLORS.keys(),
    palette=NTERM_COLORS,
    edgecolor='k',
    saturation=1,
  )
  sns.countplot(
    ax=ax,
    data=nterm_pblocks[nterm_pblocks['altTSS']],
    y='nterm',
    order=NTERM_COLORS.keys(),
    palette=NTERM_COLORS,
    edgecolor='k',
    fill=False,
    hatch='//',
  )
  ax.legend(
    loc=(0, 1),
    frameon=False,
    handles=[Patch(facecolor='w', edgecolor='k', hatch='///'), Patch(facecolor='w',
edgecolor='k')],
    labels=['Alternative transcription start site', '5\' UTR alternative splicing'],
  )
  ax.set_xlabel('Number of alternative isoforms')
  ax.set_ylabel(None)
  plt.savefig(nterm_output / 'nterm-counts-all_mechanism.png', dpi=500, facecolor=None,
bbox_inches='tight')
  # Output source data
  nterm_pblocks.query("nterm in ['Mutually exclusive starts (MSX)', 'Shared downstream
start (SDS)']")[['anchor', 'other', 'nterm', 'altTSS']].to_csv(nterm_output / 'nterm-counts-
all_mechanism.tsv', sep='\t')
  # %% Fig3 panel C: MXS vs SDS scatterplot
  font = {
    'family': 'sans-serif',
    'sans-serif': ['Arial'],
    'weight': 'normal',
    'size': 10
  }
  mpl.rc('font', **font)

  # Filter the dataframe for 'Mutually exclusive starts (MXS)' and 'Shared downstream start
(SDS)'
  msx_data = nterm_pblocks[nterm_pblocks['nterm'] == 'Mutually exclusive starts (MSX)']
  sds_data = nterm_pblocks[nterm_pblocks['nterm'] == 'Shared downstream start (SDS)']
  fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(5.5, 5.5))
```

```
    msx_color = (0.565498, 0.84243, 0.262877)
    sds_color = (0.20803, 0.718701, 0.472873)

    sns.scatterplot(data=msx_data, x='aa_loss', y='aa_gain', marker='.', ax=axes[0], alpha=0.2,
                    color=msx_color)
    axes[0].set_title('Mutually exclusive starts (MSX)', fontsize=11)
    axes[0].set_xlabel('Reference \n(amino acids)', fontsize=10)
    axes[0].set_ylabel('Alternative \n(amino acids)', fontsize=10)
    axes[0].set_xlim(0, 2000)
    axes[0].set_ylim(0, 2000)
    axes[0].set_aspect('equal')
    axes[0].grid(True, linestyle='--', linewidth=0.5)

    sns.scatterplot(data=sds_data, x='aa_loss', y='aa_gain', marker='.', ax=axes[1], alpha=0.2,
color=sds_color)
    axes[1].set_title('Shared downstream start (SDS)', fontsize=11)
    axes[1].set_xlabel('Reference \n(amino acids)', fontsize=10)
    axes[1].set_ylabel('Alternative \n(amino acids)', fontsize=10)
    axes[1].set_xlim(0, 2000)
    axes[1].set_ylim(0, 2000)
    axes[1].set_aspect('equal')
    axes[1].grid(True, linestyle='--', linewidth=0.5)
    plt.tight_layout()
    # Save plot
    plt.savefig(nterm_output / 'nterm-rel-length-change_scatterplot.png', dpi=500,
facecolor=None, bbox_inches='tight')
    # Output source data
    nterm_pblocks.query("nterm in ['Mutually exclusive starts (MSX)', 'Shared downstream
start (SDS)']")[['anchor', 'other', 'aa_loss', 'aa_gain']].to_csv(nterm_output /
'nterm_mechanism_affected_len.tsv', sep='\t')
    # %%
    ##############################################
    ## Internal region  summary illustrations ##
    ##############################################
    internal_output = output / 'internal_summary_plots'
    internal_output.mkdir(exist_ok=True)
    internal_pblocks = (
      pblocks[pblocks['internal']].
      drop(columns=[col for col in pblocks.columns if 'start' in col or 'stop' in col]).
      copy()
    )
    # convert string repr back to Python object
    internal_pblocks['tblock_events'] = internal_pblocks['tblock_events'].map(eval)
    internal_pblocks['events'] = internal_pblocks['events'].map(eval)
```

```
  internal_subcats = pd.DataFrame(
    {
      'Frameshift': internal_pblocks['frameshift'],
      'Intron': internal_pblocks['tblock_events'].isin({('I',), ('i',)}),
      'Alt. donor': internal_pblocks['tblock_events'].isin({('D',), ('d',)}),
      'Alt. acceptor': internal_pblocks['tblock_events'].isin({('A',), ('a',)}),
      'Single exon': internal_pblocks['tblock_events'].isin({('E',), ('e',)}),
      'Compound': [True for _ in internal_pblocks.index]
    }
  )
  subcat_order = ('Single exon', 'Alt. acceptor', 'Alt. donor', 'Intron', 'Compound',
'Frameshift')
  internal_pblocks['splice_event'] =
internal_subcats.idxmax(axis=1).astype(pd.CategoricalDtype(subcat_order,
ordered=True))
  # %% Fig4 panel A: Internal splicing events frequencies
  internal_pblocks_fig = plt.figure(figsize=(4.6, 3.8))
  ax = sns.countplot(
    data=internal_pblocks.sort_values('pblock_category', ascending=True),
    y='splice_event',
    dodge=True,
    hue='pblock_category',
    palette=PBLOCK_COLORS,
    saturation=1,
    edgecolor='k',
  )
  plt.legend(loc='center right', labels=['Deletions', 'Insertions', 'Substitutions'])
  ax.set_xlabel('Number of altered internal regions')
  ax.set_ylabel(None)
  internal_pblocks_fig.savefig(internal_output / 'internal-events.png', dpi=500,
facecolor=None, bbox_inches='tight')
  # Output source data
  internal_pblocks[['splice_event', 'pblock_category']].to_csv(internal_output / 'internal-
events-table.tsv', sep='\t')
  # %% Fig4 panel C: Proportion of each internal protein region that are ragged codons
  internal_pblocks_ragged_fig = plt.figure(figsize=(4.6, 3.8))
  ax = sns.countplot(
    data=internal_pblocks.sort_values('pblock_category', ascending=True),
    y='splice_event',
    palette=SPLICE_EVENT_COLORS,
    saturation=1,
    edgecolor='k',
  )
  sns.countplot(
```

```
    ax=ax,
    data=internal_pblocks[internal_pblocks['split_codons']].sort_values('pblock_category',
ascending=True),
    y='splice_event',
    fill=False,
    edgecolor='k',
    hatch='///',
  )
  plt.gca()
  ax.set_xlabel('Number of altered internal regions')
  ax.set_ylabel(None)
  internal_pblocks_ragged_fig.savefig(internal_output / 'internal-events-ragged.png',
dpi=500, facecolor=None, bbox_inches='tight')
  # Output source data
  internal_pblocks[['splice_event', 'split_codons']].to_csv(internal_output / 'internal-
events-ragged-table.tsv', sep='\t')

  alpha = 0.01
  ragged_contingency = pd.crosstab(internal_pblocks['split_codons'],
internal_pblocks['splice_event'])
  chi2, p_all, dof, expected = chi2_contingency(ragged_contingency)

  ps = dict()
  for event1, event2 in combinations(internal_subcats.columns, 2):
    sub_contingency = ragged_contingency[[event1, event2]]
    _, ps[event1, event2], _, _ = chi2_contingency(sub_contingency)

  ps_sig = {k: p for k, p in ps.items() if p < alpha/len(ps)}
  ps_insig = {k: p for k, p in ps.items() if k not in ps_sig}

  # %%
  nagnag_pblocks = internal_pblocks[(internal_pblocks['splice_event'] == 'Alt. acceptor') &
(internal_pblocks['length_change'].abs() == 1)]

  # %% Fig4 panel B: Frequency of compound splicing events
  internal_compound_pblocks = internal_pblocks[internal_pblocks['splice_event'] ==
'Compound'].copy()

  internal_compound_subcats = pd.DataFrame(
    {
      'Multi-exon skipping': internal_compound_pblocks['events'] == frozenset('e'),
      'Exon skipping + \nalt. donor/acceptor': internal_compound_pblocks['events'].isin({
        frozenset(sorted('de')),
        frozenset(sorted('De')),
```

```
            frozenset(sorted('ea')),
            frozenset(sorted('eA')),
            frozenset(sorted('dea')),
            frozenset(sorted('Dea')),
            frozenset(sorted('deA')),
            frozenset(sorted('DeA')),
        }),
        'Mutually exclusive exons': internal_compound_pblocks['tblock_events'].isin({('E', 'e'),
('e', 'E')}),
        'Multi-exon inclusion': internal_compound_pblocks['events'] == frozenset('E'),
        'Alt. donor + alt. acceptor': internal_compound_pblocks['events'].isin({
            frozenset(sorted('ad')),
            frozenset(sorted('Ad')),
            frozenset(sorted('aD')),
            frozenset(sorted('AD')),
        }),
        'Exon inclusion + \nalt. donor/acceptor': internal_compound_pblocks['events'].isin({
            frozenset(sorted('dE')),
            frozenset(sorted('DE')),
            frozenset(sorted('Ea')),
            frozenset(sorted('EA')),
            frozenset(sorted('dEa')),
            frozenset(sorted('DEa')),
            frozenset(sorted('dEA')),
            frozenset(sorted('DEA')),
        }),
        'Other': [True for _ in internal_compound_pblocks.index]
      }
    )
    internal_compound_pblocks['compound_subcat'] =
internal_compound_subcats.idxmax(axis=1).astype(pd.CategoricalDtype(internal_compo
und_subcats.columns, ordered=True))

    internal_pblocks_compound_fig = plt.figure(figsize=(3, 3))
    ax = sns.countplot(
        data=internal_compound_pblocks,
        y='compound_subcat',
        palette='Greys_r',
        saturation=1,
        edgecolor='k',
    )
    ax.set_xlabel('Number of altered\ninternal regions'),
    ax.set_ylabel(None)
```

```
  internal_pblocks_compound_fig.savefig(internal_output / 'internal-compound-events.png', dpi=500, facecolor=None, bbox_inches='tight')
  # Output source data
  internal_compound_pblocks[['anchor', 'other', 'compound_subcat']].to_csv(internal_output / 'internal-compound-events-table.tsv', sep='\t')
  #%%
  ##################################
  ## C-term summary illustrations ##
  ##################################
  cterm_output = output / 'cterm_summary_plots'
  cterm_output.mkdir(exist_ok=True)

  cterm_pblocks = pblocks[~pblocks['cterm'].isna() & (pblocks['nterm'].isna()) & (pblocks['cterm'] != "ALTERNATIVE_ORF") & (pblocks['cterm'] != "UNKNOWN")].copy()
  cterm_pblocks['cterm'] = cterm_pblocks['cterm'].map(CTERM_CLASSES).astype('category')
  # Changed string to set for intersection
  cterm_pblocks['APA'] = cterm_pblocks['events'].apply(lambda x: set(x).intersection('BbPp')).astype(bool)

  #%% Fig5 panel A: Frequency of splice-driven and frameshift-driven C-terminal events
  cterm_fig = plt.figure(figsize=(3.8, 2))
  ax = sns.countplot(
    data = cterm_pblocks,
    y = 'cterm',
    order = CTERM_CLASSES.values(),
    palette = CTERM_PALETTE,
    saturation = 1,
    linewidth = 1,
    edgecolor = 'k',
  )
  ax.set_xlabel('Number of alternative isoforms')
  ax.set_ylabel('')
  plt.savefig(cterm_output/'cterm-class-counts.png', dpi=500, facecolor=None, bbox_inches='tight')
  #Output source data
  cterm_pblocks.query("cterm in ['Splice-driven', 'Frameshift-driven']")[['anchor','other','cterm']].to_csv(cterm_output/'cterm-class-counts-table.tsv', sep='\t')
  # %% Fig5 panel B: Frequency of splice-driven patterns
  cterm_pblock_events = cterm_pblocks['up_stop_events'].combine(cterm_pblocks['down_stop_events'], lambda x, y: (x, y))
```

```
  single_ATE = (cterm_pblocks['cterm'] == 'Splice-driven') &
cterm_pblocks['tblock_events'].isin({('B', 'b'), ('b', 'B')})
  cterm_splice_subcats = pd.DataFrame(
    {
      'Exon extension introduces termination': cterm_pblocks['up_stop_events'].isin({'P', 'I',
'D'}),
      'Alternative terminal exon(s)': cterm_pblock_events.isin({('B', 'b'), ('b', 'B')}),
      'Poison exon inclusion': cterm_pblocks['up_stop_events'] == 'E',
      'Other': [True for _ in cterm_pblocks.index]
      #'Alternative last exon in UTR': cterm_pblocks['cblocks'].apply(lambda x:
'TRANSLATED' in x and 'DELETION' in x and 'UNTRANSLATED' not in x)
    }
  )
  cterm_pblocks['splice_subcat'] =
cterm_splice_subcats.idxmax(axis=1).astype(pd.CategoricalDtype(cterm_splice_subcats.
columns, ordered=True))

  cterm_splice_palette_dict = dict(zip(
    cterm_splice_subcats.columns,
    cterm_splice_palette[0:1] + cterm_splice_palette[1:2] + cterm_splice_palette[2:3] +
['#bbbbbb']
  ))
  splice_subcat_order = tuple(cterm_splice_subcats.keys())

  cterm_pblock_events =
cterm_pblocks['up_stop_events'].combine(cterm_pblocks['down_stop_events'], lambda x,
y: (x, y))
  single_ATE = (cterm_pblocks['cterm'] == 'Splice-driven') &
cterm_pblocks['tblock_events'].isin({('B', 'b'), ('b', 'B')})

  cterm_splice_subcats = pd.DataFrame(
    {
      'Exon extension introduces \n termination (EXIT)':
cterm_pblocks['up_stop_events'].isin({'P', 'I', 'D'}),
      'Alternative terminal \n exon(s) (ATE)': cterm_pblock_events.isin({('B', 'b'), ('b', 'B')}),
      'Alternative last exon \n in UTR (ALE in UTR)': cterm_pblocks.apply(lambda row:
'TRANSLATED' in row['cblocks'] and 'DELETION' in row['cblocks'] and 'UNTRANSLATED' not
in row['cblocks'] if row['cterm'] == 'Splice-driven' and row['splice_subcat'] == 'Other' else
False, axis=1),
      'Poison exon inclusion': cterm_pblocks['up_stop_events'] == 'E',
      'Cut-out splice terminal \n exon (COSTE)': cterm_pblocks.apply(lambda row:
'DELETION' in row['cblocks'] and 'INSERTION' in row['cblocks'] and 'TRANSLATED' not in
row['cblocks'] and 'UNTRANSLATED' not in row['cblocks'] and 'FRAME' not in row['cblocks']
```

```
and 'p' in row['tblock_events'] and row['tblock_events'].count('B') == 1 if row['cterm'] ==
'Splice-driven' and row['splice_subcat'] == 'Other' else False, axis=1),
        'Other': [True for _ in cterm_pblocks.index]
      }
  )
  cterm_pblocks['splice_subcat'] =
cterm_splice_subcats.idxmax(axis=1).astype(pd.CategoricalDtype(cterm_splice_subcats.
columns, ordered=True))

  cterm_splice_palette_dict = dict(zip(
    cterm_splice_subcats.columns,
    cterm_splice_palette[0:1] + cterm_splice_palette[1:2] + cterm_splice_palette[2:3] +
cterm_splice_palette[3:4] + cterm_splice_palette[4:5] +  ['#bbbbbb']
  ))
  splice_subcat_order = tuple(cterm_splice_subcats.keys())

  cterm_splice_fig, axs = plt.subplots(1, 2, figsize=(9, 4))
  sns.countplot(
    ax = axs[0],
    data = cterm_pblocks[cterm_pblocks['cterm'] == 'Splice-driven'],
    y = 'splice_subcat',
    order = splice_subcat_order,
    palette = cterm_splice_palette_dict,
    saturation = 1,
    edgecolor = 'k',
    linewidth = 1,
  )
  axs[0].set_xlabel('Number of alternative isoforms')
  axs[0].set_ylabel(None)

  plt.savefig(cterm_output/'cterm-splicing-subcats.png', dpi=500, facecolor=None,
bbox_inches='tight')
  #Output source data
  cterm_pblocks.assign(anchor_relative_length_change =
cterm_pblocks['anchor_relative_length_change'].abs())[['anchor','other',
'splice_subcat','anchor_relative_length_change']].to_csv(cterm_output/'cterm-splicing-
subcats-table.tsv', sep='\t')
  cterm_pblocks.to_csv(cterm_output / 'cterm_pblocks.tsv', sep='\t', index=False)

  # %% Alternative Last Exon in 3' UTR case from Splice-driven 'Other' category.
  cterm_pblock_splice = cterm_pblocks[cterm_pblocks['cterm'] == 'Splice-driven']
  cterm_splice_other = cterm_pblock_splice[cterm_pblock_splice['splice_subcat'] ==
'Other']
```

```
    condition1 = cterm_splice_other['cblocks'].apply(lambda x: 'DELETION' in x and
'TRANSLATED' in x)
    condition2 = cterm_splice_other['cblocks'].apply(lambda x: 'UNTRANSLATED' not in x)
    cterm_aleutr = cterm_splice_other[condition1 & condition2].copy()
    cterm_aleutr.to_csv(cterm_output / 'cterm-splice-driven-ALEinUTR.tsv', sep='\t')

    # %% Cut-out splice terminal exon case from Splice-driven 'Other' category.
    condition3 = cterm_splice_other['cblocks'].apply(lambda x: 'DELETION' in x and
'INSERTION' in x)
    condition4 = cterm_splice_other['cblocks'].apply(lambda x: 'TRANSLATED' not in x and
'UNTRANSLATED' not in x and 'FRAME' not in x)
    condition5 = cterm_splice_other['tblock_events'].apply(lambda x: x.count('B') == 1 and 'p'
in x)
    cterm_other_new = cterm_splice_other[condition3 & condition4 & condition5].copy()
    cterm_other_new.to_csv(cterm_output / 'cterm-splice-driven-other-NEW.tsv', sep='\t')
    # %% Fig5 panel C & D: 2D scatter plot v2 splice-driven vs frameshift-driven
    font = {
        'family': 'sans-serif',
        'sans-serif': ['Arial'],
        'weight': 'normal',
        'size': 10
    }
    mpl.rc('font', **font)
    msx_data = cterm_pblocks[cterm_pblocks['cterm'] == 'Splice-driven']
    sds_data = cterm_pblocks[cterm_pblocks['cterm'] == 'Frameshift-driven']
    fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(6, 6))
    msx_color = (0.5048212226066897, 0.00392156862745098, 0.47021914648212226)
    sds_color = (0.7885121107266436, 0.03238754325259515, 0.13656286043829297)

    sns.scatterplot(data=msx_data, x='aa_loss', y='aa_gain', marker='o', ax=axes[0],
alpha=0.2,
                    color=msx_color)
    axes[0].set_title('Splice-driven', fontsize=13)
    axes[0].set_xlabel('Reference \n(amino acids)', fontsize=12)
    axes[0].set_ylabel('Alternative \n(amino acids)', fontsize=12)
    axes[0].set_xlim(0, 3000)
    axes[0].set_ylim(0, 3000)
    axes[0].set_aspect('equal')
    axes[0].grid(True, linestyle='--', linewidth=0.5)

    sns.scatterplot(data=sds_data, x='aa_loss', y='aa_gain', marker='o', ax=axes[1],
alpha=0.2, color=sds_color)
    axes[1].set_title('Frameshift-driven', fontsize=13)
    axes[1].set_xlabel('Reference \n(amino acids)', fontsize=12)
```

```
    axes[1].set_ylabel('Alternative \n(amino acids)', fontsize=12)
    axes[1].set_xlim(0, 3000)
    axes[1].set_ylim(0, 3000)
    axes[1].set_aspect('equal')
    axes[1].grid(True, linestyle='--', linewidth=0.5)

    plt.tight_layout()
    plt.savefig(cterm_output / 'cterm-rel-length-change_scatterplot.png', dpi=800,
facecolor=None, bbox_inches='tight')
    # Output source data
    cterm_pblocks.query("cterm in ['Splice-driven', 'Frameshift-driven']")[['anchor', 'other',
'aa_loss', 'aa_gain']].to_csv(cterm_output / 'cterm_mechanism_affected_len.tsv', sep='\t')


    # %% Supplementary Figure S5: 2D scatter plot v2 frameshift-driven subcats
    d1 = cterm_pblocks[cterm_pblocks['splice_subcat'] == 'Exon extension introduces \n
termination (EXIT)']
    d2 = cterm_pblocks[cterm_pblocks['splice_subcat'] == 'Alternative terminal \n exon(s)
(ATE)']
    d3 = cterm_pblocks[cterm_pblocks['splice_subcat'] == 'Alternative last exon \n in UTR
(ALE in UTR)']
    d4 = cterm_pblocks[cterm_pblocks['splice_subcat'] == 'Poison exon inclusion']
    d5 = cterm_pblocks[cterm_pblocks['splice_subcat'] == 'Cut-out splice terminal \n exon
(COSTE)']

    fig, axes = plt.subplots(nrows=5, ncols=1, figsize=(6, 15))
    colors = [(0.5048212226066897, 0.00392156862745098, 0.47021914648212226),
          (0.735840061514802, 0.061960784313725495, 0.5225682429834679),
          (0.9094502114571319, 0.2894886582083814, 0.6086120722798923),
          (0.9754555940023067, 0.5330257593233372, 0.6768935024990388),
          (0.9859592464436755, 0.7293041138023837, 0.7404229142637447)]

    sns.scatterplot(data=d1, x='aa_loss', y='aa_gain', marker='o', ax=axes[0], alpha=0.2,
              color=colors[0])
    axes[0].set_title('Exon extension introduces termination', fontsize=30, pad=20)
    axes[0].set_xlabel('Reference \n(amino acids)', fontsize=25)
    axes[0].set_ylabel('Alternative \n(amino acids)', fontsize=25)
    axes[0].set_xlim(0, 3000)
    axes[0].set_ylim(0, 3000)
    axes[0].grid(True, linestyle='--', linewidth=0.5)

    sns.scatterplot(data=d2, x='aa_loss', y='aa_gain', marker='o', ax=axes[1], alpha=0.2,
              color=colors[1])
    axes[1].set_title('Alternative terminal exon(s)', fontsize=30, pad=20)
```

```
    axes[1].set_xlabel('Reference \n(amino acids)', fontsize=25)
    axes[1].set_ylabel('Alternative \n(amino acids)', fontsize=25)
    axes[1].set_xlim(0, 3000)
    axes[1].set_ylim(0, 3000)
    axes[1].grid(True, linestyle='--', linewidth=0.5)

    sns.scatterplot(data=d3, x='aa_loss', y='aa_gain', marker='o', ax=axes[2], alpha=0.2,
                    color=colors[2])
    axes[2].set_title('Alternative last exon in UTR', fontsize=30, pad=20)
    axes[2].set_xlabel('Reference \n(amino acids)', fontsize=25)
    axes[2].set_ylabel('Alternative \n(amino acids)', fontsize=25)
    axes[2].set_xlim(0, 3000)
    axes[2].set_ylim(0, 3000)
    axes[2].grid(True, linestyle='--', linewidth=0.5)

    sns.scatterplot(data=d4, x='aa_loss', y='aa_gain', marker='o', ax=axes[3], alpha=0.2,
                    color=colors[3])
    axes[3].set_title('Poison exon inclusion', fontsize=30, pad=20)
    axes[3].set_xlabel('Reference \n(amino acids)', fontsize=25)
    axes[3].set_ylabel('Alternative \n(amino acids)', fontsize=25)
    axes[3].set_xlim(0, 3000)
    axes[3].set_ylim(0, 3000)
    axes[3].grid(True, linestyle='--', linewidth=0.5)

    sns.scatterplot(data=d5, x='aa_loss', y='aa_gain', marker='o', ax=axes[4], alpha=0.2,
                    color=colors[4])
    axes[4].set_title('Cut-out splice terminal exon', fontsize=30, pad=20)
    axes[4].set_xlabel('Reference \n(amino acids)', fontsize=25)
    axes[4].set_ylabel('Alternative \n(amino acids)', fontsize=25)
    axes[4].set_xlim(0, 3000)
    axes[4].set_ylim(0, 3000)
    axes[4].grid(True, linestyle='--', linewidth=0.5)

    plt.tight_layout()
    plt.savefig(cterm_output / 'cterm-rel-splice-driven-subcat-length-
change_scatterplot.png', dpi=500, facecolor=None, bbox_inches='tight')

if __name__ == "__main__":
    run_illustrate_analysis(pblock_table, output)
```

plot_biosurfer.py

```
# %%
from pathlib import Path
from more_itertools import partition
from biosurfer.core.alignments import ProteinAlignment
from biosurfer.core.constants import APPRIS
from biosurfer.core.database import Database
from biosurfer.core.helpers import (get_ids_from_gencode_fasta,
                    get_ids_from_lrp_fasta,
                    get_ids_from_pacbio_fasta, skip_gencode,
                    skip_par_y)
from biosurfer.core.models.biomolecules import Gene, Transcript
from biosurfer.plots.plotting import IsoformPlot

#%%
def run_plot(output: Path, gene: str, db_name: str, transcript_ids: tuple[str]):
  """ Main plot function to invoke plotting for different pipelines/scripts.
  Args:
   Nothing
  Returns:
   Nothing
  """
  if not output:
   output = Path('.')
  db = Database(db_name)
  with db.get_session() as s:
    print(f'Loading transcripts from database...')

    if gene:
      gene_obj = Gene.from_name(s, gene)
      if gene_obj is None:
        print(f'Gene "{gene}" not found in database')
        transcripts = dict()
        anchor = None
      else:
        transcripts = {tx.accession: tx for tx in gene_obj.transcripts}
        anchor = max(transcripts.values(), key=lambda tx: getattr(tx, 'appris',
APPRIS.NONE))
      others = [tx for tx in transcripts.values() if tx is not anchor]
    else:
      transcripts: dict[str, Transcript] = {tx.accession: tx for tx in
Transcript.from_accessions(s, transcript_ids).values()}
      not_found, found = partition(lambda tx_id: tx_id in transcripts, transcript_ids)
```

```
        for tx_id in not_found:
            print(f'Transcript ID "{tx_id}" not found in database')
        if transcript_ids:
            anchor = transcripts.get(transcript_ids[0], None)
        else:
            print('No isoforms provided')
            anchor = None
        others = [tx for tx in map(transcripts.get, found) if tx is not anchor]

    if anchor:
        print(f'Reference isoform: {anchor}')
        gene = anchor.gene.name

        alns: dict[Transcript, ProteinAlignment] = dict()
        for other in others:
            if anchor.protein is None or other.protein is None:
                alns[other] = None
            else:
                try:
                    alns[other] = ProteinAlignment.from_proteins(anchor.protein, other.protein)
                except ValueError:
                    print(f'Could not plot isoform {other}')

        filename = f'{db_name}-{gene}.png'
        plot = IsoformPlot([anchor] + list(alns.keys()))
        plot.draw_all_isoforms()
        plot.draw_frameshifts()
        for other, aln in alns.items():
            if aln:
                plot.draw_protein_alignment_blocks(aln.blocks, anchor.protein, other.protein)
        plot.draw_legend()
        filepath = str(output/filename)
        plot.savefig(filepath)
        print(f'Saved {filepath}')




if __name__ == "__main__":
    run_plot(output, gene, db_name, transcript_ids)
```

alignments.py

```
from abc import ABC, abstractmethod
from collections import deque
from functools import cached_property, lru_cache
from itertools import chain, groupby, tee
from operator import attrgetter, itemgetter
from typing import TYPE_CHECKING
import os
from attrs import define, evolve, field, frozen
from biosurfer.core.constants import ANCHOR_EXCLUSIVE, FRAMESHIFT, CD_DEL_INS,
OTHER_EXCLUSIVE, SEQ_DEL_INS, SPLIT_CODON, CodonAlignmentCategory as
CodonAlignCat
from biosurfer.core.constants import SequenceAlignmentCategory as SeqAlignCat
from biosurfer.core.helpers import Interval, IntervalTree
from biosurfer.core.models.biomolecules import Protein, Transcript
from biosurfer.core.models.features import ProjectedFeature, ProteinFeature
from biosurfer.core.splice_events import (BasicTranscriptEvent, TranscriptEvent,
                                          call_transcript_events, sort_events)
from more_itertools import first, last, one, only, partition, windowed

if TYPE_CHECKING:
    from biosurfer.core.constants import AlignmentCategory
    from biosurfer.core.splice_events import CompoundTranscriptEvent


CACHE_SIZE = 2**8

def check_block_ranges(instance, attribute, value: 'IntervalTree'):
    starts = sorted(i.begin for i in value)
    stops = sorted(i.end for i in value)
    if starts[0] < 0:
        raise ValueError(f'Block ranges cannot be negative')
    if starts[0] > 0:
        raise ValueError(f'Block ranges do not cover (0, {starts[0]})')
    for i, (start, stop) in enumerate(zip(starts[1:], stops[:-1])):
        if start != stop:
            raise ValueError(f'Gap or overlap between block ranges ({starts[i]}, {stop}) and ({start},
{stops[i]})')


@frozen(order=True)
class AlignmentBlock(ABC):
    anchor_range: range = field(factory=range, order=attrgetter('start', 'stop'))
```

```
  other_range: range = field(factory=range, order=attrgetter('start', 'stop'))
  category: 'AlignmentCategory' = field(default=None, order=False)

  def __attrs_post_init__(self):
    A, O = len(self.anchor_range), len(self.other_range)
    if A == O == 0:
      raise ValueError(f'Invalid ranges {self.anchor_range} and {self.other_range}')

  def __repr__(self):
    return
f'{self.category}({self.anchor_range.start}:{self.anchor_range.stop}|{self.other_range.start}:{
self.other_range.stop})'

  @property
  def delta_length(self):
    return len(self.other_range) - len(self.anchor_range)

  def project_coordinate(self, coord: int, *, from_anchor: bool = True):
    source, target = (self.anchor_range, self.other_range) if from_anchor else
(self.other_range, self.anchor_range)
    index = source.index(coord)
    try:
      return target[index]
    except IndexError:
      return None


@frozen(order=True, repr=False)
class TranscriptAlignmentBlock(AlignmentBlock):
  category: 'SeqAlignCat' = field(init=False)

  def __attrs_post_init__(self):
    A, O = len(self.anchor_range), len(self.other_range)
    if A == O > 0:
      object.__setattr__(self, 'category', SeqAlignCat.MATCH)
    elif O > A == 0:
      object.__setattr__(self, 'category', SeqAlignCat.INSERTION)
    elif A > O == 0:
      object.__setattr__(self, 'category', SeqAlignCat.DELETION)
    else:
      raise ValueError(f'Invalid ranges {self.anchor_range} and {self.other_range}')


@frozen(repr=False)
```

```
class CodonAlignmentBlock(AlignmentBlock):
  category: 'CodonAlignCat' = field(kw_only=True)


@frozen(order=True, repr=False)
class ProteinAlignmentBlock(AlignmentBlock):
  category: 'SeqAlignCat' = field(kw_only=True)
  ragged5: bool = field(default=False, order=False)
  ragged3: bool = field(default=False, order=False)

  def __attrs_post_init__(self):
    super().__attrs_post_init__()
    if self.ragged and self.category is SeqAlignCat.MATCH:
      raise ValueError(f'Match protein alignment blocks cannot be ragged')

  @property
  def ragged(self):
    return self.ragged5 or self.ragged3


class ProjectionMixin:
  def range_to_blocks(self, start: int, stop: int, *, from_anchor: bool = True):
    mapper: 'IntervalTree' = self.anchor_blocks if from_anchor else self.other_blocks
    blocks: list['AlignmentBlock'] = [interval.data for interval in
sorted(mapper.overlap(start, stop))]
    if not blocks:
      raise ValueError(f'Could not locate mapping in {self.anchor if from_anchor else
self.other} for range({start}, {stop})')
    first_block = blocks[0]
    last_block = blocks[-1]
    first_block_source = first_block.anchor_range if from_anchor else
first_block.other_range
    last_block_source = last_block.anchor_range if from_anchor else
last_block.other_range
    first_block_offset = start - first_block_source.start
    last_block_offset = stop - last_block_source.start
    if first_block is last_block:
      blocks[0] = evolve(
        first_block,
        anchor_range = first_block.anchor_range[first_block_offset:last_block_offset],
        other_range = first_block.other_range[first_block_offset:last_block_offset]
      )
    else:
      blocks[0] = evolve(
```

```
            first_block,
            anchor_range = first_block.anchor_range[first_block_offset:],
            other_range = first_block.other_range[first_block_offset:]
          )
          blocks[-1] = evolve(
            last_block,
            anchor_range = last_block.anchor_range[:last_block_offset],
            other_range = last_block.other_range[:last_block_offset]
          )
        return blocks

    def project_range(self, start: int, stop: int, *, from_anchor: bool = True) -> range:
        blocks = self.range_to_blocks(start, stop, from_anchor=from_anchor)
        if from_anchor:
            return range(blocks[0].other_range.start, blocks[-1].other_range.stop)
        else:
            return range(blocks[0].anchor_range.start, blocks[-1].anchor_range.stop)

    def project_coordinate(self, coord: int, *, from_anchor: bool = True):
        mapper: 'IntervalTree' = self.anchor_blocks if from_anchor else self.other_blocks
        block: 'AlignmentBlock' = one(mapper.at(coord)).data
        return block.project_coordinate(coord, from_anchor=from_anchor)


@define(eq=False)
class TranscriptAlignment(ProjectionMixin):
    anchor: 'Transcript'
    other: 'Transcript' = field()
    events: tuple['CompoundTranscriptEvent', ...] = field(converter=sort_events, repr=False)
    anchor_events: 'IntervalTree' = field(factory=IntervalTree, repr=False)
    anchor_blocks: 'IntervalTree' = field(factory=IntervalTree, repr=False,
validator=check_block_ranges)
    other_events: 'IntervalTree' = field(factory=IntervalTree, repr=False)
    other_blocks: 'IntervalTree' = field(factory=IntervalTree, repr=False,
validator=check_block_ranges)
    event_to_block: dict['BasicTranscriptEvent', 'TranscriptAlignmentBlock'] =
field(factory=dict, repr=False)
    block_to_events: dict['TranscriptAlignmentBlock', tuple['BasicTranscriptEvent', ...]] =
field(factory=dict, repr=False)

    @other.validator
    def _check_transcripts(self, attribute, value):
        if value.gene_id != self.anchor.gene_id:
            raise ValueError(f'{self.anchor} and {value} are from different genes')
```

```
  def __attrs_post_init__(self):
    if any(i.data.is_insertion for i in self.anchor_events if isinstance(i.data,
BasicTranscriptEvent)):
      raise ValueError
    if any(i.data.is_deletion for i in self.other_events if isinstance(i.data,
BasicTranscriptEvent)):
      raise ValueError
    anchor_matches = {i.data for i in self.anchor_blocks if i.data.category is
SeqAlignCat.MATCH}
    other_matches = {i.data for i in self.other_blocks if i.data.category is
SeqAlignCat.MATCH}
    if anchor_matches != other_matches:
      raise ValueError(f'{anchor_matches} != {other_matches}')
    total_delta_nt = sum(event.delta_nt for event in self.events)
    tx_length_diff = self.other.length - self.anchor.length
    if total_delta_nt != tx_length_diff:
      raise ValueError(f'TranscriptEvent lengths add up to {total_delta_nt}; expected
{tx_length_diff}')
    for event in self.basic_events:
      block = self.event_to_block.get(event, None)
      if event not in self.block_to_events.get(block, ()):
        raise ValueError(f'Broken event-to-block mapping for {event}')

  @property
  def basic_events(self) -> tuple['BasicTranscriptEvent', ...]:
    return tuple(chain.from_iterable(event.members for event in self.events))

  @property
  def blocks(self) -> tuple['TranscriptAlignmentBlock', ...]:
    return tuple(sorted({i.data for i in chain(self.anchor_blocks, self.other_blocks)}))

  @classmethod
  @lru_cache(maxsize=CACHE_SIZE)
  def from_transcripts(cls, anchor: 'Transcript', other: 'Transcript'):
    splice_events, tss_event, apa_event = call_transcript_events(anchor, other)
    events = splice_events.copy()
    if tss_event:
      events.append(tss_event)
    if apa_event:
      events.append(apa_event)

    # map all events to transcript coordinates
    def get_transcript_interval(event: 'BasicTranscriptEvent'):
```

```
        transcript = anchor if event.is_deletion else other
        start = transcript.get_transcript_coord_from_genome_coord(event.start)
        stop = transcript.get_transcript_coord_from_genome_coord(event.stop) + 1
        return Interval(start, stop, event)

    event_to_interval: dict['BasicTranscriptEvent', 'Interval'] = dict()
    basic_to_compound: dict['BasicTranscriptEvent', 'CompoundTranscriptEvent'] = dict()
    for compound_event in events:
        for event in compound_event.members:
            event_to_interval[event] = get_transcript_interval(event)
            basic_to_compound[event] = compound_event

    def get_compound_map(basic_map: 'IntervalTree'):
        compound_map = IntervalTree()
        for compound_event, intervals in groupby(sorted(basic_map.all_intervals),
key=lambda i: basic_to_compound[i.data]):
            if len(compound_event.members) == 1:
                continue
            compound_interval = IntervalTree(intervals)
            compound_interval.merge_neighbors()
            compound_map.update(i._replace(data=compound_event) for i in
compound_interval.all_intervals)
        return compound_map

    insertions, deletions = partition(attrgetter('is_deletion'), event_to_interval.keys())
    anchor_basic = IntervalTree(event_to_interval[event] for event in deletions)
    other_basic = IntervalTree(event_to_interval[event] for event in insertions)
    anchor_compound = get_compound_map(anchor_basic)
    other_compound = get_compound_map(other_basic)
    anchor_events = anchor_compound.union(anchor_basic)
    other_events = other_compound.union(other_basic)

    # determine deletion and insertion block ranges
    del_ranges = anchor_basic.copy()
    del_ranges.merge_neighbors(data_reducer=lambda a, b: a + (b,), data_initializer=())
    ins_ranges = other_basic.copy()
    ins_ranges.merge_neighbors(data_reducer=lambda a, b: a + (b,), data_initializer=())

    # determine match block ranges
    blocks = []
    block_to_events = dict()
    event_to_block = dict()
    position = {'anchor': 0, 'other': 0}
    sorted_del_ranges = deque(sorted(map(tuple, del_ranges)))
```

```
    sorted_ins_ranges = deque(sorted(map(tuple, ins_ranges)))

    def add_match_block(length: int):
      if length > 0:
        match_block = TranscriptAlignmentBlock(
          range(position['anchor'], position['anchor'] + length),
          range(position['other'], position['other'] + length)
        )
        blocks.append(match_block)
      position['anchor'] += length
      position['other'] += length

    def add_del_block():
      del_start, del_stop, events = sorted_del_ranges.popleft()
      del_block = TranscriptAlignmentBlock(
        range(del_start, del_stop),
        range(position['other'], position['other'])
      )
      blocks.append(del_block)
      block_to_events[del_block] = events
      for event in events:
        event_to_block[event] = del_block
      position['anchor'] = del_stop

    def add_ins_block():
      ins_start, ins_stop, events = sorted_ins_ranges.popleft()
      ins_block = TranscriptAlignmentBlock(
        range(position['anchor'], position['anchor']),
        range(ins_start, ins_stop)
      )
      blocks.append(ins_block)
      block_to_events[ins_block] = events
      for event in events:
        event_to_block[event] = ins_block
      position['other'] = ins_stop

    while sorted_del_ranges or sorted_ins_ranges:
      to_next_del_block = (sorted_del_ranges[0][0] - position['anchor']) if
sorted_del_ranges else float('inf')
      to_next_ins_block = (sorted_ins_ranges[0][0] - position['other']) if sorted_ins_ranges
else float('inf')
      add_match_block(min(to_next_del_block, to_next_ins_block))
      if to_next_del_block < to_next_ins_block:
        add_del_block()
```

```
        elif to_next_ins_block < to_next_del_block:
          add_ins_block()
        else:
          anchor_pos_genomic =
anchor.get_genome_coord_from_transcript_coord(position['anchor'])
          other_pos_genomic =
other.get_genome_coord_from_transcript_coord(position['other'])
          if anchor_pos_genomic < other_pos_genomic:
            add_del_block()
            add_ins_block()
          else:
            add_ins_block()
            add_del_block()
      assert anchor.length - position['anchor'] == other.length - position['other'], f'{position=},
{anchor.length=}, {other.length=}, {anchor=}'
      add_match_block(anchor.length - position['anchor'])

      anchor_blocks = IntervalTree.from_tuples((block.anchor_range.start,
block.anchor_range.stop, block) for block in blocks if block.anchor_range)
      other_blocks = IntervalTree.from_tuples((block.other_range.start,
block.other_range.stop, block) for block in blocks if block.other_range)

      return cls(anchor, other, events, anchor_events, anchor_blocks, other_events,
other_blocks, event_to_block, block_to_events)


@define(eq=False)
class CodonAlignment(ProjectionMixin):
  anchor: 'Protein'
  other: 'Protein'
  anchor_blocks: 'IntervalTree' = field(factory=IntervalTree, repr=False,
validator=check_block_ranges)
  other_blocks: 'IntervalTree' = field(factory=IntervalTree, repr=False,
validator=check_block_ranges)
  tblock_to_cblocks: dict['TranscriptAlignmentBlock', tuple['CodonAlignmentBlock', ...]] =
field(factory=dict, repr=False)
  cblock_to_tblock: dict['CodonAlignmentBlock', 'TranscriptAlignmentBlock'] =
field(factory=dict, repr=False)

  def __attrs_post_init__(self):
    anchor_shared = {i.data for i in self.anchor_blocks if i.data.category not in
ANCHOR_EXCLUSIVE}
    other_shared = {i.data for i in self.other_blocks if i.data.category not in
OTHER_EXCLUSIVE}
```

```
      if anchor_shared != other_shared:
        raise ValueError(f'{anchor_shared} != {other_shared}')
      for cblock in filter(lambda cblock: cblock.category is not CodonAlignCat.MATCH,
self.blocks):
        tblock = self.cblock_to_tblock.get(cblock, None)
        if tblock and cblock not in self.tblock_to_cblocks.get(tblock, ()):
          raise ValueError(f'Broken tblock-to-cblock mapping for {cblock}')

    @property
    def blocks(self) -> tuple['CodonAlignmentBlock', ...]:
      return tuple(sorted({i.data for i in chain(self.anchor_blocks, self.other_blocks)}))

    def project_feature(self, anchor_feature: 'ProteinFeature'):
      if anchor_feature.protein is not self.anchor:
        raise ValueError(f'{anchor_feature} is not a feature of {self.anchor}')
      other_range = self.project_range(anchor_feature.protein_start - 1,
anchor_feature.protein_stop)
      if not other_range:
        return None
      other_blocks = self.range_to_blocks(other_range.start, other_range.stop,
from_anchor=False)
      differences = [
        ((len(block.anchor_range) if block.anchor_range else len(block.other_range)),
block.category)
        for block in other_blocks
      ]
      proj_feat = ProjectedFeature(
        feature = anchor_feature.feature,
        protein = self.other,
        protein_start = other_range.start + 1,
        protein_stop = other_range.stop,
        reference = False,
        anchor = anchor_feature,
        _differences = ','.join(f'{t[0]}{t[1]}' for t in differences)
      )
      return proj_feat

    @classmethod
    @lru_cache(maxsize=CACHE_SIZE)
    def from_proteins(cls, anchor: 'Protein', other: 'Protein'):
      tx_aln = TranscriptAlignment.from_transcripts(anchor.transcript, other.transcript)
      anchor_orf_range = range(anchor.orf.transcript_start - 1, anchor.orf.transcript_stop)
      other_orf_range = range(other.orf.transcript_start - 1, other.orf.transcript_stop)
      anchor_orf_len = len(anchor_orf_range)
```

```
    other_orf_len = len(other_orf_range)
    anchor_pr_len = anchor.length
    other_pr_len = other.length

    def compare_ranges(a: range, b: range):
      if a.start < b.stop and b.start < a.stop:
        return 0
      elif a.stop <= b.start:
        return -1
      else:
        return 1

    # convert transcript-relative coords to ORF-relative coords
    tx_blocks = deque(
      (
        block.category,
        range(block.anchor_range.start - anchor_orf_range.start, block.anchor_range.stop
- anchor_orf_range.start),
        range(block.other_range.start - other_orf_range.start, block.other_range.stop -
other_orf_range.start)
      ) for block in tx_aln.blocks
    )

    def split_paired_ranges(a: range, b: range, a_split: int):
      assert a_split in a
      a_left, a_right = range(a.start, a_split), range(a_split, a.stop)
      if b:
        b_split = b.start + len(a_left)
        b_left, b_right = range(b.start, b_split), range(b_split, b.stop)
      else:
        b_left, b_right = b, b
      return a_left, a_right, b_left, b_right

    boundaries = []
    overhangs = []
    categories = []
    frame_to_category = {
      0: CodonAlignCat.MATCH,
      1: CodonAlignCat.FRAME_AHEAD,
      2: CodonAlignCat.FRAME_BEHIND
    }
    while tx_blocks:
      tx_category, anchor_tx_range, other_tx_range = tx_blocks.popleft()
```

```
        # if block overlaps an ORF boundary, split it up
        if anchor_tx_range.start < 0 < anchor_tx_range.stop:
            anchor_tx_range, next_anchor_range, other_tx_range, next_other_range =
split_paired_ranges(anchor_tx_range, other_tx_range, 0)
            tx_blocks.appendleft((tx_category, next_anchor_range, next_other_range))
        if other_tx_range.start < 0 < other_tx_range.stop:
            other_tx_range, next_other_range, anchor_tx_range, next_anchor_range =
split_paired_ranges(other_tx_range, anchor_tx_range, 0)
            tx_blocks.appendleft((tx_category, next_anchor_range, next_other_range))
        if anchor_tx_range.start < anchor_orf_len < anchor_tx_range.stop:
            anchor_tx_range, next_anchor_range, other_tx_range, next_other_range =
split_paired_ranges(anchor_tx_range, other_tx_range, anchor_orf_len)
            tx_blocks.appendleft((tx_category, next_anchor_range, next_other_range))
        if other_tx_range.start < other_orf_len < other_tx_range.stop:
            other_tx_range, next_other_range, anchor_tx_range, next_anchor_range =
split_paired_ranges(other_tx_range, anchor_tx_range, other_orf_len)
            tx_blocks.appendleft((tx_category, next_anchor_range, next_other_range))

        # skip blocks that are outside both ORF ranges
        outside_anchor_orf = compare_ranges(anchor_tx_range, range(0,
len(anchor_orf_range)))
        outside_other_orf = compare_ranges(other_tx_range, range(0, len(other_orf_range)))
        if (outside_anchor_orf and outside_other_orf
        or outside_anchor_orf and tx_category is SeqAlignCat.DELETION
        or outside_other_orf and tx_category is SeqAlignCat.INSERTION):
            continue

        # convert block range to protein coords
        if outside_anchor_orf < 0:
            anchor_pr_start, anchor_start_overhang, anchor_pr_stop, anchor_stop_overhang =
0, 0, 0, 0
        elif outside_anchor_orf > 0:
            anchor_pr_start, anchor_start_overhang, anchor_pr_stop, anchor_stop_overhang =
(anchor_pr_len, 0, anchor_pr_len, 0)
        else:
            anchor_pr_start, anchor_start_overhang = divmod(anchor_tx_range.start, 3)
            anchor_pr_stop, anchor_stop_overhang = divmod(anchor_tx_range.stop, 3)
        if outside_other_orf < 0:
            other_pr_start, other_start_overhang, other_pr_stop, other_stop_overhang = 0, 0, 0,
0
        elif outside_other_orf > 0:
            other_pr_start, other_start_overhang, other_pr_stop, other_stop_overhang =
(other_pr_len, 0, other_pr_len, 0)
        else:
```

```
            other_pr_start, other_start_overhang = divmod(other_tx_range.start, 3)
            other_pr_stop, other_stop_overhang = divmod(other_tx_range.stop, 3)

        # infer codon block category
        if tx_category is SeqAlignCat.MATCH:
            if outside_anchor_orf:
                cd_category = CodonAlignCat.TRANSLATED
            elif outside_other_orf:
                cd_category = CodonAlignCat.UNTRANSLATED
            else:
                frameshift = (other_start_overhang - anchor_start_overhang) % 3
                cd_category = frame_to_category[frameshift]
        elif tx_category is SeqAlignCat.DELETION:
            cd_category = CodonAlignCat.DELETION
        elif tx_category is SeqAlignCat.INSERTION:
            cd_category = CodonAlignCat.INSERTION
        else:
            raise RuntimeError
        boundaries.append((anchor_pr_stop, other_pr_stop))
        overhangs.append((anchor_stop_overhang, other_stop_overhang))
        categories.append(cd_category)

    # second pass to adjust edges
    assert overhangs[-1] == (0, 0)
    anchor_boundary_shifts, other_boundary_shifts = dict(), dict()

    i = 0
    while i < len(boundaries) - 1:
        curr_category, next_category = categories[i:i+2]
        overhang = overhangs[i]

        try:
            anchor_boundary = anchor_boundary_shifts[boundaries[i][0]]
        except KeyError:
            anchor_boundary = boundaries[i][0]
            shift_anchor_boundary = (
                overhang[0] == 2 and (
                    next_category in {CodonAlignCat.MATCH, CodonAlignCat.FRAME_AHEAD}
                    or curr_category is CodonAlignCat.FRAME_AHEAD
                    or curr_category in {CodonAlignCat.DELETION,
CodonAlignCat.UNTRANSLATED} and next_category is CodonAlignCat.INSERTION
                ) or overhang[0] == 1 and curr_category is CodonAlignCat.DELETION and
next_category is CodonAlignCat.FRAME_AHEAD
            )
```

```
            if shift_anchor_boundary:
                anchor_boundary_shifts[anchor_boundary] = anchor_boundary + 1
                anchor_boundary += 1
        try:
            other_boundary = other_boundary_shifts[boundaries[i][1]]
        except KeyError:
            other_boundary = boundaries[i][1]
            shift_other_boundary = (
                overhang[1] == 2 and (
                    next_category in {CodonAlignCat.MATCH, CodonAlignCat.FRAME_BEHIND}
                    or curr_category is CodonAlignCat.FRAME_BEHIND
                    or curr_category in {CodonAlignCat.INSERTION, CodonAlignCat.TRANSLATED}
and next_category is CodonAlignCat.DELETION
                ) or overhang[1] == 1 and curr_category is CodonAlignCat.INSERTION and
next_category is CodonAlignCat.FRAME_BEHIND
            )
            if shift_other_boundary:
                other_boundary_shifts[other_boundary] = other_boundary + 1
                other_boundary += 1
        boundaries[i] = anchor_boundary, other_boundary

        # insert a single-codon block if necessary
        category_to_insert = None
        if (curr_category is CodonAlignCat.MATCH and overhang == (2, 2) or next_category is
CodonAlignCat.MATCH and overhang == (1, 1)):
            category_to_insert = CodonAlignCat.EDGE
        if (curr_category is CodonAlignCat.FRAME_AHEAD and next_category is
CodonAlignCat.DELETION and overhang == (1, 2)
        or curr_category is CodonAlignCat.INSERTION and next_category is
CodonAlignCat.FRAME_AHEAD and overhang == (1, 2)
        or curr_category is CodonAlignCat.FRAME_BEHIND and next_category is
CodonAlignCat.INSERTION and overhang == (2, 1)
        or curr_category is CodonAlignCat.DELETION and next_category is
CodonAlignCat.FRAME_BEHIND and overhang == (2, 1)):
            category_to_insert = CodonAlignCat.COMPLEX

        if category_to_insert:
            boundaries.insert(i+1, (anchor_boundary + 1, other_boundary + 1))
            overhangs.insert(i+1, overhang)
            categories.insert(i+1, category_to_insert)
            anchor_boundary_shifts[anchor_boundary] = anchor_boundary + 1
            other_boundary_shifts[other_boundary] = other_boundary + 1
            i += 1
        i += 1
```

```
    #endregion

    # merge consecutive codon blocks w/ same category
    assert len(boundaries) == len(categories)
    cd_blocks: list['CodonAlignmentBlock'] = []
    prev_boundary = (0, 0)
    for category, group in groupby(range(len(categories)), key=categories.__getitem__):
      anchor_start, other_start = prev_boundary
      idx = last(group)
      anchor_stop, other_stop = boundaries[idx]
      prev_boundary = anchor_stop, other_stop
      cblock = CodonAlignmentBlock(range(anchor_start, anchor_stop), range(other_start,
other_stop), category=category)
      cd_blocks.append(cblock)

    anchor_blocks = IntervalTree.from_tuples(
      (block.anchor_range.start, block.anchor_range.stop, block)
      for block in cd_blocks if block.anchor_range
    )
    other_blocks = IntervalTree.from_tuples(
      (block.other_range.start, block.other_range.stop, block)
      for block in cd_blocks if block.other_range
    )

    # map each cblock to one or more tblocks
    tblock_to_cblocks: dict['TranscriptAlignmentBlock', tuple['CodonAlignmentBlock', ...]]
= dict()
    cblock_to_tblock: dict['CodonAlignmentBlock', 'TranscriptAlignmentBlock'] = dict()

    def link_cblock_and_tblock(cblock, tblock):
      cblock_to_tblock[cblock] = tblock
      tblock_to_cblocks.setdefault(tblock, []).append(cblock)

    nt_shared, nt_exclusive = partition(lambda t: t[1].category in CD_DEL_INS,
enumerate(cd_blocks))
    non_frameshift, frameshift = partition(lambda t: t[1].category in FRAMESHIFT,
nt_shared)
    for i, cblock in nt_exclusive:
      if cblock.anchor_range:
        tx_start, tx_stop = map(anchor.get_transcript_coord_from_protein_coord,
(cblock.anchor_range.start, cblock.anchor_range.stop))
        mapper = tx_aln.anchor_blocks
      else:
```

```
            tx_start, tx_stop = map(other.get_transcript_coord_from_protein_coord,
(cblock.other_range.start, cblock.other_range.stop))
            mapper = tx_aln.other_blocks
          intervals = mapper.overlap(tx_start + 1, tx_stop + 1)  # use coord of the nucleotide in
the middle of the codon to avoid edge cases
          tblock = one(interval.data for interval in intervals if interval.data.category in
SEQ_DEL_INS)
          link_cblock_and_tblock(cblock, tblock)

      # map frameshifts to closest preceding del/ins tblock with length indivisible by 3
      match_tblock_to_nonsymmetric_tblock = dict()
      latest_nonsymmetric_tblock = None
      for tblock in tx_aln.blocks:
        if tblock.category is SeqAlignCat.MATCH:
          if latest_nonsymmetric_tblock:
            match_tblock_to_nonsymmetric_tblock[tblock] = latest_nonsymmetric_tblock
        else:
          if (len(tblock.other_range) - len(tblock.anchor_range)) % 3:
            latest_nonsymmetric_tblock = tblock
      for i, cblock in frameshift:
          tx_start, tx_stop = map(anchor.get_transcript_coord_from_protein_coord,
(cblock.anchor_range.start, cblock.anchor_range.stop))
          intervals = tx_aln.anchor_blocks.overlap(tx_start + 1, tx_stop + 1)
          match_tblock = one(interval.data for interval in intervals if interval.data.category is
SeqAlignCat.MATCH)
          try:
            tblock = match_tblock_to_nonsymmetric_tblock[match_tblock]
          except KeyError:
            pass
          else:
            link_cblock_and_tblock(cblock, tblock)

      for i, cblock in non_frameshift:
          tblock = None
          if cblock.category in {CodonAlignCat.UNTRANSLATED,
CodonAlignCat.TRANSLATED}:
            if not cblock.other_range:
              nterminal = cblock.other_range.start == 0
            else:
              nterminal = cblock.anchor_range.start == 0
            if nterminal:
              mapper = anchor_blocks if cblock.category is CodonAlignCat.UNTRANSLATED
else other_blocks
              start_codon_cblock = only(
```

```
                interval.data for interval in mapper.at(0)
                if interval.data.category in CD_DEL_INS
              )
              tblock = cblock_to_tblock.get(start_codon_cblock)
            else:
              if cblock.category is CodonAlignCat.UNTRANSLATED:
                mapper = other_blocks
                stop_codon_pos = other.length - 1
              else:
                mapper = anchor_blocks
                stop_codon_pos = anchor.length - 1
              stop_codon_cblock = one(interval.data for interval in
mapper.at(stop_codon_pos))
              tblock = cblock_to_tblock.get(stop_codon_cblock)
          elif cblock.category in SPLIT_CODON:
            del_ins_cblock = cd_blocks[i-1] if cd_blocks[i-1].category in CD_DEL_INS else
cd_blocks[i+1]
            tblock = cblock_to_tblock.get(del_ins_cblock)
          if tblock:
            link_cblock_and_tblock(cblock, tblock)

      tblock_to_cblocks = dict((k, tuple(v)) for (k, v) in sorted(tblock_to_cblocks.items()))
      cblock_to_tblock = dict(sorted(cblock_to_tblock.items()))

      return cls(anchor, other, anchor_blocks, other_blocks, tblock_to_cblocks,
cblock_to_tblock)


@define(eq=False)
class ProteinAlignment:
   anchor: 'Protein'
   other: 'Protein'
   anchor_blocks: 'IntervalTree' = field(factory=IntervalTree, repr=False,
validator=check_block_ranges)
   other_blocks: 'IntervalTree' = field(factory=IntervalTree, repr=False,
validator=check_block_ranges)
   cblock_to_pblock: dict['CodonAlignmentBlock', 'ProteinAlignmentBlock'] =
field(factory=dict, repr=False)
   pblock_to_cblocks: dict['ProteinAlignmentBlock', tuple['CodonAlignmentBlock', ...]] =
field(factory=dict, repr=False)

   def __attrs_post_init__(self):
      for pblock in self.blocks:
         cblocks = self.pblock_to_cblocks.get(pblock, ())
```

```
        if any(pblock is not self.cblock_to_pblock.get(cblock) for cblock in cblocks):
            raise ValueError(f'Broken pblock-to-cblock mapping for {pblock}')

    @property
    def blocks(self) -> tuple['ProteinAlignmentBlock', ...]:
        return tuple(sorted({i.data for i in chain(self.anchor_blocks, self.other_blocks)}))

    @classmethod
    @lru_cache(maxsize=CACHE_SIZE)
    def from_proteins(cls, anchor: 'Protein', other: 'Protein'):
        cd_aln = CodonAlignment.from_proteins(anchor, other)

        pblock_to_cblocks: dict['ProteinAlignmentBlock', tuple['CodonAlignmentBlock', ...]] =
dict()
        cblock_to_pblock: dict['CodonAlignmentBlock', 'ProteinAlignmentBlock'] = dict()

        pblock_bounds: list[int] = [
            last(cblock_indices) + 1
            for _, cblock_indices in groupby(
                range(len(cd_aln.blocks)),
                key = lambda i: cd_aln.blocks[i].category is CodonAlignCat.MATCH
            )
        ]
        pblock_categories = []
        pblock_split5 = [False for pblock in pblock_bounds]
        pblock_split3 = [False for pblock in pblock_bounds]
        pblock_ragged5 = [False for pblock in pblock_bounds]
        pblock_ragged3 = [False for pblock in pblock_bounds]
        for p, (c0, c1) in enumerate(windowed([0] + pblock_bounds, 2)):
            cblocks = cd_aln.blocks[c0:c1]
            anchor_start, anchor_stop = cblocks[0].anchor_range.start, cblocks[-
1].anchor_range.stop
            other_start, other_stop = cblocks[0].other_range.start, cblocks[-1].other_range.stop
            reduced_cblock_categories = {cblock.category for cblock in cblocks} - SPLIT_CODON
            anchor_sequence = anchor.sequence[anchor_start:anchor_stop]
            other_sequence = other.sequence[other_start:other_stop]
            if anchor_sequence == other_sequence:
                category = SeqAlignCat.MATCH
            elif reduced_cblock_categories <= ANCHOR_EXCLUSIVE:
                category = SeqAlignCat.DELETION
                pblock_split5[p] = cblocks[0].category in SPLIT_CODON
                pblock_split3[p] = cblocks[-1].category in SPLIT_CODON
            elif reduced_cblock_categories <= OTHER_EXCLUSIVE:
                category = SeqAlignCat.INSERTION
```

```
            pblock_split5[p] = cblocks[0].category in SPLIT_CODON
            pblock_split3[p] = cblocks[-1].category in SPLIT_CODON
        else:
            category = SeqAlignCat.SUBSTITUTION
        pblock_categories.append(category)
        pblock_ragged5[p] = pblock_split5[p] and anchor_sequence[0] != other_sequence[0]
        pblock_ragged3[p] = pblock_split3[p] and anchor_sequence[-1] != other_sequence[-
1]

    assert len(pblock_bounds) == len(pblock_categories)

    # second pass to move synonymous split codons into match pblocks
    for p, (split5, split3, ragged5, ragged3) in enumerate(zip(pblock_split5, pblock_split3,
pblock_ragged5, pblock_ragged3)):
        if split5 and not ragged5:
            pblock_bounds[p-1] += 1
        if split3 and not ragged3:
            pblock_bounds[p] -= 1

    for p, (c0, c1) in enumerate(windowed([0] + pblock_bounds, 2)):
        cblocks = cd_aln.blocks[c0:c1]
        anchor_range = range(cblocks[0].anchor_range.start, cblocks[-1].anchor_range.stop)
        other_range = range(cblocks[0].other_range.start, cblocks[-1].other_range.stop)
        pblock = ProteinAlignmentBlock(
            anchor_range,
            other_range,
            category = pblock_categories[p],
            ragged5 = pblock_ragged5[p],
            ragged3 = pblock_ragged3[p]
        )
        pblock_to_cblocks[pblock] = cblocks
        for cblock in cblocks:
            cblock_to_pblock[cblock] = pblock
    # TODO: second pass to merge match pblocks

    anchor_blocks = IntervalTree.from_tuples((block.anchor_range.start,
block.anchor_range.stop, block) for block in pblock_to_cblocks if block.anchor_range)
    other_blocks = IntervalTree.from_tuples((block.other_range.start,
block.other_range.stop, block) for block in pblock_to_cblocks if block.other_range)

    return cls(anchor, other, anchor_blocks, other_blocks, cblock_to_pblock,
pblock_to_cblocks)
```

constants.py

```
from enum import Flag, auto, Enum
from typing import Union

from biosurfer.core.helpers import OrderedEnum, StringEnum


class Strand(OrderedEnum):
   PLUS = auto()
   MINUS = auto()
   UNKNOWN = auto()

   def __str__(self):
      if self is Strand.PLUS:
         return '+'
      elif self is Strand.MINUS:
         return '-'
      else:
         return '?'

   @classmethod
   def from_symbol(cls, symbol: str) -> 'Strand':
      if symbol == '+':
         return Strand.PLUS
      elif symbol == '-':
         return Strand.MINUS
      else:
         raise ValueError(f'\'{symbol}\' is not a valid strand')


class Nucleobase(StringEnum):
   ADENINE = 'A'
   CYTOSINE = 'C'
   GUANINE = 'G'
   THYMINE = 'T'
   URACIL = 'U'
   GAP = '-'


class AminoAcid(StringEnum):
   ALANINE = 'A'
   ALA = 'A'
   ISOLEUCINE = 'I'
```

```
ILE = 'I'
LEUCINE = 'L'
LEU = 'L'
METHIONINE = 'M'
MET = 'M'
VALINE = 'V'
VAL = 'V'
PHENYLALANINE = 'F'
PHE = 'F'
TRYPTOPHAN = 'W'
TRP = 'W'
TYROSINE = 'Y'
TYR = 'Y'
ASPARAGINE = 'N'
ASN = 'N'
CYSTEINE = 'C'
CYS = 'C'
GLUTAMINE = 'Q'
GLN = 'Q'
SERINE = 'S'
SER = 'S'
THREONINE = 'T'
THR = 'T'
ASPARTATE = 'D'
ASP = 'D'
GLUTAMATE = 'E'
GLU = 'E'
ARGININE = 'R'
ARG = 'R'
HISTIDINE = 'H'
HIS = 'H'
LYSINE = 'K'
LYS = 'K'
GLYCINE = 'G'
GLY = 'G'
PROLINE = 'P'
PRO = 'P'
SELENOCYSTEINE = 'U'
SEC = 'U'
STOP = '*'  # included for ease of use
UNKNOWN = 'X'
GAP = '-'
```

```
class SequenceAlignmentCategory(StringEnum):
   MATCH = 'M'
   INSERTION = 'I'
   DELETION = 'D'
   SUBSTITUTION = 'S'
   UNKNOWN = '?'


SEQ_DEL_INS = {SequenceAlignmentCategory.DELETION,
SequenceAlignmentCategory.INSERTION}


class CodonAlignmentCategory(StringEnum):
   MATCH = 'm'
   INSERTION = 'i'
   DELETION = 'd'
   TRANSLATED = 't'
   UNTRANSLATED = 'u'
   FRAME_AHEAD = 'a'
   FRAME_BEHIND = 'b'
   EDGE = 'e'
   COMPLEX = 'x'
   UNKNOWN = '?'


ANCHOR_EXCLUSIVE = {CodonAlignmentCategory.DELETION,
CodonAlignmentCategory.UNTRANSLATED}
OTHER_EXCLUSIVE = {CodonAlignmentCategory.INSERTION,
CodonAlignmentCategory.TRANSLATED}
CD_DEL_INS = {CodonAlignmentCategory.DELETION,
CodonAlignmentCategory.INSERTION}
FRAMESHIFT = {CodonAlignmentCategory.FRAME_AHEAD,
CodonAlignmentCategory.FRAME_BEHIND}
SPLIT_CODON = {CodonAlignmentCategory.EDGE, CodonAlignmentCategory.COMPLEX}


AlignmentCategory = Union[SequenceAlignmentCategory, CodonAlignmentCategory]


class AnnotationFlag(Flag):
   NONE = 0
   SE = auto()
   IE = auto()
   A5SS = auto()
```

```
    A3SS = auto()
    IR = auto()
    IX = auto()
    SIF = auto()

    MXIC = auto()
    UIC = auto()
    DIC = auto()
    TIS = auto()
    UP_TIS = UIC | TIS
    DN_TIS = DIC | TIS
    TSS = auto()

    ACTE = auto()
    UTC = auto()
    DTC = auto()
    EXITC = auto()

    def __str__(self):
        raw = super().__str__()
        return raw.split('.')[1]


class ProteinRegion(OrderedEnum):
    NTERMINUS = auto()
    INTERNAL = auto()
    CTERMINUS = auto()

    def __str__(self):
        if self is ProteinRegion.NTERMINUS:
            return 'Nterm'
        elif self is ProteinRegion.INTERNAL:
            return 'internal'
        elif self is ProteinRegion.CTERMINUS:
            return 'Cterm'


class NTerminalChange(OrderedEnum):
    MUTUALLY_EXCLUSIVE = auto()
    DOWNSTREAM_SHARED = auto()
    UPSTREAM_SHARED = auto()
    MUTUALLY_SHARED = auto()
    ALTERNATIVE_ORF = auto()
    UNKNOWN = auto()
```

```
class CTerminalChange(OrderedEnum):
    SPLICING = auto()
    FRAMESHIFT = auto()
    ALTERNATIVE_ORF = auto()
    UNKNOWN = auto()


class FeatureType(Enum):
    DOMAIN = auto()
    IDR = auto()
    COILED = auto()
    LCR = auto()
    PTM = auto()
    SIGNALP = auto()
    TRANSMEMBRANE = auto()
    NONE = auto()
    # TODO: add more types


class APPRIS(OrderedEnum):
    NONE = auto()
    ALTERNATIVE = auto()
    PRINCIPAL = auto()


class SQANTI(OrderedEnum):
    FSM = auto()
    ISM = auto()
    NIC = auto()
    NNC = auto()
    OTHER = auto()

    def __str__(self):
        return self.name


START_CODON = 'ATG'
STOP_CODONS = {'TGA', 'TAA', 'TAG'}
```

database.py

```
import csv
import os
import re
from itertools import chain, groupby
from operator import attrgetter, itemgetter
from pathlib import Path
from sqlite3 import Connection as SQLite3Connection
from typing import TYPE_CHECKING, Callable, Dict, Iterable
from warnings import warn

from Bio import SeqIO
from biosurfer.core.alignments import CodonAlignment
from biosurfer.core.constants import (APPRIS, SQANTI, STOP_CODONS, AminoAcid,
                                      FeatureType, Strand)
from biosurfer.core.helpers import (ExceptionLogger, FastaHeaderFields,
                                    bulk_upsert, count_lines, read_gtf_line)
from biosurfer.core.models.base import Base
from biosurfer.core.models.biomolecules import (ORF, Chromosome, Exon,
                                                GencodeTranscript, Gene,
                                                PacBioTranscript, Protein,
                                                Transcript)
from biosurfer.core.models.features import (Domain, Feature, ProjectedFeature,
                                            ProteinFeature)
from more_itertools import chunked
from sqlalchemy import create_engine, delete, event, select
from sqlalchemy.dialects.sqlite import insert
from sqlalchemy.engine import Engine
from sqlalchemy.orm import contains_eager, joinedload, raiseload, scoped_session,
sessionmaker
from sqlalchemy.sql.expression import desc
from sqlalchemy.sql.functions import func
from tqdm import tqdm

CHUNK_SIZE = 5000
SQANTI_DICT = {
   'full-splice_match': SQANTI.FSM,
   'incomplete-splice_match': SQANTI.ISM,
   'novel_in_catalog': SQANTI.NIC,
   'novel_not_in_catalog': SQANTI.NNC
}

class Database:
```

```
  _databases_dir = Path(__file__).parent.parent.parent/'databases'
  registry: Dict[str, 'Database'] = {}

  @staticmethod
  def _get_db_url_from_name(name: str):
    if name:
      db_file = f'{name}.sqlite3'
      return f'sqlite:///{Database._databases_dir/db_file}'
    else:
      return 'sqlite://'

  def __new__(cls, name: str = None, *, url: str = None, **kwargs):
    if url is None:
      url = Database._get_db_url_from_name(name)
    if url in Database.registry:
      return Database.registry[url]
    else:
      obj = super().__new__(cls)
      Database.registry[url] = obj
      return obj

  def __init__(self, name: str = None, *, url: str = None, sessionfactory=None):
    if url is None:
      url = Database._get_db_url_from_name(name)
    self.url = url
    self._engine = create_engine(self.url)
    Base.metadata.create_all(self._engine)
    if sessionfactory is None:
      self._sessionmaker = scoped_session(sessionmaker(autocommit=False,
autoflush=False, bind=self.engine, future=True))
    else:
      self._sessionmaker = sessionfactory

  def __repr__(self):
    return f'Database(url=\'{self.url}\')'

  @property
  def engine(self):
    return self._engine

  def get_session(self, **kwargs):
    return self._sessionmaker(**kwargs)

  def recreate_tables(self):
```

```
        print(f'Recreating tables in {self.url} ...')
        Base.metadata.drop_all(bind=self._engine)
        Base.metadata.create_all(bind=self._engine)

    def load_gencode_gtf(self, gtf_file: str, overwrite=False) -> None:
        with self.get_session() as session:
            # if overwrite:
            #     with session.begin():
            #         for model in (Chromosome, Gene, Exon, ORF):
            #             print(f'Clearing table \'{model.__tablename__}\'...')
            #             session.execute(delete(model.__table__))
            #         print(f'Clearing GENCODE transcripts from \'{Transcript.__tablename__}\'...')
            #         session.execute(delete(Transcript.__table__).where(Transcript.type ==
'gencodetranscript'))

            chromosomes = {}
            genes_to_upsert = []
            transcripts_to_upsert = []
            exons_to_upsert = []
            orfs_to_upsert = []

            minus_transcripts = set()
            transcripts_to_exons = {}
            transcripts_to_cdss = {}

            with open(gtf_file) as gtf:
                lines = count_lines(gtf)
                t = tqdm(gtf, desc=f'Reading GENCODE annotations', total=lines, unit='lines')
                for i, line in enumerate(t, start=1):
                    if line.startswith("#"):
                        continue
                    chr, _, feature, start, stop, _, strand, _, attributes, tags = read_gtf_line(line)
                    if any('_NF' in tag for tag in tags):  # biosurfer does not handle start_NF and
end_NF transcripts very well
                        continue
                    gene_id = attributes['gene_id']
                    gene_name = attributes['gene_name']
                    if attributes['gene_type'] != 'protein_coding':
                        continue

                    if feature == 'gene':
                        if chr not in chromosomes:
                            with session.begin():
                                chromosome = session.merge(Chromosome(name=chr))
```

```
            chromosomes[chr] = chromosome
        else:
            chromosome = chromosomes[chr]
        gene = {
            'accession': gene_id,
            'name': gene_name,
            'strand': Strand.from_symbol(strand),
            'chromosome_id': chr
        }
        genes_to_upsert.append(gene)

    elif feature == 'transcript':
        transcript_id = attributes['transcript_id']
        transcript_name = attributes['transcript_name']
        appris = APPRIS.NONE
        start_nf, end_nf = False, False
        for tag in tags:
            if 'appris_principal' in tag:
                appris = APPRIS.PRINCIPAL
            if 'appris_alternative' in tag:
                appris = APPRIS.ALTERNATIVE
            start_nf, end_nf = False, False
            # start_nf = start_nf or 'start_NF' in tag
            # end_nf = end_nf or 'end_NF' in tag
        transcript = {
            'accession': transcript_id,
            'name': transcript_name,
            'type': 'gencodetranscript',
            'gene_id': gene_id,
            'strand': Strand.from_symbol(strand),
            'appris': appris,
            'start_nf': start_nf,
            'end_nf': end_nf
        }
        transcripts_to_upsert.append(transcript)
        if Strand.from_symbol(strand) is Strand.MINUS:
            minus_transcripts.add(transcript_id)

    elif feature == 'exon':
        transcript_id = attributes['transcript_id']
        exon_id = attributes['exon_id']
        exon = {
            'accession': exon_id,
            'start': start,
```

```
                'stop': stop,
                'transcript_id': transcript_id
              }
              if transcript_id in transcripts_to_exons:
                transcripts_to_exons[transcript_id].append(exon)
              else:
                transcripts_to_exons[transcript_id] = [exon,]
              exons_to_upsert.append(exon)

            elif feature == 'CDS':
              transcript_id = attributes['transcript_id']
              protein_id = attributes['protein_id']
              cds = (start, stop, protein_id)
              if transcript_id in transcripts_to_cdss:
                transcripts_to_cdss[transcript_id].append(cds)
              else:
                transcripts_to_cdss[transcript_id] = [cds,]

            if i % CHUNK_SIZE == 0:
              bulk_upsert(session, Gene.__table__, genes_to_upsert)
              bulk_upsert(session, GencodeTranscript, transcripts_to_upsert)
          bulk_upsert(session, Gene.__table__, genes_to_upsert)
          bulk_upsert(session, GencodeTranscript, transcripts_to_upsert)

        # calculate the coordinates of each exon relative to the sequence of its parent
transcript
        _process_exons(transcripts_to_exons, minus_transcripts)
        t = tqdm(
          exons_to_upsert,
          desc = 'Upserting exons',
          total = len(exons_to_upsert),
          unit = 'exons'
        )
        for exons in chunked(t, CHUNK_SIZE):
          bulk_upsert(session, Exon.__table__, exons, primary_keys=('accession',
'transcript_id'))

        # assemble CDS intervals into ORFs
        orfs_to_upsert = _process_orfs(transcripts_to_cdss, transcripts_to_exons,
minus_transcripts)
        t = tqdm(
          orfs_to_upsert,
          desc = 'Upserting ORFs',
          total = len(orfs_to_upsert),
```

```
            unit = 'ORFs'
        )
        for orfs in chunked(t, CHUNK_SIZE):
            bulk_upsert(session, ORF.__table__, orfs, primary_keys=('transcript_id', 'position'))

    def load_pacbio_gtf(self, gtf_file: str, overwrite=False) -> None:
        with self.get_session() as session:
            # if overwrite:
            #     existing_genes = {}
            #     with session.begin():
            #         for model in (Chromosome, Gene, Exon, ORF):
            #             print(f'Clearing table \'{model.__tablename__}\'...')
            #             session.execute(delete(model.__table__))
            #         print(f'Clearing PacBio transcripts from \'{Transcript.__tablename__}\'...')
            #         session.execute(delete(Transcript.__table__).where(Transcript.type ==
'pacbiotranscript'))
            # else:
            with session.begin():
                existing_genes = {row.name: row.accession for row in session.query(Gene.name,
Gene.accession).all()}

            chromosomes = {}
            genes_to_insert = []
            transcripts_to_upsert = []
            exons_to_upsert = []
            orfs_to_upsert = []

            minus_transcripts = set()
            transcripts_to_exons = {}
            transcripts_to_cdss = {}

            with open(gtf_file) as gtf:
                lines = count_lines(gtf)
                t = tqdm(gtf, desc=f'Reading PacBio annotations', total=lines, unit='lines')
                for i, line in enumerate(t, start=1):
                    if line.startswith("#"):
                        continue
                    chr, _, feature, start, stop, _, strand, _, attributes, _ = read_gtf_line(line)
                    if chr not in chromosomes:
                        with session.begin():
                            chromosome = session.merge(Chromosome(name=chr))
                        chromosomes[chr] = chromosome
                    gene_name = attributes['gene_id']
                    if gene_name not in existing_genes:
```

```
    tqdm.write(f'Could not find Ensembl accession for gene \'{gene_name}\'')
    existing_genes[gene_name] = gene_name
    gene = {
      'accession': gene_name,
      'name': gene_name,
      'strand': Strand.from_symbol(strand),
      'chromosome_id': chr
    }
    genes_to_insert.append(gene)
  gene_id = existing_genes[gene_name]

  if feature == 'transcript':
    transcript_id = attributes['transcript_id'].split('|')[1]
    transcript_name = gene_name + '|' + transcript_id
    transcript = {
      'accession': transcript_id,
      'name': transcript_name,
      'type': 'pacbiotranscript',
      'gene_id': gene_id,
      'strand': Strand.from_symbol(strand),
    }
    transcripts_to_upsert.append(transcript)
    if Strand.from_symbol(strand) is Strand.MINUS:
      minus_transcripts.add(transcript_id)

  elif feature == 'exon':
    transcript_id = attributes['transcript_id'].split('|')[1]
    exon = {
      'start': start,
      'stop': stop,
      'transcript_id': transcript_id
    }
    if transcript_id in transcripts_to_exons:
      transcripts_to_exons[transcript_id].append(exon)
    else:
      transcripts_to_exons[transcript_id] = [exon,]
    exons_to_upsert.append(exon)

  elif feature == 'CDS':
    transcript_id = attributes['transcript_id'].split('|')[1]
    cds = (start, stop)
    if transcript_id in transcripts_to_cdss:
      transcripts_to_cdss[transcript_id].append(cds)
    else:
```

```
                transcripts_to_cdss[transcript_id] = [cds,]

            if i % CHUNK_SIZE == 0:
                bulk_upsert(session, Gene.__table__, genes_to_insert)
                bulk_upsert(session, PacBioTranscript, transcripts_to_upsert)
        bulk_upsert(session, Gene.__table__, genes_to_insert)
        bulk_upsert(session, PacBioTranscript, transcripts_to_upsert)

    # calculate the coordinates of each exon relative to the sequence of its parent
transcript
    _process_exons(transcripts_to_exons, minus_transcripts)
    t = tqdm(
        exons_to_upsert,
        desc = 'Upserting exons',
        total = len(exons_to_upsert),
        unit = 'exons'
    )
    for exons in chunked(t, CHUNK_SIZE):
        bulk_upsert(session, Exon.__table__, exons, primary_keys=('accession',
'transcript_id'))

    # assemble CDS intervals into ORFs
    orfs_to_upsert = _process_orfs(transcripts_to_cdss, transcripts_to_exons,
minus_transcripts)
    t = tqdm(
        orfs_to_upsert,
        desc = 'Upserting ORFs',
        total = len(orfs_to_upsert),
        unit = 'ORFs'
    )
    for orfs in chunked(t, CHUNK_SIZE):
        bulk_upsert(session, ORF.__table__, orfs, primary_keys=('transcript_id', 'position'))

  def load_transcript_fasta(self, transcript_fasta: str, id_extractor: Callable[[str],
'FastaHeaderFields'], id_filter: Callable[[str], bool] = lambda x: False):
    with self.get_session() as session:
        with session.begin():
            existing_transcripts = {row.accession for row in
session.query(Transcript.accession)}
            existing_orfs = {
                row.transcript_id: (row.position, row.transcript_start, row.transcript_stop)
                for row in session.execute(select(ORF.position, ORF.transcript_id,
ORF.transcript_start, ORF.transcript_stop))
            }
```

```
        transcripts_to_update = []
        orfs_to_update = []
        with open(transcript_fasta) as f:
            records = count_lines(f, only=lambda line: line.startswith('>'))
        t = tqdm(SeqIO.parse(transcript_fasta, 'fasta'), desc='Reading transcripts fasta',
total=records, unit='seqs')
        for record in t:
            if id_filter(record.id):
                continue
            ids = id_extractor(record.id)
            transcript_id = ids.transcript_id
            sequence = str(record.seq)
            if transcript_id in existing_transcripts:
                transcript = {'accession': transcript_id, 'sequence': sequence}
                transcripts_to_update.append(transcript)
            if transcript_id in existing_orfs:
                position, tx_start, tx_stop = existing_orfs[transcript_id]
                # if orf is followed by a stop codon in transcript sequence, modify tx_stop to
include the stop codon
                if sequence[tx_stop:tx_stop+3] in STOP_CODONS:
                    orfs_to_update.append({
                        'transcript_id': transcript_id,
                        'position': position,
                        'transcript_start': tx_start,
                        'transcript_stop': tx_stop + 3,
                        'has_stop_codon': True
                    })

            if len(transcripts_to_update) == CHUNK_SIZE:
                bulk_upsert(session, Transcript.__table__, transcripts_to_update)
                bulk_upsert(session, ORF.__table__, orfs_to_update,
primary_keys=('transcript_id', 'position'))
        bulk_upsert(session, Transcript.__table__, transcripts_to_update)
        bulk_upsert(session, ORF.__table__, orfs_to_update, primary_keys=('transcript_id',
'position'))

    def load_translation_fasta(self, translation_fasta: str, id_extractor: Callable[[str],
'FastaHeaderFields'], id_filter: Callable[[str], bool] = lambda x: False, overwrite: bool =
False):
        with self.get_session() as session:
            # if overwrite:
            #     with session.begin():
            #         print(f'Clearing table \'{Protein.__tablename__}\'...')
            #         session.execute(delete(Protein.__table__))
```

```
        with session.begin():
            existing_orfs = {}
            for transcript_id, position, start, stop, has_stop_codon in
session.query(ORF.transcript_id, ORF.position, ORF.transcript_start, ORF.transcript_stop,
ORF.has_stop_codon):
                existing_orfs.setdefault(transcript_id, []).append((position, start, stop,
has_stop_codon))

        orfs_to_update = []
        proteins_to_upsert = []
        with open(translation_fasta) as f:
            records = count_lines(f, only=lambda line: line.startswith('>'))
        t = tqdm(SeqIO.parse(translation_fasta, 'fasta'), desc='Reading translations fasta',
total=records, unit='seqs')
        for i, record in enumerate(t, start=1):
            if id_filter(record.id):
                continue
            ids = id_extractor(record.id)
            transcript_id = ids.transcript_id
            protein_id = ids.protein_id
            sequence = str(record.seq)
            seq_length = len(sequence)

            protein = {'accession': protein_id, 'sequence': sequence}
            proteins_to_upsert.append(protein)
            if transcript_id in existing_orfs:
                orfs = existing_orfs[transcript_id]
                for position, start, stop, has_stop_codon in orfs:
                    orf_nt_length = stop - start + 1
                    orf_aa_length = seq_length + int(has_stop_codon)
                    if orf_nt_length == orf_aa_length*3:
                        orfs_to_update.append({
                            'transcript_id': transcript_id,
                            'position': position,
                            'transcript_start': start,
                            'transcript_stop': stop,
                            'protein_id': protein_id
                        })
                        if has_stop_codon:
                            protein['sequence'] = protein['sequence'] + AminoAcid.STOP.value
                        break

            if i % CHUNK_SIZE == 0:
```

```
            bulk_upsert(session, Protein.__table__, proteins_to_upsert)
            bulk_upsert(session, ORF.__table__, orfs_to_update,
primary_keys=('transcript_id', 'position'))
        bulk_upsert(session, Protein.__table__, proteins_to_upsert)
        bulk_upsert(session, ORF.__table__, orfs_to_update, primary_keys=('transcript_id',
'position'))

  def load_sqanti_classifications(self, sqanti_file: str):
    with self.get_session() as session:
      with session.begin():
        existing_gencode_transcripts = {row.accession for row in
session.query(GencodeTranscript.accession)}
        existing_pacbio_transcripts = {row.accession for row in
session.query(PacBioTranscript.accession)}
        transcripts_to_update = []
        with open(sqanti_file) as f:
          f.readline()  # skip header
          lines = count_lines(f)
          reader = csv.DictReader(f, delimiter='\t')
          for row in tqdm(reader, desc='Reading SQANTI classifications', total=lines,
unit='lines'):
            if row['isoform'] in existing_pacbio_transcripts:
              tx = {
                'accession': row['isoform'],
                'sqanti': SQANTI_DICT.get(row['structural_category'], SQANTI.OTHER)
              }
              associated_tx = row['associated_transcript']
              if tx['sqanti'] in {SQANTI.FSM, SQANTI.ISM} and associated_tx in
existing_gencode_transcripts:
                tx['gencode_id'] = associated_tx
              else:
                tx['gencode_id'] = None
              transcripts_to_update.append(tx)
            if len(transcripts_to_update) == CHUNK_SIZE:
              bulk_upsert(session, PacBioTranscript.__table__, transcripts_to_update)
          bulk_upsert(session, PacBioTranscript.__table__, transcripts_to_update)

  def load_domains(self, domain_file: str, overwrite: bool = False):
    # if overwrite:
    #   print(f'Clearing domains from table \'{Feature.__tablename__}\'...')
    #   with self.get_session() as session:
    #     with session.begin():
    #       session.execute(delete(Feature.__table__).where(Feature.type ==
FeatureType.DOMAIN))
```

```
    with open(domain_file) as f:
      lines = count_lines(f)
      reader = csv.reader(f, delimiter='\t')
      t = tqdm(reader, desc='Reading domain info', total=lines, unit='domains')
      domains_to_upsert = (
        {
          'type': FeatureType.DOMAIN,
          'accession': acc,
          'name': name,
          'description': desc
        } for acc, name, _, desc, *_ in t
      )
      domains_to_upsert = list(domains_to_upsert)
    with self.get_session() as session:
      bulk_upsert(session, Domain.__table__, domains_to_upsert)

  def load_patterns(self, pattern_file: str):
    with open(pattern_file) as f:
      name_getter = re.compile(r'(ID   )(\S+)(; PATTERN.)')
      acc_getter = re.compile(r'(AC   )(\S+)(;)')
      desc_getter = re.compile(r'(DE   )(.+)')

      lines = count_lines(f, only=lambda s: name_getter.match(s))
      t = tqdm(None, desc='Reading pattern info', total=lines, unit='patterns')
      patterns_to_upsert = []
      for line in f:
        if (name := name_getter.match(line)):
          pattern = {'name': name.group(2), 'type': FeatureType.NONE}
          t.update()
        elif (acc := acc_getter.match(line)):
          pattern['accession'] = acc.group(2)
        elif (desc := desc_getter.match(line)):
          pattern['description'] = desc.group(2)
          patterns_to_upsert.append(pattern)
    with self.get_session() as session:
      bulk_upsert(session, Feature.__table__, patterns_to_upsert)

  def load_feature_mappings(self, domain_mapping_file: str, appris_only: bool = True,
overwrite: bool = False):
    with self.get_session() as session:
      feature_types = [
        {
          'type': FeatureType.IDR,
```

```
            'accession': 'mobidb-lite',
            'name': 'MobiDB',
            'description': 'intrinsically disordered region (MobiDB)'
          }
        ]
        bulk_upsert(session, Feature.__table__, feature_types)

        with session.begin():
          if overwrite:
            print(f'Clearing table \'{ProteinFeature.__tablename__}\'...')
            session.execute(delete(ProteinFeature.__table__))
          if appris_only:
            protein_length = func.length(Protein.sequence)
            subq = (
              select(GencodeTranscript.gene_id, Protein.accession,
GencodeTranscript.appris, protein_length).
                select_from(Protein).
                join(Protein.orf).
                join(ORF.transcript).
                order_by(GencodeTranscript.gene_id).
                order_by(desc(GencodeTranscript.appris)).
                order_by(desc(protein_length)).
                order_by(GencodeTranscript.name).
                subquery()
            )
            proteins_to_map = set(
              session.execute(
                select(subq.c.accession).select_from(subq).group_by(subq.c.gene_id)
              ).scalars()
            )
          else:
            proteins_to_map = set(session.execute(select(Protein.accession)).scalars())
          existing_features = set(session.execute(select(Feature.accession)).scalars())
        with open(domain_mapping_file) as f:
          f.readline()  # skip header
          lines = count_lines(f)
          reader = csv.DictReader(f, delimiter='\t')
          t = tqdm(reader, desc='Reading reference domain mappings', total=lines,
unit='mappings')
          domains_to_insert = (
            {
              'feature_id': row['feature id'],
              'protein_id': row['Protein stable ID version'],
              'protein_start': row['start'],
```

```
                'protein_stop': row['end'],
                'reference': True
            }
            for row in t if row['feature id'] in existing_features and row['Protein stable ID
version'] in proteins_to_map
        )
        domains_to_insert = list(domains_to_insert)
      with session.begin():
        session.execute(insert(ProteinFeature.__table__).on_conflict_do_nothing(),
domains_to_insert)

  def project_feature_mappings(self, gene_ids: Iterable[str] = None, overwrite: bool =
False):
    with self.get_session() as session:
      # if overwrite:
      #   with session.begin():
      #     print(f'Clearing non-reference mappings from table
\'{ProteinFeature.__tablename__}\'...')
      #
session.execute(delete(ProteinFeature.__table__).where(~ProteinFeature.reference))
      protein_tx = contains_eager(Protein.orf).contains_eager(ORF.transcript)
      q = (
        select(Protein, Transcript.gene_id).
        join(Protein.orf).
        join(ORF.transcript).
        order_by(Transcript.gene_id).
        order_by(desc(Transcript.__table__.c.appris)).
        order_by(desc(ORF.length)).
        order_by(Transcript.name).
        options(
          protein_tx.joinedload(Transcript.orfs),
          protein_tx.joinedload(Transcript.exons).joinedload(Exon.transcript),
          protein_tx.joinedload(Transcript.gene),
          contains_eager(Protein.orf).joinedload(ORF.protein),
          joinedload(Protein.features).joinedload(ProteinFeature.protein),
          joinedload(Protein.features).joinedload(ProteinFeature.feature),
          raiseload('*')
        )
      )
      # tqdm.write(str(q))
      if not gene_ids:
        gene_ids = list(session.execute(
          select(Transcript.gene_id).distinct().
          select_from(ORF).
```

```
            join(ORF.transcript).
            order_by(Transcript.gene_id)
          ).scalars())
        nrows = session.execute(
          select(func.count(Transcript.accession)).
          select_from(ORF).
          join(ORF.transcript).
          where(Transcript.gene_id.in_(gene_ids))
        ).scalars().first()
        rows = chain.from_iterable(
          session.execute(
            q.where(Transcript.gene_id.in_(gene_chunk))
          ).unique()
          for gene_chunk in chunked(gene_ids, 200)
        )
        t = tqdm(None, desc='Projecting domain mappings', total=nrows, unit='proteins',
mininterval=0.2)
        domains_to_insert = []
        for gene_chunk in chunked(gene_ids, 200):
          rows = session.execute(
            q.where(Transcript.gene_id.in_(gene_chunk))
          ).unique()
          rows_by_gene = groupby(rows, key=itemgetter(1))
          for gene, group in rows_by_gene:
            proteins = [row[0] for row in group]
            anchor = proteins[0]
            for other in proteins[1:]:
              if not anchor.features:
                continue
              try:
                aln = CodonAlignment.from_proteins(anchor, other)
              except Exception as e:
                tqdm.write(f'{anchor}|{other}\t{e}')
                continue
              for feat in anchor.features:
                proj_feat = aln.project_feature(feat)
                if proj_feat:
                  record = {
                    k: getattr(proj_feat, k)
                    for k in (
                      'protein_start',
                      'protein_stop',
                      'reference',
                      '_differences'
```

```
                    )
                }
                record['feature_id'] = proj_feat.feature.accession
                record['protein_id'] = other.accession
                record['anchor_id'] = proj_feat.anchor.id
                domains_to_insert.append(record)
            t.update(len(proteins))
        # if len(domains_to_insert) > CHUNK_SIZE:
        if domains_to_insert:
            session.execute(insert(ProteinFeature.__table__).on_conflict_do_nothing(),
domains_to_insert)
            session.commit()
            domains_to_insert[:] = []
    # if domains_to_insert:
    #     session.execute(insert(ProteinFeature.__table__).on_conflict_do_nothing(),
domains_to_insert)
    # session.commit()
    # Alignment.__new__.cache_clear()


def _process_exons(transcripts_to_exons, minus_transcripts):
  # calculate the coordinates of each exon relative to the sequence of its parent transcript
  t = tqdm(
    transcripts_to_exons.items(),
    desc = 'Calculating transcript-relative exon coords',
    total = len(transcripts_to_exons),
    unit = 'transcripts'
  )
  for transcript_id, exon_list in t:
    exon_list.sort(key=itemgetter('start'), reverse=transcript_id in minus_transcripts)
    tx_idx = 1
    for i, exon in enumerate(exon_list, start=1):
      exon['position'] = i
      if 'accession' not in exon:
        exon['accession'] = transcript_id + f':EXON{i}'
      exon_length = exon['stop'] - exon['start'] + 1
      exon['transcript_start'] = tx_idx
      exon['transcript_stop'] = tx_idx + exon_length - 1
      tx_idx += exon_length


def _process_orfs(transcripts_to_cdss, transcripts_to_exons, minus_transcripts):
  # assemble CDS intervals into ORFs
  orfs_to_upsert = []
```

```
  t = tqdm(
    transcripts_to_cdss.items(),
    desc = 'Calculating transcript-relative ORF coords',
    total = len(transcripts_to_cdss),
    unit = 'transcripts'
  )
  for transcript_id, cds_list in t:
    exon_list = transcripts_to_exons[transcript_id]
    cds_list.sort(key=itemgetter(0), reverse=transcript_id in minus_transcripts)
    first_cds, last_cds = cds_list[0], cds_list[-1]
    # assuming that the first and last CDSs are the ORF boundaries -- won't work when
dealing with multiple ORFs
    orf_start = first_cds[0]
    orf_stop = last_cds[1]
    if transcript_id in minus_transcripts:
      orf_start, orf_stop = orf_stop, orf_start
    first_exon = next((exon for exon in exon_list if exon['start'] <= first_cds[0] and first_cds[1]
<= exon['stop']), None)
    last_exon = next((exon for exon in reversed(exon_list) if exon['start'] <= last_cds[0] and
last_cds[1] <= exon['stop']), None)
    # find ORF start/end relative to exons
    if transcript_id not in minus_transcripts:
      first_offset = first_cds[0] - first_exon['start']
      last_offset = last_exon['stop'] - last_cds[1]
    else:
      first_offset = first_exon['stop'] - first_cds[1]
      last_offset = last_cds[0] - last_exon['start']
    # convert to transcript-relative coords
    orf_tx_start = first_exon['transcript_start'] + first_offset
    orf_tx_stop = last_exon['transcript_stop'] - last_offset
    orf_length = orf_tx_stop - orf_tx_start + 1
    if orf_length % 3 != 0:
      continue  # ORFs with nt lengths indivisible by 3 should not be considered
    orfs_to_upsert.append({
      'transcript_id': transcript_id,
      'position': 1,
      'transcript_start': orf_tx_start,
      'transcript_stop': orf_tx_stop,
      'has_stop_codon': False,
    })
  return orfs_to_upsert


# Create databases folder if it doesn't exist
```

```
if not Database._databases_dir.exists():
   Database._databases_dir.mkdir()


# Make sure SQLite enforces foreign key constraints
# https://www.scrygroup.com/tutorial/2018-05-07/SQLite-foreign-keys/
@event.listens_for(Engine, "connect")
def _set_sqlite_pragma(dbapi_connection, connection_record):
   if isinstance(dbapi_connection, SQLite3Connection):
      cursor = dbapi_connection.cursor()
      cursor.execute("PRAGMA foreign_keys=ON;")
      cursor.close()
```

helpers.py

```
# utility functions that don't fit in other modules
import sys
import traceback
from bisect import bisect
from collections.abc import Mapping
from contextlib import AbstractContextManager
from copy import copy
from dataclasses import dataclass, field, fields
from enum import Enum
from itertools import chain, count
from operator import itemgetter
from typing import TYPE_CHECKING, Callable, Generic, Iterable, Iterator, List, Optional,
TextIO, Tuple, TypeVar

from graph_tool import Graph
from intervaltree import Interval, IntervalTree
from sqlalchemy.dialects.sqlite.dml import insert

if TYPE_CHECKING:
   from io import TextIOBase

T = TypeVar('T')


class OrderedEnum(Enum):
   # https://docs.python.org/3/library/enum.html#orderedenum
   def __ge__(self, other):
      if self.__class__ is other.__class__:
         return self.value >= other.value
```

```
      return NotImplemented
    def __gt__(self, other):
      if self.__class__ is other.__class__:
        return self.value > other.value
      return NotImplemented
    def __le__(self, other):
      if self.__class__ is other.__class__:
        return self.value <= other.value
      return NotImplemented
    def __lt__(self, other):
      if self.__class__ is other.__class__:
        return self.value < other.value
      return NotImplemented


class StringEnum(Enum):
    def __str__(self):
      return self.value


class BisectDict(Mapping, Generic[T]):
    def __init__(self, items: Iterable[Tuple[int, T]]):
      self.breakpoints, self._values = zip(*sorted(items, key=itemgetter(0)))

    def __getitem__(self, key: int) -> T:
      if key < 0:
        raise KeyError('Key must be non-negative')
      i = bisect(self.breakpoints, key)
      try:
        return self._values[i]
      except IndexError as e:
        raise KeyError(key) from e

    def __iter__(self) -> Iterator[int]:
      yield from (0,) + self.breakpoints[:-1]

    def __len__(self):
      return len(self.breakpoints)


def frozendataclass(cls):
    frozencls = dataclass(cls, frozen=True)
    field_names = {field.name for field in fields(frozencls)}
    def replace(self, **kwargs):
```

```
      """Return new instance of frozendataclass with updated values."""
      new_field_values = {name: kwargs.get(name, getattr(self, name)) for name in
field_names}
      return frozencls(**new_field_values)
   frozencls.replace = replace
   return frozencls


class ExceptionLogger(AbstractContextManager):
   def __init__(self, info=None, output: TextIO = None, callback=None):
      self.info = info
      self.callback = callback if callable(callback) else None
      self.output = output if output is not None else sys.stderr

   def __exit__(self, exc_type, exc_val, exc_tb):
      if exc_type is not None:
         self.output.write('---------\n')
         if self.info:
            self.output.write(str(self.info) + '\n')
         traceback.print_exc(file=self.output)
         self.output.write('---------\n')
         if self.callback:
            self.callback(exc_type, exc_val, exc_tb)
         return True


def run_length_encode(text: str) -> str:
   if not text:
      return ''
   encoding = []
   run_length = 1
   prev_char = text[0]
   for char in text[1:]:
      if char == prev_char:
         run_length += 1
      else:
         encoding.append(f'{run_length}{prev_char}')
         prev_char = char
         run_length = 1
   encoding.append(f'{run_length}{prev_char}')
   return ','.join(encoding)


def run_length_decode(encoding: str) -> str:
```

```
    return ''.join(int(token[:-1]) * token[-1] for token in encoding.split(',')) if encoding else ''


def get_interval_overlap_graph(intervals: Iterable[Tuple[int, int]], labels: Optional[Iterable]
= None, label_type: str = 'string') -> 'Graph':
    # inspired by https://stackoverflow.com/a/19088519
    # build graph of labels where labels are adjacent if their intervals overlap
    if not labels:
        labels = count()
    g = Graph(directed=False)
    g.vp.label = g.new_vertex_property(label_type)
    label_to_vertex = dict()
    active_labels = set()
    boundaries = sorted(
        chain.from_iterable(
            [(a, True, label), (b, False, label)]
            for (a, b), label in zip(intervals, labels)
        ),
        key = itemgetter(0, 1)
    )
    for _, start_of_interval, label in boundaries:
        if start_of_interval:
            if label not in label_to_vertex:
                v = g.add_vertex()
                g.vp.label[v] = label
                label_to_vertex[label] = v
            for other_label in active_labels:
                i = label_to_vertex[label]
                j = label_to_vertex[other_label]
                g.add_edge(i, j)
            active_labels.add(label)
        else:
            active_labels.discard(label)
    return g, label_to_vertex


# Helper functions/classes for loading into database
@dataclass
class FastaHeaderFields:
    transcript_id: str = None
    protein_id: str = None


def count_lines(file_handle: 'TextIOBase', only: Optional[Callable[..., bool]] = None):
```

```
  lines = sum(1 for _ in filter(only, file_handle))
  file_handle.seek(0)
  return lines


def bulk_upsert(session, table, records, primary_keys=('accession',)):
  if records:
    fields = [field for field in records[0] if field not in primary_keys]
    with session.begin():
      stmt = insert(table)
      session.execute(
        stmt.on_conflict_do_update(
          index_elements = primary_keys,
          set_ = {field: stmt.excluded[field] for field in fields}
        ),
        records
      )
    records[:] = []


def read_gtf_line(line: str) -> list:
  """Read and parse a single gtf line

  Args:
    line (str): unbroken line of a gtf file

  Returns:
    list: gtf attributes
    chromosome : str
    source : str
    feature : str
    start : int
    stop : int
    score : str
    strand : str
    phase : str
    attributes: dict
    tags: list

  """
  chromosome, source, feature, start, stop, score, strand, phase, attributes = line.split('\t')
  start = int(start)
  stop = int(stop)
  attributes = attributes.split(';')[:-1]
```

```
    attributes = [att.strip(' ').split(' ') for att in attributes]
    tags = [att[1].strip('"') for att in attributes if att[0] == 'tag']
    attributes = {att[0]: att[1].strip('"') for att in attributes if att[0] != 'tag'}
    return chromosome, source, feature, start, stop, score, strand, phase, attributes, tags


def get_ids_from_gencode_fasta(header: str):
    fields = [field for field in header.split('|') if field and not field.startswith(('UTR', 'CDS'))]
    transcript_id = next((field for field in fields if field.startswith(('ENST', 'ENSMUST'))), None)
    protein_id = next((field for field in fields if field.startswith(('ENSP', 'ENSMUSP'))), None)
    return FastaHeaderFields(transcript_id, protein_id)

def get_ids_from_pacbio_fasta(header: str):
    return FastaHeaderFields(header, None)

def get_ids_from_lrp_fasta(header: str):
    fields = header.split('|')
    return FastaHeaderFields(fields[1], fields[1] + ':PROT1')

def skip_par_y(header: str):  # these have duplicate ENSEMBL accessions and that makes
SQLAlchemy very sad
    return 'PAR_Y' in header

def skip_gencode(header: str):
    return header.startswith('gc')
```

splice_events.py

```
import heapq
import warnings
from abc import ABC, abstractmethod
from operator import attrgetter, methodcaller
from typing import Iterable, Union

from attrs import evolve, field, frozen
from biosurfer.core.helpers import get_interval_overlap_graph
from biosurfer.core.models.biomolecules import Exon, Transcript
from biosurfer.core.models.nonpersistent import GenomeRange, Position, Junction
from graph_tool import GraphView
from graph_tool.topology import is_bipartite, shortest_path, label_components
from more_itertools import first, windowed
```

```
@frozen(eq=True)
class TranscriptEvent(ABC):
   @abstractmethod
   def __neg__(self) -> 'TranscriptEvent':
      raise NotImplementedError

   @property
   @abstractmethod
   def delta_nt(self) -> int:
      raise NotImplementedError

   @property
   @abstractmethod
   def start(self) -> 'Position':
      raise NotImplementedError

   @property
   @abstractmethod
   def stop(self) -> 'Position':
      raise NotImplementedError


def sort_events(x: Iterable['TranscriptEvent']):
   return tuple(sorted(x, key=attrgetter('start', 'stop')))


@frozen(eq=True)
class BasicTranscriptEvent(TranscriptEvent):
   is_deletion: bool

   def __neg__(self):
      return evolve(self, is_deletion=not self.is_deletion)

   @property
   def is_insertion(self):
      return not self.is_deletion

   @property
   def delta_nt(self) -> int:
      return (-1 if self.is_deletion else 1) * self.length

   @property
   def length(self) -> int:
      return (self.stop - self.start) + 1
```

```
@frozen(eq=True)
class CompoundTranscriptEvent(TranscriptEvent):
  members: tuple['BasicTranscriptEvent', ...] = field(converter=sort_events)

  def __neg__(self):
    return self.from_basic_events(-event for event in self.members)

  @property
  def delta_nt(self) -> int:
    return sum(event.delta_nt for event in self.members)

  @property
  def start(self):
    return self.members[0].start

  @property
  def stop(self):
    return self.members[-1].stop

  @classmethod
  def from_basic_events(cls, basic_events: Iterable['BasicTranscriptEvent']):
    return cls(members=basic_events)


@frozen(eq=True)
class IntronSpliceEvent(BasicTranscriptEvent):
  junction: 'Junction'

  @property
  def anchor_junctions(self):
    return () if self.is_deletion else (self.junction,)

  @property
  def other_junctions(self):
    return (self.junction,) if self.is_deletion else ()

  @property
  def start(self) -> 'Position':
    return self.junction.donor

  @property
  def stop(self) -> 'Position':
```

```
      return self.junction.acceptor


@frozen(eq=True)
class DonorSpliceEvent(BasicTranscriptEvent):
   upstream_junction: 'Junction'
   downstream_junction: 'Junction' = field()
   @downstream_junction.validator
   def _check_junctions(self, attribute, value: 'Junction'):
      if self.upstream_junction.donor >= value.donor:
         raise ValueError(f'{value} has donor upstream of {self.upstream_junction}')

   @property
   def anchor_junctions(self):
      return (self.downstream_junction,) if self.is_deletion else (self.upstream_junction,)

   @property
   def other_junctions(self):
      return (self.upstream_junction,) if self.is_deletion else (self.downstream_junction,)

   @property
   def start(self) -> 'Position':
      return self.upstream_junction.donor

   @property
   def stop(self) -> 'Position':
      return self.downstream_junction.donor - 1


@frozen(eq=True)
class AcceptorSpliceEvent(BasicTranscriptEvent):
   upstream_junction: 'Junction'
   downstream_junction: 'Junction' = field()
   @downstream_junction.validator
   def _check_junctions(self, attribute, value: 'Junction'):
      if self.upstream_junction.acceptor >= value.acceptor:
         raise ValueError(f'{value} has acceptor upstream of {self.upstream_junction}')

   @property
   def anchor_junctions(self):
      return (self.upstream_junction,) if self.is_deletion else (self.downstream_junction,)

   @property
   def other_junctions(self):
```

```
      return (self.downstream_junction,) if self.is_deletion else (self.upstream_junction,)

    @property
    def start(self) -> 'Position':
      return self.upstream_junction.acceptor + 1

    @property
    def stop(self) -> 'Position':
      return self.downstream_junction.acceptor


@frozen(eq=True)
class ExonSpliceEvent(BasicTranscriptEvent):
    skip_junction: 'Junction'
    upstream_junction: 'Junction'
    downstream_junction: 'Junction' = field()
    @downstream_junction.validator
    def _check_short_junctions(self, attribute, value: 'Junction'):
      if self.upstream_junction & value:
        raise ValueError(f'{self.upstream_junction} and {value} overlap')

    @property
    def anchor_junctions(self):
      return (self.upstream_junction, self.downstream_junction) if self.is_deletion else
(self.skip_junction,)

    @property
    def other_junctions(self):
      return (self.skip_junction,) if self.is_deletion else (self.upstream_junction,
self.downstream_junction)

    @property
    def start(self) -> 'Position':
      return self.upstream_junction.acceptor + 1

    @property
    def stop(self) -> 'Position':
      return self.downstream_junction.donor - 1


BasicSpliceEvent = Union[IntronSpliceEvent, DonorSpliceEvent, AcceptorSpliceEvent,
ExonSpliceEvent]

@frozen(eq=True)
```

```
class ExonBypassEvent(BasicTranscriptEvent):
  exon: 'GenomeRange'  # TODO: replace with updated Exon object
  is_partial: bool = False

  @property
  def start(self) -> 'Position':
    return self.exon.begin

  @property
  def stop(self) -> 'Position':
    return self.exon.end


EVENT_CODES = {
  IntronSpliceEvent: ('I', 'i'),
  DonorSpliceEvent: ('D', 'd'),
  AcceptorSpliceEvent: ('A', 'a'),
  ExonSpliceEvent: ('E', 'e'),
  ExonBypassEvent: ('B', 'b')
}


def get_event_code(events: Iterable['BasicTranscriptEvent']):
  # return ''.join(EVENT_CODES[type(event)][event.is_deletion] for event in events)
  code = ''
  for event in events:
    if getattr(event, 'is_partial', False):
      code += 'p' if event.is_deletion else 'P'
    else:
      code += EVENT_CODES[type(event)][event.is_deletion]
  return code


@frozen(eq=True)
class SpliceEvent(CompoundTranscriptEvent):
  members: tuple['BasicSpliceEvent', ...] = field(converter=sort_events, repr=False)
  code: str = field(default='')
  anchor_junctions: tuple['Junction', ...] = field(factory=tuple)
  other_junctions: tuple['Junction', ...] = field(factory=tuple)

  @members.validator
  def _check_members(self, attribute, value):
    if len(value) > 1:
      if any(isinstance(event, IntronSpliceEvent) for event in value):
```

```
            raise ValueError('Cannot combine IntronSpliceEvent with other BasicSpliceEvents')
        if any(isinstance(event, DonorSpliceEvent) for event in value[1:]):
            raise ValueError('DonorSpliceEvent must be first')
        if any(isinstance(event, AcceptorSpliceEvent) for event in value[:-1]):
            raise ValueError('AcceptorSpliceEvent must be last')

    @classmethod
    def from_basic_events(cls, events: Iterable['BasicSpliceEvent']):
        events = tuple(events)
        code = get_event_code(events)
        anchor_junctions = sorted({junc for event in events for junc in event.anchor_junctions},
key=attrgetter('donor'))
        other_junctions = sorted({junc for event in events for junc in event.other_junctions},
key=attrgetter('donor'))
        return cls(members=events, code=code, anchor_junctions=tuple(anchor_junctions),
other_junctions=tuple(other_junctions))


@frozen(eq=True)
class TSSEvent(CompoundTranscriptEvent):
    members: tuple['ExonBypassEvent', ...] = field(converter=sort_events)

    @members.validator
    def _check_members(self, attribute, value: tuple['ExonBypassEvent', ...]):
        if any(event.is_partial for event in value[:-1]):
            raise ValueError(f'{value}')
        if len({event.is_deletion for event in value[:-1]}) > 1:
            raise ValueError(f'{value}')


@frozen(eq=True)
class APAEvent(CompoundTranscriptEvent):
    members: tuple['ExonBypassEvent', ...] = field(converter=sort_events)

    @members.validator
    def _check_members(self, attribute, value: tuple['ExonBypassEvent', ...]):
        if any(event.is_partial for event in value[1:]):
            raise ValueError(f'{value}')
        if len({event.is_deletion for event in value[1:]}) > 1:
            raise ValueError(f'{value}')


def call_splice_event(comp: 'GraphView') -> 'SpliceEvent':
    def by_donor(v):
```

```
      junc = comp.vp.label[v]
      return junc.donor, junc.acceptor

    def by_acceptor(v):
      junc = comp.vp.label[v]
      return junc.acceptor, junc.donor

    basic_events = []
    N = comp.num_vertices()
    if N == 1:
      # call alt. intron event
      v = first(comp.vertices())
      if not (comp.vp.overlaps_tss[v] or comp.vp.overlaps_pas[v]):
        basic_events = [IntronSpliceEvent(is_deletion=not comp.vp.from_anchor[v],
junction=comp.vp.label[v])]
    else:
      v0, v1 = heapq.nsmallest(2, comp.vertices(), key=by_donor)
      vN, vM = heapq.nlargest(2, comp.vertices(), key=by_acceptor)
      # check for alt. donor usage
      junc0 = comp.vp.label[v0]
      junc1 = comp.vp.label[v1]
      if junc0.donor != junc1.donor and not comp.vp.overlaps_tss[v0]:
        donor_event = DonorSpliceEvent(
          is_deletion = bool(comp.vp.from_anchor[v1]),
          upstream_junction = junc0,
          downstream_junction = junc1
        )
        donor_event = [donor_event]
      else:
        donor_event = []
      # check for alt. acceptor usage
      juncM = comp.vp.label[vM]
      juncN = comp.vp.label[vN]
      if juncM.acceptor != juncN.acceptor and not comp.vp.overlaps_pas[vN]:
        acceptor_event = AcceptorSpliceEvent(
          is_deletion = bool(comp.vp.from_anchor[vM]),
          upstream_junction = juncM,
          downstream_junction = juncN
        )
        acceptor_event = [acceptor_event]
      else:
        acceptor_event = []
      # call any alt. exon events
      exon_events = []
```

```
      if N > 2:
        path, _ = shortest_path(
          comp,
          source = min(v0, v1, key=methodcaller('out_degree')),
          target = min(vM, vN, key=methodcaller('out_degree'))
        )
        for skip in path[1:-1]:
          neighbors = sorted(skip.out_neighbors(), key=by_donor)
          other_has_skip = not comp.vp.from_anchor[skip]
          for upstream, downstream in windowed(neighbors, 2):
            exon_event = ExonSpliceEvent(
              is_deletion = other_has_skip,
              upstream_junction = comp.vp.label[upstream],
              downstream_junction = comp.vp.label[downstream],
              skip_junction = comp.vp.label[skip]
            )
            exon_events.append(exon_event)
      basic_events = donor_event + exon_events + acceptor_event
    return SpliceEvent.from_basic_events(basic_events) if basic_events else None


def call_transcript_events(anchor: 'Transcript', other: 'Transcript'):
    chr = anchor.gene.chromosome_id
    strand = anchor.strand
    anchor_start = Position(chr, strand, anchor.start)
    anchor_stop = Position(chr, strand, anchor.stop)
    if anchor_start > anchor_stop:
      anchor_start, anchor_stop = anchor_stop, anchor_start
    other_start = Position(chr, strand, other.start)
    other_stop = Position(chr, strand, other.stop)
    if other_start > other_stop:
      other_start, other_stop = other_stop, other_start
    downstream_start = max(anchor_start, other_start)
    upstream_stop = min(anchor_stop, other_stop)

    anchor_junctions = set(anchor.junctions)
    diff_junctions = anchor_junctions ^ set(other.junctions)
    diff_junctions = {junc for junc in diff_junctions if downstream_start <= junc.acceptor and
junc.donor <= upstream_stop}
    tss_overlap_junction = first((junc for junc in diff_junctions if junc.donor <=
downstream_start <= junc.acceptor + 1), None)
    pas_overlap_junction = first((junc for junc in diff_junctions if junc.donor - 1 <=
upstream_stop <= junc.acceptor), None)
```

```
    g, _ = get_interval_overlap_graph(((j.donor, j.acceptor+1) for j in diff_junctions),
diff_junctions, label_type='object')
    if not is_bipartite(g):
        warnings.warn(f'Overlap graph not bipartite for {anchor} | {other}')
    g.vp.from_anchor = g.new_vertex_property('bool')
    g.vp.overlaps_tss = g.new_vertex_property('bool')
    g.vp.overlaps_pas = g.new_vertex_property('bool')
    for v in g.vertices():
        junc = g.vp.label[v]
        g.vp.from_anchor[v] = junc in anchor_junctions
        g.vp.overlaps_tss[v] = junc == tss_overlap_junction
        g.vp.overlaps_pas[v] = junc == pas_overlap_junction

    g.vp.comp, hist = label_components(g)
    components = {c: GraphView(g, vfilt=lambda v: g.vp.comp[v] == c) for c in range(len(hist))}

    with warnings.catch_warnings():
        warnings.simplefilter('ignore')
        splice_events = sorted(
            filter(None, (call_splice_event(comp) for comp in components.values())),
            key = lambda e: min(j.donor for j in (e.anchor_junctions + e.other_junctions))
        )

    # TODO: simplify this when Exons are refactored
    def get_exon(exon_obj: 'Exon'):
        return GenomeRange(*sorted((Position(chr, strand, exon_obj.start), Position(chr,
strand, exon_obj.stop))))

    anchor_exons = [get_exon(exon) for exon in anchor.exons]
    other_exons = [get_exon(exon) for exon in other.exons]
    upstream_exons = sorted((exon for exon in set(anchor_exons) | set(other_exons) if
exon.begin < downstream_start), key=attrgetter('begin'))
    downstream_exons = sorted((exon for exon in set(anchor_exons) | set(other_exons) if
upstream_stop < exon.end), key=attrgetter('begin'))
    # call TSS event
    if upstream_exons:
        is_deletion = downstream_start == other_start
        bypass_events = [ExonBypassEvent(is_deletion, exon) for exon in upstream_exons]
        if tss_overlap_junction:
            alt_downstream_exon = (other_exons if is_deletion else anchor_exons)[0]
            last_bypass_event = ExonBypassEvent(
                not is_deletion,
                GenomeRange(
                    downstream_start,
```

```
            min(alt_downstream_exon.end, tss_overlap_junction.acceptor)
          ),
          is_partial = tss_overlap_junction.acceptor < alt_downstream_exon.end
        )
        bypass_events.append(last_bypass_event)
      elif any(exon.begin < downstream_start <= exon.end for exon in (anchor_exons if
is_deletion else other_exons)):
        upstream_exons[-1] = evolve(upstream_exons[-1], end=downstream_start-1)
        bypass_events[-1] = evolve(bypass_events[-1], exon=upstream_exons[-1],
is_partial=True)
      tss_event = TSSEvent.from_basic_events(bypass_events)
    else:
      tss_event = None
    # call APA event
    if downstream_exons:
      is_deletion = upstream_stop == other_stop
      bypass_events = [ExonBypassEvent(is_deletion, exon) for exon in downstream_exons]
      if pas_overlap_junction:
        alt_upstream_exon = (other_exons if is_deletion else anchor_exons)[-1]
        first_bypass_event = ExonBypassEvent(
          not is_deletion,
          GenomeRange(
            max(alt_upstream_exon.begin, pas_overlap_junction.donor),
            upstream_stop
          ),
          is_partial = alt_upstream_exon.begin < pas_overlap_junction.donor
        )
        bypass_events.insert(0, first_bypass_event)
      elif any(exon.begin <= upstream_stop < exon.end for exon in (anchor_exons if
is_deletion else other_exons)):
        downstream_exons[0] = evolve(downstream_exons[0], begin=upstream_stop+1)
        bypass_events[0] = evolve(bypass_events[0], exon=downstream_exons[0],
is_partial=True)
      apa_event = APAEvent.from_basic_events(bypass_events)
    else:
      apa_event = None
    return splice_events, tss_event, apa_event
```

base.py

```
from typing import Iterable, Type
from sqlalchemy import Column, String, select
from sqlalchemy.ext.declarative import declarative_base, declared_attr
from sqlalchemy.orm.exc import NoResultFound

Base = declarative_base()


class TablenameMixin:
    @declared_attr
    def __tablename__(cls: Type['Base']):
        return cls.__name__.lower()


class NameMixin:
    name = Column(String, index=True)

    @classmethod
    def from_name(cls: Type['Base'], session, name: str, unique: bool = True):
        statement = select(cls).where(cls.name == name)
        result = session.execute(statement).scalars()
        if unique:
            try:
                return result.one()
            except NoResultFound:
                return None
        else:
            return result.all()

    @classmethod
    def from_names(cls: Type['Base'], session, names: Iterable[str]):
        statement = select(cls).where(cls.name.in_(names))
        return {inst.name: inst for inst in session.execute(statement).scalars()}


class AccessionMixin:
    accession = Column(String, primary_key=True, index=True)

    @classmethod
    def from_accession(cls: Type['Base'], session, accession: str):
        statement = select(cls).where(cls.accession == accession)
        result = session.execute(statement).scalars()
```

```
        try:
            return result.one()
        except NoResultFound:
            return None

    @classmethod
    def from_accessions(cls: Type['Base'], session, accessions: Iterable[str]):
        statement = select(cls).where(cls.accession.in_(accessions))
        return {inst.name: inst for inst in session.execute(statement).scalars()}
```

biomolecules.py

```
from functools import cached_property
from operator import attrgetter
from typing import Dict, Iterable, List, Optional, Tuple, Type
from warnings import warn

from Bio.Seq import Seq
from biosurfer.core.constants import APPRIS, SQANTI, Strand
from biosurfer.core.helpers import BisectDict
from biosurfer.core.models.base import AccessionMixin, Base, NameMixin,
TablenameMixin
from biosurfer.core.models.nonpersistent import *
from more_itertools import only
from sqlalchemy import Boolean, Column, Enum, ForeignKey, Integer, String, func
from sqlalchemy.ext.hybrid import hybrid_property
from sqlalchemy.ext.orderinglist import ordering_list
from sqlalchemy.orm import relationship


# TODO: replace with Enum?
class Chromosome(Base, TablenameMixin, NameMixin):
    name = Column(String, primary_key=True)
    genes = relationship('Gene', back_populates='chromosome')

    def __repr__(self) -> str:
        return self.name


class Gene(Base, TablenameMixin, NameMixin, AccessionMixin):
    strand = Column(Enum(Strand))
    chromosome_id = Column(String, ForeignKey('chromosome.name'))
    chromosome = relationship('Chromosome', back_populates='genes')
```

```
    transcripts = relationship(
        'Transcript',
        back_populates = 'gene',
        order_by = 'Transcript.name',
        lazy = 'selectin'  # always load transcripts along with gene
    )

    def __repr__(self) -> str:
        return self.name

    # FIXME: make this work in SQL queries
    @property
    def start(self) -> int:
        return min(exon.start for transcript in self.transcripts for exon in transcript.exons)

    @property
    def stop(self) -> int:
        return max(exon.stop for transcript in self.transcripts for exon in transcript.exons)



class Transcript(Base, TablenameMixin, NameMixin, AccessionMixin):
    strand = Column(Enum(Strand))
    type = Column(String)
    sequence = Column(String)
    gene_id = Column(String, ForeignKey('gene.accession'))
    gene = relationship('Gene', back_populates='transcripts')
    exons = relationship(
        'Exon',
        order_by = 'Exon.transcript_start',
        collection_class = ordering_list('position', count_from=1),
        back_populates = 'transcript',
        uselist = True,
        lazy = 'selectin'  # always load exons along with transcript
    )
    orfs = relationship(
        'ORF',
        order_by = 'ORF.transcript_start',
        back_populates = 'transcript',
        uselist = True
    )

    __mapper_args__ = {
        'polymorphic_on': type,
```

```
    'polymorphic_identity': 'transcript'
  }

  # The reason we use cached properties here instead of setting things up in __init__ or
init_on_load
  # is to make sure the ORM eagerly loads all exons first.
  @cached_property
  def _exon_mapping(self) -> BisectDict:
    return BisectDict((exon.transcript_stop+1, i) for i, exon in enumerate(self.exons))

  @cached_property
  def _junction_mapping(self) -> Dict['Junction', Tuple['Exon', 'Exon']]:
    mapping = dict()
    for i in range(1, len(self.exons)):
      up_exon = self.exons[i-1]
      down_exon = self.exons[i]
      chr = self.gene.chromosome_id
      if self.strand is Strand.MINUS:
        up_exon_stop = up_exon.start
        down_exon_start = down_exon.stop
      else:
        up_exon_stop = up_exon.stop
        down_exon_start = down_exon.start
      donor = Position(chr, self.strand, up_exon_stop) + 1
      acceptor = Position(chr, self.strand, down_exon_start) - 1
      junction = Junction.from_splice_sites(donor, acceptor)
      mapping[junction] = (up_exon, down_exon)
    return mapping

  @cached_property
  def nucleotides(self):
    if not self.sequence:
      raise AttributeError(f'{self.name} has no sequence')
    nucleotides = []
    self._nucleotide_mapping: Dict[int, 'Nucleotide'] = dict()
    if self.exons:
      i = 0
      for exon in self.exons:
        coords = range(exon.start, exon.stop+1)
        if self.strand is Strand.MINUS:
          coords = reversed(coords)
        for coord in coords:
          nt = Nucleotide(self, coord, i+1)
          nucleotides.append(nt)
```

```
            self._nucleotide_mapping[coord] = nt
            i += 1
    else:
        nucleotides = [Nucleotide(self, None, i+1) for i in range(len(self.sequence))]
    return nucleotides

def __repr__(self) -> str:
    return self.name

# FIXME: make this work in SQL queries
@property
def start(self) -> int:
    return self.exons[0].stop if self.strand is Strand.MINUS else self.exons[0].start

@property
def stop(self) -> int:
    return self.exons[-1].start if self.strand is Strand.MINUS else self.exons[-1].stop

@hybrid_property
def length(self):
    return len(self.sequence)

@length.expression
def length(cls):
    return func.length(cls.sequence)

@property
def chromosome(self) -> 'Chromosome':
    return self.gene.chromosome

@property
def primary_orf(self) -> Optional['ORF']:
    if not self.orfs:
        return None
    return max(self.orfs, key=attrgetter('length'))

@property
def protein(self) -> Optional['Protein']:
    """Get the "primary" protein produced by this transcript, if it exists."""
    return self.primary_orf.protein if self.primary_orf else None

@property
def junctions(self):
    return list(self._junction_mapping.keys())
```

```
  # These methods may seem redundant, but the idea is to keep the publicly accessible interface separate from the implementation details
  def get_exon_containing_position(self, position: int) -> 'Exon':
    """Given a position (1-based) within the transcript's nucleotide sequence, return the exon containing that position."""
    return self.exons[self._exon_mapping[position]]

  def get_exon_index_containing_position(self, position: int) -> int:
    """Given a position (1-based) within the transcript's nucleotide sequence, return the index (0-based) of the exon containing that position."""
    return self._exon_mapping[position]

  def get_nucleotide_from_coordinate(self, coordinate: int) -> 'Nucleotide':
    """Given a genomic coordinate (1-based) included in the transcript, return the Nucleotide object corresponding to that coordinate."""
    if not hasattr(self, '_nucleotide_mapping'):
      self.nucleotides
    if coordinate in self._nucleotide_mapping:
      return self._nucleotide_mapping[coordinate]
    else:
      return None

  def contains_coordinate(self, coordinate: int) -> bool:
    """Given a genomic coordinate (1-based), return whether or not the transcript contains the nucleotide at that coordinate."""
    return coordinate in self._nucleotide_mapping

  def get_exons_from_junction(self, junction: 'Junction') -> Tuple['Exon', 'Exon']:
    try:
      return self._junction_mapping[junction]
    except KeyError as e:
      raise KeyError(f'{self} does not use junction {junction}') from e

  def get_genome_coord_from_transcript_coord(self, tx_coord: int) -> Position:
    try:
      nt = self.nucleotides[tx_coord]
      return Position(self.gene.chromosome_id, self.strand, nt.coordinate)
    except IndexError:
      pos = Position(self.gene.chromosome_id, self.strand, self.nucleotides[-1].coordinate)
      return pos + (tx_coord - self.length + 1)

  def get_transcript_coord_from_genome_coord(self, gn_coord: Position) -> int:
    if self.strand is not gn_coord.strand:
```

```
            raise ValueError(f'{gn_coord} is different strand from {self}')
        elif self.gene.chromosome_id != gn_coord.chromosome:
            raise ValueError(f'{gn_coord} is different chromosome from {self}')
        nt = self.get_nucleotide_from_coordinate(gn_coord.coordinate)
        if nt:
            return nt.position - 1
        else:
            raise KeyError


class GencodeTranscript(Transcript):
    __tablename__ = None
    appris = Column(Enum(APPRIS, values_callable=lambda x: [str(m.value) for m in x]))
    start_nf = Column(Boolean)
    end_nf = Column(Boolean)
    pacbio = relationship(
        'PacBioTranscript',
        back_populates = 'gencode',
        uselist = True
    )

    def __init__(self, **kwargs):
        if 'strand' in kwargs:
            kwargs['strand'] = Strand.from_symbol(kwargs['strand'])
        super().__init__(**kwargs)

    __mapper_args__ = {
        'polymorphic_identity': 'gencodetranscript'
    }

    @hybrid_property
    def basic(self):
        return ~(self.start_nf | self.end_nf)


class PacBioTranscript(Transcript):
    __tablename__ = None
    sqanti = Column(Enum(SQANTI, values_callable=lambda x: [str(m.value) for m in x]))
    gencode_id = Column(String, ForeignKey('transcript.accession'))
    gencode = relationship(
        'GencodeTranscript',
        back_populates = 'pacbio',
        uselist = False,
        remote_side = [Transcript.accession]
```

```
    )

    __mapper_args__ = {
        'polymorphic_identity': 'pacbiotranscript'
    }


class Exon(Base, TablenameMixin, AccessionMixin):
    # type = Column(String)
    position = Column(Integer)  # exon ordinal within parent transcript
    # genomic coordinates
    start = Column(Integer)
    stop = Column(Integer)
    # transcript coordinates
    transcript_start = Column(Integer)
    transcript_stop = Column(Integer)
    # sequence = Column(String, default='')
    transcript_id = Column(String, ForeignKey('transcript.accession'), primary_key=True)
    transcript = relationship(
        'Transcript',
        back_populates = 'exons'
    )

    def __repr__(self) -> str:
        return f'{self.transcript}:exon{self.position}'

    @hybrid_property
    def length(self):
        return self.stop - self.start + 1

    @property
    def gene(self):
        return self.transcript.gene

    @property
    def chromosome(self):
        return self.gene.chromosome

    @property
    def strand(self) -> 'Strand':
        return self.transcript.strand

    @property
    def sequence(self):
```

```
        return self.transcript.sequence[self.transcript_start-1:self.transcript_stop]

    @property
    def nucleotides(self):
        return self.transcript.nucleotides[self.transcript_start-1:self.transcript_stop]

    @property
    def coding_nucleotides(self):
        return [nt for nt in self.nucleotides if nt.residue]


class ORF(Base, TablenameMixin):
    # genomic coordinates
    # start = Column(Integer)
    # stop = Column(Integer)
    # transcript coordinates
    transcript_start = Column(Integer)
    transcript_stop = Column(Integer)
    has_stop_codon = Column(Boolean)
    position = Column(Integer, primary_key=True)
    transcript_id = Column(String, ForeignKey('transcript.accession'), primary_key=True)
    transcript = relationship(
        'Transcript',
        back_populates = 'orfs',
        lazy = 'joined'
    )
    protein_id = Column(String, ForeignKey('protein.accession'))
    protein = relationship(
        'Protein',
        back_populates = 'orf',
        uselist = False,
        lazy = 'joined'
    )

    @property
    def _first_exon_index(self):
        return self.transcript.get_exon_index_containing_position(self.transcript_start)

    @property
    def _last_exon_index(self):
        try:
            return self.transcript.get_exon_index_containing_position(self.transcript_stop)
        except KeyError as e:
            warn(
```

```
            f'KeyError: {e} when getting ORF._last_exon_index for {self}'
          )
          return len(self.transcript.exons) - 1

    @cached_property
    def utr5(self):
      utr5_boundary_exon_index = self._first_exon_index
      if self.transcript.exons[utr5_boundary_exon_index].transcript_start ==
self.transcript_start:
        utr5_boundary_exon_index -= 1
      if utr5_boundary_exon_index >= 0:
        return FivePrimeUTR(self, utr5_boundary_exon_index)
      else:
        return None

    @cached_property
    def utr3(self):
      utr3_boundary_exon_index = self._last_exon_index
      if self.transcript.exons[utr3_boundary_exon_index].transcript_stop ==
self.transcript_stop:
        utr3_boundary_exon_index += 1
      if utr3_boundary_exon_index < len(self.transcript.exons):
        return ThreePrimeUTR(self, utr3_boundary_exon_index)
      else:
        return None

    @cached_property
    def nmd(self):
      # ORFs with stop codons at least 50 bp upstream of the last splice site in the mature
transcript
      # (i.e. the beginning of the last exon) are considered candidates for nonsense-mediated
decay (NMD)
      last_junction = self.transcript.exons[-1].transcript_start
      return last_junction - self.transcript_stop >= 50

    def __repr__(self) -> str:
      return f'{self.transcript}:orf({self.transcript_start}-{self.transcript_stop})'

    # FIXME: make this work in SQL queries
    @property
    def start(self):
      if self.transcript.strand is Strand.PLUS:
        return self.nucleotides[0].coordinate
      elif self.transcript.strand is Strand.MINUS:
```

```
      return self.nucleotides[-1].coordinate

@property
def stop(self):
   if self.transcript.strand is Strand.PLUS:
      return self.nucleotides[-1].coordinate
   elif self.transcript.strand is Strand.MINUS:
      return self.nucleotides[0].coordinate

@hybrid_property
def length(self) -> int:
   return self.transcript_stop - self.transcript_start + 1

@property
def sequence(self) -> str:
   return self.transcript.sequence[self.transcript_start - 1:self.transcript_stop]

@property
def nucleotides(self) -> List['Nucleotide']:
   return self.transcript.nucleotides[self.transcript_start - 1:self.transcript_stop]

@property
def gene(self) -> 'Gene':
   return self.transcript.gene

@property
def exons(self) -> List['Exon']:
   return self.transcript.exons[self._first_exon_index:self._last_exon_index+1]

@property
def junctions(self) -> List['Junction']:
   return self.transcript.junctions[self._first_exon_index:self._last_exon_index]

def _link_aa_to_nt(self, residue_list):
   aa_sequence = Seq(self.protein.sequence)
   nt_sequence = Seq(self.sequence)
   translation = nt_sequence.translate(to_stop=True)
   aa_match_index = aa_sequence.find(translation)
   if aa_match_index == -1:
      warn(
         f'Could not match amino acid sequence to nucleotide sequence of {self}'
      )
      return
```

```
        nt_match_index = aa_match_index*3
        nt_list = self.nucleotides[nt_match_index:]
        for i, aa in enumerate(residue_list):
            aa.codon = tuple(nt_list[3*i:3*i + 3])
            for nt in aa.codon:
                nt.residue = aa


class Protein(Base, TablenameMixin, AccessionMixin):
    sequence = Column(String)
    orf = relationship(
        'ORF',
        back_populates = 'protein',
        uselist = False,
        lazy = 'joined'
    )
    features = relationship(
        'ProteinFeature',
        back_populates = 'protein',
        uselist = True,
        order_by = 'ProteinFeature.protein_start'
    )

    # def __init__(self, **kwargs):
    #     super().__init__(**kwargs)
    #     self.residues = []

    # @reconstructor
    # def init_on_load(self):
    #     self.residues = [Residue(self, aa, i) for i, aa in enumerate(self.sequence + '*', start=1)]
    #     if self.orf and self.orf.nucleotides:
    #         self._link_aa_to_orf_nt()

    @cached_property
    def residues(self):
        # if not self.sequence.endswith('*'):
        #     self.sequence += '*'
        _residues = [Residue(self, i+1) for i in range(len(self.sequence))]
        if self.orf:
            self.orf._link_aa_to_nt(_residues)
        return _residues

    def __repr__(self):
        if self.orf:
```

```
            return f'{self.orf.transcript}:protein'
        else:
            return self.accession

    @property
    def gene(self):
        return self.orf.transcript.gene

    @property
    def transcript(self) -> 'Transcript':
        return self.orf.transcript

    @hybrid_property
    def length(self):
        return len(self.sequence)

    def get_protein_coord_from_transcript_coord(self, transcript_coord: int):
        return (transcript_coord - self.orf.transcript_start + 1) // 3

    def get_transcript_coord_from_protein_coord(self, protein_coord: int):
        return protein_coord*3 + self.orf.transcript_start – 1
```

features.py

```
import json
from functools import cached_property
from typing import TYPE_CHECKING, List

from biosurfer.core.constants import FeatureType
from biosurfer.core.constants import \
    CodonAlignmentCategory as TranscriptAlignCat
from biosurfer.core.helpers import run_length_decode
from biosurfer.core.models.base import AccessionMixin, Base, NameMixin,
TablenameMixin
from sqlalchemy.ext.hybrid import hybrid_property
from sqlalchemy.orm import relationship
from sqlalchemy.sql.schema import Column, ForeignKey, Table, UniqueConstraint
from sqlalchemy.sql.sqltypes import Boolean, Enum, Integer, PickleType, String

if TYPE_CHECKING:
    from biosurfer.core.alignments import FeatureAlignment
    from biosurfer.core.models.biomolecules import Residue

# feature_base_table = Table(
```

```
#    'proteinfeature', Base.metadata,
#    Column('type', Enum(FeatureType)),
#    Column('accession', String, primary_key=True, index=True),
#    Column('name', String),
#    Column('description', String)
# )


# feature_mapping_table = Table(
#    'proteinfeature_mapping', Base.metadata,
#    Column('feature_id', String, ForeignKey('proteinfeature.accession'), primary_key=True),
#    Column('protein_id', String, ForeignKey('protein.accession'), primary_key=True),
#    Column('protein_start', Integer, primary_key=True),
#    Column('protein_stop', Integer, primary_key=True),
# )


class Feature(Base, TablenameMixin, NameMixin, AccessionMixin):
  type = Column(Enum(FeatureType))
  description = Column(String)

  __mapper_args__ = {
    'polymorphic_on': type,
    'polymorphic_identity': FeatureType.NONE
  }


class Domain(Feature):
  __tablename__ = None
  __mapper_args__ = {
    'polymorphic_identity': FeatureType.DOMAIN
  }


class IDR(Feature):
  __tablename__ = None
  __mapper_args__ = {
    'polymorphic_identity': FeatureType.IDR
  }


class ProteinFeature(Base, TablenameMixin):
  id = Column(Integer, primary_key=True, autoincrement=True)
  feature_id = Column(String, ForeignKey('feature.accession'), nullable=False)
```

```
  protein_id = Column(String, ForeignKey('protein.accession'), nullable=False)
  protein_start = Column(Integer, nullable=False)
  protein_stop = Column(Integer, nullable=False)
  reference = Column(Boolean, nullable=False)

  feature = relationship('Feature', uselist=False, lazy='selectin')
  protein = relationship('Protein', back_populates='features', uselist=False)

  __table_args__ = (UniqueConstraint(feature_id, protein_id, protein_start, protein_stop),)

  __mapper_args__ = {
    'polymorphic_on': reference,
    'polymorphic_identity': True
  }

  def __repr__(self):
    return f'{self.protein}:{self.name}({self.protein_start}-{self.protein_stop})'

  @property
  def type(self) -> FeatureType:
    return self.feature.type if self.feature else FeatureType.NONE

  @property
  def name(self) -> str:
    return self.feature.name if self.feature else None

  @property
  def description(self) -> str:
    return self.feature.description if self.feature else None

  @hybrid_property
  def length(self):
    return self.protein_stop - self.protein_start + 1

  @property
  def sequence(self) -> str:
    return self.protein.sequence[self.protein_start-1:self.protein_stop]

  @property
  def residues(self) -> List['Residue']:
    return self.protein.residues[self.protein_start-1:self.protein_stop]


class ProjectedFeature(ProteinFeature, TablenameMixin):
```

```
    __tablename__ = None
    anchor_id = Column(Integer, ForeignKey('proteinfeature.id'))
    anchor = relationship('ProteinFeature', foreign_keys=[anchor_id], uselist=False)
    _differences = Column(String)  # run-length encoding of FeatureAlignment with anchor feature

    __mapper_args__ = {
        'polymorphic_identity': False
    }

    @cached_property
    def altered_residues(self):
        alt_res = []
        i = 0
        for token in self._differences.split(','):
            char = token[-1]
            length = int(token[:-1])
            if char != TranscriptAlignCat.MATCH.value:
                alt_res.extend(self.residues[i:i+length])
            i += length
        return alt_res

    # def __init__(self, feature_alignment: 'FeatureAlignment'):
    #     self.alignment = feature_alignment
    #     self.anchor = feature_alignment.proteinfeature
    #     self.feature = feature_alignment.proteinfeature.feature
    #     self.protein = feature_alignment.other

    #     residues = feature_alignment.other_residues
    #     self.protein_start = residues[0].position
    #     self.protein_stop = residues[-1].position

    #     self.altered_residues = [res_aln.other for res_aln in feature_alignment if res_aln.category not in {TranscriptAlignCat.MATCH, TranscriptAlignCat.EDGE_MATCH, TranscriptAlignCat.DELETION}]

    def __repr__(self):
        return super().__repr__() + '*'
```

nonpersistent.py

```
from abc import ABC, abstractmethod
from collections import Counter
from functools import cached_property
from operator import attrgetter
from typing import TYPE_CHECKING, List, Optional, Tuple, Union
from warnings import warn

from attrs import define, frozen, field, evolve, validators
from biosurfer.core.constants import Nucleobase, AminoAcid, Strand

if TYPE_CHECKING:
  from biosurfer.core.models.biomolecules import Chromosome, Gene, Transcript, Exon,
ORF, Protein


@frozen(hash=True)
class Position:
  chromosome: str  # TODO: convert to dynamic Enum?
  strand: 'Strand' = field(validator=validators.instance_of(Strand))
  coordinate: int = field(validator=validators.gt(0))

  def __repr__(self):
    return f'{self.chromosome}({self.strand}):{self.coordinate}'

  def _is_comparable(self, other: 'Position'):
    return (self.chromosome, self.strand) == (other.chromosome, other.strand)

  def __lt__(self, other: 'Position'):
    if not isinstance(other, Position):
      return NotImplemented
    if not self._is_comparable(other):
      raise ValueError(f'{self} and {other} are from different strands')
    elif self.strand is Strand.MINUS:
      return self.coordinate > other.coordinate
    else:
      return self.coordinate < other.coordinate

  def __le__(self, other: 'Position'):
    if not isinstance(other, Position):
      return NotImplemented
    if not self._is_comparable(other):
      raise ValueError(f'{self} and {other} are from different strands')
```

```
    elif self.strand is Strand.MINUS:
        return self.coordinate >= other.coordinate
    else:
        return self.coordinate <= other.coordinate

def __gt__(self, other: 'Position'):
    if not isinstance(other, Position):
        return NotImplemented
    if not self._is_comparable(other):
        raise ValueError(f'{self} and {other} are from different strands')
    elif self.strand is Strand.MINUS:
        return self.coordinate < other.coordinate
    else:
        return self.coordinate > other.coordinate

def __ge__(self, other: 'Position'):
    if not isinstance(other, Position):
        return NotImplemented
    if not self._is_comparable(other):
        raise ValueError(f'{self} and {other} are from different strands')
    elif self.strand is Strand.MINUS:
        return self.coordinate <= other.coordinate
    else:
        return self.coordinate >= other.coordinate

def __add__(self, offset: int):
    if not isinstance(offset, int):
        return NotImplemented
    else:
        if self.strand is Strand.MINUS:
            offset = -offset
        return evolve(self, coordinate=self.coordinate + offset)

def __radd__(self, offset: int):
    return self.__add__(offset)

def __sub__(self, other: Union['Position', int]):
    if isinstance(other, int):
        return self.__add__(-other)
    elif not isinstance(other, Position):
        return NotImplemented
    elif not self._is_comparable(other):
        raise ValueError(f'{self} and {other} are from different strands')
    delta = self.coordinate - other.coordinate
```

```
      if self.strand is Strand.MINUS:
        delta = -delta
      return delta


class Nucleotide:
  __slots__ = ('parent', 'coordinate', 'position', '_base', 'residue')

  def __init__(self, parent, coordinate: int, position: int) -> None:
    self.parent = parent
    self.coordinate = coordinate  # genomic coordinate
    self.position = position  # position within parent
    self._base = None
    self.residue = None  # associated Residue, if any

  def __repr__(self) -> str:
    return f'{self.parent.chromosome}:{self.coordinate}({self.parent.strand}){self.base}'

  @property
  def base(self) -> 'Nucleobase':
    if not self._base:
      self._base = Nucleobase(self.parent.sequence[self.position - 1])
    return self._base

  @property
  def chromosome(self) -> 'Chromosome':
    return self.parent.chromosome

  @property
  def strand(self) -> 'Strand':
    return self.parent.strand

  @property
  def gene(self) -> 'Gene':
    if isinstance(self.parent, Gene):
      return self.parent
    return self.parent.gene

  @property
  def exon(self) -> 'Exon':
    return self.parent.get_exon_containing_position(self.position)

  def __eq__(self, other: 'Nucleotide'):
    if not isinstance(other, Nucleotide):
```

```
        raise TypeError(f'Cannot compare Nucleotide with {type(other)}')
    return self.coordinate, self.position, self.base == other.coordinate, other.position,
other.base


OptNucleotide = Optional['Nucleotide']
Codon = Tuple[OptNucleotide, OptNucleotide, OptNucleotide]

class Residue:
  __slots__ = ('protein', 'position', '_aa', 'codon')

  def __init__(self, protein: 'Protein', position: int) -> None:
    self._aa = None
    self.protein = protein
    self.position = position  # position within protein peptide sequence
    self.codon: Codon = (None, None, None)  # 3-tuple of associated Nucleotides; filled in
later

  def __repr__(self) -> str:
    return f'{self.amino_acid}{self.position}'

  @property
  def amino_acid(self) -> 'AminoAcid':
    if not self._aa:
      self._aa = AminoAcid(self.protein.sequence[self.position - 1])
    return self._aa

  @property
  def codon_str(self) -> str:
    return ''.join(str(nt.base) for nt in self.codon)

  @property
  def exons(self) -> List['Exon']:
    return list(set(nt.exon for nt in self.codon))

  @property
  def primary_exon(self) -> 'Exon':
    exons = Counter([nt.exon for nt in self.codon])
    return exons.most_common(1)[0][0]

  @property
  def junction(self) -> Optional['Junction']:
    exons = self.exons
    if len(exons) < 2:
```

```
            return None
        transcript = exons[0].transcript
        # this is a bit kludgy, may need to add properties to Exon class
        return transcript.junctions[exons[0].position - 1]

    @property
    def is_gap(self):
        return self.amino_acid is AminoAcid.GAP


@frozen(hash=True)
class GenomeRange:
    begin: 'Position' = field(validator=validators.instance_of(Position))
    end: 'Position' = field(validator=validators.instance_of(Position))

    def __attrs_post_init__(self):
        if self.begin > self.end:
            raise ValueError(f'{self.begin} is downstream of {self.end}')

    @property
    def chromosome(self):
        return self.begin.chromosome

    @property
    def strand(self):
        return self.begin.strand

    def __repr__(self):
        return
f'{self.begin.chromosome}({self.begin.strand}):{self.begin.coordinate}^{self.end.coordinate
}'

    def __eq__(self, other: 'GenomeRange'):
        if not isinstance(other, GenomeRange):
            return NotImplemented
        delta_begin = abs(self.begin - other.begin)
        delta_end = abs(self.end - other.end)
        if delta_begin <= 2 and delta_end <= 2 and (delta_begin != 0 or delta_end != 0):
            warn(f'Possible off-by-one error for ranges {self} and {other}')
        return delta_begin == 0 and delta_end == 0

    def __and__(self, other: 'GenomeRange'):
        if not isinstance(other, GenomeRange):
            return NotImplemented
```

```
      begin = max(self.begin, other.begin)
      end = min(self.end, other.end)
      return evolve(self, begin=begin, end=end) if begin <= end else None

    def __or__(self, other: 'GenomeRange'):
      if not isinstance(other, GenomeRange):
        return NotImplemented
      begin = min(self.begin, other.begin)
      end = max(self.end, other.end)
      return evolve(self, begin=begin, end=end) if begin <= end else None

    @property
    def length(self) -> int:
      return (self.end - self.begin) + 1

    def as_tuple(self):
      return self.chromosome, self.strand, self.begin.coordinate, self.end.coordinate

    @classmethod
    def from_coordinates(cls, chromosome: str, strand: 'Strand', begin: int, end: int):
      return cls(Position(chromosome, strand, begin), Position(chromosome, strand, end))


@frozen(hash=True)
class Junction:
    range: 'GenomeRange' = field(validator=validators.instance_of(GenomeRange))

    @property
    def donor(self):
      return self.range.begin

    @property
    def acceptor(self):
      return self.range.end

    def __repr__(self):
      return
f'{self.range.chromosome}({self.range.strand}):{self.donor.coordinate}^{self.acceptor.coord
inate}'

    def __eq__(self, other: 'Junction'):
      if not isinstance(other, Junction):
        return NotImplemented
      return self.range == other.range
```

```
  def __and__(self, other: 'Junction'):
    if not isinstance(other, Junction):
      return NotImplemented
    intersection = self.range & other.range
    return Junction(intersection) if intersection else None

  def __or__(self, other: 'Junction'):
    if not isinstance(other, Junction):
      return NotImplemented
    union = self.range | other.range
    return Junction(union) if union else None

  @property
  def length(self):
    return self.range.length

  def as_tuple(self):
    return self.range.as_tuple()

  @classmethod
  def from_coordinates(cls, chromosome: str, strand: 'Strand', donor: int, acceptor: int):
    return Junction(GenomeRange.from_coordinates(chromosome, strand, donor,
acceptor))

  @classmethod
  def from_splice_sites(cls, donor: 'Position', acceptor: 'Position'):
    return Junction(GenomeRange(donor, acceptor))


class UTR(ABC):
  def __init__(self, orf: 'ORF', boundary_exon_index: int):
    self.orf = orf
    self.transcript: 'Transcript' = orf.transcript
    self._boundary_exon_index = boundary_exon_index
    self.transcript_start = None
    self.transcript_stop = None

  @property
  def length(self):
    return self.transcript_stop - self.transcript_start + 1

  @property
  def nucleotides(self) -> List['Nucleotide']:
```

```
      return self.transcript.nucleotides[self.transcript_start-1:self.transcript_stop]

    @property
    def sequence(self) -> str:
      return self.transcript.sequence[self.transcript_start-1:self.transcript_stop]

    @property
    def start(self) -> int:
      return self.transcript.nucleotides[self.transcript_start-1].coordinate

    @property
    def stop(self) -> int:
      return self.transcript.nucleotides[self.transcript_stop-1].coordinate

    @property
    @abstractmethod
    def exons(self):
      raise NotImplementedError

    @abstractmethod
    def __repr__(self):
      raise NotImplementedError


class FivePrimeUTR(UTR):
    def __init__(self, orf: 'ORF', boundary_exon_index: int):
      super().__init__(orf, boundary_exon_index)
      self.transcript_start = 1
      self.transcript_stop = orf.transcript_start - 1

    @property
    def exons(self) -> List['Exon']:
      return self.transcript.exons[:self._boundary_exon_index+1]

    def __repr__(self):
      return f'{self.transcript}:utr5({self.transcript_start}-{self.transcript_stop})'


class ThreePrimeUTR(UTR):
    def __init__(self, orf: 'ORF', boundary_exon_index: int):
      super().__init__(orf, boundary_exon_index)
      self.transcript_start = orf.transcript_stop + 1
      self.transcript_stop = self.transcript.length
```

```
  @property
  def exons(self) -> List['Exon']:
    return self.transcript.exons[self._boundary_exon_index:]

  def __repr__(self):
    return f'{self.transcript}:utr3({self.transcript_start}-{self.transcript_stop})'
```

plotting.py

```
# functions to create different visualizations of isoforms/clones/domains/muts
from copy import copy
from dataclasses import dataclass
from itertools import chain, groupby, islice, tee
from operator import attrgetter, sub
from typing import (TYPE_CHECKING, Any, Callable, Collection, Dict, Iterable,
            List, Literal, Optional, Set, Tuple, Union)
from warnings import filterwarnings, warn

import matplotlib.lines as mlines
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from Bio import Align
from biosurfer.core.alignments import CodonAlignment, ProjectedFeature
from biosurfer.core.constants import (FRAMESHIFT, SPLIT_CODON, AminoAcid,
                        CodonAlignmentCategory, FeatureType,
                        SequenceAlignmentCategory, Strand)
from biosurfer.core.helpers import (ExceptionLogger, Interval, IntervalTree,
                       get_interval_overlap_graph)
from biosurfer.core.models.biomolecules import (GencodeTranscript,
                            PacBioTranscript, Transcript)
from biosurfer.core.splice_events import (AcceptorSpliceEvent,
                           DonorSpliceEvent, ExonBypassEvent,
                           ExonSpliceEvent, IntronSpliceEvent)
from brokenaxes import BrokenAxes
from graph_tool import Graph
from graph_tool.topology import sequential_vertex_coloring
from matplotlib._api.deprecation import MatplotlibDeprecationWarning
from more_itertools import first, last, only

if TYPE_CHECKING:
```

```
    from biosurfer.core.alignments import ProteinAlignmentBlock, CodonAlignmentBlock
    from biosurfer.core.models.biomolecules import Protein
    from matplotlib.axes import Axes
    from matplotlib.figure import Figure

StartStop = Tuple[int, int]

filterwarnings("ignore", category=MatplotlibDeprecationWarning)

TRANSCRIPT_SOURCE = {
    'gencodetranscript': 'GENCODE',
    'pacbiotranscript': 'PacBio'
}
def get_transcript_source(tx: 'Transcript'):
    return TRANSCRIPT_SOURCE.get(tx.type, '')

# colors for different transcript types
TRANSCRIPT_COLORS = {
    None: ('#404040', '#777777'),
    GencodeTranscript: ('#343553', '#5D5E7C'),
    PacBioTranscript: ('#61374D', '#91677D')
}

# colors for transcript events
EVENT_COLORS = {
    IntronSpliceEvent: '#e69138',
    DonorSpliceEvent: '#6aa84f',
    AcceptorSpliceEvent: '#674ea7',
    ExonSpliceEvent: '#3d85c6',
    ExonBypassEvent: '#bebebe',
}

# alpha values for different absolute reading frames
ABS_FRAME_ALPHA = {0: 1.0, 1: 0.45, 2: 0.15}

# hatching styles for different relative frameshifts
REL_FRAME_STYLE = {
    CodonAlignmentCategory.FRAME_AHEAD: '////',
    CodonAlignmentCategory.FRAME_BEHIND: 'xxxx'
}

# colors for CodonAlignmentBlocks
CBLOCK_COLORS = {
    CodonAlignmentCategory.TRANSLATED: '#9bf3ff',
```

```
  CodonAlignmentCategory.INSERTION: '#05e0ff',
  CodonAlignmentCategory.FRAME_AHEAD: '#fff099',
  CodonAlignmentCategory.FRAME_BEHIND: '#ffd700',
  CodonAlignmentCategory.UNTRANSLATED: '#ff99ce',
  CodonAlignmentCategory.DELETION: '#ff0082',
  CodonAlignmentCategory.EDGE: '#8270c1',
  CodonAlignmentCategory.COMPLEX: '#aaaaaa'
}

PBLOCK_COLORS = {
  SequenceAlignmentCategory.DELETION: '#FF0082',
  SequenceAlignmentCategory.INSERTION: '#05E0FF',
  SequenceAlignmentCategory.SUBSTITUTION: '#FFD700'
}

FEATURE_COLORS = {
  'MobiDB': '#AAAAAA'
}

@dataclass
class IsoformPlotOptions:
  """Bundles various options for adjusting plots made by IsoformPlot."""
  intron_spacing: int = 30  # number of bases to show in each intron
  track_spacing: float = 2  # ratio of space between tracks to max track width
  subtle_splicing_threshold: int = 20  # maximum difference (in bases) between exon
boundaries to display subtle splicing

  @property
  def max_track_width(self) -> float:
    return 1/(self.track_spacing + 1)

  @max_track_width.setter
  def max_track_width(self, width: float):
    self.track_spacing = (1 - width)/width


TableColumn = Callable[[Transcript], str]
class IsoformPlot:
  """Encapsulates methods for drawing one or more isoforms aligned to the same genomic
x-axis."""
  def __init__(self, transcripts: Iterable['Transcript'], columns: Dict[str, TableColumn] =
None, **kwargs):
    self.transcripts: List['Transcript'] = list(transcripts)  # list of orf objects to be drawn
    gene = {tx.gene for tx in filter(None, self.transcripts)}
```

```
    if len(gene) > 1:
      raise ValueError(f'Found isoforms from multiple genes: {", ".join(g.name for g in
gene)}')
    strand = only(
      {tx.strand for tx in filter(None, self.transcripts)},
      too_long = ValueError("Can't plot isoforms from different strands")
    )
    self.strand: Strand = strand

    self.fig: Optional['Figure'] = None
    self._bax: Optional['BrokenAxes'] = None
    self._columns: Dict[str, TableColumn] = {'Source': get_transcript_source} | (columns if
columns else dict())
    self.opts = IsoformPlotOptions(**kwargs)
    self.reset_xlims()

    # keep track of artists for legend
    self._handles = dict()


  # Internally, IsoformPlot stores _subaxes, which maps each genomic region to the
subaxes that plots the region's features.
  # The xlims property provides a simple interface to allow users to control which genomic
regions are plotted.
  @property
  def xlims(self) -> Tuple[StartStop]:
    """Coordinates of the genomic regions to be plotted, as a tuple of (start, end) tuples."""
    return self._xlims

  @xlims.setter
  def xlims(self, xlims: Iterable[StartStop]):
    xregions = IntervalTree.from_tuples((min(start, stop), max(start, stop)+1) for start, stop
in xlims)  # condense xlims into single IntervalTree object
    xregions.merge_equals()
    xregions.merge_overlaps()
    xregions.merge_neighbors()
    xregions = sorted(xregions.all_intervals)
    if self.strand is Strand.MINUS:
      xregions.reverse()
    self._subaxes = IntervalTree.from_tuples((start, end, i) for i, (start, end, _) in
enumerate(xregions))
    self._xlims = tuple((start, end-1) if self.strand is Strand.PLUS else (end-1, start) for start,
end , _ in xregions)
```

```
  def reset_xlims(self):
    """Set xlims automatically based on exons in isoforms."""
    space = self.opts.intron_spacing
    self.xlims = tuple((exon.start - space, exon.stop + space) for tx in filter(None,
self.transcripts) for exon in tx.exons)

  # This method speeds up plotting by allowing IsoformPlot to add artists only to the
subaxes where they are needed.
  def _get_subaxes(self, xcoords: Union[int, StartStop]) -> Tuple['Axes']:
    """For a specific coordinate or range of coordinates, retrieve corresponding subaxes."""
    if isinstance(xcoords, tuple):
      if xcoords[0] > xcoords[1]:
        xcoords = (xcoords[1], xcoords[0])
      xcoords = slice(*xcoords)
    subax_ids = [interval[-1] for interval in self._subaxes[xcoords]]
    if not subax_ids:
      raise ValueError(f"{xcoords} is not within plot's xlims")
    return tuple(self._bax.axs[id] for id in subax_ids)

  def draw_point(self, track: int, pos: int,
           ylims: tuple[float, float] = None,
           marker='', linewidth=1, **kwargs):
    """Draw a feature at a specific point. Appears as a vertical line with an optional
marker."""
    if ylims is None:
      ylims = -0.5*self.opts.max_track_width, 0.5*self.opts.max_track_width
    artist = mlines.Line2D(
      xdata = (pos, pos),
      ydata = (track + ylims[0], track + ylims[1]),
      linewidth = linewidth,
      marker = marker,
      markevery = 2,
      **kwargs
    )

    try:
      subaxes = self._get_subaxes(pos)[0]
    except ValueError as e:
      warn(str(e))
    else:
      subaxes.add_artist(artist)
    return artist

  def draw_region(self, track: int, start: int, stop: int,
```

```
            y_offset: Optional[float] = None,
            height: Optional[float] = None,
            type='rect', **kwargs):
    """Draw a feature that spans a region. Appearance types are rectangle and line."""
    if start == stop:
        return
    # TODO: make type an enum?
    if type == 'rect':
        if height is None:
            height = self.opts.max_track_width
        if y_offset is None:
            y_offset = -0.5*height
        artist = mpatches.Rectangle(
            xy = (start, track + y_offset),
            width = stop - start,
            height = height,
            **kwargs
        )
    elif type == 'line':
        if y_offset is None:
            y_offset = 0
        artist = mlines.Line2D(
            xdata = (start, stop),
            ydata = (track + y_offset, track + y_offset),
            **kwargs
        )
    else:
        raise ValueError(f'Region type "{type}" is not defined')

    subaxes = self._get_subaxes((start, stop))
    for ax in subaxes:
        ax.add_artist(copy(artist))
    return artist

def draw_background_rect(self, start: int, stop: int,
                         track_first: int = None, track_last: int = None,
                         padding: float = None, **kwargs):
    """Draw a rectangle in the background of the plot."""
    if start == stop:
        return
    if track_first is None:
        track_first = 0
    if track_last is None:
        track_last = len(self.transcripts) - 1
```

```
        if padding is None:
            padding = self.opts.max_track_width
        top = track_first - padding
        bottom = track_last + padding
        artist = mpatches.Rectangle(
            xy = (start, top),
            width = stop - start,
            height = bottom - top,
            zorder = 0.5,
            **kwargs
        )

        subaxes = self._get_subaxes((start, stop))
        for ax in subaxes:
            ax.add_artist(copy(artist))
        return artist

    def draw_text(self, x: int, y: float, text: str, **kwargs):
        """Draw text at a specific location. x-coordinate is genomic, y-coordinate is w/ respect
to tracks (0-indexed).
        Ex: x=20000, y=2 will center text on track 2 at position 20,000."""
        # TODO: make this use Axes.annotate instead
        # we can't know how much horizontal space text will take up ahead of time
        # so text is plotted using BrokenAxes' big_ax, since it spans the entire x-axis
        big_ax = self._bax.big_ax
        try:
            subaxes = self._get_subaxes(x)[0]  # grab coord transform from correct subaxes
        except ValueError as e:
            warn(str(e))
        else:
            big_ax.text(x, y, text, transform=subaxes.transData, **kwargs)

    def draw_legend(self, only_labels: Optional[Iterable[str]] = None, except_labels:
Optional[Iterable[str]] = None, **kwargs):
        if only_labels and except_labels:
            raise ValueError('Cannot set both "only_labels" and "except_labels"')
        elif only_labels:
            labels = [label for label in self._handles if label in only_labels]
        elif except_labels:
            labels = [label for label in self._handles if label not in except_labels]
        else:
            labels = list(self._handles.keys())
        handles = [self._handles[label] for label in labels]
        self.fig.legend(
```

```
      handles = handles,
      labels = labels,
      # ncol = 1,
      # loc = 'center left',
      # mode = 'expand',
      # bbox_to_anchor = (1.05, 0.5),
      **kwargs
    )

  def draw_isoform(self, tx: 'Transcript', track: int):
    """Plot a single isoform in the given track."""
    start, stop = tx.start, tx.stop
    align_start, align_stop = 'right', 'left'
    if self.strand is Strand.MINUS:
      align_start, align_stop = align_stop, align_start

    # plot intron line
    self.draw_region(
      track,
      start = start,
      stop = stop,
      type = 'line',
      linewidth = 1.5,
      color = 'gray',
      zorder = 1.5
    )

    # plot exons
    utr_kwargs = {
      'type': 'rect',
      'edgecolor': 'k',
      'facecolor': TRANSCRIPT_COLORS[type(tx)][1],
      'height': 0.5*self.opts.max_track_width,
      'zorder': 1.5
    }
    cds_kwargs = {
      'type': 'rect',
      'edgecolor': 'k',
      'facecolor': TRANSCRIPT_COLORS[type(tx)][0],
      'zorder': 1.5
    }
    if tx.orfs:
      orf = tx.primary_orf
      if orf.utr5:
```

```
            for exon in orf.utr5.exons:
              if self.strand is Strand.PLUS:
                start = exon.start
                stop = min(exon.stop, orf.start)
              elif self.strand is Strand.MINUS:
                start = max(exon.start, orf.stop)
                stop = exon.stop
              self.draw_region(track, start=start, stop=stop, **utr_kwargs)
          for exon in orf.exons:
            start = max(exon.start, orf.start)
            stop = min(exon.stop, orf.stop)
            self.draw_region(track, start=start, stop=stop, **cds_kwargs)
          if orf.utr3:
            for exon in orf.utr3.exons:
              if self.strand is Strand.PLUS:
                start = max(exon.start, orf.stop)
                stop = exon.stop
              elif self.strand is Strand.MINUS:
                start = exon.start
                stop = min(exon.stop, orf.start)
              self.draw_region(track, start=start, stop=stop, **utr_kwargs)
      else:
        for exon in tx.exons:
          self.draw_region(track, start=exon.start, stop=exon.stop, **utr_kwargs)

      for exon in tx.exons:
        # label every 5th exon in anchor isoform for easier navigation
        if track == 0 and exon.position % 5 == 0:
          self.draw_text((exon.start + exon.stop)//2, track - self.opts.max_track_width,
f'E{exon.position}', ha='center', va='baseline')

        # add subtle splice (delta) amounts, if option turned on
        # first, make sure the exon contains a (coding) cds object
        # if exon.cds:
        #   # TODO: pull subtle splice detection code out into this method?
        #   delta_start, delta_end = retrieve_subtle_splice_amounts(exon.cds)
        #   if delta_start:
        #     self.draw_text(exon.start, track-0.1, delta_start, va='bottom', ha=align_start,
size='x-small')
        #   if delta_end:
        #     self.draw_text(exon.stop, track-0.1, delta_end, va='bottom', ha=align_stop,
size='x-small')

      for orf in tx.orfs:
```

```
            first_res = orf.protein.residues[0]
            last_res = orf.protein.residues[-1]
            if first_res.amino_acid is AminoAcid.MET:
                start_codon = first_res.codon[0].coordinate
                self.draw_point(track, start_codon, color='lime')
            if last_res.amino_acid is AminoAcid.STOP:
                stop_codon = last_res.codon[2].coordinate
                self.draw_point(track, stop_codon, color='red')

        if hasattr(tx, 'start_nf') and tx.start_nf:
            self.draw_text(tx.start if self.strand is Strand.PLUS else tx.stop, track, '! ', ha='right',
va='center', weight='bold', color='r')
        if hasattr(tx, 'end_nf') and tx.end_nf:
            self.draw_text(tx.stop if self.strand is Strand.PLUS else tx.start, track, ' !', ha='left',
va='center', weight='bold', color='r')

    def draw_all_isoforms(self, subplot_spec = None):
        """Plot all isoforms."""
        R = len(self.transcripts)
        C = len(self._columns)
        self.fig = plt.figure()
        self._bax = BrokenAxes(fig=self.fig, xlims=self.xlims, ylims=((R-0.5, -0.5),), wspace=0,
d=0.008, subplot_spec=subplot_spec)
        self._handles['Intron'] = mlines.Line2D([], [], linewidth=1.5, color='gray')
        self._handles['Exon (translated)'] =
mpatches.Patch(facecolor=TRANSCRIPT_COLORS[None][0], edgecolor='k')
        self._handles['Exon (untranslated)'] =
mpatches.Patch(facecolor=TRANSCRIPT_COLORS[None][1], edgecolor='k')
        self._handles['Start codon'] = mlines.Line2D([], [], linestyle='None', color='lime',
marker='|', markersize=10, markeredgewidth=1)
        self._handles['Stop codon'] = mlines.Line2D([], [], linestyle='None', color='red',
marker='|', markersize=10, markeredgewidth=1)



        # process orfs to get ready for plotting
        # find_and_set_subtle_splicing_status(self.transcripts,
self.opts.subtle_splicing_threshold)

        for i, tx in enumerate(self.transcripts):
            with ExceptionLogger(f'Error plotting {tx}'):
                if tx:
                    self.draw_isoform(tx, i)

        # plot genomic region label
```

```
    # gene = self.transcripts[0].gene
    # start, end = self.xlims[0][0], self.xlims[-1][1]
    # self._bax.set_title(f'{gene.chromosome}({self.strand}):{start}-{end}')

    # hide y axis spine
    left_subaxes = self._bax.axs[0]
    left_subaxes.spines['left'].set_visible(False)
    left_subaxes.set_yticks([])

    # plot table
    # https://stackoverflow.com/a/57169705
    table = self._bax.big_ax.table(
      rowLabels = [getattr(tx, 'name', '') for tx in self.transcripts],
      colLabels = list(self._columns.keys()),
      cellText = [[f(tx) if tx else '' for f in self._columns.values()] for tx in self.transcripts],
      cellLoc = 'center',
      edges = 'open',
      bbox = (-0.1*C, 0.0, 0.1*C, (R+1)/R)
    )
    # table.auto_set_font_size(False)
    # table.set_fontsize(10)

    # rotate x axis tick labels for better readability
    for subaxes in self._bax.axs:
      subaxes.xaxis.set_major_formatter('{x:.0f}')
      for label in subaxes.get_xticklabels():
        label.set_va('top')
        label.set_rotation(90)
        label.set_size(8)

  def draw_frameshifts(self, anchor: Optional['Transcript'] = None, hatch_color='white'):
    """Plot relative frameshifts on all isoforms. Uses first isoform as the anchor by
default."""
    self._handles['Frame +1'] = mpatches.Patch(facecolor='k', edgecolor='w',
hatch=REL_FRAME_STYLE[CodonAlignmentCategory.FRAME_AHEAD])
    self._handles['Frame +2'] = mpatches.Patch(facecolor='k', edgecolor='w',
hatch=REL_FRAME_STYLE[CodonAlignmentCategory.FRAME_BEHIND])

    if anchor is None:
      anchor = next(filter(None, self.transcripts))
    if not anchor or not anchor.protein:
      warn(
        'Cannot draw frameshifts without an anchor ORF'
      )
```

```
            return
        for i, other in enumerate(self.transcripts):
            if not other or not other.protein or other is anchor:
                continue
            aln = CodonAlignment.from_proteins(anchor.protein, other.protein)
            for block in filter(lambda block: block.category in FRAMESHIFT, aln.blocks):
                for exons, residues in
groupby(other.protein.residues[block.other_range.start:block.other_range.stop],
key=attrgetter('exons')):
                    if len(exons) > 1:
                        continue
                    r1, r2 = tee(residues, 2)
                    start = first(r1).codon[1].coordinate
                    stop = last(r2).codon[1].coordinate
                    self.draw_region(
                        track = i,
                        start = start,
                        stop = stop,
                        facecolor = 'none',
                        edgecolor = hatch_color,
                        linewidth = 0.0,
                        zorder = 1.9,
                        hatch = REL_FRAME_STYLE[block.category]
                    )

    def draw_codon_alignment_blocks(self, cd_aln: 'CodonAlignment', alpha: float = 0.5):
        for category, color in CBLOCK_COLORS.items():
            label = category.name.capitalize().replace('_', ' ')
            if label not in self._handles:
                self._handles[label] = mpatches.Patch(facecolor=color)
        height = 0.25*self.opts.max_track_width
        track = self.transcripts.index(cd_aln.other.transcript)
        for block in filter(lambda block: block.category is not
CodonAlignmentCategory.MATCH, cd_aln.blocks):
            if block.other_range:
                start = cd_aln.other.residues[block.other_range[0]].codon[1].coordinate
                stop = cd_aln.other.residues[block.other_range[-1]].codon[1].coordinate
            else:
                start = cd_aln.anchor.residues[block.anchor_range[0]].codon[1].coordinate
                stop = cd_aln.anchor.residues[block.anchor_range[-1]].codon[1].coordinate
            if block.category in SPLIT_CODON:
                self.draw_point(  # TODO: fix
                    track,
                    start,
```

```
            height = height,
            type = 'lollipop',
            marker = '.',
            color = CBLOCK_COLORS[block.category],
            zorder = 1.9,
            alpha = alpha
          )
        else:
          self.draw_region(
            track,
            start,
            stop,
            y_offset = -0.5*self.opts.max_track_width,
            height = -height,
            facecolor = CBLOCK_COLORS[block.category],
            alpha = alpha
          )

  def draw_protein_alignment_blocks(self, pblocks: Iterable['ProteinAlignmentBlock'],
anchor: 'Protein', other: 'Protein', alpha: float = 1.0):
    for category, color in PBLOCK_COLORS.items():
      label = category.name.capitalize().replace('_', ' ')
      if label not in self._handles:
        self._handles[label] = mpatches.Patch(facecolor=color)
    self._handles['Ragged 5\' end'] = mlines.Line2D([], [], linestyle='None', color='#999999',
marker='<', markersize=8, markeredgewidth=1)
    self._handles['Ragged 3\' end'] = mlines.Line2D([], [], linestyle='None', color='#999999',
marker='>', markersize=8, markeredgewidth=1)

    for pblock in filter(lambda block: block.category is not
SequenceAlignmentCategory.MATCH, pblocks):
      anchor_start, anchor_stop, other_start, other_stop = None, None, None, None
      if pblock.category is not SequenceAlignmentCategory.INSERTION:
        anchor_start = anchor.transcript.get_genome_coord_from_transcript_coord(
          anchor.get_transcript_coord_from_protein_coord(pblock.anchor_range[0]) + 1
        ).coordinate
        anchor_stop = anchor.transcript.get_genome_coord_from_transcript_coord(
          anchor.get_transcript_coord_from_protein_coord(pblock.anchor_range[-1]) + 1
        ).coordinate
      if pblock.category is not SequenceAlignmentCategory.DELETION:
        other_start = other.transcript.get_genome_coord_from_transcript_coord(
          other.get_transcript_coord_from_protein_coord(pblock.other_range[0]) + 1
        ).coordinate
        other_stop = other.transcript.get_genome_coord_from_transcript_coord(
```

```
            other.get_transcript_coord_from_protein_coord(pblock.other_range[-1]) + 1
          ).coordinate

        other_track = self.transcripts.index(other.transcript)
        lollipop_direction = 1 if pblock.category is SequenceAlignmentCategory.INSERTION
else -1

        if pblock.ragged5:
          self.draw_point(
            other_track,
            pos = anchor_start,
            ylims = (lollipop_direction*0.75*self.opts.max_track_width, 0),
            linewidth = 0,
            marker = '<',
            markersize = 6,
            color = PBLOCK_COLORS[pblock.category],
            zorder = 1.9
          )
        if pblock.ragged3:
          self.draw_point(
            other_track,
            pos = anchor_stop,
            ylims = (lollipop_direction*0.75*self.opts.max_track_width, 0),
            linewidth = 0,
            marker = '>',
            markersize = 6,
            color = PBLOCK_COLORS[pblock.category],
            zorder = 1.9
          )
        self.draw_region(
          other_track,
          start = anchor_start,
          stop = anchor_stop,
          y_offset = -1.0*self.opts.max_track_width,
          height = 0.5*self.opts.max_track_width,
          edgecolor = 'none',
          facecolor = PBLOCK_COLORS[pblock.category],
          alpha = alpha
        )
        self.draw_region(
          other_track,
          start = other_start,
          stop = other_stop,
          y_offset = 0.5*self.opts.max_track_width,
```

```
            height = 0.5*self.opts.max_track_width,
            edgecolor = 'none',
            facecolor = PBLOCK_COLORS[pblock.category],
            alpha = alpha
        )

    def draw_features(self):
        h = self.opts.max_track_width
        feature_names = sorted({feature.name for tx in filter(None, self.transcripts) if tx.protein
for feature in tx.protein.features if feature.type is not FeatureType.IDR})
        cmap = sns.color_palette('pastel', len(feature_names))
        colors = dict(zip(feature_names, cmap))
        colors.update(FEATURE_COLORS)
        self._handles.update({name: mpatches.Patch(facecolor=color) for name, color in
colors.items()})
        for track, tx in enumerate(self.transcripts):
            if not tx or not tx.protein:
                continue
            features = tx.protein.features
            if not features:
                continue
            subtracks, n_subtracks = generate_subtracks(
                ((feature.protein_start, feature.protein_stop) for feature in features),
                (feature.name for feature in features)
            )
            for feature in features:
                subtrack = subtracks[feature.name]
                color = colors[feature.name]
                if feature.reference:
                    subfeatures = groupby(feature.residues, key=lambda res: (False,
res.primary_exon))
                    # n_subtracks_temp = n_subtracks
                else:
                    subfeatures = groupby(feature.residues, key=lambda res: (res in
feature.altered_residues, res.primary_exon))
                    # n_subtracks_temp = 2*n_subtracks
                for (altered, _), subfeature in subfeatures:
                    subfeature = list(subfeature)
                    start = subfeature[0].codon[1].coordinate
                    stop = subfeature[-1].codon[1].coordinate
                    self.draw_region(
                        track,
                        start = start,
                        stop = stop,
```

```
                y_offset = (-0.5 + subtrack/n_subtracks)*h,
                height = h/n_subtracks,
                edgecolor = 'none',
                facecolor = color,
                alpha = 0.5 if altered else 1.0,
                zorder = 1.8,
                label = feature.name
            )
        # draw box behind entire feature
        self.draw_region(
            track,
            start = feature.residues[0].codon[1].coordinate,
            stop = feature.residues[-1].codon[1].coordinate,
            y_offset = (-0.5 + subtrack/n_subtracks)*h,
            height = h/n_subtracks,
            edgecolor = 'none',
            facecolor = color,
            alpha = 0.5,
            zorder = 1.4
        )

    def savefig(self, fig_path):
        self.fig.set_size_inches(20, 0.8 + 0.4*len(self.transcripts))
        plt.figure(self.fig)
        plt.savefig(fig_path, facecolor='w', transparent=False, dpi=200, bbox_inches='tight')


def generate_subtracks(intervals: Iterable[Tuple[int, int]], labels: Iterable):
    # inspired by https://stackoverflow.com/a/19088519
    # build graph of labels where labels are adjacent if their intervals overlap
    g, vertex_labels = get_interval_overlap_graph(intervals, labels)
    # find vertex coloring of graph
    # all labels w/ same color can be put into same subtrack
    coloring = sequential_vertex_coloring(g)
    label_to_subtrack = dict(zip(vertex_labels, coloring))
    subtracks = max(label_to_subtrack.values(), default=0) + 1
    return label_to_subtrack, subtracks
```

Biosurfer_analysis codebase

The directory structure of the **Biosurfer**_analysis codebase is outlined below:

```
biosurfer_analysis/
├── scripts
│   ├── c_termini_summary.py
│   ├── download_gencode_toy.sh
│   ├── download_gencode_v42.sh
│   ├── download_wtc11.sh
│   ├── genome_wide_summary.py
│   ├── install_biosurfer.sh
│   ├── internal_summary.py
│   ├── isoform_plotting.sh
│   ├── manuscript_stats.ipynb
│   ├── n_termini_summary.py
│   └── plot_config.py
```

c_termini_summary.py

```
#!/usr/bin/env python

#%%Importing libraries
from pathlib import Path
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import numpy as np
import scipy.stats as stats
import matplotlib as mpl

from plot_config import CTERM_CLASSES, CTERM_PALETTE, cterm_splice_palette,
cterm_frameshift_palette, pblocks

# %% Output paths
output = Path('../E_cterm_summary_plots')
output.mkdir(exist_ok=True)

#%%
cterm_pblocks = pblocks[~pblocks['cterm'].isna() & (pblocks['nterm'].isna()) &
(pblocks['cterm'] != "ALTERNATIVE_ORF") & (pblocks['cterm'] != "UNKNOWN")].copy()
cterm_pblocks['cterm'] =
cterm_pblocks['cterm'].map(CTERM_CLASSES).astype('category')
# Changed string to set for intersection
```

```
cterm_pblocks['APA'] = cterm_pblocks['events'].apply(lambda x:
set(x).intersection('BbPp')).astype(bool)

#%% Fig5 panel A: Frequency of splice-driven and frameshift-driven C-terminal events

cterm_fig = plt.figure(figsize=(3.8, 2))
ax = sns.countplot(
   data = cterm_pblocks,
   y = 'cterm',
   order = CTERM_CLASSES.values(),
   palette = CTERM_PALETTE,
   saturation = 1,
   linewidth = 1,
   edgecolor = 'k',
)
ax.set_xlabel('Number of alternative isoforms')
ax.set_ylabel('')
plt.savefig(output/'cterm-class-counts.png', dpi=200, facecolor=None,
bbox_inches='tight')

#Output source data
cterm_pblocks.query("cterm in ['Splice-driven', 'Frameshift-
driven']")[['anchor','other','cterm']].to_csv(output/'cterm-class-counts-table.tsv', sep='\t')

# %% Fig5 panel B: Frequency of splice-driven patterns
cterm_pblock_events =
cterm_pblocks['up_stop_events'].combine(cterm_pblocks['down_stop_events'], lambda x,
y: (x, y))
single_ATE = (cterm_pblocks['cterm'] == 'Splice-driven') &
cterm_pblocks['tblock_events'].isin({('B', 'b'), ('b', 'B')})
cterm_splice_subcats = pd.DataFrame(
   {
      'Exon extension introduces termination': cterm_pblocks['up_stop_events'].isin({'P', 'I',
'D'}),
      'Alternative terminal exon(s)': cterm_pblock_events.isin({('B', 'b'), ('b', 'B')}),
      'Poison exon inclusion': cterm_pblocks['up_stop_events'] == 'E',
      'Other': [True for _ in cterm_pblocks.index]
      #'Alternative last exon in UTR': cterm_pblocks['cblocks'].apply(lambda x: 'TRANSLATED'
in x and 'DELETION' in x and 'UNTRANSLATED' not in x)
   }
)
cterm_pblocks['splice_subcat'] =
cterm_splice_subcats.idxmax(axis=1).astype(pd.CategoricalDtype(cterm_splice_subcats.
columns, ordered=True))
```

```
cterm_splice_palette_dict = dict(zip(
   cterm_splice_subcats.columns,
   cterm_splice_palette[0:1] + cterm_splice_palette[1:2] + cterm_splice_palette[2:3] +
['#bbbbbb']
))
splice_subcat_order = tuple(cterm_splice_subcats.keys())

cterm_pblock_events =
cterm_pblocks['up_stop_events'].combine(cterm_pblocks['down_stop_events'], lambda x,
y: (x, y))
single_ATE = (cterm_pblocks['cterm'] == 'Splice-driven') &
cterm_pblocks['tblock_events'].isin({('B', 'b'), ('b', 'B')})


cterm_splice_subcats = pd.DataFrame(
   {
      'Exon extension introduces \n termination (EXIT)':
cterm_pblocks['up_stop_events'].isin({'P', 'I', 'D'}),
      'Alternative terminal \n exon(s) (ATE)': cterm_pblock_events.isin({('B', 'b'), ('b', 'B')}),
      'Alternative last exon \n in UTR (ALE in UTR)': cterm_pblocks.apply(lambda row:
'TRANSLATED' in row['cblocks'] and 'DELETION' in row['cblocks'] and 'UNTRANSLATED' not
in row['cblocks'] if row['cterm'] == 'Splice-driven' and row['splice_subcat'] == 'Other' else
False, axis=1),
      'Poison exon inclusion': cterm_pblocks['up_stop_events'] == 'E',
      'Cut-out splice terminal \n exon (COSTE)': cterm_pblocks.apply(lambda row:
'DELETION' in row['cblocks'] and 'INSERTION' in row['cblocks'] and 'TRANSLATED' not in
row['cblocks'] and 'UNTRANSLATED' not in row['cblocks'] and 'FRAME' not in row['cblocks']
and 'p' in row['tblock_events'] and row['tblock_events'].count('B') == 1 if row['cterm'] ==
'Splice-driven' and row['splice_subcat'] == 'Other' else False, axis=1),
      'Other': [True for _ in cterm_pblocks.index]
   }
)
cterm_pblocks['splice_subcat'] =
cterm_splice_subcats.idxmax(axis=1).astype(pd.CategoricalDtype(cterm_splice_subcats.
columns, ordered=True))

cterm_splice_palette_dict = dict(zip(
   cterm_splice_subcats.columns,
   cterm_splice_palette[0:1] + cterm_splice_palette[1:2] + cterm_splice_palette[2:3] +
cterm_splice_palette[3:4] + cterm_splice_palette[4:5] +  ['#bbbbbb']
))
splice_subcat_order = tuple(cterm_splice_subcats.keys())
```

```
cterm_splice_fig, axs = plt.subplots(1, 2, figsize=(9, 4))
sns.countplot(
   ax = axs[0],
   data = cterm_pblocks[cterm_pblocks['cterm'] == 'Splice-driven'],
   y = 'splice_subcat',
   order = splice_subcat_order,
   palette = cterm_splice_palette_dict,
   saturation = 1,
   edgecolor = 'k',
   linewidth = 1,
)
axs[0].set_xlabel('Number of alternative isoforms')
axs[0].set_ylabel(None)

plt.savefig(output/'cterm-splicing-subcats.png', dpi=700, facecolor=None,
bbox_inches='tight')

#Output source data
cterm_pblocks.assign(anchor_relative_length_change =
cterm_pblocks['anchor_relative_length_change'].abs())[['anchor','other',
'splice_subcat','anchor_relative_length_change']].to_csv(output/'cterm-splicing-subcats-
table.tsv', sep='\t')

#%%
#TODO: Mann-Whitney U Test signed ranked test here between SE, alt Acc .. vs Intron
cterm_frameshift=cterm_pblocks[cterm_pblocks['cterm'] == 'Frameshift-driven']
cterm_intron = cterm_pblocks[cterm_pblocks['frame_subcat'] == 'Intron']
cterm_se = cterm_pblocks[cterm_pblocks['frame_subcat'] == 'Single exon']
cterm_altacc = cterm_pblocks[cterm_pblocks['frame_subcat'] == 'Alt. acceptor']
cterm_altdonor = cterm_pblocks[cterm_pblocks['frame_subcat'] == 'Alt. donor']

data = [[cterm_intron],[cterm_frameshift],[cterm_se],[cterm_altacc],[cterm_altdonor]]
data = [[4890, 3499, 3301, 551], [6819, 2247, 1185, 1096, 457, 437]]
stat, p, dof, expected = chi2_contingency(data)


# %% Alternative Last Exon in 3' UTR case from Splice-driven 'Other' category.

cterm_pblock_splice = cterm_pblocks[cterm_pblocks['cterm'] == 'Splice-driven']
cterm_splice_other = cterm_pblock_splice[cterm_pblock_splice['splice_subcat']=='Other']
condition1 = cterm_splice_other['cblocks'].apply(lambda x: 'DELETION' in x and
'TRANSLATED' in x)
condition2 = cterm_splice_other['cblocks'].apply(lambda x: 'UNTRANSLATED' not in x)
cterm_aleutr = cterm_splice_other[condition1 & condition2].copy()
```

```
cterm_aleutr.to_csv(output/'cterm-splice-driven-ALEinUTR.tsv', sep='\t')

# %% Cut-out splice terminal exon case from Splice-driven 'Other' category.

condition3 = cterm_splice_other['cblocks'].apply(lambda x: 'DELETION' in x and
'INSERTION' in x)
condition4 = cterm_splice_other['cblocks'].apply(lambda x: 'TRANSLATED' not in x and
'UNTRANSLATED' not in x and 'FRAME' not in x)
condition5 = cterm_splice_other['tblock_events'].apply(lambda x: x.count('B') == 1 and 'p'
in x)
cterm_other_new = cterm_splice_other[condition3 & condition4 & condition5].copy()
cterm_other_new.to_csv(output/'cterm-splice-driven-other-NEW.tsv', sep='\t')

# %%  Fig5 panel C & D: 2D scatter plot v2 splice-driven vs frameshift-driven

font = {
   'family': 'sans-serif',
   'sans-serif': ['Arial'],
   'weight': 'normal',
   'size': 10
}
mpl.rc('font', **font)
msx_data = cterm_pblocks[cterm_pblocks['cterm'] == 'Splice-driven']
sds_data = cterm_pblocks[cterm_pblocks['cterm'] == 'Frameshift-driven']
fig, axes = plt.subplots(nrows=1, ncols=2, figsize=(6, 6))
msx_color = (0.5048212226066897, 0.00392156862745098, 0.47021914648212226)
sds_color = (0.7885121107266436, 0.03238754325259515, 0.13656286043829297)

sns.scatterplot(data=msx_data, x='aa_loss', y='aa_gain', marker='o', ax=axes[0], alpha=0.2,
            color=msx_color)
axes[0].set_title('Splice-driven', fontsize=13)
axes[0].set_xlabel('Reference \n(amino acids)', fontsize=12)
axes[0].set_ylabel('Alternative \n(amino acids)', fontsize=12)
axes[0].set_xlim(0, 3000)
axes[0].set_ylim(0, 3000)
axes[0].set_aspect('equal')
axes[0].grid(True, linestyle='--', linewidth=0.5)

sns.scatterplot(data=sds_data, x='aa_loss', y='aa_gain', marker='o', ax=axes[1], alpha=0.2,
color=sds_color)
axes[1].set_title('Frameshift-driven', fontsize=13)
axes[1].set_xlabel('Reference \n(amino acids)', fontsize=12)
axes[1].set_ylabel('Alternative \n(amino acids)', fontsize=12)
axes[1].set_xlim(0, 3000)
```

```
axes[1].set_ylim(0, 3000)
axes[1].set_aspect('equal')
axes[1].grid(True, linestyle='--', linewidth=0.5)

plt.tight_layout()
plt.savefig(output/'cterm-rel-length-change_scatterplot.png', dpi=800, facecolor=None,
bbox_inches='tight')
plt.show()

#Output source data
cterm_pblocks.query("cterm in ['Splice-driven', 'Frameshift-
driven']")[['anchor','other','aa_loss','aa_gain']].to_csv(output/'cterm_mechanism_affected_le
n.tsv', sep='\t')

# %% Supplementary Figure S5: 2D scatter plot v2 frameshift-driven subcats

d1 = cterm_pblocks[cterm_pblocks['splice_subcat'] == 'Exon extension introduces \n
termination (EXIT)']
d2 = cterm_pblocks[cterm_pblocks['splice_subcat'] == 'Alternative terminal \n exon(s)
(ATE)']
d3 = cterm_pblocks[cterm_pblocks['splice_subcat'] == 'Alternative last exon \n in UTR (ALE
in UTR)']
d4 = cterm_pblocks[cterm_pblocks['splice_subcat'] == 'Poison exon inclusion']
d5 = cterm_pblocks[cterm_pblocks['splice_subcat'] == 'Cut-out splice terminal \n exon
(COSTE)']

fig, axes = plt.subplots(nrows=5, ncols=1, figsize=(6, 15))
colors = [(0.5048212226066897, 0.00392156862745098, 0.47021914648212226),
      (0.735840061514802, 0.061960784313725495, 0.5225682429834679),
(0.9094502114571319, 0.2894886582083814, 0.6086120722798923),
(0.9754555940023067, 0.5330257593233372, 0.6768935024990388),
(0.9859592464436755, 0.7293041138023837, 0.7404229142637447)]

sns.scatterplot(data=d1, x='aa_loss', y='aa_gain', marker='o', ax=axes[0], alpha=0.2,
          color=colors[0])
axes[0].set_title('Exon extension introduces termination', fontsize=30, pad=20)
axes[0].set_xlabel('Reference \n(amino acids)', fontsize=25)
axes[0].set_ylabel('Alternative \n(amino acids)', fontsize=25)
axes[0].set_xlim(0, 3000)
axes[0].set_ylim(0, 3000)
axes[0].grid(True, linestyle='--', linewidth=0.5)

sns.scatterplot(data=d2, x='aa_loss', y='aa_gain', marker='o', ax=axes[1], alpha=0.2,
          color=colors[1])
```

```
axes[1].set_title('Alternative terminal exon(s)', fontsize=30, pad=20)
axes[1].set_xlabel('Reference \n(amino acids)', fontsize=25)
axes[1].set_ylabel('Alternative \n(amino acids)', fontsize=25)
axes[1].set_xlim(0, 3000)
axes[1].set_ylim(0, 3000)
axes[1].grid(True, linestyle='--', linewidth=0.5)

sns.scatterplot(data=d3, x='aa_loss', y='aa_gain', marker='o', ax=axes[2], alpha=0.2,
        color=colors[2])
axes[2].set_title('Alternative last exon in UTR', fontsize=30, pad=20)
axes[2].set_xlabel('Reference \n(amino acids)', fontsize=25)
axes[2].set_ylabel('Alternative \n(amino acids)', fontsize=25)
axes[2].set_xlim(0, 3000)
axes[2].set_ylim(0, 3000)
axes[2].grid(True, linestyle='--', linewidth=0.5)

sns.scatterplot(data=d4, x='aa_loss', y='aa_gain', marker='o', ax=axes[3], alpha=0.2,
        color=colors[3])
axes[3].set_title('Poison exon inclusion', fontsize=30, pad=20)
axes[3].set_xlabel('Reference \n(amino acids)', fontsize=25)
axes[3].set_ylabel('Alternative \n(amino acids)', fontsize=25)
axes[3].set_xlim(0, 3000)
axes[3].set_ylim(0, 3000)
axes[3].grid(True, linestyle='--', linewidth=0.5)

sns.scatterplot(data=d5, x='aa_loss', y='aa_gain', marker='o', ax=axes[4], alpha=0.2,
        color=colors[4])
axes[4].set_title('Cut-out splice terminal exon', fontsize=30, pad=20)
axes[4].set_xlabel('Reference \n(amino acids)', fontsize=25)
axes[4].set_ylabel('Alternative \n(amino acids)', fontsize=25)
axes[4].set_xlim(0, 3000)
axes[4].set_ylim(0, 3000)
axes[4].grid(True, linestyle='--', linewidth=0.5)

plt.tight_layout()
plt.savefig(output/'cterm-rel-splice-driven-subcat-length-change_scatterplot.png',
dpi=900, facecolor=None, bbox_inches='tight')
plt.show()

# Output source data
#cterm_pblocks.query("splice_subcat in ['Exon extension introduces \n termination (EXIT)',
'Alternative terminal \n exon(s) (ATE)', 'Alternative last exon \n in UTR (ALE in UTR)', 'Poison
exon inclusion', 'Cut-out splice terminal \n exon
```

(COSTE)']")[['anchor','other','aa_loss','aa_gain']].to_csv(output/'cterm_mechanism_affected_len.tsv', sep='\t')

download_gencode_toy.sh

```
#!/bin/sh
#Author: Mayank Murali
#Project: Biosurfer

#Script to download GENCODE toy files from Zenodo (https://zenodo.org/record/7297008)

echo "================================================================="
echo "  Downloading GENCODE toy data ..."
echo "================================================================="

cd data
mkdir A_gencode_toy
cd A_gencode_toy

wget https://zenodo.org/records/10822882/files/biosurfer_gencode_toy_data.zip
unzip biosurfer_gencode_toy_data.zip
rm -rf __MACOSX biosurfer_gencode_toy_data.zip
```

download_gencode_v42.sh

```
#!/bin/sh
#Author: Mayank Murali
#Project: Biosurfer

#Script to download GENCODE v42 files from Zenodo (https://zenodo.org/record/7297008)

echo "================================================================="
echo "  Downloading GENCODE v42 data ..."
echo "================================================================="

cd data
mkdir A_gencode_v42
cd A_gencode_v42

wget https://zenodo.org/records/10822882/files/biosurfer_gencode_v42_data.zip
unzip biosurfer_gencode_v42_data.zip
rm -rf __MACOSX biosurfer_gencode_v42_data.zip
```

download_wtc11.sh

```
#!/bin/sh
#Author: Mayank Murali
#Project: Biosurfer

#Script to download PacBio WTC11 from Zenodo (https://zenodo.org/record/7297008)

echo "============================================================="
echo "  Downloading PacBio WTC11 data ..."
echo "============================================================="

mkdir A_wtc11
cd A_wtc11

wget https://zenodo.org/records/10822882/files/biosurfer_wtc11_data.zip
unzip biosurfer_wtc11_data.zip
rm -rf __MACOSX biosurfer_wtc11_data.zip
```

genome_wide_summary.py

```
# %% Importing libraries
from pathlib import Path
from re import M
import scipy.stats as stats
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import numpy as np
import csv
from plot_config import PBLOCK_COLORS, SECTION_COLORS, pblocks

# %% Output paths
output = Path('../C_altered_region_summary_plots')
output.mkdir(exist_ok=True)

# %% Fig2 panel A: Number of altered isoforms per gene vs number of genes
fig = plt.figure(figsize=(4, 2.4))
bins = list(range(1, 11)) + [100]
ax = sns.histplot(
  x = pd.cut(
    pblocks.groupby('anchor')['other'].nunique(),
    bins = bins,
    right = False,
```

```
      labels = [str(x) for x in bins[:-2]] + [f'{bins[-2]}+'],
    ),
    shrink = 0.75,
    color = '#888888',
    edgecolor = 'k',
    alpha = 1,
)
ax.set_xlabel('Number of alternative isoforms\nper gene')
ax.set_ylabel('Number of genes')
ax.set_ylim(0, 5000)
### output
fig.savefig(output/'alternative-isoforms-per-gene.png', dpi=200, facecolor=None,
bbox_inches='tight')

#Output source data
###
pblocks.groupby('anchor')['other'].nunique().to_frame(name='count').to_csv(output/'altern
ative-isoforms-per-gene-table.tsv', sep='\t')

# %% Fig2 panel B: Number of observed pblocks per alternative protein isoforms
fig = plt.figure(figsize=(4, 2.4))
ax = sns.histplot(
    x = pd.cut(
      pblocks.groupby(['anchor', 'other']).size(),
      bins = [1, 2, 3, 4, 5, 14],
      right = False,
      labels = ['1', '2', '3', '4', '5+']
    ),
    shrink = 0.75,
    color = '#888888',
    edgecolor = 'k',
    alpha = 1,
)
ax.set_xlabel('Number of altered regions\nper isoform')
ax.set_ylabel('Number of alternative\nprotein isoforms')
# ax.set_ylim(0, 30000)
#ax.ticklabel_format(axis='y', style='sci', scilimits=(0, 0))

### output
fig.savefig(output/'altered-regions-per-isoform.png', dpi=200, facecolor=None,
bbox_inches='tight')

#Output source data
### output
```

```
pblocks.groupby(['anchor',
'other']).size().to_frame(name='num_alt_regions').to_csv(output/'altered-regions-per-
isoform-table.tsv', sep='\t')

# %% Fig2 panel C: Distribution of lengths of the insertion, deletion and substituion
affected regions for proteins
aa_loss = pblocks[pblocks['pblock_category'].isin({'DELETION',
'SUBSTITUTION'})].reset_index()[['anchor','other','pblock_category', 'aa_loss']]
aa_loss['pblock_category'].replace('SUBSTITUTION', 'SUBSTITUTION (reference)',
inplace=True)
aa_loss.rename(columns={'aa_loss': 'length'}, inplace=True)
aa_gain = pblocks[pblocks['pblock_category'].isin({'INSERTION',
'SUBSTITUTION'})].reset_index()[['anchor','other','pblock_category', 'aa_gain']]
aa_gain['pblock_category'].replace('SUBSTITUTION', 'SUBSTITUTION (alternative)',
inplace=True)
aa_gain.rename(columns={'aa_gain': 'length'}, inplace=True)
affected_lengths = pd.concat([aa_loss, aa_gain])

binwidth = 50
xmax = 600
xtick = 200

fig = plt.figure(figsize=(5, 2))
data = affected_lengths[affected_lengths['pblock_category'] != 'SUBSTITUTION
(alternative)']
ax = sns.histplot(
   data = data,
   x = 'length',
   binwidth = binwidth,
   binrange = (0, xmax),
   stat = 'count',
   color = '#808080',
   alpha = 1,
)
ax.set_xlabel('Length of altered region (amino acids)')
ax.set_ylabel('Number of\naltered regions')
ax.ticklabel_format(axis='y', style='sci', scilimits=(-1, 1))
ax.vlines(data['length'].median(), *ax.get_ylim(), color='#b0b0b0', linestyle='-', linewidth=1)

### output
fig.savefig(output/'altered-region-affected-lengths.png', dpi=200, facecolor=None,
bbox_inches='tight')

#Output source data
```

```
### output
affected_lengths[affected_lengths['pblock_category'] != 'SUBSTITUTION
(alternative)'].to_csv(output/'altered-region-affected-lengths-table.tsv', sep='\t')

# %% Fig2 panel D: Distribution of the length of altered protein regions across the
annotated proteome
facets = sns.displot(
   data = affected_lengths,
   x = 'length',
   binwidth = binwidth,
   binrange = (0, xmax),
   stat = 'count',
   row = 'pblock_category',
   hue = 'pblock_category',
   palette = PBLOCK_COLORS,
   row_order = ('DELETION', 'INSERTION', 'SUBSTITUTION (reference)', 'SUBSTITUTION
(alternative)'),
   legend = False,
   alpha = 1,
   height = 2,
   aspect = 2.5
)
facets.set_xlabels('Length of altered region (amino acids)')
facets.set_ylabels('Number of\naltered regions')
for category, ax in facets.axes_dict.items():
   ax.set_title(category.capitalize())
   ax.set_xticks(range(0, xmax+1, xtick))
   ax.ticklabel_format(axis='y', style='sci', scilimits=(-1, 1))
   ax.vlines(affected_lengths[affected_lengths['pblock_category'] ==
category]['length'].median(), *ax.get_ylim(), color='#808080', linestyle='-', linewidth=1)

### output
facets.fig.savefig(output/'altered-region-affected-lengths-categories.png', dpi=200,
facecolor=None, bbox_inches='tight')

#Output source data
### output
affected_lengths.to_csv(output/'altered-region-affected-lengths-categories-table.tsv',
sep='\t')

# %%
#  T-test for the means of two independent samples of scores.
insertion = affected_lengths[affected_lengths['pblock_category'] == 'INSERTION']['length']
deletion = affected_lengths[affected_lengths['pblock_category'] == 'DELETION']['length']
```

```
substitution_ref = affected_lengths[affected_lengths['pblock_category'] == 'SUBSTITUTION (reference)']['length']
substitution_alt = affected_lengths[affected_lengths['pblock_category'] == 'SUBSTITUTION (alternative)']['length']

stats.mannwhitneyu(insertion, deletion) #MannwhitneyuResult(statistic=37940405.5, pvalue=0.0)
stats.mannwhitneyu(insertion, substitution_ref)
#MannwhitneyuResult(statistic=20566509.0, pvalue=0.0)
stats.mannwhitneyu(insertion, substitution_alt)
#MannwhitneyuResult(statistic=51275501.0, pvalue=1.1174708861070221e-07)

# Also alternative package
import pingouin as pg
pg.mwu(insertion, deletion, alternative='two-sided')
pg.mwu(insertion, substitution_ref, alternative='two-sided')

# Mann-Whitney U Test in R using stats library
# wilcox.test(insertion_data$V1, deletion_data$V1, alternative = "two.sided")
# data:  insertion_data$V1 and deletion_data$V1
# W = 37940406, p-value < 2.2e-16
# alternative hypothesis: true location shift is not equal to 0


# %% Fig2 panel I =: Substitution scatter plot
plt.figure(figsize=(4.8, 3.6))
ax = sns.histplot(
   data = pblocks[pblocks['pblock_category'] == 'SUBSTITUTION'],
   x = 'aa_gain',
   y = 'aa_loss',
   binwidth = binwidth/2,
   stat = 'count',
   color = PBLOCK_COLORS['SUBSTITUTION'],
   legend = False,
   cbar = True,
   cbar_kws = {
      'label': 'Number of regions',
   },
   alpha = 1,
)
ax.set_xlim(0, xmax)
ax.set_ylim(0, xmax)
ax.set_xticks(range(0, xmax+1, xtick))
ax.set_yticks(range(0, xmax+1, xtick))
```

```
ax.set_xlabel('Length of substitution region \nin alternative isoform (AA)')
ax.set_ylabel('Length of substitution region \nin reference isoform (AA)')
### output
plt.savefig(output/'substitution-reference-alternative-lengths.png', dpi=200,
facecolor=None, bbox_inches='tight')

#Output source data
### output
pblocks.query("pblock_category == 'SUBSTITUTION'")[['anchor',
'other','pblock_category','aa_gain','aa_loss']].to_csv(output/'substitution-reference-
alternative-lengths-table.tsv', sep='\t')

# %% Fig2 panel D: Pie chart
category_counts = pblocks['pblock_category'].value_counts()
total_pblocks = category_counts.sum()
fig, ax = plt.subplots()
wedges, texts, autotexts = plt.pie(
  category_counts,
  colors = category_counts.index.map(PBLOCK_COLORS),
  wedgeprops = {'width': 0.4},
  startangle = 180,
  counterclock = False,
  autopct = lambda x: f'{np.round(total_pblocks*x/100):.0f}\n({x:.0f}%)',
  pctdistance = 1.3,
)
for i, wedge in enumerate(wedges):
  wedge.set_edgecolor('k')
### output
fig.savefig(output/'altered-region-category-donut.png', dpi=200, facecolor=None,
bbox_inches='tight')

#Output source data
### output
pblocks['pblock_category'].value_counts().to_csv(output/'altered-region-category-donut-
table.tsv', sep='\t')

# %%
def get_section(nterm, cterm):
  if nterm and cterm:
    return 'Full-length'
  elif nterm:
    return 'N-terminal'
  elif cterm:
    return 'C-terminal'
```

```
    else:
        return 'Internal'

pblocks['protein_section'] = list(map(get_section, ~pblocks['nterm'].isna(),
~pblocks['cterm'].isna()))
pblock_sections = pblocks['protein_section'].value_counts()

fig, ax = plt.subplots(figsize=(6, 1))
left = 0
for section, color in SECTION_COLORS.items():
    val = pblock_sections[section]
    label = f'{val:g}\n({100*val/pblock_sections.sum():0.1f}%)'
    if section == 'Full-length':
        left += 5000
        label_type = 'edge'
        padding = 5
    else:
        label_type = 'center'
        padding = 0
    bar = plt.barh(
        [0],
        val,
        left = left,
        color = color,
        edgecolor = 'k',
        label = section,
    )
    plt.bar_label(bar, labels=[label], label_type=label_type, padding=padding)
    left = left + pblock_sections[section]
ax.legend(loc='upper left', bbox_to_anchor=(0, 0, 1, -0.1), ncols=2, frameon=False)
plt.axis('off')
### output
fig.savefig(output/'protein-section-counts.png', dpi=200, facecolor=None,
bbox_inches='tight')

#Output source data
### output
with open(output/'protein-section-counts-table.tsv', 'w', newline='') as file:
    writer = csv.DictWriter(file, fieldnames=SECTION_COLORS.keys(), delimiter='\t')
    writer.writeheader()
    writer.writerow(SECTION_COLORS)

# %% Identifying frameshift cases
pblocks[pblocks['cblocks'].str.count('FRAME') > 2]
```

install_biosurfer.sh

```
%%writefile create_conda_env.sh
#!/usr/bin/env bash
#Author: Mayank Murali
#Project: Biosurfer

#Script to download and install Biosufer from GitHub

# Clone the repository
git clone -b dev --single-branch https://github.com/sheynkman-lab/biosurfer.git

# Move to the folder
cd biosurfer

# Run setup
pip install --editable .
```

internal_summary.py

```
# %%
from pathlib import Path
from matplotlib.patches import Patch
from scipy.stats.contingency import chi2_contingency
from itertools import combinations

import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

from plot_config import PBLOCK_COLORS, SPLICE_EVENT_COLORS, pblocks

# %% Output paths
output = Path('../F_internal_summary_plots')
output.mkdir(exist_ok=True)

# %%
internal_pblocks = (
    pblocks[pblocks['internal']].
    drop(columns=[col for col in pblocks.columns if 'start' in col or 'stop' in col]).
    copy()
```

```
)
# convert string repr back to Python object
internal_pblocks['tblock_events'] = internal_pblocks['tblock_events'].map(eval)
internal_pblocks['events'] = internal_pblocks['events'].map(eval)

internal_subcats = pd.DataFrame(
  {
    'Frameshift': internal_pblocks['frameshift'],
    'Intron': internal_pblocks['tblock_events'].isin({('I',), ('i',)}),
    'Alt. donor': internal_pblocks['tblock_events'].isin({('D',), ('d',)}),
    'Alt. acceptor': internal_pblocks['tblock_events'].isin({('A',), ('a',)}),
    'Single exon': internal_pblocks['tblock_events'].isin({('E',), ('e',)}),
    'Compound': [True for _ in internal_pblocks.index]
  }
)
subcat_order = ('Single exon','Alt. acceptor','Alt. donor','Intron','Compound','Frameshift')
internal_pblocks['splice_event'] =
internal_subcats.idxmax(axis=1).astype(pd.CategoricalDtype(subcat_order,
ordered=True))

# %% Fig4 panel A: Internal splicing events frequencies
internal_pblocks_fig = plt.figure(figsize=(4.6, 3.8))
ax = sns.countplot(
  data = internal_pblocks.sort_values('pblock_category', ascending=True),
  y = 'splice_event',
  dodge = True,
  hue = 'pblock_category',
  palette = PBLOCK_COLORS,
  saturation = 1,
  edgecolor = 'k',
)
plt.legend(loc='center right', labels=['Deletions', 'Insertions', 'Substitutions'])
ax.set_xlabel('Number of altered internal regions')
ax.set_ylabel(None)
internal_pblocks_fig.savefig(output/'internal-events.png', dpi=500, facecolor=None,
bbox_inches='tight')

#Output source data
internal_pblocks[['splice_event','pblock_category']].to_csv(output/'internal-events-
table.tsv', sep='\t')

# %% Fig4 panel C: Proportion of each internal protein region that are ragged codons
internal_pblocks_ragged_fig = plt.figure(figsize=(4.6, 3.8))
ax = sns.countplot(
```

```
    data = internal_pblocks.sort_values('pblock_category', ascending=True),
    y = 'splice_event',
    palette = SPLICE_EVENT_COLORS,
    saturation = 1,
    edgecolor = 'k',
)
sns.countplot(
    ax = ax,
    data = internal_pblocks[internal_pblocks['split_codons']].sort_values('pblock_category',
ascending=True),
    y = 'splice_event',
    fill = False,
    edgecolor = 'k',
    hatch = '///',
)
# # # Creating the combined legend handles and labels
# handles = [
#     Patch(facecolor='w', edgecolor='k', hatch='///'),
#     Patch(facecolor='w', edgecolor='k')
# ]
# labels = ['Contains \nragged codons', 'Clean \ncodons']

# # # Creating the combined legend
# combined_legend = plt.legend(handles=handles, labels=labels, loc='center right')

# Adding the combined legend to the plot
plt.gca()
ax.set_xlabel('Number of altered internal regions')
ax.set_ylabel(None)
internal_pblocks_ragged_fig.savefig(output/'internal-events-ragged.png', dpi=700,
facecolor=None, bbox_inches='tight')

#Output source data
internal_pblocks[['splice_event','split_codons']].to_csv(output/'internal-events-ragged-
table.tsv', sep='\t')
# %%
alpha = 0.01
ragged_contingency = pd.crosstab(internal_pblocks['split_codons'],
internal_pblocks['splice_event'])
chi2, p_all, dof, expected = chi2_contingency(ragged_contingency)

ps = dict()
for event1, event2 in combinations(internal_subcats.columns, 2):
    sub_contingency = ragged_contingency[[event1, event2]]
```

```
    _, ps[event1, event2], _, _ = chi2_contingency(sub_contingency)

ps_sig = {k: p for k, p in ps.items() if p < alpha/len(ps)}
ps_insig = {k: p for k, p in ps.items() if k not in ps_sig}

# %%
nagnag_pblocks = internal_pblocks[(internal_pblocks['splice_event'] == 'Alt. acceptor') &
(internal_pblocks['length_change'].abs() == 1)]

# %% Fig4 panel B: Frequency of compound splicing events
internal_compound_pblocks = internal_pblocks[internal_pblocks['splice_event'] ==
'Compound'].copy()

internal_compound_subcats = pd.DataFrame(
    {
        'Multi-exon skipping': internal_compound_pblocks['events'] == frozenset('e'),
        'Exon skipping + \nalt. donor/acceptor': internal_compound_pblocks['events'].isin({
            frozenset(sorted('de')),
            frozenset(sorted('De')),
            frozenset(sorted('ea')),
            frozenset(sorted('eA')),
            frozenset(sorted('dea')),
            frozenset(sorted('Dea')),
            frozenset(sorted('deA')),
            frozenset(sorted('DeA')),
        }),
        'Mutually exclusive exons': internal_compound_pblocks['tblock_events'].isin({('E', 'e'),
('e', 'E')}),
        'Multi-exon inclusion': internal_compound_pblocks['events'] == frozenset('E'),
        'Alt. donor + alt. acceptor': internal_compound_pblocks['events'].isin({
            frozenset(sorted('ad')),
            frozenset(sorted('Ad')),
            frozenset(sorted('aD')),
            frozenset(sorted('AD')),
        }),
        'Exon inclusion + \nalt. donor/acceptor': internal_compound_pblocks['events'].isin({
            frozenset(sorted('dE')),
            frozenset(sorted('DE')),
            frozenset(sorted('Ea')),
            frozenset(sorted('EA')),
            frozenset(sorted('dEa')),
            frozenset(sorted('DEa')),
            frozenset(sorted('dEA')),
            frozenset(sorted('DEA')),
```

```
    }),
    'Other': [True for _ in internal_compound_pblocks.index]
  }
)
internal_compound_pblocks['compound_subcat'] =
internal_compound_subcats.idxmax(axis=1).astype(pd.CategoricalDtype(internal_compo
und_subcats.columns, ordered=True))

internal_pblocks_compound_fig = plt.figure(figsize=(3, 3))
ax = sns.countplot(
    data = internal_compound_pblocks,
    y = 'compound_subcat',
    palette = 'Greys_r',
    saturation = 1,
    edgecolor = 'k',
)
ax.set_xlabel('Number of altered\ninternal regions'),
ax.set_ylabel(None)
internal_pblocks_compound_fig.savefig(output/'internal-compound-events.png', dpi=200,
facecolor=None, bbox_inches='tight')

#Output source data
internal_compound_pblocks[['anchor','other','compound_subcat']].to_csv(output/'internal-
compound-events-table.tsv', sep='\t')



# %%
from scipy.stats import chi2_contingency

# Define the observed frequencies (counts) as a 2D array
observed = [[578, 829, 1806]]

# Perform the chi-square test
chi2, p, dof, expected = chi2_contingency(observed)

# Output the results
print(f"Chi-Square Statistic: {chi2}")
print(f"P-Value: {p}")
print(f"Degrees of Freedom: {dof}")
print("Expected Frequencies:")
print(expected)

# Set the significance level (alpha)
```

```
alpha = 0.05

# Determine whether to reject the null hypothesis
if p < alpha:
   print("Reject the null hypothesis: There is a significant association.")
else:
   print("Fail to reject the null hypothesis: There is no significant association.")

# %%
```

isoform_plotting.sh

```
%%writefile create_conda_env.sh
#!/usr/bin/env bash
#Author: Mayank Murali
#Project: Biosurfer

#Script to plot isoforms using Biosurfer
biosurfer plot -d gencode_toy --gene CRYBG2
```

n_termini_summary.py

```
# %%
from pathlib import Path
from matplotlib.patches import Patch
from scipy.stats import mannwhitneyu
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import matplotlib as mpl

from plot_config import NTERM_CLASSES, NTERM_COLORS, pblocks

# %% Output paths
output = Path('../D_nterm_summary_plots')
output.mkdir(exist_ok=True)

# %%
nterm_pblocks = pblocks[~pblocks['nterm'].isna() & (pblocks['nterm'] !=
'ALTERNATIVE_ORF') & (pblocks['cterm'].isna())].copy()
nterm_pblocks['nterm'].replace(NTERM_CLASSES, inplace=True)
nterm_pblocks['altTSS'] = nterm_pblocks['events'].apply(lambda x:
eval(x).intersection('BbPp')).astype(bool)
```

```
# %% Fig3 panel A (both Alt TSS and 5' UTR AS)
tss_fig = plt.figure(figsize=(5, 4))
ax = sns.countplot(
   data = nterm_pblocks,
   y = 'nterm',
   order = NTERM_COLORS.keys(),
   palette = NTERM_COLORS,
   edgecolor = 'k',
   saturation = 1,
)
sns.countplot(
   ax = ax,
   data = nterm_pblocks[nterm_pblocks['altTSS']],
   y = 'nterm',
   order = NTERM_COLORS.keys(),
   palette = NTERM_COLORS,
   edgecolor = 'k',
   fill = False,
   hatch = '//',
)
ax.legend(
   loc = (0, 1),
   frameon = False,
   handles = [Patch(facecolor='w', edgecolor='k', hatch='///'), Patch(facecolor='w',
edgecolor='k')],
   labels = ['Alternative transcription start site', '5\' UTR alternative splicing'],
)
ax.set_xlabel('Number of alternative isoforms')
ax.set_ylabel(None)
plt.savefig(output/'nterm-counts-all_mechanism.png', dpi=500, facecolor=None,
bbox_inches='tight')

#Output source data
nterm_pblocks.query("nterm in ['Mutually exclusive starts (MSX)', 'Shared downstream
start (SDS)']")[['anchor','other','nterm','altTSS']].to_csv(output/'nterm-counts-
all_mechanism.tsv', sep='\t')

# %% Fig3 panel C: MXS vs SDS scatterplot
font = {
   'family': 'sans-serif',
   'sans-serif': ['Arial'],
   'weight': 'normal',
   'size': 10
}
```

```
mpl.rc('font', **font)

# Filter the dataframe for 'Mutually exclusive starts (MXS)' and 'Shared downstream start
(SDS)'
msx_data = nterm_pblocks[nterm_pblocks['nterm'] == 'Mutually exclusive starts (MSX)']
sds_data = nterm_pblocks[nterm_pblocks['nterm'] == 'Shared downstream start (SDS)']
fig, axes = plt.subplots(nrows=2, ncols=1, figsize=(5.5, 5.5))
msx_color = (0.565498, 0.84243, 0.262877)
sds_color = (0.20803, 0.718701, 0.472873)

sns.scatterplot(data=msx_data, x='aa_loss', y='aa_gain', marker='.', ax=axes[0], alpha=0.2,
        color=msx_color)
axes[0].set_title('Mutually exclusive starts (MXS)', fontsize=11)
axes[0].set_xlabel('Reference \n(amino acids)', fontsize=10)
axes[0].set_ylabel('Alternative \n(amino acids)', fontsize=10)
axes[0].set_xlim(0, 2000)
axes[0].set_ylim(0, 2000)
axes[0].set_aspect('equal')
axes[0].grid(True, linestyle='--', linewidth=0.5)

sns.scatterplot(data=sds_data, x='aa_loss', y='aa_gain', marker='.', ax=axes[1], alpha=0.2,
color=sds_color)
axes[1].set_title('Shared downstream start (SDS)', fontsize=11)
axes[1].set_xlabel('Reference \n(amino acids)', fontsize=10)
axes[1].set_ylabel('Alternative \n(amino acids)', fontsize=10)
axes[1].set_xlim(0, 2000)
axes[1].set_ylim(0, 2000)
axes[1].set_aspect('equal')
axes[1].grid(True, linestyle='--', linewidth=0.5)

plt.tight_layout()

# Save plot
plt.savefig(output/'nterm-rel-length-change_scatterplot.png', dpi=600, facecolor=None,
bbox_inches='tight')
plt.show()

#Output source data
nterm_pblocks.query("nterm in ['Mutually exclusive starts (MSX)', 'Shared downstream
start
(SDS)']")[['anchor','other','aa_loss','aa_gain']].to_csv(output/'nterm_mechanism_affected_le
n.tsv', sep='\t')
```

plot_config.py

```
import colorsys

import matplotlib as mpl
import matplotlib.colors as mc
import matplotlib.font_manager as fm
import pandas as pd
from seaborn import color_palette

# Setting configurations for plotting
# for font_path in fm.findSystemFonts():
#     fm.fontManager.addfont(font_path)

font = {
   'family': 'sans-serif',
   'sans-serif': ['Arial'],
   'weight': 'normal',
   'size': 16
}
mpl.rc('font', **font)

# from https://stackoverflow.com/a/49601444
def adjust_lightness(color, amount=0.5):
   try:
      c = mc.cnames[color]
   except:
      c = color
   c = colorsys.rgb_to_hls(*mc.to_rgb(c))
   cnew = colorsys.hls_to_rgb(c[0], max(0, min(1, amount * c[1])), c[2])
   return mc.to_hex(cnew)

PBLOCK_COLORS = {
   'DELETION': '#f800c0',
   'INSERTION': '#00c0f8',
   'SUBSTITUTION': '#f8c000',
}

PBLOCK_COLORS['SUBSTITUTION (reference)'] =
adjust_lightness(PBLOCK_COLORS['SUBSTITUTION'], 1)
PBLOCK_COLORS['SUBSTITUTION (alternative)'] =
adjust_lightness(PBLOCK_COLORS['SUBSTITUTION'], 1)

SECTION_COLORS = {
```

```
    'N-terminal': color_palette('pastel')[2],
    'Internal': color_palette('pastel')[7],
    'C-terminal': color_palette('pastel')[3],
    'Full-length': 'none',
}

NTERM_CLASSES = {
    'MUTUALLY_EXCLUSIVE': 'Mutually exclusive starts (MSX)',
    'DOWNSTREAM_SHARED': 'Shared downstream start (SDS)',
    'UPSTREAM_SHARED': 'Shared upstream start (SUS)',
    'MUTUALLY_SHARED': 'Mutually shared starts (MSS)'
}
NTERM_COLORS = dict(zip(
    NTERM_CLASSES.values(),
    color_palette('viridis_r', n_colors=len(NTERM_CLASSES)+1)[:-1]
))

SPLICE_EVENT_COLORS = {
    'Intron': '#EBA85F',
    'Single exon': '#649FD2',
    'Alt. donor': '#86BB6F',
    'Alt. acceptor': '#A26FBB',
    'Compound': '#888888',
    'Frameshift': '#F7D76E',
}

CTERM_CLASSES = {
    'SPLICING' :  'Splice-driven',
    'FRAMESHIFT' : 'Frameshift-driven',
}
cterm_splice_palette = color_palette('RdPu_r', n_colors=6)
cterm_frameshift_palette = color_palette('YlOrRd_r', n_colors=5)
CTERM_PALETTE = [cterm_splice_palette[0], cterm_frameshift_palette[0]]

# # GENCODE v42
pblocks = pd.read_csv('../B_hybrid_aln_gencode_v42/pblocks.tsv', sep='\t')
# # WTC11
#pblocks = pd.read_csv('../B_hybrid_aln_wtc11/pblocks.tsv', sep='\t')
```