Data Biology: A quantitative exploration of gene regulation and underlying mechanisms

by

Benjamin J Schiller

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of DOCTOR OF PHILOSOPHY in Biochemistry and Molecular Biology in the GRADUATE DIVISION of the UNIVERSITY OF CALIFORNIA, SAN FRANCISCO
Data Biology: A quantitative exploration of gene regulation and underlying mechanisms

by

Benjamin J Schiller

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Biochemistry and Molecular Biology

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, SAN FRANCISCO

Approved: 

Chair

Committee in Charge
Acknowledgements

To all my family, friends, companions, labmates, classmates, colleagues, collaborators, advisers and mentors from the last five years: I cannot adequately describe how much I have learned from all of you. Pursuing my doctorate has been a transformative experience. I am indebted to everyone who assisted me and I am confident that the lessons I have learned will guide my actions for many years to come.
Certain chapters of this dissertation are a reprint of material from other publications. Chapter 2 contains portions adapted from [1]. Detailed author contributions for Chapter 2 are listed in Section 2.5 on page 39. Chapter 3 contains portions adapted from a manuscript in preparation. Detailed author contributions for Chapter 3 are listed in Section 3.8 on page 100. Chapter 5 contains portions adapted from [2]. Detailed author contributions for Chapter 5 are listed in Section 5.6 on page 147.
Understanding gene regulation in the era of deep sequencing

Benjamin J. Schiller

Abstract

Regulation of gene expression is a fundamental biological process required to adapt the full set of hereditary information (i.e., the genome) to the varied environments that any organism encounters. Here, we elucidate two distinct forms of gene regulation – of endogenous genes by binding of transcription factors to information-containing genomic sequences and of selfish genes (“transposons”) by targeting of small RNAs to repetitive genomic sequences – using a wide array of approaches.

To study regulation by transcription factors, we used glucocorticoid receptor (GR), a hormone-activated, DNA-binding protein that controls inflammation, metabolism, stress responses and other physiological processes. In vitro, GR binds as an inverted dimer to two imperfectly palindromic “half sites” separated by a “spacer”. Moreover, GR binds different sequences with distinct conformations, as demonstrated by nuclear magnetic resonance spectroscopy (NMR) and other biophysical methods.

In vivo, GR employs different functional surfaces when regulating different genes. We investigated whether sequences bound by GR in vivo might be a composite of several motifs, each biased toward utilization of a particular pattern of functional surfaces of GR. Using microarrays and deep sequencing, we characterized gene expression and genomic occupancy by GR, with and without glucocorticoid treatment, of cells expressing GR alleles bearing differences in three known functional surfaces. We found a “sub-motif”, the GR “half site”, that relates to utilization of the dimerization interface and directs genomic binding by GR in a distinct conformation.
To study repression of transposons, we characterized the production and function of small RNAs in the yeast *Cryptococcus neoformans*. We found that target transcripts are distinguished by suboptimal introns and inefficient splicing. We identified a complex, SCANR, required for synthesis of small RNAs and demonstrate that it physically associates with the spliceosome. We propose that recognition of gene products by SCANR is in kinetic competition with splicing, thereby further promoting small RNA production from target transcripts.

To achieve these results, we developed new bioinformatics tools: twobitreader, a small Python package for efficient extraction of genomic sequences; scripter, a flexible back-end for easily creating scripts and pipeline; and seriesoftubes, a pipeline built upon scripter for the analysis of deep sequencing data.
## Contents

Acknowledgements iii

Abstract v

Table of Contents vii

List of Figures xvi

List of Tables xx

List of Source Code xxii

1 Introduction 1

2 The glucocorticoid receptor dimer interface allosterically transmits sequence-specific DNA signals 4

2.1 Background 5

2.2 Results 7

2.2.1 GBS spacer affects GR occupancy, activity and structure 7

2.2.2 D-loop conformation depends on spacer but not half-site 9

2.2.3 The A477T mutation disrupts D-loop conformation 13

2.2.4 A477T affects cooperativity but not stoichiometry 16

2.2.5 A477T disrupts signaling between dimer partners 23
2.3 Discussion .................................................. 28
2.4 Methods .................................................... 33
  2.4.1 Protein expression and purification ..................... 33
  2.4.2 Protein-DNA complex formation ....................... 34
  2.4.3 Protein nuclear magnetic resonance assignment .......... 35
  2.4.4 Chemical-shift difference and ZZ-exchange nuclear magnetic resonance ......................... 36
  2.4.5 Transcriptional reporter assays ....................... 36
  2.4.6 Surface plasmon resonance .......................... 37
  2.4.7 Electrophoretic mobility shift assay ................... 38
  2.4.8 GR chromatin immunoprecipitation sequencing .......... 38
  2.4.9 DNA oligonucleotide sequences ....................... 39
2.5 Chapter Acknowledgements ............................... 39

3 Glucocorticoid receptor binds half sites to induce a distinct conformational state and regulate specific target genes 41
3.1 Background .................................................. 42
3.2 Results ...................................................... 45
  3.2.1 Different GR alleles induce diverse, gene-specific transcriptional responses ....................... 45
  3.2.2 A477T and GRγ selectively occupy GBRs near genes that gain activation .......................... 49
  3.2.3 Most A477T-selectively occupied GBRs contain a half site motif ............................... 52
  3.2.4 A subset of genes regulated by GRα contains only half sites .................................. 56
  3.2.5 GRα binds half sites with a distinct conformation compared to full sites ....................... 59
3.7.2 Additional file 2: Genome-wide description of GR binding regions (GBRs) ................................................................. 78
3.7.3 Additional file 3: Assignments of GBRs to genes (GREAT output) 79
3.7.4 Additional files 4 – 6: MACS2 analysis of differential occupancy between GRα and mutants ........................................ 79
3.7.5 Additional file 7: GBR positions and strengths of occupancy at genes that lost activation with A477T ............................ 80
3.7.6 Additional file 8: GBR positions and strengths of occupancy at genes that gained activation with A477T ....................... 80
3.7.7 Additional file 9: GRα and A477T selectively activate transcription with transcriptional reporters containing selectively occupied GBRs 80
3.7.8 Additional file 10: Motif analysis of selectively and nonselectively occupied GBRs .................................................... 83
3.7.9 Additional file 11: GBR positions and strength of occupancy at nonselectively activated genes ................................. 85
3.7.10 Additional file 12: Motif information at GBRs occupied by GRα or A477T ................................................................. 85
3.7.11 Additional files 13 – 15: Pairwise comparisons of GRα GBRs and GBRs of mutant alleles ............................................. 85
3.7.12 Additional file 16: R script for analysis of gene expression microarray data (text file) ................................................ 87
3.7.13 Additional file 17: R script for GBR analysis (text file) .......... 93
3.7.14 Additional file 18: Table of primers used for reporters (.xls) ... 100
3.8 Chapter Acknowledgements ................................................. 100

4 Perspective: Glucocorticoid receptor activity is modulated by a precise
interplay between binding sequence and structural conformation, dynamics

5 Stalled spliceosomes are a signal for RNAi-mediated genome defense 106

5.1 Introduction ................................................................. 107

5.2 Results ................................................................. 109

5.2.1 Endogenous siRNA of C. neoformans targets transposon mRNA precursors ......................................................... 109

5.2.2 Identification and characterization of proteins associated with RNA-dependent RNA polymerase .................................. 112

5.2.3 SCANR associates with the spliceosome ................................ 117

5.2.4 RNAi target transcripts display high spliceosome occupancy ... 119

5.2.5 Mutations that stall splicing trigger RNAi ................................ 123

5.2.6 Entry into the splicing pathway is required for RNAi ............ 126

5.2.7 Splicing intermediates may be a preferred substrate for siRNA biogenesis ............................................................... 127

5.3 Discussion ............................................................. 128

5.3.1 Intron-containing mRNA precursors are a template for siRNA in C. neoformans ......................................................... 130

5.3.2 Spliceosomal stalling of transposon transcripts suggests a kinetic competition .............................................................. 131

5.3.3 The inefficient splicing of pre-mRNAs that contain transposons . 132

5.4 Methods ................................................................. 134

5.4.1 Yeast strains .............................................................. 134

5.4.2 Tandem affinity protein purification .................................... 134

5.4.3 RNA immunoprecipitation ............................................. 135
# Table of Contents

6.1 License and Warranty .............................................. 154
6.2 Source Code ...................................................... 154
6.3 twobitreader module .............................................. 154
6.4 twobitreader.download ............................................ 157
6.5 Run module as script ............................................. 157
   6.5.1 twobitreader ................................................ 157
   6.5.2 twobitreader.download ..................................... 158

7 scripter: a tool for parallel execution of functions on many files 159
   7.1 License and Warranty .............................................. 159
   7.2 Source Code ...................................................... 159
   7.3 Philosophy ....................................................... 160
   7.4 Setting up the Environment .................................... 160
   7.5 Attaching a FilenameParser .................................... 160
   7.6 Defining an action .............................................. 161
   7.7 Modifying the script options .................................. 162
   7.8 Dealing with errors/exceptions ............................... 162
   7.9 Logging .......................................................... 163
   7.10 Default command-line options ............................... 163
   7.11 scripter API .................................................... 164
      7.11.1 Classes .................................................... 164
      7.11.2 Exceptions ................................................ 166
      7.11.3 Decorators ................................................ 167
      7.11.4 Functions .................................................. 167

8 seriesoftubes: An extended pipeline for Solexa ChIP-seq data 169
   8.1 License and Warranty .............................................. 169
A twobitreader Source Code

A.1 twobitreader module ........................................ 200
A.2 twobitreader.download module ............................... 214
A.3 twobitreader Unit Tests ...................................... 215

B scripter Source Code ............................................. 219

B.1 scripter module .................................................. 219

C seriesoftubes Source Code ..................................... 234

C.1 seriesoftubes scripts ......................................... 234
  C.1.1 preprocess_reads.py .................................. 234
  C.1.2 align2.py .............................................. 241
  C.1.3 call_peaks.py ......................................... 252
  C.1.4 analyze.py ............................................. 256
  C.1.5 align.py (deprecated) .................................. 257
  C.1.6 subtract_bam.py (deprecated) ......................... 268

C.2 seriesoftubes package ........................................ 270
  C.2.1 seriesoftubes.bedgraph module ....................... 270
  C.2.2 seriesoftubes.cPreprocess module (Cython extension) .... 272

C.3 seriesoftubes.converters package .......................... 284
  C.3.1 seriesoftubes.converters.discover module ......... 284
  C.3.2 seriesoftubes.converters.cat module ............... 286
  C.3.3 seriesoftubes.converters.anytotab module ....... 287
  C.3.4 seriesoftubes.converters.bamtofastq module ...... 288
  C.3.5 seriesoftubes.converters.bamtofastq2 module ..... 291
<table>
<thead>
<tr>
<th>Section</th>
<th>Module/Package</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.3.6</td>
<td>seriesoftubes.converters.fastqtotab module</td>
<td>295</td>
</tr>
<tr>
<td>C.4</td>
<td>seriesoftubes.fnparsers package</td>
<td>296</td>
</tr>
<tr>
<td></td>
<td>C.4.1 seriesoftubes.fnparsers module</td>
<td>296</td>
</tr>
<tr>
<td></td>
<td>C.4.2 seriesoftubes.fnparsers.bowtie module</td>
<td>297</td>
</tr>
<tr>
<td></td>
<td>C.4.3 seriesoftubes.fnparsers.illumina module</td>
<td>299</td>
</tr>
<tr>
<td></td>
<td>C.4.4 seriesoftubes.fnparsers.aligned module</td>
<td>301</td>
</tr>
<tr>
<td></td>
<td>C.4.5 seriesoftubes.fnparsers.peaks module</td>
<td>302</td>
</tr>
<tr>
<td>C.5</td>
<td>seriesoftubes.tubes package</td>
<td>304</td>
</tr>
<tr>
<td></td>
<td>C.5.1 seriesoftubes.tubes module</td>
<td>304</td>
</tr>
<tr>
<td></td>
<td>C.5.2 seriesoftubes.tubes.polledpipe module</td>
<td>305</td>
</tr>
<tr>
<td>C.6</td>
<td>bioplus package</td>
<td>307</td>
</tr>
<tr>
<td></td>
<td>C.6.1 bioplus.fasta module</td>
<td>307</td>
</tr>
<tr>
<td></td>
<td>C.6.2 bioplus.genometools module</td>
<td>309</td>
</tr>
<tr>
<td></td>
<td>C.6.3 bioplus.motif module</td>
<td>312</td>
</tr>
<tr>
<td></td>
<td>C.6.4 bioplus.peaktools module</td>
<td>320</td>
</tr>
<tr>
<td></td>
<td>C.6.5 bioplus.seqtools module</td>
<td>321</td>
</tr>
<tr>
<td></td>
<td>C.6.6 bioplus.sitefinder module</td>
<td>325</td>
</tr>
<tr>
<td></td>
<td>C.6.7 bioplus.tabfile module</td>
<td>331</td>
</tr>
<tr>
<td></td>
<td>C.6.8 bioplus.wrappers module</td>
<td>347</td>
</tr>
</tbody>
</table>

References 349
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Nonspecific GBS bases modulate GR structure and activity</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>NMR assignment of the DNA-bound GR DBD complex</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>GBS spacer affects the conformation of the D-loop</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>NMR chemical shift difference analysis of GBSs that differ in the half-site or spacer sequence</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>Disruption of the dimerization interface affects lever arm and DNA-recognition helix conformation</td>
<td>15</td>
</tr>
<tr>
<td>2.6</td>
<td>DNA-bound A477T conformation differs from that of unbound WT DBD. The overlay of $^{15}$N-HSQC spectra of WT and A477T DBD bound to the Fkbp5 GBS, compared to the unbound DBD (WT Apo) at 35°C.</td>
<td>17</td>
</tr>
<tr>
<td>2.7</td>
<td>A477T impairs dimerization but not monomer DNA binding</td>
<td>19</td>
</tr>
<tr>
<td>2.8</td>
<td>A477T disrupts cooperativity and GBS-specific dissociation</td>
<td>21</td>
</tr>
<tr>
<td>2.9</td>
<td>Chemical shift difference between WT DBD – GBS complexes with different levels of transcriptional induction by GR</td>
<td>24</td>
</tr>
<tr>
<td>2.10</td>
<td>Conformational exchange within the lever arm and dimer interface</td>
<td>26</td>
</tr>
<tr>
<td>2.11</td>
<td>Sequence-specific lever-arm conformation is dependent on the intact dimerization interface</td>
<td>27</td>
</tr>
<tr>
<td>2.12</td>
<td>NMR chemical shift variance across seven different GBS complexes</td>
<td>29</td>
</tr>
<tr>
<td>2.13</td>
<td>Comparison of sequence-specific DNA bend as determined by FRET assay</td>
<td>31</td>
</tr>
</tbody>
</table>
3.1 GR alleles used in this study ................................................. 44
3.2 Glucocorticoid regulation occurs with gene-specific utilization of GR functional surfaces ......................................................... 47
3.3 Differential regulation analysis reveals distinct classes of regulated genes ................................................................. 48
3.4 Location, strength of occupancy and inferred activity of GBRs compared pairwise between GRα and mutants ............................................. 50
3.5 Both A477T and GRα occupy "half site" GBRs in vivo ...................... 53
3.6 GR half sites and full sites are similar but distinct motifs .................. 58
3.7 GR half sites predominate at GBRs near nonselective, activated genes . 59
3.8 GRα and A477T bind a half site in similar conformations .................... 62
3.9 See Additional file 7 in Section 3.7.5 ........................................ 81
3.10 See Additional file 8 in Section 3.7.6 ...................................... 82
3.11 See Additional file 9 in Section 3.7.7 ....................................... 84
3.12 See Additional file 11 in Section 3.7.9 ...................................... 86

5.1 Endogenous siRNA of C. neoformans ..................................... 111
5.2 Functional characterization of Rdpl-associated proteins ................. 112
5.3 PRSC and SCANR subunits are physically and functionally distinct .... 116
5.4 SCANR physically associates with Srr1 and the spliceosome ............ 117
5.5 Transcripts targeted by RNAi display suboptimal splicing features and are stalled on spliceosomes in vivo ........................................ 120
5.6 Stalled spliceosomes promote RNAi in a manner that depends on entry into the splicing pathway ................................................. 125
5.7 Lariat debranchase is required for siRNA production ..................... 128
5.8 Characterization of Ago1 cytoplasmic foci, Related to Figure 5.2 ........ 149
5.9 Co-immunoprecipitation of SCANR subunit Rdp1 and spliceosome component Syf1, Related to Figure 5.4 .......................... 150
5.10 Spliceosome occupancy of RNAi target transcripts in the absence of siRNA, Related to Figure 5.5 ................................. 151
5.11 Stalled spliceosomes promote siRNA production by Rdp1, Related to Figure 5.6 ................................................... 152
List of Tables

2.1 Summary of SPR fit parameters for WT and A477T DBD binding to
GBS surfaces at 35°C or 8°C. Dissociation constant, $K_{1/2}$, and Hill co-
efficient, $n_H$, were calculated from equilibrium binding isotherms fit to a
the Hill equation. Fitting of the dissociation phases of the SPR traces by
conventional models gave systematic residuals consistent with a biphasic-
sequential dissociation process, likely resulting from the complex kinetics
and strong cooperativity of this system. Instead, the stabilities of the
different complexes were reliably determined by a standard exponential de-
cay model, and $t_{1/2}$ values were extrapolated from the apparent off-rates.
Errors are the mean ± two s.d. from 2-3 replicates. .......................... 20

2.2 Correlation between transcriptional activity and DNA binding affinity
($K_{1/2}$) or half-life ($t_{1/2}$). Correlations for WT and A477T were calculated
for seven different GBS complexes. Because this relationship is expected
to be non-linear [30], a statistical analysis with non-linear fitting of data
are presented. .................................................................................. 23

3.1 Occurrences of STAT and AP1 motifs found using DREME ............ 58

5.1 Non-essential Supplementary Tables ............................................. 147
5.2 Frequency of convergently oriented genes among RNAi target genes, Related to Figure 5.1 .......................... 149
5.3 siRNAs mapping to genes in wild-type C. neoformans, Related to Figure 5.1 149
5.4 Quantitative yeast two-hybrid results, Related to Figure 5.2 ............... 150
5.5 Strains used in this study, Related to Experimental Procedures .......... 150
5.6 Primers used in this study, Related to Experimental Procedures .......... 150
5.7 Proteins removed from mass spectrometry results by background filtering criteria, Related to Experimental Procedures ......................... 150
# List of Source Code

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>See Additional file 16 in Section 3.7.12</td>
<td>87</td>
</tr>
<tr>
<td>3.2</td>
<td>See Additional file 17 in Section 3.7.13</td>
<td>94</td>
</tr>
<tr>
<td>5.1</td>
<td>Scripts for siRNA read alignment</td>
<td>137</td>
</tr>
<tr>
<td>A.1</td>
<td>twobitreader module</td>
<td>200</td>
</tr>
<tr>
<td>A.2</td>
<td>twobitreader.download module</td>
<td>214</td>
</tr>
<tr>
<td>A.3</td>
<td>twobitreader Unit Tests</td>
<td>215</td>
</tr>
<tr>
<td>B.1</td>
<td>scripter module</td>
<td>219</td>
</tr>
<tr>
<td>C.1</td>
<td>preprocess_reads.py</td>
<td>234</td>
</tr>
<tr>
<td>C.2</td>
<td>align2.py</td>
<td>241</td>
</tr>
<tr>
<td>C.3</td>
<td>call_peaks.py</td>
<td>252</td>
</tr>
<tr>
<td>C.4</td>
<td>analyze.py</td>
<td>256</td>
</tr>
<tr>
<td>C.5</td>
<td>align.py (deprecated)</td>
<td>257</td>
</tr>
<tr>
<td>C.6</td>
<td>subtract_bam.py (deprecated)</td>
<td>268</td>
</tr>
<tr>
<td>C.7</td>
<td>seriesoftubes.bedgraph module</td>
<td>271</td>
</tr>
<tr>
<td>C.8</td>
<td>seriesoftubes.cPreprocess module (Cython extension)</td>
<td>272</td>
</tr>
<tr>
<td>C.9</td>
<td>seriesoftubes.converters.discover module</td>
<td>284</td>
</tr>
<tr>
<td>C.10</td>
<td>seriesoftubes.converters.cat module</td>
<td>286</td>
</tr>
<tr>
<td>C.11</td>
<td>seriesoftubes.converters.anytotab module</td>
<td>287</td>
</tr>
<tr>
<td>C.12</td>
<td>seriesoftubes.converters.bamtofastq module</td>
<td>288</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>C.13</td>
<td>seriesoftubes.converters.bamtofastq2 module</td>
<td>291</td>
</tr>
<tr>
<td>C.14</td>
<td>seriesoftubes.converters.fastqtotab module</td>
<td>295</td>
</tr>
<tr>
<td>C.15</td>
<td>seriesoftubes.fnparsers module</td>
<td>296</td>
</tr>
<tr>
<td>C.16</td>
<td>seriesoftubes.fnparsers.bowtie module</td>
<td>297</td>
</tr>
<tr>
<td>C.17</td>
<td>seriesoftubes.fnparsers.illumina module</td>
<td>300</td>
</tr>
<tr>
<td>C.18</td>
<td>seriesoftubes.fnparsers.aligned module</td>
<td>301</td>
</tr>
<tr>
<td>C.19</td>
<td>seriesoftubes.fnparsers.peaks module</td>
<td>302</td>
</tr>
<tr>
<td>C.20</td>
<td>seriesoftubes.tubes module</td>
<td>304</td>
</tr>
<tr>
<td>C.21</td>
<td>seriesoftubes.tubes.polledpipe module</td>
<td>305</td>
</tr>
<tr>
<td>C.22</td>
<td>bioplus.fasta module</td>
<td>307</td>
</tr>
<tr>
<td>C.23</td>
<td>bioplus.genometools module</td>
<td>309</td>
</tr>
<tr>
<td>C.24</td>
<td>bioplus.motif module</td>
<td>312</td>
</tr>
<tr>
<td>C.25</td>
<td>bioplus.peaktools module</td>
<td>320</td>
</tr>
<tr>
<td>C.26</td>
<td>bioplus.seqtools module</td>
<td>321</td>
</tr>
<tr>
<td>C.27</td>
<td>bioplus.sitefinder module</td>
<td>325</td>
</tr>
<tr>
<td>C.28</td>
<td>bioplus.tabfile module</td>
<td>331</td>
</tr>
<tr>
<td>C.29</td>
<td>bioplus.wrappers module</td>
<td>347</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Recent advances in our understanding of gene regulation have been driven largely by rapidly advancing technology for sequencing DNA. A mere decade since the human genome was first sequenced [3] over the course of five years and at a cost of more than a hundred million dollars, the throughput of DNA sequencing has advanced to the point where a human genome may be sequenced in under two weeks and at a cost of less than ten thousand dollars – a more than 100-fold reduction in both price and time! The revolutionary technology of “deep sequencing” has been made possible by the marriage of the polymerase chain reaction (PCR) with advanced microscopy, using fluorescence-labeled nucleotides, yielding massively parallel sequences for large numbers of clonally amplified molecules (“clusters”). In equal part, this revolution in molecular biology rests upon computational technology and methods that enable processing of “Big Data”. These methods have transformed bioinformatics from an isolated discipline to a set of powerful tools and methods that are used by all biologists. This new “data biology” has become a basic underpinning of current and future progress.

We employed diverse methods in bioinformatics, biochemistry and molecular biology to
yield new insights into diverse biological processes including gene regulation by the glucocorticoid receptor (GR) – a hormone-activated, DNA-binding transcriptional regulatory factor that controls inflammation, metabolism, stress responses and other physiological processes – and suppression of transposons – selfish, mobile genetic elements – in the yeast Cryptococcus neoformans. In this dissertation, I describe in detail the applications and methods of new computational tools that leverage sequencing data to improve not only our understanding of biology but also the scope of questions we are now able to pose.

Previous work by Rogatsky et al. [4] and Meijsing et al. [5] show that GR utilizes distinct functional surfaces in the regulation of different genes and that it binds to different GBSs in different conformations. These results together support the notion that GR assumes different conformations at bound sequences in vivo. Chapter 2 describes measurements of GR strutural conformation and dynamics using a variety of biophysical approaches, including nuclear magnetic resonance spectroscopy (NMR), of GR. The results confirm that GR conformation and dynamics are modulated in response to binding different GR binding sequences (GBSs), and further show that the dimerization interface of GR is structurally coupled to its DNA recognition helix.

In Chapter 3, we examined whether sequence motifs, including the canonical GBS, found within genomic binding regions occupied by GR might be a composition of several specific motifs, each biased toward utilization of particular functional surface of GR. We present measurements of gene expression and genomic occupancy of isogenic cultured human cell lines, with and without glucocorticoid treatment, expressing alleles of GR bearing differences in particular functional surfaces. We found differences in the sequence motifs that are present at GREs, and we describe a particular “sub-motif” that relates to utilization of the dimerization interface. This “sub-motif”, the GR “half site”, directs genomic binding by GR in a distinct conformation, most likely as a monomer. These results provide
insights into the evolution of transcription networks and combinatorial regulation, and may contribute to predictive models for GR activity and therapy.

Chapter 5 describes a mechanism by which transposons – mobile selfish genomic elements – are targeted for RNAi-mediated genome defense in the yeast Cryptococcus neoformans. We showed that intron-containing mRNA precursors template siRNA synthesis. We identified a Spliceosome-Coupled And Nuclear RNAi (SCANR) complex required for siRNA synthesis and demonstrate that it physically associates with the spliceosome. Using bioinformatics, we found that RNAi target transcripts are distinguished by suboptimal introns and abnormally high occupancy on spliceosomes, likely due to inefficient splicing at these suboptimal introns. We propose that recognition of mRNA precursors by the SCANR complex is in kinetic competition with splicing, thereby promoting siRNA production from transposon transcripts stalled on spliceosomes.

Chapters 6, 7 and 8 describe a series of software projects that were independently developed to assist in the analysis of data described in the earlier chapters. twobitreader (Chapter 6) is a small Python package for efficient extraction of genomic sequences. scripter (Chapter 7) is a flexible back-end for easily creating scripts and pipelines. scripter was used a basis for developing other projects including seriesoftubes. seriesoftubes is a pipeline developed for the analysis of ChIP-seq data. Source code for these software projects are contained in A, B and C. All software/source code is licensed under Perl Artistic License 2.0. No warranty is provided, express or implied.
Chapter 2

The glucocorticoid receptor dimer interface allosterically transmits sequence-specific DNA signals

Abstract

Glucocorticoid receptor (GR) binds to genomic response elements and regulates gene transcription with cell and gene specificity. Within a response element, the precise sequence to which the receptor binds has been implicated in directing its structure and activity. Here, we use NMR chemical-shift difference mapping to show that nonspecific interactions with bases at particular positions in the binding sequence, such as those of the 'spacer', affect the conformation of distinct regions of the rat GR DNA-binding domain. These regions include the DNA-binding surface, the 'lever arm' and the dimerization interface, suggesting an allosteric pathway that signals between the DNA-binding sequence and the associated dimer partner. Disrupting this pathway by mutating the dimer interface alters sequence-specific conformations, DNA-binding kinetics and transcriptional activity. Our study demonstrates that GR
dimer partners collaborate to read DNA shape and to direct sequence-specific gene activity.

2.1 Background

Gene expression is tailored to the needs of specific tissues and in response to environmental and developmental changes. Transcriptional regulators coordinate this task by integrating input signals at specific genomic regions [6, 7] to effect precise transcriptional outputs at target genes. This intricate process relies on combinatorial control, in which distinct combinations of factors assemble into functional regulatory complexes that control the transcriptional activity of associated genes. However, the determinants that define the gene-specific assembly and activity of these regulatory complexes are poorly understood.

GR, a glucocorticoid-activated member of the nuclear receptor superfamily, utilizes combinatorial control to regulate hundreds to thousands of genes in a cell- and gene-specific manner. This specificity arises partly from context-dependent GR-binding regions (GBRs), which can be defined in vivo using genome-wide approaches. Some, but not all, GBRs appear to function in vivo as glucocorticoid response elements (GREs), which confer context-specific glucocorticoid regulation on nearby genes. Although GBR and GRE activities are clearly separable, both rely on the effects of multiple signals, such as hormonal ligands, other regulatory factors and post-translational modifications. Each of these signals drives distinct conformational changes in the receptor [8, 9, 10, 11, 12, 13], thereby modulating its transcriptional regulatory activity [14, 15, 16]. For example, two GR ligands, dexamethasone and mifepristone, differentially affect the formation of a coactivator interaction surface of the ligand-binding domain [13] and induce different transcriptional profiles.

GBRs and GREs are composite elements consisting of binding motifs for non-GR tran-
scriptional regulatory factors and, often, one or more GR binding sequences (GBSs) [17]. Purified GR binds GBSs with high affinity in vitro, and mutational studies have confirmed that GBSs within a particular GBR are responsible for GRE activity [17, 18] (S. Meijsing, personal communication). GBSs vary loosely around a 15-base-pair (15-bp) consensus sequence consisting of two hexameric half-sites separated by a 3-bp spacer [18]. GR binds to a GBS as a homodimer, and each dimer partner specifically contacts, at most, three bases in each GBS half-site. Structural studies of free and DNA-bound GR DNA-binding domain (DBD) suggest that DNA binding imparts structural changes in the second zinc finger of the DBD, forming the dimerization interface [19, 20, 21].

We have previously demonstrated that DNA-binding sequences serve as distinct signals that direct GR structure and activity [5, 22]. Crystallographic studies comparing GR bound to different GBSs have revealed alternate protein conformations that are dependent on the precise DNA-binding sequence [5]. The alternate conformations observed were localized to a loop region within the DBD termed the lever arm, which does not itself contact the DNA. Moreover, GBSs that produced different lever-arm conformations were invariant at all nucleotide positions that make direct contacts with GR, indicating that nonspecific bases affect GR structure. The presence of alternate lever-arm conformations suggests that GBS-specific conformational dynamics have a role in gene-specific regulation by GR.

Our previous crystallography studies motivated the following questions: (i) how does GR detect sequence differences among GBSs? (ii) Do GBSs drive distinct 'allosteric paths' of conformational changes that extend into and through the lever arm? (iii) How do GBS-dependent differences in GR conformation affect GR activity? To address these questions, we used solution techniques to assess the effects of changing nucleotides at specific positions within the GBS and of perturbing a functional surface of the GR DBD.
2.2 Results

2.2.1 GBS spacer affects GR occupancy, activity and structure

We sought to determine the degree of sequence variability among endogenous GBSs to estimate the potential for DNA sequences to be unique signals that produce distinct GR activities. We identified GR binding regions in human U2OS cells exogenously expressing full-length rat GR by GR chromatin immunoprecipitation followed by deep sequencing of the precipitated DNA fragments (ChIP-sequencing), (B.J.S., L.C.W. and K.R.Y., data not shown). An unbiased search for sequence motifs within 1,000 GR binding regions with the highest number of reads revealed a GBS motif composed of imperfect palindromic hexamers separated by a 3-bp spacer (Figure 2.1a), similar to motifs previously identified on the basis of smaller sample sets [17]. Scanning for this motif among the 30,000 observed GR binding regions revealed that 90% of GBSs are unique, suggesting that there is sufficient diversity for each to be a gene-specific signal. This model would require that nonspecific bases contribute to sequence specificity. The GBS positions with the highest information content (>1 bit) correspond to the six bases that are directly contacted by the GR dimer [21, 5]. The remaining nine nucleotide positions each contain <1 bit of information, with half-site positions 3 and 13 nearly devoid of sequence preference (0.05 and 0.1 bits, respectively). Notably, however, GR showed appreciable base preference at nucleotide positions that it does not contact directly: pyrimidines at spacer positions 7-9, as well as A and T at positions 6 and 10, respectively, adjacent to the spacer.

It was previously shown that GBSs differentially modulate GR transcriptional induction using luciferase reporters consisting of a single GBS upstream of a minimal promoter [5]. To investigate how varying the GBS spacer affects GR transcriptional induction, we compared reporter activity in the presence of 100 nM dexamethasone for GBSs that differ
Figure 2.1: Nonspecific GBS bases modulate GR structure and activity. (a) The GR binding motif identified by GR ChIP-sequencing in GR-expressing U2OS cells. Asterisks indicate specific bases directly contacted by GR. (b) Luciferase induction of single GBS reporters in U2OS cells treated with 100 nM dexamethasone or ethanol control. Error bars represent s.d. from four or more independent experiments. Significant difference in transcriptional response (*P < 0.05, two-tailed t-test) is indicated for GBSs that differ only by spacer. (c) GBS sequences used in this study, including spacer (lowercase letters) and bases that differ from the palindromic Pal-R sequence (red letters). The 15-bp GBSs were centered within identical flanking sequences, resulting in a 24-bp double-stranded DNA. (d) Alignment of GR DBD–Pal-F (gray) (PDB 3G99) and GR DBD–Fkbp5 (blue) (PDB 3G6U) crystal structures. (e) Close-up view of interaction between Lys490 and the spacer of the Pal-F GBS (gray) and of the Fkbp5 GBS (blue).
only in spacer sequence (Figure 2.1b,c). Changing the spacer of Sgk from TTT to GGG (Sgk-ggg) resulted in a 69% decrease in transcriptional activation, and changing the spacer by only one base (from TTT to TTG; Sgk-ttg) resulted in a 42% decrease. Alternatively, changing the spacer sequence (GGG) of a GBS associated with the Fkbp5 gene, to AAA (Pal-F), but not to TTT (Pal-R), resulted in decreased transcriptional activity. Thus, spacer sequence, within the context of the whole GBS, influences GR activity.

As GR does not directly contact the GBS spacer, we investigated other potential structural mechanisms by which spacer sequence influences GR function. Though prior crystallography studies did not include the Sgk-ggg GBS, we aligned structures of GR DBD bound to the Pal-F and Fkbp5 GBSs, which differ only in spacer sequence (Figure 2.1d). These structures revealed that the minor groove of the Pal-F spacer is narrower than that of the Fkbp5 spacer, with average widths of 3.8 Å and 6.4 Å, respectively, as measured by Curves+ [23]. As the Pal-F and Fkbp5 GBSs have spacer sequences of AAA or GGG, respectively, the sequence-specific difference in minor-groove width is consistent with previous studies showing that short A-tracts narrow the minor groove [24]. This led us to hypothesize that DNA shape, defined by the nucleotide sequence of the GBS spacer, serves as a signal that regulates GR activity. Thus, we predicted that structural features associated with the DNA spacer impart structural changes that are propagated through the GR DBD. Consistent with this prediction, our examination of GR contacts with the DNA phosphate backbone near the GBS spacer indicated that the orientation of the side chain of Lys490 is dependent on the spacer sequence (Figure 2.1e).

### 2.2.2 D-loop conformation depends on spacer but not half-site

To further investigate how GBS spacer affects GR structure, we monitored GR DBD conformation by $^{15}$N-HSQC, which measures the chemical environment of the amide bond
of individual amino acid residues. For GR DBD bound to a high-affinity GBS (Gha), we assigned >90% of the chemical-shift peaks to their corresponding residues (Figure 2.2). We overlaid the HSQC spectra for GR bound to GBSs that differ only at nonspecific bases of the spacer or at half-site positions 13 and 15 (hereafter referred to as half-site_{13:15}). In both comparisons, many of the spectra peaks did not overlap, which indicates that each GBS complex is structurally distinct (Figure 2.3a) and suggests that the bases of the spacer and the half-site_{13:15} influence GR structure.

We used chemical-shift perturbation, which is sensitive to local changes in conformation [25, 26, 27], to distinguish which regions of GR DBD are affected by spacer sequence and which are affected by half-site_{13:15}. GBSs that differed only in the spacer showed substantial peak shifts mapping to Ala477 and Gly478 of the GR dimerization loop (D-loop) (Figure 2.3b), despite a distance of at least 18 Å between the GR backbone and the nucleotide bases of the spacer. In contrast, the bases at half-site_{13:15} had little effect on Ala477 and Gly478 but affected residues at other surfaces of the DBD (Figure 2.4). Thus, sequence variation within particular regions of the GBS corresponds with conformational changes in distinct subdomains of the DBD.

Extending this analysis across the GR DBD further demonstrated that changing the spacer, but not half-site_{13:15}, induced peak shifts mapping to the D-loop of GR (Figure 2.3c,d). Alternatively, half-site_{13:15}, but not the spacer, influenced peaks mapping to outward-facing surfaces of the DBD near the DNA (Figure 2.3d and Figure 2.4). Additionally, the DNA-recognition helix H1 and the lever arm were affected by sequence changes in the spacer and in half-site_{13:15}. This effect of GBS on the lever-arm conformation further

---

**Figure 2.2 (following page):** NMR assignment of the DNA-bound GR DBD complex. (a) HSQC of $^{2}\text{H}^{15}\text{N}$-labeled GR wildtype (WT) DBD bound to the Gha GBS at 35°C. (b) Overlay of the $^{15}\text{N}$-HSQC spectra of the unbound WT DBD (WT Apo) and WT DBD – Gha complex at 25°C.
Figure 2.3: GBS spacer affects the conformation of the D-loop. (a) Comparison of \textsuperscript{15}N-HSQC spectra of GR DBD–GBS complexes that differ at spacer only (Sgk and Sgk-ggg) or at spacer and half-site (Sgk and Gilz; Sgk-ggg and Gilz). (b) Close-up of spectra showing the chemical-shift perturbation of D-loop residues Ala477 and Gly478 resulting from changing specific nucleotides in the spacer or half-site. (c) The magnitude of combined \textsuperscript{1}H and \textsuperscript{15}N chemical-shift difference between GR DBD–Sgk and GR DBD–Sgk-ggg spectra for each assigned residue, colored onto the crystal structure (PDB 3G6U). Unassigned residues are shown in white. (d) Chemical-shift difference (CSD) analysis for pairwise comparison of GR DBD complexes with TTT and TTG spacers (top) or TGTTCT and TGTCCG half-sites (bottom). Peaks unambiguously arising from peak splitting were assigned to their corresponding residues and CSD values for both peaks are plotted. Grey bars indicate residues that have a CSD greater or equal to the mean CSD across pairwise comparisons. UA, unassigned; p.p.m., parts per million.
corroborates the alternate conformations observed by crystallography [5].

How might information in the GBS spacer be transmitted across the substantial distance to the D-loop to elicit specific rearrangements? As the lever arm structurally links the DNA-recognition helix to the D-loop, the simplest model suggests structural coupling of the D-loop to the DNA-recognition helix via the lever arm.

2.2.3 The A477T mutation disrupts D-loop conformation

To investigate the functional role of spacer-specific structural changes, we tested whether perturbing the D-loop affected GR activity in a GBS-specific manner. We focused on Ala477 of the D-loop, which makes one of the four dimerization contacts within the GR DBD [21] – a backbone hydrogen bond between the carbonyl of Ala477 and the amide of Ile483 from the associated dimer partner. The A477T substitution has been shown to alter GR activity in a gene-specific manner [28]. As reported previously [5, 29], the extent to which transcriptional induction differed between A477T mutant GR and wild-type GR was GBS specific. Among eight GBSs tested in reporter assays, the A477T mutant showed increased, decreased or unchanged transcriptional induction compared to wild-type GR (Figure 2.5a). In contrast to wild-type GR, the A477T mutant showed no difference in activity for two GBSs that differ in spacer (Sgk and Sgk-igg). Thus, this mutation in the dimerization interface did not abolish GR activity but instead resulted in reinterpretation of GBS signals by the mutant GR.

Figure 2.4 (following page): NMR chemical shift difference analysis of GBSs that differ in the half-site or spacer sequence. (a) Pairwise comparison of six GBS complexes showing the effect of changing either spacer of half-site positions 13 and 15 (half-site\textsubscript{13:15}). NMR chemical shift differences are mapped on to the crystal structure of GR DBD (PDB ID:3g6u). (b) The locations within the GR DBD and (c) the overlay of $^{15}$N-HSQC peaks for residues that differ in half-site\textsubscript{13:15}.  

13
Figure 2.5: Disruption of the dimerization interface affects lever arm and DNA-recognition helix conformation. (a) Comparison of transcriptional induction of GBS luciferase reporters in U2OS cells expressing either wild-type (WT) or mutant (A477T) GR after treatment with 100 nM dexamethasone. Error bars represent s.d. from four or more independent experiments, and significant differences in transcriptional response between WT and A477T were determined by two-tailed t-test (*P < 0.05). The transcriptional response of A477T is equivalent for Sgk and Sgk-ggg (P = 0.44) or Sgk-ttg (P = 0.19). (b) $^{15}$N-HSQC comparing WT and A477T DBD bound to the Fkbp5 GBS. (c) Magnitude of combined $^1$H and $^{15}$N chemical-shift differences between WT and A477T DBD bound to the Fkbp5 GBS, colored onto the WT DBD crystal structure (PDB 3G6U). Unassigned WT residues are shown in white. (d) Chemical-shift difference (CSD) between spectra of WT DBD and A477T DBD complexes. Grey bars highlight residues with CSD ≥ 0.05 parts per million (p.p.m.) between WT and A477T across all three GBSs. UA, unassigned.
To assess the mechanism by which the mutation differentially affects GBS-specific activity, we characterized the structural and biophysical impacts of A477T. We compared the HSQC spectra of wild-type and A477T GR bound to the Fkbp5 GBS (Figure 2.5b,c). The A477T DBD spectrum showed many peaks that overlaid well with that of wild-type GR, but >30% of residues were shifted as a result of the A477T mutation. These peaks did not overlap with those corresponding to the unbound wild-type DBD, confirming that the protein was completely bound to DNA (Figure 2.6).

We quantified the chemical-shift difference as the distance between each peak in the GR wild-type spectra and the peak in the A477T spectra nearest to it [29]. Comparison of wild type- and A477T-bound Gha, Fkbp5 and Sgk complexes (Figure 2.5d) revealed A477T-specific shifts that mapped to the D-loop and the residues surrounding Ile483, consistent with a disruption of the dimerization interface. Additional chemical-shift differences mapped to the N-terminal region of the lever arm and the recognition helix of GR. Thus, the A477T mutation generates local structural changes as well as structural reorganization in regions outside of the dimerization surface. Together with the observation that wild-type GR produced GBS-specific structural changes in the dimerization interface (Figure 2.3b,c), our findings with the A477T mutation indicated that the dimerization and DNA interfaces are structurally coupled.

2.2.4 A477T affects cooperativity but not stoichiometry

To pursue the mechanism by which the A477T mutation affects the GBS-specific activity of GR, we assessed DNA binding by an electrophoretic mobility shift assay (EMSA). Comparison of wild-type DBD and A477T DBD binding to the Pal-R and Sgk GBSs revealed that GR dimer complexes were formed at saturating concentrations of both wild-type and A477T DBD, although the mutant showed reduced DNA-binding affinity (Figure
Figure 2.6: DNA-bound A477T conformation differs from that of unbound WT DBD. The overlay of $^{15}$N-HSQC spectra of WT and A477T DBD bound to the Fkbp5 GBS, compared to the unbound DBD (WT Apo) at 35°C.
2.7a). For the wild type, the transition from free DNA to dimer complex occurred at lower concentrations of GR, and only a minor population of DNA-bound monomer was present, indicative of strong positive cooperativity. In contrast, A477T showed little cooperativity and had nearly saturated the DNA as a monomer before dimer formation. To distinguish whether the reduced overall affinity of A477T was due to impaired DNA recognition resulting from the mutation, we compared binding of wild-type GR and A477T to a GBS half-site (Figure 2.7b). We found that half-site binding was equivalent for wild type and A477T, indicating that the A477T mutation does not disrupt the DNA-binding ability of the monomer (Figure 2.7b,c). Thus, the reduction in overall affinity of the A477T mutant is due to diminished cooperativity.

As the EMSA is a measure of non-equilibrium, we also used surface plasmon resonance (SPR) to monitor the effects of binding sequence on the DNA-recognition properties of wild type and A477T, under conditions similar to those used for the NMR studies. We compared binding of wild-type DBD or A477T DBD in two GBSs whose transcriptional induction in reporter assays was reduced (Pal-R) or unaffected (Gilz) by the A477T mutation (Figure 2.8a). Isotherms constructed from maximal binding of wild-type or A477T DBD to GBS-immobilized surfaces at equilibrium showed that the mutation results in a decrease in binding affinity, by a factor of 10 for Pal-R and of 5 for Gilz (Figure 2.8b and Supplementary Table 2.1). Consistent with the EMSA results, maximal binding responses were similar for wild type and A477T, indicating that the mutant binds to DNA at the same stoichiometry as wild type (Figure 2.8a). As the concentration dependence was non-Langmuir, we fit the SPR binding isotherms to the Hill equation. Fit Hill coefficients for wild type were $1.83 \pm 0.28$ and $2.13 \pm 0.26$ for the Gilz and Pal-R GBSs, respectively (Figure 2.8b and Supplementary Table 2.1). The A477T isotherms were well described by Hill coefficients of $1.34 \pm 0.16$ s.d. (Gilz) and $1.41 \pm 0.1$ (Pal-R), indicating that cooperativity was reduced but not abolished by the A477T mutation. As the Hill coef-
Figure 2.7: A477T impairs dimerization but not monomer DNA binding. (a,b) EMSA monitoring binding of wild-type (WT) or A477T DBD to GBSs conjugated with an Alexa 488 fluorophore at a concentration of 5 nM (a) or to mutated GBSs, where one half-site is changed to the least favorable nucleotide at each position on the basis of the ChIP-sequencing binding motif shown in Figure 2.1a (b). (c) Quantitative comparison of WT DBD (open squares) or A477T DBD (filled triangles) binding by EMSA. Error bars represent s.d. from 2-4 replicates.
### Table 2.1: Summary of SPR fit parameters for WT and A477T DBD binding to GBS surfaces at 35°C or 8°C. Dissociation constant, $K_{1/2}$, and Hill coefficient, $n_H$, were calculated from equilibrium binding isotherms fit to a the Hill equation. Fitting of the dissociation phases of the SPR traces by conventional models gave systematic residuals consistent with a biphasic-sequential dissociation process, likely resulting from the complex kinetics and strong cooperativity of this system. Instead, the stabilities of the different complexes were reliably determined by a standard exponential decay model, and $t_{1/2}$ values were extrapolated from the apparent off-rates. Errors are the mean ± two s.d. from 2-3 replicates.

<table>
<thead>
<tr>
<th>GBS</th>
<th>$K_{1/2}$ (nM)</th>
<th>$n_H$</th>
<th>$t_{1/2}$ (s)</th>
<th>$K_{1/2}$ (nM)</th>
<th>$n_H$</th>
<th>$t_{1/2}$ (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>35°C</td>
<td></td>
<td></td>
<td>35°C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pal-R</td>
<td>1.6 ± 0.2</td>
<td>1.6 ± 0.2</td>
<td>55 ± 4</td>
<td>16 ± 1</td>
<td>1.4 ± 0.1</td>
<td>4.8 ± 0.7</td>
</tr>
<tr>
<td>Gilz</td>
<td>5.7 ± 0.6</td>
<td>1.6 ± 0.2</td>
<td>23 ± 2</td>
<td>28 ± 1</td>
<td>1.3 ± 0.2</td>
<td>4.7 ± 0.8</td>
</tr>
<tr>
<td></td>
<td>8°C</td>
<td></td>
<td></td>
<td>8°C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gha</td>
<td>2.7 ± 0.2</td>
<td>2.2 ± 0.2</td>
<td>160 ± 20</td>
<td>21 ± 1</td>
<td>1.2 ± 0.05</td>
<td>18 ± 1</td>
</tr>
<tr>
<td>Pal-R</td>
<td>3.1 ± 0.1</td>
<td>2.1 ± 0.2</td>
<td>120 ± 20</td>
<td>39 ± 1</td>
<td>0.9 ± 0.03</td>
<td>13 ± 0.8</td>
</tr>
<tr>
<td>Sgk</td>
<td>9.7 ± 0.2</td>
<td>1.8 ± 0.06</td>
<td>50 ± 7</td>
<td>42 ± 1</td>
<td>1.0 ± 0.02</td>
<td>10 ± 0.7</td>
</tr>
<tr>
<td>Fkbp5</td>
<td>12 ± 0.2</td>
<td>2.0 ± 0.06</td>
<td>22 ± 3</td>
<td>55 ± 1</td>
<td>1.4 ± 0.02</td>
<td>5.9 ± 0.5</td>
</tr>
<tr>
<td>Gilz</td>
<td>13 ± 0.2</td>
<td>1.9 ± 0.05</td>
<td>28 ± 4</td>
<td>71 ± 5</td>
<td>1.1 ± 0.03</td>
<td>7.6 ± 0.7</td>
</tr>
<tr>
<td>Sgk-ttg</td>
<td>13 ± 0.2</td>
<td>2.3 ± 0.2</td>
<td>34 ± 2</td>
<td>48 ± 17</td>
<td>1.4 ± 0.02</td>
<td>12 ± 0.7</td>
</tr>
<tr>
<td>Sgk-ggg</td>
<td>20 ± 0.2</td>
<td>1.9 ± 0.06</td>
<td>18 ± 3</td>
<td>166 ± 23</td>
<td>1.1 ± 0.05</td>
<td>5.8 ± 0.6</td>
</tr>
</tbody>
</table>

The Hill coefficient represents an exponential component of the Hill equation, the differences in Hill coefficients between the two GBSs (0.3 for wild type and 0.07 for A477T) are substantial for wild type and diminished for A477T. This suggests that differential cooperativity contributes to the discrimination between binding sites for wild type but has a lesser role for A477T.

As the transcriptional activity of GR is affected by the fractional occupancy of the active dimer complex on a given response element, we also compared the dissociation kinetics of wild-type and A477T DBD-GBS complexes (Figure 2.8a). To simplify the comparison between wild type and A477T across the different binding sites, dissociation traces were fit to a single exponential decay process and the fit parameters were presented as half-life ($t_{1/2}$) values (Supplementary Table 2.1). For wild type, Pal-R and Gilz showed fit $t_{1/2}$ values of 55 ± 4 s and 23 ± 2 s, respectively. In contrast, A477T dissociated from both
Figure 2.8: A477T disrupts cooperativity and GBS-specific dissociation. (a) Representative SPR binding traces for wild-type (WT) and A477T DBD binding to immobilized GBSs at 35 °C. Comparison of dissociation curves for WT DBD (top right) and A477T DBD (bottom right). (b) Binding isotherms for a GR DBD concentration series (0.70-200 nM for WT, 1.4-400 nM for A477T) binding to immobilized GBS surfaces at 35 °C from three separate titrations, normalized to maximal binding. (c) Comparison of transcriptional activity (fold induction) and binding affinity ($K_{1/2}$) or dissociation half-life ($t_{1/2}$), for WT DBD (blue) and A477T DBD (red) across seven GBS surfaces at 8 °C. Error bars are s.d.
Pal-R and Gilz GBSs with a $t_{1/2}$ of $\sim$5 s; a decrease of 90% and of 80%, respectively, relative to wild type. Under these conditions, the dissociation of wild type was dependent on the GBS, whereas A477T kinetics seemed to be undiscriminating of sequence. This suggests that an intact dimerization surface is crucial for interpreting GBS-specific signals that modulate GR dissociation.

To assess more broadly the biophysical parameters that might influence GBS-specific transcriptional activity, we extended our analysis to include five GBSs in addition to the Pal-R and Gilz GBSs. Across all GBSs, A477T showed lower affinity, faster dissociation and lower cooperativity than did wild type (Supplementary Table 2.1). Notably, although Pal-R and Gha had similar binding parameters, their transcriptional regulatory activities in reporters were not aligned ($P = 0.004$). To assess the relationship between DNA-binding properties and GBS-specific transcription, we compared transcriptional activity to the $K_{1/2}$ or $t_{1/2}$ values across this panel of GBSs (Figure 2.8c). We found that transcriptional induction of wild-type GR did not vary as a simple function of DNA affinity. For example, the A477T mutation resulted in reduced binding affinity, but enhanced transcriptional induction at the Gha GBS (Figure 2.5a). Additionally, GBSs sharing similar GR binding properties, such as Gha and Pal-R or Fkbp5 and Gilz, differed in transcriptional activity (Figure 2.1a) as well as conformation (Figure 2.9). These results are consistent with those of Bain et al. [30], who describe GBSs with binding affinities and regulatory activities that are not aligned. Therefore, we were surprised that Bain et al. [30] asserted that DNA binding affinity defines transcriptional activity at GBSs. Although DNA binding affinity clearly has a role in the activities of transcriptional regulators, our results as well as those of Bain et al. [30] demonstrate that other factors must also contribute substantially. Having previously proposed that GBS-specific structural changes determine transcriptional activity [5, 22], we assessed the relationship between binding and activity in the A477T mutant. Compared to wild type, A477T showed a stronger correlation between transcription and
### Table 2.2: Correlation between transcriptional activity and DNA binding affinity ($K_{1/2}$) or half-life ($t_{1/2}$).

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Wildtype</th>
<th>A477T</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R^2$ (log transformed)</td>
<td>0.22</td>
<td>0.17</td>
</tr>
<tr>
<td>Pearson’s ($R^2$)</td>
<td>0.31</td>
<td>0.06</td>
</tr>
</tbody>
</table>

Correlations for WT and A477T were calculated for seven different GBS complexes. Because this relationship is expected to be non-linear [30], a statistical analysis with non-linear fitting of data are presented.

affinity, as measured by multiple correlation analyses (Supplementary Table 2.2). Thus, disruption of the dimer interface seems to dampen allosteric signaling, rendering GBS affinity a stronger determinant.

### 2.2.5 A477T disrupts signaling between dimer partners

To further dissect the structural mechanism by which A477T alters GBS-specific activity, we used $^{15}$N-HSQC to monitor the effects of the A477T mutation on GBS-specific conformations of GR. We focused on the lever-arm region because its structure is sensitive to GBS [5]. We found that for wild type, Gly470 showed peak-splitting in a GBS-dependent manner (Figure 2.11a). Peak-splitting indicates two unique chemical environments for a single residue and reflects two possibilities: either the GR dimer partners have non-equivalent conformations, or GR dimers undergo slow conformational exchange between two distinct states. To distinguish between these possibilities, we used ZZ-exchange NMR to detect conformational exchange. This experiment is similar to HSQC except that a mixing period is introduced between recordings of the $^{15}$N chemical-shift and of the $^1$H chemical shift for each amide. Chemical exchange that occurs during the mixing period is detected as cross-peaks in the NMR spectrum corresponding to the $^{15}$N chemical shift of conformation A and the $^1$H chemical shift of conformation B – and vice versa (Figure 2.10a,b). We fit the individual intensities of exchange cross-peaks at different mixing
Figure 2.9: Chemical shift difference between WT DBD – GBS complexes with different levels of transcriptional induction by GR. Structures are colored according to the magnitude of combined $^1$H and $^{15}$N NMR chemical shift differences for pairs of GBS complexes that have similar binding affinity, cooperativity and dissociation kinetics. Unassigned residues are colored white.
periods [31] (Figure 2.11b) and determined that the conformational exchange rate of Gly470 (4.36s$^{-1}$ and 3.10s$^{-1}$ for conformation A to B and conformation B to A, respectively) is two orders of magnitude faster than its rate of dissociation from DNA (0.03s$^{-1}$) (Figure 2.10e). This suggests that doublet peaks result from conformational exchange of DNA-bound GR dimer partners rather than dissociation from one half-site and subsequent re-binding to the adjacent half-site (Figure 2.10b).

Because the Gilz GBS consists of two non-identical half-sites, the simplest view is that Gly470 conformations may be determined solely by the sequence of the GBS half-site to which each GR dimer partner is bound. This would predict that GBSs that are identical at one half-site and differ at the other will have one overlaid Gly470 peak and one non-overlaid peak (compare Pal-R and Sgk). We found, however, that the GBS complexes Gilz, Sgk and Pal-ttg, which are identical at one half-site, each had two unique Gly470 chemical shifts (Figure 2.11a). Thus, the lever-arm conformation for each GR dimer depends not only on the sequence of the half-site to which it was directly bound, but also on the sequence bound by the adjacent dimer partner. Taken together with comparisons of Gly470 chemical shifts among different GBS complexes, our data suggest that each GR dimer partner integrates sequence-specific signals from both GBS half-sites (Figure 2.11c).

Unlike those for wild type, the Gly470 peaks for A477T bound to different GBSs were overlaid (Figure 2.11a). Therefore, disrupting the dimer interface at Ala477 abolished the sequence-specific conformations of Gly470 within the lever arm. We expanded this comparison to investigate the extent to which each residue samples distinct conformations when bound to different GBSs by determining the chemical-shift variance across a panel of seven GBSs (Figure 2.11d and Figure 2.12). For wild-type GR, residues with considerable chemical-shift variance included those in the recognition helix and lever arm, with the
Figure 2.10: Conformational exchange within the lever arm and dimer interface.
(a) Zoomed view of ZZ-exchange NMR spectra for G470 and Ile484 with a 200 ms mixing period. These peaks indicate two conformational states, arbitrarily labeled A and B, at 0 ms mixing period and the exchange cross-peaks are labeled A→B and B→A. (b) Model for conformational swapping between dimer partners while bound to DNA, based on the observed rate of conformational exchange ($k_A$ and $k_B$), which is faster than the measured GR dissociation rate from DNA (0.03 s$^{-1}$). (c) The location of Gly470 and Ile484 (red spheres) within the GR DBD monomer. Recognition helix (orange), lever arm (yellow), and zinc finger 2 of the dimer interface (light blue). (d) Mapping of $^{15}$N-HSQC peaks of the WT DBD – Gilz complex that undergo peak-splitting from slow exchange. (e) The parameters from curves fit to ZZ-exchange peaks, where $R$ = longitudinal relaxation rate of magnetization, $k$ = exchange rate for converting from site A to B, and $I(0)$ = longitudinal magnetization at the start of the mixing period [31].
Figure 2.11: Sequence-specific lever-arm conformation is dependent on the intact dimerization interface. (a) $^{15}$N-HSQC close-ups (left and middle) of the Gly470 peak of the lever arm of wild-type (WT) DBD bound to the asymmetric GBSs (Gilz, Sgk and Pal-ttg) and a palindromic site (Pal-R). Right, overlay of Gly470 peaks from all for GBSs for WT (top) and A477T (bottom) DBD. (b) The peak intensity for Gly470 and Ile484 WT DBD residues in ZZ-exchange spectra at five mixing periods. The auto peak and the corresponding exchange peak are shaded equivalently. Insets, spectra for auto and exchange peaks at a mixing period of 0.2 s. (c) Model of the role of the dimerization interface in defining GBS-specific conformations. Both half-sites and the spacer determine the conformation of each GR dimer partner by transmitting information from the adjacent GBS half-site across the dimerization interface (left, colored arrows). The unique conformations of GR dimer partners at each half-site are represented by the hexagon and square (top), and are distinct from the conformations at a second GBS, represented by the triangle and trapezoid (bottom). Right, disruption of the dimerization interface by A477T impairs signal transmission and results in lever-arm conformations that are insensitive to GBS sequence. (d) Comparison of WT and A477T chemical-shift variance among seven GBS complexes colored by amino acid onto the wild-type GR DBD crystal structure. Amino acids in which the chemical shift variance is affected by the A477T mutation are colored red.
highest variance at Arg466, which makes a direct contact with the GBS. Consistent with the impact of A477T on Gly470, we found that A477T showed lower chemical-shift variance throughout the recognition helix and lever arm than did wild type. Thus, this global chemical-shift perturbation analysis supports our conclusion that the GBS-specific conformations of GR depend on an intact dimer interface.

Building on NMR chemical-shift mapping showing that GBS modulates the GR dimer interface, we conclude that the intact dimerization surface allows for allosteric communication between dimer partners and integration of sequence signals from the GBS as a whole. Consistent with this view, disruption of this communication by the A477T mutation reduced the sequence-specific effects of the GBS on GR conformation and simplified the relationship between GR DNA binding affinity and regulatory activity. Thus, we propose that signals are transmitted from the DNA-binding interface through the lever arm and the dimerization interface and into GR domains outside of the DBD that confer transcriptional regulatory activity [32, 33].

2.3 Discussion

Genomic response elements are composed of combinations of sequence motifs that specify binding of distinct transcriptional regulators to execute gene-specific control of transcription. Even combinations of bases within a single binding motif can affect gene-specific activity [17]. Crystallographic studies of the GR DBD have revealed that the conformation of the 'lever arm', a region between the DNA-recognition helix and the dimer interface, differs at distinct GBSs [17]. Through NMR analysis of wild-type and mutant GR DBD-GBS complexes, we defined both the origin and the consequence of the lever-arm conformational transitions – a path of affected residues including parts of the DNA interface, the lever arm and the dimerization interface that facilitates allosteric communication between
Figure 2.12: NMR chemical shift variance across seven different GBS complexes. WT variance and A477T variance among $^{15}$N-HSQC spectra for each amino acid calculated as $\sigma^2_H$ or $\sigma^2_N = \sum (x - \mu)^2/(n-1)$, where $x = ^1H$ or $^{15}$N chemical shift (ppm), $\mu$ = the mean chemical shift (ppm), and $n = 7$; and combined as the variance sum: $\sigma^2_{NH} = (1/5 \times \sigma^2_N) + \sigma^2_H$ (top and middle panel). The difference between WT and A477T chemical shift variance (bottom panel). The dotted lines represent the first quartile (Q1), median (Q2) and the third quartile (Q3). Unassigned amino acids are plotted as zero.
GR dimer partners. This path enables integration of sequence-specific signals from both GBS half-sites, exponentially increasing the informational complexity of the GBS.

We considered the possibility that chemical-shift differences in the DNA and dimer interfaces result from the GBS-dependent reorientation of rigid GR dimer partners relative to the DNA or each other – for example, from differential DNA bending. We found that the magnitude of chemical-shift differences did not correlate with the proximity of GR DBD residues to the DNA (data not shown), contrary to the rigid-but-reoriented model. We assessed GBS-dependent DNA bending by fluorescence resonance energy transfer (FRET) analysis. GR binding produced very small increases in GBS FRET efficiency compared to DNA alone (Figure 2.13), suggesting minimal DNA bending induced by GR. Furthermore, A477T and wild type showed equivalent behavior among the five GBSs tested. Therefore, the chemical-shift differences between GBS complexes and between wild type and A477T represent GBS-specific differences in GR structure, not DNA bending.

Protein allostery is a crucial aspect of combinatorial control. Structural studies of isolated ligand-binding domains (LBDs) from nuclear receptors suggest how signaling information residing in small-molecule ligands is transmitted to a co-regulator recognition surface [13, 34, 35, 36, 37, 38]. Here we show that GBSs drive structural changes at the DNA-binding interface that are coupled with changes in the GR dimerization interface and dimer partner as well as correlated with distinct biophysical and transcriptional outcomes. It is likely that such structural transitions extend into distinct domains of intact GR to specify regulatory complex assembly and activity. These findings provide perhaps the clearest mechanistic perspective to date on functional studies showing that binding sequences modulate receptor interactions with co-regulators [5, 39, 40], that ligands modulate interactions with DNA [41] and that both DNA and ligands direct interactions with co-regulators [42]. Thus, the DBD residues identified here, together with LBD residues
Figure 2.13: Comparison of sequence-specific DNA bend as determined by FRET assay. Alexa488 and Alexa546 fluorophores were conjugated to opposite 3' ends of 24 bp GBS-containing oligos, serving as the donor and acceptor, respectively. FRET was measured for five GBSs at 50 nM with and without GR DBD (500 nM WT or A477T). By assuming a bend at the center of the GBS, GR binding-induced DNA bend is less than 1° for all GBSs, using the formula $R = R_0[(1/E)−1]^{1/6}$, where $R =$distance, $E =$efficiency and $R_0 =$Förster distance of the FRET pair. All FRET assays were performed in NMR buffer using a Molecular Devices SpectraMax M5 with a fixed excitation of 444 nM and an emission scan from 500-650 nm. Equivalent results were obtained using 1 µM protein, indicating saturated binding under these conditions. Data is shown as the mean ± two s.d. from 12 replicates.
that interpret ligand signals [43] and affect gene-specific regulation [4, 44, 45], could begin to define a molecular ‘map’ that, in the cellular context, integrates GBSs, ligands, chromatin, co-regulators and post-transcriptional modifications to determine the composition and function of gene-specific transcriptional regulatory complexes.

Other than specific base contacts, what DNA signals might trigger changes in GR structure? Our data suggest that GR ‘measures’ the spacer minor-groove width as an indirect readout of spacer sequence. In the Pal complex, the Lys490 side chain reaches across the spacer minor groove to contact the phosphate backbone at the complement of spacer position 7. In the Fkbp5 complex, which has a wider spacer minor groove, Lys490 contacts the phosphate backbone of the proximal strand at position 11 (Figure 2.1e). Indirect recognition of narrow minor grooves by insertion of positively charged side chains is a general feature of DNA recognition [24] and a contributor to specificity among transcription factors [46, 47]. For GR, we observe a distinct mechanism whereby minor-groove width imposes structural constraints on lysine-mediated backbone contacts to DNA. Thus, flexible regions of the protein may adopt conformations that accommodate differences in DNA shape, as has been demonstrated for other transcriptional regulators [48].

In addition to identifying regions of GR that can adopt distinct conformational states among different GBS complexes, we found that dimer partners undergo dynamic exchange between two discrete conformations while bound to a particular GBS (Figure 2.11a,b and Figure 2.10). This is consistent with the structural asymmetry between dimer partners observed by crystallography [5]. Gly470 of the lever arm and Ile484 of the dimer interface showed dynamics with similar timescales, consistent with structural coupling between these regions of the DBD. We speculate that GR-GBS complexes may differentially access conformational states that interact preferentially with particular transcriptional co-regulators, thus providing ‘assembly instructions’ for different regulatory complexes. GBS-specific
dissociation kinetics may, in part, affect GR activity by altering the turnover of GR-DNA complexes. Indeed, interactions with response elements are highly dynamic, occurring on the timescale of seconds [49], and regulatory complexes are actively and continuously disassembled [50]. How the DNA-binding kinetics of GR are regulated and how they affect transcriptional activity remain open questions.

Structural characterization of related nuclear receptors has shown that DNA binding mediates conformational changes in the dimerization surface, providing a mechanism for cooperative dimerization [51]. Here we have shown that GR cooperativity is affected by the precise binding sequence and impaired by the A477T mutation. Differential cooperativity is well established as a mechanism for achieving gene-specific activity and suppressing transcriptional noise [52, 53]. We propose that sequence-dependent conformational changes in the dimer interface modulate gene-specific cooperativity, in turn regulating the level of transcriptional activation by GR. While this manuscript was in revision, Hudson et al. [54] reported that GR binds with negative cooperativity at “nGREs,” where GR represses transcription. Thus, GBS-mediated allosteric regulation of cooperativity may enable GR to exhibit exceptional specificity in gene-regulation activities ranging from transcriptional activation to repression.

2.4 Methods

2.4.1 Protein expression and purification

Expression vector pET28a containing rat GR wild-type DBD residues 440-525 has been described previously [5]. Vector for A477T DBD was derived by PCR site-directed mutagenesis. BL21 Gold E. coli cells were grown in 50 mL LB medium to an absorbance ($A$) of ~0.5-1.0 then pelleted and resuspended in 1 l minimal medium containing $^{15}$NHCl$_4$
(2 g/l) as the only nitrogen source. Cultures were grown to an absorbance of ~0.7 and expression was induced with 0.5 mM IPTG for ~16 h at 18 °C or 8 h at 30 °C (both produced equivalent spectra). Cells were pelleted and resuspended at 40 mL/L in lysis buffer containing 50 mM Tris (pH 7.5), 500 mM NaCl and 15 mM imidazole then frozen in liquid nitrogen and stored at -80 °C. Cells were lysed using an EmulsiFlex C5 homogenizer. Lysate was centrifuged for 45 min at 40,000 r.p.m. at 4 °C and run over a nickel-Sepharose (GE Healthcare) column and eluted with a linear gradient to 350 mM imidazole. Pooled fractions were dialyzed into 20 mM Tris (pH 7.5), 50 mM NaCl, 2.5 mM CaCl$_2$ and 0.5 mM β-mercaptoethanol and cleaved at 4 °C overnight with 50-100 U thrombin. Protein was further purified over a Resource S ion exchange column with a linear gradient of 50-300 mM NaCl, 20 mM Tris (pH 7.5) and 0.5 mM β-mercaptoethanol. Protein was concentrated using Amicon Ultra 5-kDa MWCO (Millipore) and run over a 16/60 Superdex75 gel filtration column in NMR buffer (20 mM sodium phosphate (pH 6.7), 100 mM NaCl, 1 mM DTT).

2.4.2 Protein-DNA complex formation

Single-stranded GBS oligonucleotides (IDT) were purified by 10/10 MonoQ as described previously [5] and resuspended at 2 mM. Oligonucleotides were annealed at 1 mM in boiling water and cooled slowly to room temperature. To ensure solubility of GR DBD-DNA complexes, dilute protein was combined with double-stranded DNA (dsDNA) 1× diluted in cold NMR buffer. DNA was present in ~33% excess of DBD to ensure saturated binding. Dilute GR DBD-DNA complexes were concentrated slowly at 4 °C using a 3-kDa MWCO centrifugal filter (Amicon) to ~300 µM dimer complex and filtered with Ultrafree PVDF 0.22 µm columns (Millipore).
2.4.3 Protein nuclear magnetic resonance assignment

Triple-labeled GR DBD was prepared as described above except that 50 mL LB cultures were resuspended in 1 l unlabeled minimal media, grown to an absorbance of ~0.7, then pelleted and resuspended in 1 l of minimal media containing 2 g of $^{15}\text{N}\text{HCl}_4$ and 2 g $^{13}\text{C}$-glucose in 90-100% D$_2$O. Expression was induced at 30 °C for 12 h. Assignments in the absence of DNA were with 1.7 mM GR DBD. The following experiments were run at 25 °C on a Bruker 500-MHz spectrometer: $^{15}\text{N}$-HSQC, HNCO, HNCA, HNCOCA, HNOCACB [55, 56] and CC(CO)NH-TOCSY. A $^{15}\text{N}$-edited NOESY [57] was run on a Bruker 800-MHz spectrometer. For assignment of DNA-bound GR, purified DBD and the Gha GBS were combined at a ratio of 2:1 (monomer to GBS) and concentrated to 500 $\mu$M complex (1 mM GR DBD). NMR assignments were generated from $^{15}\text{N}$-edited NOESY [57], TROSY-HNCO, TROSY-HNCA, TROSY-HN(CO)CA, TROSY-HNOCAB and TROSY HN(CA)CO [58] experiments conducted on 600-, 800- and 900-MHz spectrometers at 25 °C and 35 °C, because some peaks gave stronger signal at 35 °C. Assignments were aided by $^{15}\text{N}$-HSQC of $^{15}\text{N}$-specific amino acid labeling of isoleucine, leucine, valine, phenylalanine, tyrosine, lysine and reverse-labeled arginine using DL39 cells for expression. All NMR data were processed in NMRPipe [59] and analyzed using Sparky (http://www.cgl.ucsf.edu/home/sparky/). Wild-type DBD-GBS assignments were transferred to A477T DBD complexes generally according to the minimal combined $^1\text{H}$ and $^{15}\text{N}$ chemical-shift difference for each assigned GR wild-type peak to the nearest A477T peak [29]. Assignment transfer was aided by $^{15}\text{N}$ reverse-labeling of arginine or lysine A477T residues.
2.4.4 Chemical-shift difference and ZZ-exchange nuclear magnetic resonance

$^{15}$N-HSQ spectra were acquired on a Bruker 800-MHz spectrometer at 35 °C. Peak assignments were transferred from the wild-type DBD-Gha complex to additional GBS complexes by measuring the minimal chemical-shift difference from each assigned wild-type DBD-Gha peak to the nearest A477T peak using the formula: chemical-shift difference $\% \Delta \delta = [(\Delta^1H \text{ p.p.m.})^2 + (\Delta^{15}N \text{ p.p.m.}/5)]^{1/2}$ [29]. Similarly, peak assignments for each A477T-GBS complex were transferred by minimal chemical-shift difference using the A477T DBD-Gha complex as a reference. For ZZ-exchange, $^{15}$N-HSQ spectra were acquired on a Bruker 800-MHz spectrometer at 35 °C using a pulse sequence modified to include mixing times of 0, 0.025, 0.05, 0.1, 0.2 and 0.4 s. The intensities of auto and exchange peaks were plotted against mixing period, and curves were fit individually for Gly470 and Ile484 residues according to previously described formulas [31]. The exchange rate ($k_{ex}$) and relaxation rate for each conformation were fit separately.

2.4.5 Transcriptional reporter assays

GBS reporter plasmids were either previously generated [5] or were constructed equivalently. Reporter assays were performed as previously described [5], except that we used human GR wild-type and human GR A477T (generated by PCR site-directed mutagenesis). Briefly, U2OS cells were seeded in DMEM and 5% FBS in 24-well plates at ~20,000 cells per well 1 d before transfection. Cells were transfected with 20 ng GR plasmid, 20 ng GBS-luciferase plasmid, 200 pg pRL Renilla, 120 ng empty p6R plasmid, 1 µL PLUS reagent and 0.7 µL Lipofectamine reagent (Invitrogen) per well for 4 h in serum-free DMEM media. Cells were washed and recovered in DMEM and 5% FBS for 3 h,
then treated with 100 nM dexamethasone or ethanol for 12 h. Luciferase induction was measured using the Dual-Luciferase Reporter kit (Promega) in a 96-well format using a Tecan plate reader. Data were normalized to Renilla for each well, then to ethanol-treated control and empty pGL3 vector control.

2.4.6 Surface plasmon resonance

SPR analysis of wild-type and A477T DBD interaction with GBSs was carried out on a Biacore T100. Matrix-free surfaces were prepared by injection of Neutravidin (Invitrogen) across a planar saccharide monolayer with covalently coupled biotin (BP chips, Xantec Bioanalytics) at 25 °C in 20 mM HEPES (pH 7.2), 150 mM NaCl. Double-stranded GBS oligonucleotides with a single 5' biotin-TEG label (IDT) were subsequently captured at immobilization levels ranging from 20 to 65 RU. Wild-type and A477T DBDs were dialyzed overnight in NMR buffer. Following dialysis, 0.1 mg/mL BSA (Sigma) was added to the assay buffer and the protein samples. Fifteen-point concentration series were prepared by serial dilutions spanning 0.700-200 nM for wild-type and 1.4-400 nM for A477T. Association and dissociation times were selected to ensure equilibrium and complete dissociation. All data were processed and analyzed in Matlab. Isotherms were fit to the Hill equation: fractional occupancy $= \left(\frac{c}{c + K_{1/2}}\right)^{n_H}$, where $c$ is the GR concentration, $K_{1/2}$ is the GR concentration producing half occupancy and $n_H$ is the Hill coefficient. Error in $K_{1/2}$ and $n_H$ was determined by a bootstrap method with replacement: after scaling of $n$ equilibrium responses for each GBS, a random set of $n$ data points was selected with the possibility of selecting the same data point multiple times. After 500 iterations, the 100 best-fit parameters (s.s.e.) were used to find mean values and s.d. Parameters for $t_{1/2}$ were fit from the following equations: $R = R_0 e^{-t k_{off}}$ and $t_{1/2} = ln(2)/k_{off}$, where $R =$ response units.
2.4.7 **Electrophoretic mobility shift assay**

Unlabeled GR wild-type DBD and A477T DBD were purified and annealed as described above. Double-stranded GBSs with an Alexa 488 fluorophore conjugated to one of the 3’ ends (IDT) were incubated for 30 min at a final concentration of 5 nM with GR DBD titrations in binding buffer (20 mM Tris (pH 8), 50 mM NaCl, 1 mM EDTA, 10 µg/mL poly(dI-dC), 5 mM MgCl₂, 200 µl/mL BSA, 5% glycerol, and 1 mM DTT) on ice. Native 8% polyacrylamide gels were run at 200 V in 0.5× TBE buffer at 4 °C. Alexa 488 signal was imaged on a Typhoon scanner (GE Healthcare) and quantified using ImageQuant. Fraction of bound GR DBD was determined as $1 - \text{DNA}_{\text{free}}$.

2.4.8 **GR chromatin immunoprecipitation sequencing**

Details of GR ChIP-sequencing methods and data will be described in a separate publication. Briefly, U2OS cell lines stably expressing GR wild-type or A477T were treated with 100 nM dexamethasone for 90 min, then cross-linked with 1% formaldehyde and quenched with 125 mM glycine. Cells were lysed for 30 min at 4 °C, and nuclei were pelleted at $600 \times g$ for 5 min and resuspended in RIPA buffer (10 mM Tris (pH 8.0), 1 mM EDTA, 150 mM NaCl, 5% glycerol, 0.1% sodium deoxycholate, 0.1% SDS, 1% Triton X-100). Chromatin was fragmented by Diagenode Biorupter sonication and incubated for 16 h at 4 °C with prewashed antibodies to GR bound to Protein G Dynabeads in RIPA buffer containing protease inhibitor cocktail and 2 µg/µL BSA. Complexes were washed extensively with RIPA buffer containing 500 mM NaCl then LiCl buffer, and cross-links were reversed. DNA was column-purified (Zymogen Clean and Concentrator Kit). ChIP-sequencing libraries were prepared by end-repair, dATP-addition, and ligation of sequencing adaptors containing in-house barcodes. Libraries were amplified by 17 cycles of PCR and purified by PAGE. Libraries were sequenced using the Genome Analyzer II (Illumina) with 2 ×
75-bp paired-end reads, and aligned with Bowtie [60]. Motif analysis was performed using MEME [61] in 'zero or one motif per site' mode with a second-order background Markov model based on the top 1,000 peaks.

2.4.9 DNA oligonucleotide sequences

The sequences of oligonucleotides used in NMR, EMSA, SPR and FRET are as follows (GBS 5' to 3': Fkbp5, gtacAGAACAgggTGTTCTtcgac; Gha, gtacGGAACAtaaTGTTCTtcgac; Gilz, gtacAGAACAttgGTTTCCtcgac; Pal-F, gtacAGAACAaaaTGTTCTtcgac; Pal-R, gtacAGAACAtttTGTTCTtcgac; Pal-ttg, gtacAGAACAtttTGTTCTtcgac; Sgk gtac, AGAACAtttTGTCCGtcgac; Sgk-agg, gtacAGAACAgggTGTCCGtcgac; Sgk-ttg, gtacAGAACAttgTGTCCGtcgac. The half-sites are in uppercase and the spacer and flanking nucleotides are in lowercase.

2.5 Chapter Acknowledgements

The following people contributed to the manuscript contained in this chapter: Lisa C Watson\textsuperscript{12}, Kristopher M Kuchenbecker\textsuperscript{34}, Benjamin J Schiller\textsuperscript{12}, John D Gross\textsuperscript{5}, Miles A Pufall\textsuperscript{26}, and Keith R Yamamoto.\textsuperscript{2} L.C.W. designed, performed and analyzed experiments. J.D.G. and L.C.W. designed and performed experiments for NMR sequential assignment. K.M.K. and L.C.W. designed, performed and analyzed SPR and FRET experiments. B.J.S. and L.C.W. designed and performed ChIP-sequencing experiments, and B.J.S. analyzed

\textsuperscript{1}Tetrad Graduate Program, University of California, San Francisco, San Francisco, California, USA.
\textsuperscript{2}Department of Cellular and Molecular Pharmacology, University of California, San Francisco, San Francisco, California, USA.
\textsuperscript{3}Biophysics Graduate Group, University of California, San Francisco, San Francisco, California, USA.
\textsuperscript{4}Department of Biochemistry and Biophysics, University of California, San Francisco, San Francisco, California, USA.
\textsuperscript{5}Department of Pharmaceutical Chemistry, University of California, San Francisco, San Francisco, California, USA.
\textsuperscript{6}Present address: Department of Biochemistry, University of Iowa, Iowa City, Iowa, USA.
the data. M.A.P. and K.R.Y. contributed to the direction of project, and L.C.W. and K.R.Y. wrote the manuscript.

We thank M. Kelly (University of California (UC), San Francisco) and J. Pelton (UC Berkeley) for NMR support, J. Cheney and S. Floor for providing scripts for NMR analysis, and S. Cooper and S. Meijsing for developing GR ChIP-sequencing protocols. We thank R. Fletterick, A. Johnson, G. Narlikar and members of K.R.Y.’s lab for thoughtful discussions; and M. Knuesel and S. Meijsing for critical reading of the manuscript. This work was supported by US National Institutes of Health (NIH) grant CA020535 (K.R.Y), the Biophysics Graduate Group training grant NIHT32GM008284 (K.M.K), NIH grants GM08537 and 5K99CA149088 (M.A.P.), the Leukemia and Lymphoma Society Fellowship (M.A.P), the Larry S. Hillblom Foundation Fellowship (L.C.W), the Genentech and Sandler Foundation Graduate Fellowship (L.C.W) and the Cancer Research Coordinating Committee Fellowship (L.C.W).
Chapter 3

Glucocorticoid receptor binds half sites to induce a distinct conformational state and regulate specific target genes

Abstract

Glucocorticoid receptor (GR) is a hormone-activated, DNA-binding transcriptional regulatory factor that controls inflammation, metabolism, stress responses and other physiological processes. In vitro, GR binds as an inverted dimer to a motif consisting of two imperfectly palindromic 6 bp "half sites" separated by 3 bp "spacers". In vivo, GR employs different patterns of functional surfaces of GR to regulate different target genes. The relationships between GR genomic binding and functional surface utilization have not been defined.

We found that A477T, a GR mutant that disrupts the dimerization interface, differed from wildtype GRα in binding and regulation of target genes. Genomic
regions strongly occupied by A477T were enriched for a novel half site motif. *In vitro*, GRα bound half sites non-cooperatively and in a distinctive conformation similar to that of A477T bound at "full sites", implying that half sites are bound by GRα monomers. Through the overlap between GRα- and A477T-bound regions, we identified GRα-bound regions containing only half sites. We further identified GR target genes linked with half sites and not with the full motif.

Genomic regions bound by GR differ in underlying DNA sequence motifs and in the GR functional surfaces employed for regulation. Identification of GR binding regions that selectively utilize particular GR surfaces or depend upon certain structural changes may discriminate "sub-motifs", including the half site motif, that favor those surfaces or conformational shifts. This approach may contribute to predictive models for GR activity and therapy.

### 3.1 Background

Glucocorticoid receptor (GR, HUGO symbol NR3C1) is a DNA-binding transcriptional regulatory factor that is activated specifically by binding glucocorticoid hormones, and which regulates diverse aspects of physiology. Genome-wide measurements of gene expression in U2OS human osteosarcoma cells stably expressing GR [62] have revealed that thousands of genes are differentially expressed in response to glucocorticoid treatment (that is, regulated by GR) [4]. Moreover, U2OS lines stably expressing three mutant GR alleles, each disrupting one functional surface, have revealed that different GR target genes depend on different patterns of functional surfaces of GR for proper regulation; A477T disrupts the dimerization interface; and 30iiB (E219K/F220L/W234) and E773R, disrupt the activation function 1 and 2 (AF1, AF2) domains, respectively (Figure 3.1A) [4]. Similarly, GRγ, a minor isoform differing from the major isoform GRα by insertion of a single arginine in the "lever arm" region [5], differentially regulates some GR regulated genes, implicating
this region as another functional surface. X-ray crystallography [5] and nuclear magnetic resonance spectroscopy (NMR) [1] of GR bound to different oligonucleotides containing high-affinity binding sites indicate that GR assumes different distinct conformations at different sites. These results imply that GR also assumes different conformations in vivo at different glucocorticoid response elements (GREs), genomic regions that confer specific context-dependent GR responsiveness upon a nearby gene.

In vitro, purified GR recognizes specifically a GR binding sequence (GBS) motif composed of imperfect palindromic six base pair (bp) "half sites" separated by three bp "spacers", binding as an inverted dimer [63]. In vivo, GR occupies specific genomic GR binding regions (GBRs), identified using chromatin immunoprecipitation coupled with quantitative PCR (ChIP) or deep sequencing (ChIP-seq). The genomic locations of GBRs differ strikingly in different cell or physiologic contexts, and not all GBRs contain an identifiable GBS motif. Hence, the relationship between sequences recognized in vitro and genomic regions occupied in vivo is complex. It is reassuring that GBS motifs drive GR-regulated transcription in a simple reporter context [63], and therefore reasonable to assume that GR occupancy at GBRs that contain GBSs reflects in vivo sequence recognition. Interestingly, different GBSs, tested in isolation in vivo, trigger utilization of different patterns of GR functional surfaces [5], supporting the notion that GR assumes different conformations at these sequences in vivo. However, rules dictating the relationship between DNA sequence, GR conformation and utilization of distinct GR functional surfaces remain unknown.

We therefore examined whether sequence motifs, including the canonical GBS, found within GBRs might in fact be an aggregate of specific motifs, each biased toward utilization of particular functional surfaces of GR. In this study, we measured changes in gene expression and genomic occupancy upon glucocorticoid treatment of isogenic cultured human cell lines expressing alleles of GR bearing differences in particular functional sur-
Figure 3.1: GR alleles used in this study. Human GRα, the major isoform of GR, is a 777 amino acid protein with a modular structure typical of nuclear receptors: an N-terminal activation function 1 (AF1) domain, a DNA-binding domain (DBD), and a C-terminal ligand binding domain (LDB), which contains the activation function 2 (AF2) domain. We used mutations of GRα in AF1 (30iiB); AF2 (E773R); the dimerization interface of the DBD (A477T); and a minor isoform GRγ, which bears a single amino acid (arginine) insertion at position 471 in GRα as a result of alternative splicing. Domain boundaries are approximate, based on functional assays. Western blots show that GR was expressed at similar levels in all cell lines, which were similar to the level of endogenous GR found in the lung adenocarcinomic human alveolar basal epithelial cell line A549.
faces. We further used statistical, computational and biophysical methods to characterize changes in GR conformation and function, as well as the sequence motifs that are present at GREs, in relation to these functional surfaces.

3.2 Results

3.2.1 Different GR alleles induce diverse, gene-specific transcriptional responses

To improve our understanding of the patterns of GR functional surfaces used during transcriptional regulation, we carried out a genome-wide analysis in a series of human U2OS osteosarcoma cell lines with stably integrated GR alleles [4] (unpublished observations, M Thomas-Chollier, LC Watson, SB Cooper, M Pufall, J Liu, K Borzym, E Einfeldt, M Vingron, SH Meijsing and KR Yamamoto). We isolated RNA from U2OS cells expressing integrated GRα, GRγ, GRα 30iiB (30iiB; E219K/F220L/W234R), GRα A477T (A477T), or GRα E773R (E773R) (Figure 3.1A-B) after treatment for 2, 4, or 24 hours with 100 nM dexamethasone, a synthetic glucocorticoid that binds GR with high affinity, or for four hours with ethanol only as a "0 hour" control. Each GRα mutation disrupts the function of a distinct regulatory surface; among cells expressing the 30iiB, A477T or E773R mutations, a panel of ten regulated genes displayed six out of the eight possible patterns of surface utilization of three surfaces compared to GRα; i.e., (2 states, utilizes or does not)^(3 mutations) = 8 patterns [4]. These results imply that other functional surfaces and a multitude of patterns are likely used at regulated genes.

We assayed all gene expression from isolated RNA samples (Additional file 1). After four hours of dexamethasone treatment, thousands of genes were regulated by GR [4]. Because of the statistical complexities with assessing utilization of each of the four GR surfaces
at every regulated gene, we instead analyzed pairwise the regulation by GRα compared individually with our four GR alleles (Additional file 1). As expected, we found that regulation was affected in a highly allele-specific and gene-specific manner (Additional File 1, Figure 3.2).

We calculated the statistical significance of the difference in regulation between GRα and each allele for each gene after four hours of dexamethasone treatment (Figure 3.3), using limma, a software package for the analysis of gene expression microarray data using linear models. We instructed limma to estimate the regression coefficients (i.e., the per-gene differences in regulation between a given allele and GRα) for each such simple contrast (the difference of GRγ and GRα, etc.), as well as the standard errors of those coefficients [64]. This method allows the application of the t-test to determine significance, which may reduce the number of identified targets but ensures a high confidence gene list by properly controlling the false discovery rate [64].

We then assigned a simple phenotypic classification to each GR regulated gene based on its valence of regulation by GRα (activation or repression) and the difference in regulation by that allele and by GRα (gain, loss, or change of valence). Some phenotypic classifications were less common than others: in particular, genes that change the valence of regulation were relatively rare. Moreover, certain phenotypes appeared to be relatively allele-restricted; e.g., many genes gained activation with A477T and GRγ (103 and 36 genes, respectively), whereas few gained activation with 30iiB or E773R (6 and 1 genes, respectively) (Figure 3.3). Thus, A477T and GRγ may drive activation through one or more mechanisms unavailable to GRα, 30iiB or E773R.
Glucocorticoid regulation occurs with gene-specific utilization of GR functional surfaces. U2OS cells expressing GR alleles were treated for four hours with dexamethasone or vehicle (ethanol). Scatterplots of expression changes (log2-fold change) at 4 hours are shown with the GRα response (log2-fold change) on the x-axis and the response for another allele on the y-axis: A477T, GRγ, 30iIB, or E773R. Only genes with significant (adjusted p < 0.05) changes in expression are shown, as open circles: those with changes that were significant in GRα are pink, significant in the other allele are green, and significant in both are orange.
Figure 3.3: Differential regulation analysis reveals distinct classes of regulated genes. Scatterplots of expression changes (log2-fold change) at four hours are shown (left) as in Figure 3.2. Genes with significant differences in regulation (differences in differences in expression) with a given allele as compared to GRα are shown as filled circles with a black outline; genes lacking significant differences are shown as open circles. Only genes with significant (adjusted p < 0.05) regulation in at least one condition (GRα or a given allele) are shown. We categorized genes showing significant differences in regulation (compared to GRα) by the functional consequence of the alternative allele/mutation (right).
3.2.2 A477T and GRγ selectively occupy GBRs near genes that gain activation

Next, we performed ChIP-seq to define GBRs genome-wide in U2OS cells expressing GRα or one of the three mutant alleles (30iiB, A477T, and E773R), after treatment with dexamethasone. Defining GBRs enables identification of putative primary regulated genes (those regulated by a proximal GR-occupied presumptive GRE) and allows us to distinguish genes differentially regulated due to changes in occupancy from those with altered GR activity. GRγ has been similarly examined by others (unpublished observations, M Thomas-Chollier, LC Watson, SB Cooper, MA Pufall, J Liu, K Borzym, E Einfeldt, M Vingron, SH Meijsing and KR Yamamoto). Importantly, we did paired-end sequencing of the ChIP-seq library, which improves the dynamic range of occupancy by allowing use of both ends of the double stranded molecule to determine whether the originating fragment was unique [65]. We identified GBRs (Additional file 2) using MACS2\(^1\), a software package for the analysis of ChIP-seq data, and assigned the GBRs to genes (Additional file 3) based on proximity (see Materials and Methods, "GBR to gene assignment") [68]. We also compared occupancy between GRα and each mutant (Additional files 4 – 6) using the "diffstats" module of MACS2 (see Materials and Methods, "Computational analysis of ChIP-seq data"). To avoid considering the same GBR twice, we considered all GBRs from GRα and those GBRs from a given mutant that did not overlap with GBRs from GRα (see Additional files 13 – 15 and 17); additionally, we analyzed only GBRs with signals of at least two reads per million total reads.

\(^1\)MACS2 will be described in more detail in a forthcoming publication. Briefly, peak calling uses a Poisson test as in the first version of MACS [66] and differential occupancies are determined using a log-likelihood ratio test, and transformed to p-values using Wilks’ theorem [67]. Pre-release versions of MACS2 are available online at <https://github.com/taoliu/MACS/>. 
of GBRs differed\(^2\); and we observed the same fractional difference among cells expressing GR\(\alpha\) or E773R. Similarly, the magnitude of occupancy (i.e., the number of observed sequence reads normalized by total reads) differed at only one quarter of GBRs among cells expressing GR\(\alpha\), 30iiB, or E773R (Figure 3.4C and 3.4D). In contrast, the locations of over half of GBRs differed\(^2\) among cells expressing GR\(\alpha\) or A477T (for example, Figure 3.4A). Moreover, the magnitude of occupancy varied at most GBRs: more than half had greater occupancy by GR\(\alpha\) than by A477T (GR\(\alpha\)-selectively occupied) and a quarter had greater occupancy by A477T than by GR\(\alpha\) (A477T-selectively occupied) (Figure 3.4B).

Comparing these results with our gene expression findings, we observed substantial enrichment of GR\(\alpha\)-selectively occupied GBRs near genes that lost activation in cells expressing A477T (Figure 3.4E, Additional files 7 and 12), and of both A477T- and GR\(\alpha\)-selectively occupied GBRs near genes that gained activation (Figure 3.4F, Additional files 8 and 12). Thus, it appeared that the differentially regulated genes represent a mixture, some with altered strength of occupancy (i.e., a direct relationship between binding and regulation) and others with altered GR activity (i.e., an indirect relationship).

\(^2\)GBRs with different locations among two samples were determined as those GBRs that were detectable in one sample and not the other (by overlap) and that did not significantly differ in occupancy among the two samples. This quantification of GBRs with differing locations is only an estimate, as some GBRs that differ in occupancy may appear to have similar occupancy. This effect results from variation in the observed numbers at a given GBR, primarily due to Poisson counting error.

Figure 3.4 (following page): Location, strength of occupancy and inferred activity of GBRs compared pairwise between GR\(\alpha\) and mutants. (A) Example of allele-specific GBR location: HOXD1 gene displays proximal GR\(\alpha\)-specific and A477T-specific GBRs. (B-D) Strength of occupancy: Pie charts of relative magnitude of GBR occupancy (p < 0.05) in pairwise comparisons of GR\(\alpha\) relative to (B) A477T, (C) 30iiB, and (D) E773R: greater mutant occupancy (blue), greater GR\(\alpha\) occupancy (yellow) or not significantly different occupancy (grey). There was enrichment of GR\(\alpha\)-selectively occupied GBRs near genes that lost activation (E) in cells expressing A477T. Similarly, there was enrichment of A477T-selectively occupied GBRs near genes that gained activation (F), although such genes also displayed enrichment for GR\(\alpha\)-selectively occupied GBRs. In contrast, 30iiB and E773R GBRs were generally similar to GR\(\alpha\) GBRs in both location and strength of occupancy.
Normalized Occupancy (reads per million reads)

A

GRα

HOXD1

A477T

E773R

30iiB

B

7,310

16,231

37,311

A477T

C

4,222

3,094

27,478

30iiB

D

4,244

3,094

27,437

E773R

Δ Occupancy

(p < 0.05)

GRα-selective

Mutant-selective

N.S.D.

E

Loss of activation with A477T

enrichment (a.u.)

Distance from promoter (kb)

-100

-50

0

50

100

F

Gain of activation with A477T

enrichment (a.u.)

Distance from promoter (kb)

-100

-50

0

50

100
We further investigated whether the differentially occupied GBRs near differentially regu-
lated genes were responsible for the observed regulation. As a simple measure of regulatory
function, we cloned GBRs, selected on the basis of proximity to differentially regulated
genes and of strength of occupancy, into luciferase-based reporters and transfected them
into U2OS cells expressing either GRα or A477T. We found GRα-selective activation
with four GRα-selectively occupied GBRs, A477T-selective activation with seven A477T-
selectively occupied GBRs, and nonselective activation with five nonselectively occupied
GBRs (Additional file 9). Six of 24 GBRs did not activate transcription in either cell line,
but only two drove aberrant activation (opposing selectivity of occupancy and activation).
These results imply that certain of the differentially occupied GBRs may indeed function
to differentially regulate nearby genes. Additional experiments, using recent methods
such as endogenous genome editing [69, 70, 71, 72], will be required to assess unequivo-
cally whether a given GBR is actually a GRE, and, if so, whether it confers differential
regulation.

3.2.3 Most A477T-selectively occupied GBRs contain a half
site motif

In order to understand what sequence constraints, if any, were responsible for differential
occupancy by GRα and A477T, we performed motif analysis (Additional file 10) using
MEME-ChIP [61]. Three groups of GBRs were analyzed: (1) the thousand most-occupied
GBRs that were occupied by GRα and did not have detectable occupancy by A477T ("GRα
only", Figure 3.5), (2) the thousand most-occupied GBRs that were occupied by both GRα
and A477T with less than a 1.4-fold difference ("GRα & A477T", Figure 3.5), and (3) the
thousand most-occupied GBRs that were occupied by A477T and did not have detectable
occupancy by GRα ("A477T only", Figure 3.5). We excluded from motif analysis GBRs
that overlap with known repeats (see Materials and Methods, "Computational analysis of ChIP-seq data").

De novo motif searches with MEME [73] revealed that GBRs occupied by only GRα were enriched for the canonical GR "full site" motif, which specifies cooperative binding as a GR dimer (Figure 3.5). Allowing MEME to determine the optimal length for motifs in GBRs occupied by only GRα revealed a 17 bp GR motif (Additional file 10), which is longer than the previously reported 14 bp [17] and 15 bp motifs [74], having additional, asymmetric specificity flanking the more strongly specified half site. In contrast, GBRs occupied by only A477T were enriched for a single half site motif (Figure 3.5), which may provoke binding of a GR monomer to DNA. These results are consistent with the observations that A477T has an impaired dimerization interface and decreased cooperativity [1].

Interestingly, the A477T-derived half site motif contained an "anti-motif" that specifies preferences nearly opposite to the canonical full site motif (i.e., those nucleotides observed least frequently at full site motifs) at the positions where the second half site would normally appear (Figure 3.5). GBRs that were occupied by both GRα and A477T were enriched for an intermediate motif that consisted of a closer-to-consensus half site and a

---

**Figure 3.5 (following page): Both A477T and GRα occupy "half site" GBRs in vivo.** Motif analysis was performed using MEME-ChIP on three groups of GBRs (results shown by row): occupied by GRα only (top row), occupied by both GRα and A477T with ≤ 1.4-fold difference (second row), and occupied by A477T only (third row). GR motifs (left column) from GBRs occupied by GRα only are similar to the canonical full site. In contrast, GR motifs from GBRs occupied by A477T only consist of a "half site" motif with an "anti-motif" in place of the second half site. GR motifs from GBRs occupied by both GRα and A477T contain a very weak second half site, separated by the normal three bp spacer. GR motifs were forced to 17 bp in all samples, but an unbiased search revealed shorter 11 bp motifs at GBRs occupied by GRα and A477T, or A477T alone (Additional file 10). GR motifs are centrally enriched in all samples (p < 0.01). Motifs that were similar to those for STAT (center column) and AP1 (right column) family transcription factors were also found, although the AP1 motif was not enriched at GBRs occupied by only A477T. The bottom row shows the JASPAR motifs for STAT3 (MA0144.1) and AP1 (MA0099.2) as a reference, and the vertical dashed lines indicate their alignment with discovered motifs.
very weak second half site, separated by the normal three bp spacer (Figure 3.5). Due to
the semi-quantitative nature of ChIP-seq, some GBRs that we characterized as similarly
occupied by both GRα and A477T may actually be differentially occupied (and conversely,
some GBRs that we characterized as differentially occupied may not be). Therefore,
the weak second half site seen at GBRs occupied by both GRα and A477T may be an
experimental artifact; moreover, the presence of the "anti-motif" at A477T-selectively
occupied GBRs indicates that such GBRs indeed contain a motif distinct from that found
at GRα-occupied GBRs. Supporting this view, length-unbiased searches with MEME for
motifs at GBRs occupied by only A477T or both A477T and GRα revealed truncated
motifs that included only the first two positions corresponding to the second half site
(Additional file 10). In Figure 3.5, we display GR motifs found when forcing MEME to
search for motifs that are exactly 17 bp in length, for ease of comparison. CentriMo,
which searches for central/local enrichment of motifs in ChIP-seq data [75], showed that
GR motifs were strongly enriched at GBR signal maxima (Additional file 10), implying
direct binding by GR at these GBRs.

Discriminatory motif analysis of the three different classes (GRα-selective, A477T-
selective, and nonselective) of occupied GBRs with DREME, which exhaustively searches
for all three to eight bp regular expression motifs that are enriched relative to di-nucleotide
shuffled sequences [76], revealed motifs similar to that of activation protein 1 (AP1) and
Signal Transducers and Activators of Transcription (STAT) families of transcription fac-
tors (Figure 3.5 and Additional file 10). We did not detect any centrally enriched motifs
that might indicate a cooperative binding partner (i.e., a heterodimer) for DNA-bound
GRα or A477T.

Using DREME, we identified a STAT-like motif (CWGGAA) in 233 and 251 of 1000
GBRs occupied by GRα alone, or GRα and A477T, respectively; a different STAT-like
motif (GGAAYG) was found in 117 of 1000 GBRs occupied by A477T alone (Table 3.1). AP1-like motifs, STGAGTCA and ATGABTCA, were positively identified in 35 of 1000 GBRs occupied by GRα alone, and 105 of 1000 GBRs occupied by GRα and A477T, respectively; AP1-like motifs were not significantly enriched at GBRs occupied by A477T alone (Table 3.1). Of all 23,525 GBRs found in cells expressing GRα, we found that 1719 and 1214 had STAT3 (a canonical member of the STAT family) or AP1 motifs, respectively, but lacked GR full sites, and 3277 and 3694 had STAT3 or AP1 motifs, respectively, and contained GR full sites (see Additional files 12, 17 and Table 3.1).

GR interacts physically and functionally with both the STAT and AP1 families of transcription factors [77]. The presence of AP1 and STAT motifs at GBRs is consistent with both "tethering" of GR – protein:protein association of GR with DNA-bound AP1 or STAT – and with the existence of composite elements, elements that contain binding sites for GR and for one or more additional transcriptional regulatory factors. The greater abundances of GBRs that contained STAT3 or AP1 motifs and also GR full sites as compared to GBRs that contained these motifs and lacked GR full sites (1.4- and 3.0–fold, respectively; Table 3.1) implies that composite elements may be more common than tethering elements.

### 3.2.4 A subset of genes regulated by GRα contains only half sites

Motivated by the discovery of a half site motif near genes regulated by A477T, we investigated whether any genes were regulated by GRα recognition of half site motifs. We identified a set of genes whose regulation at 4 hours is similar (less than twofold different) between GRα and A477T at a 95% confidence level\(^2\). GBR occupancy near these

\(^2\)To determine whether fold-changes in gene expression between two conditions were less than 2-fold different at a significance level of 95%, we constructed 97.5% confidence intervals for the fold-change in gene expression under each condition and required that they do not overlap at all. The 97.5% confidence
"nonselective" genes (Additional file 10) did not reveal significant enrichment of GR\(\alpha\)- or A477T-selectively occupied GBRs relative to genes that are selectively regulated, suggesting that GBRs near these genes are bound by both GR\(\alpha\) and A477T. Consistent with this explanation, GR\(\alpha\) and A477T bind specifically and with equal affinity to half sites \textit{in vitro} [1].

We investigated whether GBRs within 20 kilo-base pairs (kb) of the transcriptional start site of nonselective, activated genes contained half sites, full sites or both. Since the sets of sequences described by the full site motif and by the half site motif are partially overlapping (Figure 3.6), we assessed the individual information, a quantitative and absolute measurement of the similarity of a sequence to a given motif [78], of the best match to each GR motif (GR\(\alpha\)-derived full site motif and A477T-derived half site motif) at each GBR (± 75 bp from the signal summit; Additional files 11 and 12). Consistent with the observed differential occupancy, genes that lost activation with A477T generally had GBRs that contained GR motifs that are good full sites (Figure 3.7), which logically also constitute good half sites. Inversely, genes that gained activation with A477T had GBRs that contained poor full sites and good half sites. Nonselective genes also had GBRs that contained poor full sites and good half sites, suggesting that GBRs near nonselective genes, like those near genes that gained activation, are bound by GR\(\alpha\) at half sites (Figure 3.7).

Next, we investigated whether there were GR\(\alpha\)-regulated genes bearing half site GBRs and apparently lacking full site GBRs (Additional files 12 and 13). This restrictive criterion should define genes most likely to be regulated by GR\(\alpha\) via half sites. We defined as a full site motif any sequence with an individual information of at least 8.94 bits (the 95th intervals were computed as the fold-change plus or minus 2.447 – the \(t\)-statistic for six degrees of freedom (eight total sample measurements minus two degrees for determining their respective means) – divided by the square root of two (because subtracting two groups of four yields an equivalent \(n\) of 2), and multiplied by the posterior standard deviation for a given gene as determined by \textit{limma}.
Found by DREME within training set of 1000 peaks (see Figure 3.5 and Additional File 10)

Found at GBRs in cells expressing GRα
STAT motif        CWGGAA        CWGGAA        GGAAYG
# of 1000 with motif 233          251           117
AP1 motif          STGAGTCA      ATGABTCA      (NA)
# of 1000 with motif 35           105           0

Found among all 23,525 GBRs in cells expressing GRα

<table>
<thead>
<tr>
<th>Motif present at GBR</th>
<th># with GR motif</th>
<th># without GR motif</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>STAT3</td>
<td>3277</td>
<td>1719</td>
<td>1.4</td>
</tr>
<tr>
<td>AP1</td>
<td>3694</td>
<td>1214</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Table 3.1: Occurrences of STAT and AP1 motifs found using DREME.

Figure 3.6: GR half sites and full sites are similar but distinct motifs. More 17mers containing half sites than whole sites are expected because the information-containing region of the motif is shorter; similarly, most whole sites contain a good half site because the constraints are similar and there are two positions available for a good half site due to the palindromic nature of dimer.
percentile of randomly permuted GBR sequences), and as a half site motif any sequence
with an individual information of at least 8.00 bits, with respect to the given motif. By
comparison, the full site and half site motifs had 11.9 bits and 9.3 bits of information,
respectively, relative to GBR background sequence (Additional file 10). We additionally
required that the full site motif’s individual information at candidate GBRs be less than the
half site motif’s individual information. In total, there were 1441 GRα-regulated genes,
607 (42%) of which had associated GBRs within 20 kb of the transcriptional start site.
We found that 35 of these 607 genes met our definition, and that this number declined
slowly with an increasing window of association (27 within 30 kb, 21 within 40 kb, 17
within 50 kb).

3.2.5 GRα binds half sites with a distinct conformation compared to full sites

To compare the conformations of GR bound to half sites and to full sites, we used nu-
clear magnetic resonance spectroscopy (NMR) to measure the chemical environment of
the amide bond of individual amino acid residues (see Materials and Methods, "Nuclear
magnetic resonance spectroscopy"). NMR spectra of complexes of purified GRα DBD
or A477T DBD bound to oligonucleotides containing either a full site GBS (gtacAGAA-
CAtttTGTTCTtcgac) or a half site (gtacAGAACAttt) were compared by chemical shift

Figure 3.7 (following page): GR half sites predominate at GBRs near nonselective,
activated genes. Individual information (measured in bits) of the best full site (x-axis) and
half site (y-axis) found at GBRs (circles) located within 20 kb of the transcriptional start
sites of a set of genes; contour lines encompass same number of GBRs, and thus localize
regions of high GBR density. Genes that lost activation with A477T (red) generally had
peaks with good full sites (> 10 bits) and good half sites (> 9 bits). Genes that gained
activation with A477T (green) generally had GBRs with poor full sites (< 5 bits) and good
half-sites (> 10 bits). Genes that were nonselectively1 activated (blue) also generally had
GBRs with poor full sites and good half sites.
Loss of activation
Gain of activation
Nonselective

Monomer sites
Dimer sites

Full site information (bits)
Half site information (bits)
Predicted

0 5 1 0 1 5

Full site information (bits)
difference analysis, which infers local conformational differences by measuring relative position, in parts per million (ppm), of each residue amide moiety. As a half site, we used a simple truncation of the full site after the spacer, rather than mutating the second half site, so as to avoid potential interference in the NMR spectrum from a minor population of GR dimers that have one partner non-specifically bound.

Previous results showed that nearly 20% of residues in the DBD bound to the full site were shifted more than 0.15 ppm with the A477T mutation [1]. Similarly, we found that 20% of residues were shifted more than 0.15 ppm with GRα DBD bound to a full site versus a half site, with a median shift of 0.07 ppm (Figure 3.8). In contrast, comparing A477T DBD bound to a full site with GRα DBD bound to a half site, we found that only 10% of residues were shifted more than 0.15 ppm (Figure 3.8) with a median shift of only 0.03 ppm. Notably, many of these residues were in the dimerization loop (residues 475-491) itself, which we expect to be perturbed because of the A477T mutation. Of the detectable amide peaks for the A477T full site complex, only two residues outside the dimerization interface were shifted compared to GRα DBD bound to a half site. Therefore, GRα binds half sites in a very similar conformation as A477T. Moreover, we fitted data from an electrophoretic mobility shift assay with GRα bound to an oligonucleotide containing a half site adjacent to the "anti-motif" (gtacAGAACAtttGTGAGGtcgac) [1] to the Hill equation and found a Hill coefficient of 1.080 ± 0.065 (mean ± standard deviation), consistent with no cooperativity and binding of a monomer.
Figure 3.8: GRα and A477T bind a half site in similar conformations. (A) Top: Overlay of $^{15}$N-HSQC spectra (see Materials and Methods, "Nuclear magnetic resonance spectroscopy") for GRα DBD bound to oligonucleotides containing either a full site GBS (gtacAGAACAtttTGGTCTtcgac; black) or a half site (gtacAGAACAttt; red). Bottom: Overlay of $^{15}$N-HSQC spectra for A477T DBD bound to an oligonucleotide containing a full site (black) and GRα DBD bound to an oligonucleotide containing a half site (red). (B) The chemical shift differences at assigned NMR peaks are displayed for the comparison between GRα DBD bound to oligonucleotides containing either a full site GBS or a half site (top) and between A477T DBD bound to an oligonucleotide containing a full site and GRα DBD bound an oligonucleotide containing a half site (bottom). Dashed vertical gray lines demarcate the dimerization loop (475-491). (C) Hill plot derived from an electrophoretic mobility shift assay describing the fraction of oligonucleotides containing a half site with an adjacent "anti-motif" (gtacAGAACAtttGTGAGGtgcag) bound at increasing GRα concentrations [1]. The Hill coefficient was computed from the Hill Equation (right) to be 1.080 ± 0.065.

Hill equation \[ \theta = \frac{1}{(1 + K^{1/2} / [GR])^n} \]

Fitted Curve

Affinity ($K^{1/2}$): 185 ± 11 nM
Hill coefficient (n): 1.080 ± 0.065
$\chi^2 = 0.009$
3.3 Discussion

3.3.1 Half site motifs suggest GR monomers bind DNA and regulate specific genes

We set out to find whether sequence motifs bound by GR might be decomposed into multiple specific motifs that each relate to the utilization of specific functional surfaces of GR. We found that A477T, a mutation in the dimerization interface of GR leading to reduced DNA-binding cooperativity and increased dissociation rates in vitro [1], resulted in gains and losses in glucocorticoid-induced gene regulation and in GBR occupancy in vivo. Analysis of A477T-occupied GBRs revealed a hitherto unrecognized half site motif. The half site motif, coupled with a failure to detect non-GR partner motifs, suggested that A477T might bind GBRs as a monomer in vivo. Furthermore, A477T regulated many genes, suggesting that GR monomers are sufficient for gene regulation. The question remains, then, whether GRα monomers bind to DNA in a sequence-specific manner and regulate a set of target genes.

Curiously, we found that some genes were regulated by A477T and not by GRα, and some GBRs were occupied by A477T and not by GRα. We speculate that GRα fails to bind such GBRs because it is titrated away by cooperatively bound, high affinity full sites, composed of "degenerate" or weak half sites; such cooperatively bound sites would be unavailable to A477T due to its reduced cooperativity. Indeed, many GRα-selectively occupied GBRs near GRα-selectively regulated genes contained full sites lacking "consensus" or perfect half sites; rather, these full sites were composed of two degenerate half sites (Figure 3.6 and 3.7). These results suggest two orthogonal binding modes for GRα: i) consensus half sites, most likely bound by monomers; and ii) degenerate full sites, most likely bound by
Alternatively, GR\(\alpha\) occupancy may not be observed at A477T-specific GBRs because GR\(\alpha\) dissociates from these regions more rapidly than A477T, reducing the signal below detection limits. However, A477T dissociates from DNA more rapidly than GR\(\alpha\) \textit{in vitro} [1] and \textit{in vivo} [79], arguing against this model. Importantly, at any given site, some fraction of binding events may involve monomers and some fraction may involve dimers. We found that a minimum of 35 genes are regulated entirely by direct binding of GR\(\alpha\) monomers; while many genes may be regulated entirely by direct, cooperative binding of GR\(\alpha\) as a dimer, our results suggest that others may be regulated in part by monomers and in part by dimers.

Additionally, we found that GBRs were enriched within 20 kb from the promoters of regulated genes (relative to the whole genome), suggesting that promoter-proximal GBRs were responsible for many observed changes in gene expression. Similarly, we found that GR\(\alpha\)- and A477T-selectively occupied GBRs were enriched near the promoters of genes that lost and gained activation, respectively, with A477T, consistent with these GBRs being responsible for gene regulation. Nonetheless, more than half of GBRs were located distal to genes (more than 50 kb from the transcriptional start site), and only 42% of GR\(\alpha\)-regulated genes were associated with promoter-proximal GBRs (\(\pm\)20 kb). Further exploration is needed to understand the function (if any) of distal GBRs.

### 3.3.2 Monomeric GR\(\alpha\) is functional in regulation

The ability of a transcription factor to function in gene regulation alternatively as a monomer or a dimer has been previously observed for SRY (sex determining region Y)-box 10 (SOX10) [80]. There are several lines of evidence suggesting that GR\(\alpha\), too, binds DNA alternatively as a monomer or a dimer. GR\(\alpha\) and A477T bind specifically and with
equal affinity to half sites \textit{in vitro} [1], with a Hill coefficient of approximately 1, consistent with binding as a monomer to half sites. Additionally, single-molecule imaging of GR binding \textit{in vivo} indicates that A477T dissociates from DNA tenfold faster than GR\(\alpha\) [79], consistent with the observation that A477T has reduced cooperativity when binding full sites \textit{in vitro} [1]. These results collectively suggest that A477T binds as a monomer \textit{in vivo}. Furthermore, deep sequencing of DNase I-treated chromatin, in combination with ChIP-seq, reveals that hormone-induced binding of androgen receptor (AR, HUGO symbol NR3C4), a related nuclear receptor, was associated with DNase footprints the size of AR monomers at two thirds of binding regions [81]. Given the significant homology between AR and GR, these results suggest that GR\(\alpha\) is also capable of binding as a monomer \textit{in vivo}.

Our results show that the equilibrium binding and conformational state of GR\(\alpha\) bound to half sites is similar to the conformational state of DNA-bound A477T and is distinct from a dimer bound with both partners making sequence-specific contacts. However, we have not excluded the possibility that both A477T and GR\(\alpha\) bind half sites as dimers \textit{in vivo} despite decreased cooperativity at half sites \textit{in vitro}. The observed half site motif at A477T-occupied GBRs requires that only one GR monomer need bind DNA in a sequence-specific manner; its partner might make only DNA backbone contacts. Because the conformation of GR\(\alpha\) bound to half sites is nearly identical to that of A477T, the same constraints must apply to both. DNase footprints measured in aggregate, and even direct imaging of individual binding events, are insufficient to demonstrate that certain GBR binding events promote regulation of target genes. Joint measurements of gene expression and GBR occupancy are the minimum necessary data to infer function of GBRs. Our finding that at least 35 regulated genes contain only GBRs with half sites within 20 kb of their transcriptional start site provides the best evidence to date that GR\(\alpha\) binds as a functional monomer \textit{in vivo}, resulting in regulation of target genes.
3.3.3 Implications of GR monomers for physiology

If GR and other nuclear receptors bind DNA alternatively as dimers or as monomers, then these two receptor species may represent distinct drug targets [82]. Glucocorticoids, like other steroid hormones, regulate diverse aspects of physiology, making them potent but blunt tools for medicine. Significant interest in "selective" or "dissociated" glucocorticoid receptor agonists [41, 83] that separate the beneficial and adverse clinical effects of glucocorticoid treatment has been driven by the success of similar drugs targeting the estrogen receptor, especially in treating breast cancer and osteoporosis [84]. Drugs targeting nuclear receptors commonly bind to a pocket in the ligand binding domain, normally bound by an endogenous hormone (cortisol in the case of human GR). Alternatively targeting the dimerization interface or DNA-binding interface to separate dimer and monomer activity may help produce improved selective modulators for GR and other receptors.

Encouragingly, drugs modulating AR conformation and activity, subsequent to hormone binding, have been previously found [85]. Our data make it difficult to infer the potential effects of such selective modulators on GR activity. However, observations of mice harboring the equivalent mutation to A477T indicate that inhibiting GR dimerization may prevent glucocorticoid-induced decreases in skin and bone collagen synthesis, as well as decreases in insulin sensitivity, while maintaining or even exaggerating glucocorticoid-induced increases in triglyceride synthesis [82]. Moreover, such an inhibitor may cause an increase in lipogenesis, possibly due to the regulation (by monomers) of target genes not normally regulated by GR [82]. A selective modulator that inhibits the regulatory function of GR dimers may have different effects from one that inhibits their formation; both would inhibit regulation of dimerization-dependent genes, but the latter may also drive regulation of new target genes, similar to those observed in the dimerization interface mutant.
3.3.4 Evolution and structure of hormone-regulated gene networks

Evidence for existence of functional AR [81] and GR monomers bound to consensus half sites suggests that similar monomeric forms may exist for all nuclear receptor subfamily 3 group C transcription factors (GR, mineralocorticoid receptor, progesterone receptor, and AR), and possibly more distantly related nuclear receptors. These results may have important implications for how hormone-regulated gene networks evolved. Binding of receptor monomers to consensus half sites may present an evolutionary path for establishing robust response elements wherein one perfect half site appears, at random or by insertion or rearrangement, and allows for regulation by monomers, while also promoting selection of a second site with the proper spacing for cooperative binding by dimers; the full site would then be resistant to mutations that might have perturbed binding by monomers. Alternatively, the half site may have preferred function, and may therefore remain stable. Analysis of mutations equivalent to A477T in other nuclear receptors and cell lines may help clarify the functional and evolutionary role of half sites.

Additionally, other GR functional surfaces, including but not limited to AF1, AF2 and the lever arm, may be associated with more narrowly defined motifs that may either be subsets of the full site GBS motif, or may differ from it. Such motifs may be detected by analyzing differentially occupied GBRs near regulated genes selectively affected by perturbations to a functional surface. In cases where binding is not perturbed, discriminatory motif analysis between GBRs proximal to differentially regulated genes and GBRs proximal to nonselectively regulated genes may yield similar answers. Moreover, structural methods such as X-ray crystallography, electron microscopy, and NMR may define directly the conformational changes induced in GR upon binding to these motifs. These approaches may confirm that GR binding to a suspected "sub-motif" induces changes in the functional
surface being studied and drives allostertic changes in remote functional surfaces.

3.4 Conclusions

GREs differ in their precise sequence motifs, in their modes of GR binding (monomer or dimer; direct or indirect), and in the functional GR surfaces required for binding or regulation. Structural changes induced in GR upon binding different "sub-motifs" (refinements of the canonical motif) may be important determinants of GR activity at particular response elements. We have shown that GR occupies in vivo many genomic regions containing half sites, and that these regions are likely responsible for regulation of a subset of target genes. Moreover, GR specifically binds to half sites in vitro, without apparent cooperativity, and thus as a monomer. Half sites are easily overlooked among those sequences that conform to the known GR full site motif, but they have distinct functional properties compared to degenerate full sites, which may be predicted to bind with similar affinity. Identification of GREs with other sub-motifs, the conformations those GREs induce, and the functional surfaces they utilize will be important steps in defining predictive models of GR activity and in developing therapeutic strategies to modulate GR activity.

3.5 Materials and Methods

3.5.1 Cell culture

U2OS cells stably transfected with rat GRα, GRα 30iiB (30iiB; E219K/F220L/W234R), GRα A477T (A477T), GRγ, or GRα E773R (E773R) were grown as previously described [4]. Parent U2OS cells do not manifest any endogenous response to glucocorticoid treatment. DNA sequencing analysis was used to confirm the presence of point mutations
in all cell lines. A549 cells were obtained from American Type Culture Collection and maintained in DMEM supplemented with 10% volume/volume (v/v) fetal bovine serum.

3.5.2 Immunoblotting

Whole cell extracts were collected using 20 mM Tris-Cl pH 8.0, 150 mM NaCl, 1% NonidetP-40 (NP-40), 1% sodium deoxycholate, 0.1% sodium dodecyl sulphate (SDS), 2 mM ethylenediaminetetraacetic acid (EDTA). Immunoblot analysis was performed as previously described [86]. Antibodies to GR (N499) and β-actin (Sigma-Aldrich) were used.

3.5.3 Gene expression measurements using microarrays

Cells were plated in 6-well plates using DMEM supplemented with 5% v/v fetal bovine serum. Dexamethasone (Sigma) at a final concentration of 100 nM was added to the medium for 2, 4, or 24 hours (hr); for the "0 hr" time point, cells were treated with vehicle (ethanol) only for 4 hr. Cells were lysed and total RNA was isolated using QIAshredder and RNeasy mini columns (Qiagen). The quality of RNA samples was evaluated by $A_{260}/A_{280}$ ratio which was at least 1.9 and the integrity was analyzed using the Bioanalyzer 2100 (Agilent) with the Experion RNA Stdsens analysis kit (Biorad). For each experimental condition 2 µg of high quality total RNA was submitted to the University of Southern California Epigenome Center. Four biological replicates of each experimental sample from independent experiments that were collected on different days were randomly placed on Illumina Human Ref8 beadchips and processed following standard Illumina procedures.
3.5.4 Genome build

All coordinates reported are for the UCSC human genome build hg19 (GRCh37).

3.5.5 Computational analysis of gene expression data

Bead-level data outputted from Illumina’s BeadScan software were read using the beadarray package [87] from Bioconductor [88] with the R programming language [89]. BASH [90] was used to correct for compact and diffuse spatial artifacts on the arrays. Bead-level data were summarized using beadarray, filtering out beads that were more than three mean absolute deviations from the median. Probe annotations were retrieved using the illuminaHumanv3.db package from Bioconductor; poor quality probes and probes not corresponding to known HUGO gene symbols were removed from the data before further processing. Bead summary data were log2 transformed and quantile normalized. We then used the non-parametric empirical Bayes method described by [91] to correct for chip-to-chip batch effects. The raw data and batch-corrected, normalized data have been deposited in NCBI’s Gene Expression Omnibus [92] and are accessible through GEO Series accession number GSE45407 [93].

The limma package [64] from Bioconductor was used to determine differential gene expression/regulation: the lmFit function was used to determine relative expression at each condition (time point, expressed allele); the contrasts.fit function was used twice, first to determine fold-change at 2, 4, and 24 hr relative to the "0 hr" time point (4 hr ethanol treatment), and second to determine the difference in fold-change between each expressed allele and GRα; the eBayes function was used to compute statistics in each case. An adjusted [94] p-value cutoff of p < 0.05 was used to determine which genes are differentially expressed. We found there was insufficient statistical power to identify many differentially
regulated genes at 2 hours. This is likely due in part to the limitations of measuring steady state total RNA levels rather than newly generated transcripts, which was not feasible at the time of this study. The ggplot2 package [95] was used to visualize the results. For a more detailed procedure see (Additional file 16).

### 3.5.6 Chromatin immunoprecipitation (ChIP)

U2OS cells stably expressing one of GRα, A477T, 30iiB, or E773R were grown in $3 \times 75 \text{cm}^2$ dishes and treated with 100 nM dexamethasone for 90 minutes (min). The following procedure is adapted from [74]. We added formaldehyde to a final concentration of 1% for 3 min at room temperature and then quenched by adding 2.5 M glycine to a final concentration of 125 mM glycine. Cells were incubated at 4°C for 10 min and then washed with PBS for 5 min. We added 5 ml ice-cold IP lysis buffer (50 mM HEPES-KOH pH 7.4, 1 mM EDTA, 150 mM NaCl, 10% v/v glycerol, 0.5% v/v Triton X-100) with 1X protease inhibitor cocktail added (PIC; Roche Complete) to each flask to lyse cells and scraped cells into 50 ml conical tubes. Lysed cells were nutated for 30 min and pelleted by centrifugation at 4°C for 5 min at 600 times gravity. After removing the supernatant, nuclei were resuspended in 900 µL RIPA buffer (10 mM Tris pH 8.0, 1 mM EDTA, 150 mM NaCl, 5% v/v glycerol, 0.1% sodium deoxycholate, 0.1% SDS, 1% v/v Triton X-100) with 1X PIC added and frozen at -80°C. The remaining ChIP procedure was performed at 4°C. Protein G Dynabeads (Invitrogen) were prepared for IP as follows: beads were pelleted (using a magnet); liquid was removed; beads were washed twice with RIPA, then resuspended in RIPA with 1X PIC and 0.5 mg/ml BSA at the original bead concentration; 12 µg N499 α-GR antibody were added per 100 µL beads, and the beads were nutated for 2 hr.

Nuclei were thawed in ice water. Chromatin from nuclei were fragmented by sonication
at 4°C in 300 μL aliquots in 1.5 ml microcentrifuge tubes using the Diagenode Biorupter for 8 cycles of 7 min (0.5 min on / 0.5 min off). Samples were spun at 14,000 revolutions per minute (rpm) and for 15 min to pellet insoluble molecules, and the supernatant was transferred into new microcentrifuge tubes. We saved 10% of each sample to isolate later as input DNA (GRα and A477T only). Protein G Dynabeads with antibody bound were pelleted, washed twice with RIPA, and resuspended at the original bead concentration in RIPA with 4x PIC and 2 mg/mL BSA. Sonicated chromatin (900 μL) from each sample was incubated with 100 μL of antibody-bound beads for 16 hr.

After pulldown, beads were washed four times with RIPA with 500 mM NaCl, then four times with LiCl buffer (20 mM Tris, 1 mM EDTA, 250 mM LiCl, 0.5% NP-40, 0.5% sodium deoxycholate), to decrease nonspecific binding. To reverse crosslinks, beads were then resuspended in 11 μL RIPA with 500 mM NaCl plus 89 μL Rev-Xlink buffer (0.7% SDS in TE pH 8.0 with 0.2 mg/ml Proteinase K). 10 μL of each input DNA sample was similarly treated by adding 1 μL 5 M NaCl and 89 μL Rev-Xlink buffer. Samples were incubated in a PCR block at 55°C for 3 hr, then 16°C for 16 hr, then column purified using the Clean & Concentrator-5 kit (Zymogen) at a DNA binding buffer:sample ratio of 5:1.

### 3.5.7 Library preparation for ChIP sequencing (ChIP-seq)

Fragments were end-repaired for 30 min at 20°C in 25 μL reactions with 1X T4 DNA Ligase Buffer w/ 10 mM ATP (NEB), 100 uM dNTPs (Invitrogen), 3.75 U T4 DNA Polymerase (NEB), 1.25 U DNA Polymerase I, Large (Klenow) Fragment (NEB), and 12.5 U T4 Polynucleotidse Kinase. Samples were column purified using the Clean & Concentrator-5 kit (Zymogen) at a DNA binding buffer:sample ratio of 2:1 following this and subsequent enzymatic reactions. dATP was added to the 3’ end of molecules by
incubating for 30 min at 37°C in 25 µL reactions with Buffer 2 (NEB), 200 µM dATP (Invitrogen), and 7.5 U Klenow Fragment (3’-＞5’ exo-). Sample concentrations were measured using PicoGreen (Invitrogen), and then ligated to sequencing adapters at a 2:1 (adapter:sample) molar ratio in 20 µL reactions with 1X T4 DNA ligase buffer w/ 10 mM ATP (NEB) with 6% PEG-8000 and 200 U T4 DNA Ligase (NEB). Libraries were amplified by PCR for 17 cycles using the standard Illumina paired-end primers, and purified by PAGE as described in [96]. Sample concentrations and size distribution were measured using the High Sensitivity DNA kit for the Bioanalyzer 2100 (Agilent) prior to sequencing.

3.5.8 Adapters and barcodes for ChIP-seq

In-house barcodes were used for ChIP samples, which were added to the 3’ ends of the sequencing adapters and determined using the first four bases of the read. Barcodes are TCAT (BC1), GACG (BC2), AGTC (BC3), and CTGA (BC4). The ChIPs for GRα and A477T were each ligated to all barcodes, pooled, and sequenced in their own lanes. Samples for 30iiB and E773R were ligated to BC1–2 and BC3–4, respectively, and pooled. Input samples were ligated to indexed TruSeq adapters using indices 5 and 7 for GRα and A477T inputs, respectively.

3.5.9 ChIP-seq procedure

ChIP samples were sequenced at the University of California San Francisco Center for Advanced Technology using the Genome Analyzer II (Illumina) with 2 × 75 bp paired end reads as per the manufacturer’s instructions. Input DNA samples were sequenced using the HiSeq 2000 (Illumina) with 2 × 100 bp paired end reads. Raw sequence reads were deposited in the Sequence Read Archive (SRA) at the National Center for Biotechnology Information (NCBI) [97] and are accessible through SRA study accession
number SRP020242.

3.5.10 Computational analysis of ChIP-seq data

We automated the analysis procedure using the Python programming language in order to make our results easily reproducible and to facilitate similar analysis by others. The scripts used – preprocess_reads.py, align2.py, call_peaks.py, and analyze.py – are described here and are available in the software package seriesoftubes on the Python Package Index (PyPI). seriesoftubes depends in part on Biopython [98]. We used preprocess_reads.py (seriesoftubes) to separate the sequencing results by barcode/index, saving only reads with valid barcodes/indices (on both ends in a pair for internal barcodes), and merged barcodes/indices as applicable to consolidate sequence data for each sample. We then used align2.py (seriesoftubes) to align sequences to the genome using bowtie2 [60], allowing a maximum insert size of 600 bp (based on our observation of an average library size of <150 bp) and saved in BAM format using samtools [99]. Note: for internal barcodes, the first five bases contain the barcode plus an additional "T"; these positions were ignored when running bowtie2.

We used call_peaks.py (seriesoftubes), which relies on the MACS2\(^2\) callpeak module, to find significantly occupied peak regions ("peaks") and summit positions using an adjusted p-value cutoff of \(p < 0.01\) and to generate pileup tracks (signal per million reads). Additionally, the MACS2 diffstats module was used to compute differential occupancy statistics and find (mutant versus GR\(\alpha\)) differentially occupied peaks at using an adjusted p-value cutoff of \(p < 0.05\). For convenience, we have created a track hub formatted for the UCSC genome browser [100] containing paired-end read alignments, signal pileups, peak region

\(^2\)MACS2 will be described in more detail in a forthcoming publication. Briefly, peak calling uses a Poisson test as in the first version of MACS [66] and differential occupancies are determined using a log-likelihood ratio test, and transformed to p-values using Wilks’ theorem [67]. Pre-release versions of MACS2 are available online at <https://github.com/taoliu/MACS/>.
locations, and peak summit locations [101].

3.5.11 GBR to gene assignment

Peaks were assigned to genes using GREAT [68], with a basal domain of ±20 kb of a gene’s primary transcriptional start site (where assignment is guaranteed) and an extended domain of ±100 kb (where assignment is made only if it does not overlap with another gene’s basal domain); if a peak was assigned to multiple genes, it was ignored in subsequent analyses.

3.5.12 Motif analysis

MEME-ChIP [61] was used for motif discovery as described in the text. For all motif analyses, we used only peak regions (± 75 bp from the peak summit) that do not overlap with repeats, found by using bedtools [102] to intersect peaks regions with the "rmsk" table from the RepeatMasker [103] track for hg19, extracted using the Table Browser [104]. Peak sequences (± 75 bp from the peak summit) were then extracted from the UCSC hg19 build of the human genome using the twobitreader package (available on PyPI). Occurrences of motifs were found using analyze.py (seriesoftubes), which relies on MOODS [105] for efficiency. JASPAR [106] motifs MA0144.1 and MA0099.2 were used to find STAT3 and AP1 sites, respectively. Peak sequences were randomly permuted and the 95th percentile of individual information for the best motif in each permuted peak sequence was used as a cutoff for determining presence of GR full site, STAT3, or AP1 motifs (8.94, 8.06, and 9.49 bits, respectively). Data were integrated for further analysis using the R programming language (Additional file 17) and visualized with ggplot2 [95].
3.5.13 Luciferase transcriptional reporters

Selected GR binding regions were cloned into the firefly luciferase reporter vector pGL4.10-E4TATA\(^3\) using conventional methods (i.e., PCR and restriction enzymes). A table of reporter names, primers, and genomic coordinates is provided in Additional file 18. We used the *Renilla* luciferase reporter vector pGL4.70-MLE2\(^4\) as a normalization control, and the mammalian expression vector p6R as "filler" DNA to reach the recommended concentration of DNA for transfections. U2OS cells stably expressing either GR\(\alpha\) or A477T were grown in three 10 cm dishes each, split, and seeded into two 96-well plates at a final density of about \(5 \times 10^4\) cells/well. The next day, cells were transfected with 3.33 ng firefly reporter, 150 pg *Renilla* reporter, and 20 ng p6R per well using PLUS Reagent and Lipofectamine (Invitrogen) according to manufacturer instructions. After 16 hours, cells were treated with 100 nM dexamethasone or vehicle only (ethanol) for 3 hours, and then luciferase expression was assayed using the Dual-Luciferase Reporter Assay System (Promega) according to manufacturer’s instructions.

3.5.14 Nuclear magnetic resonance spectroscopy

Rat GR\(\alpha\) DBD [5] and A477T DBD were purified as described [1]. Single-stranded oligonucleotides (IDT) were purified and annealed as described [5] to obtain double-stranded DNA (dsDNA) containing either a full site GBS (gtacAGAACAtttTGTTCTtcgac) or a half site (tttTGTTCTtcgac). Complexes of Nitrogen-15-\((^{15}N)\)-labeled GR\(\alpha\) or A477T DBD and unlabeled dsDNA were formed as described [1]. \(^{15}N\) heteronuclear single quantum coher-

\(^3\)pGL4.10-E4TATA was constructed by ligating the E4TATA minimal promoter 5′-tttttagtcc tatatatact cgctcgtac ttggcccttt ttacactgtg to the promoter-less vector pGL4.10 at BgIII and HindIII restriction sites. The E4TATA promoter was selected because it does not respond to dexamethasone.

\(^4\)pGL4.70-MLE2 was constructed by ligating the MLE2 promoter (engineered ML promoter with stronger induction ability) 5′-gaaggggggc tataaaaaact cgctcgtac cgctcgtc to the promoter-less vector pGL4.10 at BgIII and HindIII restriction sites. The MLE2 promoter was chosen because it does not respond to dexamethasone and drives strong expression.
ence (HSQC) spectra of GR DBD-DNA complexes were acquired on a Bruker 800 MHz spectrometer at 35°C as described [1]. Published residue assignments [1] were used to determine chemical shifts differences among spectra for GRα or A477T DBD bound to a full site GBS. Residue assignments for GRα DBD bound to a half site were determined by assignment transfer from DNA-free GRα DBD [1] by titration of half site dsDNA at a molar ratio of 0, 0.25, 0.5, 0.75 and 1.5 (half site dsDNA to DBD monomer). Chemical shift differences between equivalent residues in each pair of spectra were evaluated as the minimal Euclidean distance from each assigned GRα peak to the nearest A477T peak, after down-scaling the $^{15}$N chemical shift by one fifth [29].

### 3.6 Abbreviations

AF1: activation function 1; AF2: activation function 2; AP1: activation protein 1; AR: androgen receptor; bp: base pairs; ChIP: chromatin immunoprecipitation; ChIP-seq: chromatin immunoprecipitation sequencing; DNA: deoxyribonucleic acid; GBR: glucocorticoid receptor binding region; GBS: glucocorticoid receptor binding sequence; GR: glucocorticoid receptor; GRE: glucocorticoid receptor response element; kb: kilo-base pairs; NMR: nuclear magnetic resonance spectroscopy; STAT: Signal Transducers and Activators of Transcription

### 3.7 Additional files

The additional files are described as in the original manuscript and will be available online. Figures and code listings are reproduced here with the applicable descriptions.
3.7.1 Additional file 1: Genome-wide expression analysis

Compressed folder (zip archive) containing the following tab-delimited text files with probe-wise results of gene expression analysis using limma, including statistics and gene-level annotations: Expression_Levels_log2.txt, expression levels at all conditions; Fold-changes_vs_0hr_log2.txt, relative gene expression in all cell lines after 2, 4, and 24 hours of dexamethasone treatment versus "0 hours"; Fold-difference_vs_GRalpha_2hr_log2.txt, difference in regulation (changes in gene expression) after 2 hours of dexamethasone treatment in cells expressing A477T, 30iiB, E773R, or GRγ versus regulation in cells expressing GRα; "Fold-difference_vs_GRalpha_4hr_log2.txt" and "Fold-difference_vs_GRalpha_24hr_log2.txt", same as "Fold-difference_vs_GRalpha_2hr_log2.txt" but after 4 and 24 hours of dexamethasone treatment, respectively. Low quality and non-genic probes have been removed.

3.7.2 Additional file 2: Genome-wide description of GR binding regions (GBRs)

Gzip-compressed tarball (.tar.gz file) containing tab-delimited text files and BED files (UCSC format), created with MACS2, describing significantly enriched GBRs from ChIP-seq data for GRα (GR-WT), A477T (GR-Dim), 30iiB (GR-AF1), E773R (GR-AF2). Each allele is organized into a folder that contains three files (named as described above): [NAME]_peaks.bed and [NAME]_summits.bed files contain peak/GBR locations and (signal peak) summit locations as BED files, respectively, and [NAME]_peaks.xls files describing statistics about each peak. Chromosomal coordinates in .xls files start with 1 and intervals contain both the start and end position.
3.7.3 Additional file 3: Assignments of GBRs to genes (GREAT output)

Gzip-compressed tarball (.tar.gz file) containing tab-delimited text files with assignments of peaks to genes and vice versa for each GR allele. Assignments were made with GREAT (see Methods, "Analysis of ChIP-seq data"). Alleles and GBRs are named as in Additional file 2.

3.7.4 Additional files 4 – 6: MACS2 analysis of differential occupancy between GR$\alpha$ and mutants

Comparisons are between GR$\alpha$ and A477T (Additional file 4), 30iiB (Additional file 5), E773R (Additional file 6). Gzip-compressed tarballs (.tar.gz files) containing tab-delimited text files describing differentially occupied regions genome-wide or at peaks (GBRs) in each separate sample. Three files created with MACS2 are provided for each comparison (alleles and GBRs are named as in Additional file 6): GR-WT_vs_[NAME]_diffpeaks.xls, which lists all differentially occupied peaks; GR-WT_vs_[NAME]_diffpeaks_by_peaks1.xls and GR-WT_vs_[NAME]_diffpeaks_by_peaks2.xls, which list all peaks originally found with each mutant (annotated sample 1) or GR-WT/GR$\alpha$ (annotated as sample 2), respectively, and statistics about their differential occupancy. Chromosomal coordinates in .xls files start with 1 and intervals contain both the start and end position.
3.7.5 Additional file 7: GBR positions and strengths of occupancy at genes that lost activation with A477T

Genes are aligned at their transcription start sites with coding regions in the plus direction and promoter regions in the minus direction. GBRs are indicated as circles; size indicates the magnitude of occupancy; color indicates differential occupancy: blue indicates more binding by A477T and yellow indicates more binding by GRα. A yellow box from -20 kb to +20 kb indicates where there is significant enrichment of GRα-selective GBRs. See Figure 3.9.

3.7.6 Additional file 8: GBR positions and strengths of occupancy at genes that gained activation with A477T

Blue box from -20 kb to +20 kb indicates region of significant enrichment of GRα-selectively occupied GBRs. See Additional file 7 legend for further description of features.

3.7.7 Additional file 9: GRα and A477T selectively activate transcription with transcriptional reporters containing selectively occupied GBRs

(A) Selected GBRs (~500 bp, centered on the signal maximum) were cloned into a luciferase-based transcriptional reporter and transfected into U2OS cells expressing either GRα or A477T. (B) GBRs were selected on the basis of proximity to activated genes and strength of occupancy: GRα-selectively occupied GBRs proximal to GRα-selectively activated genes; A477T-selectively occupied GBRs proximal to A477T-selectively activated GBRs genes (with the exception of HOXD1_GRE1, which showed GRα-selective
Figure 3.9: See Additional file 7 in Section 3.7.5.
Figure 3.10: See Additional file 8 in Section 3.7.6.
occupancy); and nonselectively occupied GBRs proximal to nonselectively activated genes. GBRs drove GR\(\alpha\)-selective activation with several GR\(\alpha\)-selectively occupied GBRs, A477T-selective activation with several A477T-selectively occupied GBRs, and nonselective activation with several nonselectively occupied GBRs. Some GBRs did not activate transcription in either cell line (e.g., KCNA5_GRE1–3), but only two (HOXD1_GRE1 and CHST15_GRE1) drove aberrant activation (opposing selectivity of occupancy and activation).

3.7.8 Additional file 10: Motif analysis of selectively and non-selectively occupied GBRs

Gzip-compressed tarball (.tar.gz file) containing folders with motif results (MEME-ChIP) for three groups of GBRs, with and without enforcement of a 17 bp motif in MEME (six folders total). "GRalpha_only" and "GRalpha_only.17bp": the thousand most-occupied GBRs that were occupied by GR\(\alpha\) and did not have detectable occupancy by A477T. "GRalpha_and_A477T" and "GRalpha_and_A477T.17bp": the thousand most-GR\(\alpha\)-occupied GBRs that were occupied by both GR\(\alpha\) and A477T with less than a 1.4-fold difference. "A477T_only" and "A477T_only.17bp": The thousand most-occupied GBRs that were occupied by A477T and did not have detectable occupancy by GR\(\alpha\). Each folder contains a file named "index.html", which can be opened in a web browser to view the results. See Figure 3.5 for a summary.
Fold change (Dex/EtOH) of pGL4.10-E4TATA

Transfect cells expressing GRα or A477T
Treat with dexamethasone or ethanol
Measure luciferase after 3 hours

Figure 3.11: See Additional file 9 in Section 3.7.7.
3.7.9 Additional file 11: GBR positions and strength of occupancy at nonselectively\textsuperscript{5} activated genes

See Additional file 7 legend for a description of features.

3.7.10 Additional file 12: Motif information at GBRs occupied by GR\textsubscript{\alpha} or A477T

Gzip-compressed tarballs containing tab-delimited text files with information about the best match to the full site or half site motifs at GBRs occupied by GR\textsubscript{\alpha} or A477T. Additionally, there are files containing STAT3 (MA0144.1) and AP1 (MA0099.2) for GBRs occupied by GR\textsubscript{\alpha}. Alleles are named as in Additional file 2.

3.7.11 Additional files 13 – 15: Pairwise comparisons of GR\textsubscript{\alpha} GBRs and GBRs of mutant alleles

Gzip-compressed tab-delimited text files (.txt.gz files) with tables comparing GBRs between GR\textsubscript{\alpha} and A477T (Additional file 13), 30iiB (Additional file 14), or E773R (Additional file 15). The comparison between GR\textsubscript{\alpha} and A477T (Additional file 13) includes information about full site and half motifs. Each file contains all GBRs from GR\textsubscript{\alpha} and those GBRs from a given mutant that do not overlap with GBRs from GR\textsubscript{\alpha}. Applicable information is merged from Additional files 3 – 6, 10, and 12. Alleles are named as in Additional file 2.

\textsuperscript{5}To determine whether fold-changes in gene expression between two conditions were less than 2-fold different at a significance level of 95%, we constructed 97.5% confidence intervals for the fold-change in gene expression under each condition and required that they do not overlap at all. The 97.5% confidence intervals were computed as the fold-change plus or minus \(2.447 \times \text{t}^{\text{-statistic for six degrees of freedom}}\) divided by the square root of two (because subtracting two groups of four yields an equivalent n of 2), and multiplied by the posterior standard deviation for a given gene as determined by limma.
Figure 3.12: See Additional file 11 in Section 3.7.9.
Additional file 2. See Additional file 17 for details about the construction of these files.

### 3.7.12 Additional file 16: R script for analysis of gene expression microarray data (text file)

See Materials and Methods, "Computational analysis of gene expression data" for details.


```r
# Written by Benjamin Schiller
# Keith Yamamoto lab, UC San Francisco 2013
# Using Bioconductor 2.10
# Using R 2.15.0
source("http://bioconductor.org/biocLite.R")
biocLite()
biocLite('beadarray')
biocLite("illuminaHumanv3.db")
biocLite("limma")

library("illuminaHumanv3.db")
library("beadarray")
library("limma")

# Raw data are available at GEO (GSE45407)
# You must extract each set of 8 arrays into its own directory, i.e.
# 4858944020A–4858944020H go in "4858944020", etc.
# some files are duplicated within each sample. it's okay to overwrite/ignore
# those, but some files names are the same between directories
USE_IMAGES=FALSE # Use bead-level data, not images
# Set this yourself after extracting data
inputDirectory = getwd()
# create a new output directory
repeat {
  i = 0
  outputDirectory = file.path(inputDirectory, paste(Sys.Date(), i, sep='.'))
  if(!file.exists(outputDirectory)) {
    i = i + 1
  } else break
}
dir.create(outputDirectory)
arrayDirs = c("4858944020", "5098941038", "5098941045", "5262237036", "5262237068")
```

87
arrayPaths = file.path(inputDirectory, arrayDirs)

resultsDir <- file.path(outputDirectory, "results")
dir.create(resultsDir)

targets <- lapply(paste(arrayPaths,"targets.txt",sep="_
"), read.table, sep="\t", header=TRUE, as.is=TRUE, na.strings=NA, comment='')

# Populate some convenient variables
alltargets = Reduce(rbind, targets)
ids=alltargets$SampleID
originNames <- c("U2OS-GRalpha", "U2OS-Dim", "U2OS-30iib", "U2OS-AF2", "U2OS-GRgamma")
cellLabels = c("alpha", "A477T", "30iib", "E773R", "gamma")
cells=factor(alltargets$Origin, levels=originNames, labels=cellLabels)
times=factor(alltargets$Time)

dataOrder of cells of interest (WT first)
cellTypes = levels(cells)

# Order of time points
timeZeros = paste(cellTypes, "0hr", sep=".
")
timeSeries = c("2hr", "4hr", "24hr")
timeSeriesFull <- as.vector(t(sapply(cellLabels, paste, timeSeries, sep=".
")))
timeOrder <- c(timeZeros, timeSeriesFull)
sampleConditions = factor(paste(cells, times, sep=".
"), levels=timeOrder)

# function to read bead-level data
readMyData <- function(arrayPath) {
  output <- readIllumina(
    dir=arrayPath,
    useImages=USE_IMAGES,
    illuminaAnnotation='Humanv3'
  )
  return(output)
}

# function to apply BASH (masks spatial regions with outliers)
applyBash <- function(BLData) {
  for (i in 1:arraySize) {
    bashOutput <- BASH(BLData, array=i, useLocs=FALSE, extended=FALSE)
    BLData <- setWeights(BLData, wts=bashOutput$wts, array=i)
    remove(bashOutput)
    gc()
  }
  return(BLData)
}
# read data
BLDatas <- lapply(arrayPaths, readMyData)
# apply BASH to data
BLDatas <- lapply(BLDatas, applyBash)

# Create bead summary data
myMedian = function(x) median(x, na.rm=TRUE)
myMAD = function(x) mad(x, na.rm=TRUE)
greenCorrectedTransform <- function(BLData, array) {
  x = getBeadData(BLData, array = array, what = "Grn. Corrected")
}

greenChannel = new("illuminaChannel", greenChannelTransform =
  illuminaOutlierMethod, myMedian, myMAD, "Grn")

BSDatas = lapply(BLDatas, summarize, list(greenChannel))
beadSummary = Reduce(combine, BSDatas)

# IMPORTANT ANNOTATION DATA – CHANGE THIS IF YOU ARE NOT USING HUMANv3 ARRAYS
library(illuminaHumanv3.db)
illuminalDs = rownames(fData(beadSummary))
chr = mget(illuminalDs, illuminaHumanv3CHR, ifnotfound=NA)
chrloc = mget(illuminalDs, illuminaHumanv3CHRLOC, ifnotfound=NA)
chrlocend = mget(illuminalDs, illuminaHumanv3CHRLOCEND, ifnotfound=NA)
refseq = mget(illuminalDs, illuminaHumanv3REFSEQ, ifnotfound = NA)
genename = mget(illuminalDs, illuminaHumanv3GENENAME, ifnotfound = NA)
geneID = mget(illuminalDs, illuminaHumanv3SYMBOL, ifnotfound=NA)
egID = mget(illuminalDs, illuminaHumanv3ENTREZREANNOTATED, ifnotfound=NA)

anno = cbind(ID = as.character(illuminalDs), Symbol = as.character(geneID),
  Chr = paste("chr", as.character(chr), sep=""),
  Loc = as.character(chrloc), Locend = as.character(chrlocend ),
  EntrezGene = as.character(egID), Name = as.character(genename))

# note and remove control probes
control_probes = which(fData(beadSummary)[,3] != "regular")
# and bad quality probes
qual <- unlist(mget(illuminalDs, illuminaHumanv3PROBEQUALITY, ifnotfound=NA))
rem <- qual == 'No match' | qual == 'Bad' | is.na(qual)
# and non-genic transcripts (without a symbol)
no_symbol <- geneID=="NA"

# remove probes that are bad quality or do not map to genes
goodRegularProbes <- which(fData(beadSummary)[,3]=='regular' & !rem & !
  no_symbol)
beadSummary.quantileOnlyGoodProbes <- normaliseIllumina(beadSummary[ goodRegularProbes,],
  method='quantile')
gamma.NAs=FALSE
tmp
b. prior
n.

n.batches
n.batch

}
an.prior

annoGenes = anno[goodRegularProbes,]

# negativeControlProbes <- which(fData(beadSummary)[,3]=='negative')

# Modified Bayes method, based on ComBat.R
arrayNumber <- unlist(lapply(arrayDirs, rep, 8))
saminfo <- data.frame("Array name" = arrayNumber, 
       Batch= factor(arrayNumber, labels=c(1:10)), 
       "Covariate 1" = cells, "Covariate 2" = times)
design <- design.mat(saminfo)
batches <- list.batch(saminfo)
n.batch <- 10
n.batches <- sapply(batches, length)
n.array <- sum(n.batches)
dat <- exprs(beadSummary.quantileOnlyGoodProbes)

cat("Standardizing Data across genes\n")
NAs=FALSE
if (!NAs){B. hat <- solve(t(design)%*%design)%*%t(as.matrix(design))}else{B. hat=apply(dat,1,Beta.NA,design)} #Standerization Model
grand.mean <- t(n.batches/n.array)%*%B. hat[1:n.batch,]
if (!NAs){var.pooled <- ((dat-t(design%*%B. hat))^2)%*%rep(1/n.array,n.array)}else{var.pooled <- apply(dat-t(design%*%B. hat),1, var,na.rm=T)}

stand.mean <- t(grand.mean)%*%t(rep(1,n.array))
if(!is.null(design)){tmp <- design ; tmp[,c(1:n.batch)] <- 0; stand.mean <- stand.mean+t(tmp%*%B. hat)}
s.data <- (dat-stand.mean)/(sqrt(var.pooled)%*%t(rep(1,n.array))
)
##Get regression batch effect parameters
cat("Fitting L/S model and finding priors\n")
batch.design <- design[,1:n.batch]
if (!NAs){gamma.hat <- solve(t(batch.design)%*%batch.design)%*%t(batch.design)}else{gamma.hat=apply(s.data,1,Beta.NA ,batch.design)}
delta.hat <- NULL
for (i in batches){
delta.hat <- rbind(delta.hat, apply(s.data[,i], 1, var,na.rm=T))
}
##Find Priors
gamma.bar <- apply(gamma.hat, 1, mean)
t2 <- apply(gamma.hat, 1, var)
a. prior <- apply(delta.hat, 1, aprior)
b. prior <- apply(delta.hat, 1, bprior)
par(mfrow=c(2,2))
tmp <- density(gamma.hat[1,])
plot(tmp, type='l', main="Density Plot")
xx <- seq(min(tmp$x), max(tmp$x), length=100)
lines(xx,dnorm(xx,gamma.bar[1],sqrt(t2[1])),col=2)
qqnorm(gamma.bar[1],col=2)
qqline(gamma.bar[1],col=2)
tmp <- density(delta.hat[1,])
invgam <- 1/rgamma(ncol(delta.hat),a.prior[1],b.prior[1])
tmpl <- density(invgam)
plot(tmpl, typ='l', main="Density Plot", ylim=c(0,max(tmp$y,tmpl$y)))
lines(tmpl1, col=2)
qqplot(delta.hat[1,] , invgam , xlab="Sample Quantiles", ylab='Theoretical Quantiles')
lines(c(0,max(invgam)), c(0,max(invgam)), col=2)
title('Q-Q Plot')
gamma.star <- delta.star <- NULL
cat("Finding nonparametric adjustments
")
for (i in 1:n.batch){
temp <- int.eprior(as.matrix(s.data[,batches[[i]]]),gamma.hat[i,],delta.hat[i,])
gamma.star <- rbind(gamma.star ,temp[1,])
delta.star <- rbind(delta.star ,temp[2,])
}
### Normalize the Data ###
cat("Adjusting the Data\n")
bayesdata <- s.data
j <- 1
for (i in batches){
bayesdata[,i] <- (bayesdata[,i]−t(batch.design[i,]%*%gamma.star))/(sqrt(delta.star[j,]%*%t(rep(1,n.batches[j]))))
j <- j+1
}
bayesdata <- (bayesdata* (sqrt(var.pooled)%*%t(rep(1,n.array))))+stand.mean
# These data were saved and submitted to GEO
detection(beadSummary) <- calculateDetection(beadSummary)
i <- as.vector(matrix(t(apply(1:80,1:160)),c(1,160)))
# Raw data
write.table(file="Raw data for GEO.txt", cbind(x,x2)[,i], quote=F, sep='\t')
# Batch-corrected data
write.table(bayesdata, file="Batch-corrected data for GEO.txt", sep="\t")
# Justification of batch-correction
colorByChip <- rep(sapply(c("red","orange","yellow","green","lightblue",
"darkblue", "purple", "gold", "deeppink", "black"), rep, 8))
# MDS before correction
plotMDS(exprs(beadSummary.quantileOnlyGoodProbes)[,sampleOrder],
               cbind(x,x2)[,i], quote=F, sep='\t')
dim.plot=c(1,2), cex=.75, col=colorByChip[sampleOrder],
labels=sampleConditions[sampleOrder])
# MDS after correction
plotMDS(bayesdata[, sampleOrder[i:j]],
      dim.plot=c(1,2), cex=.75, col=colorByChip[sampleOrder],
      labels=sampleConditions[sampleOrder])

# This makes Additional file 1

# Proceed with limma on Bayes-corrected data
design <- model.matrix(~0 + sampleConditions)
lmAllFitBD <- lmFit(bayesdata, limmaDesign)
elmAllFitBD <- eBayes(lmAllFitBD)
# make contrast matrix for fold-change calculations (vs 0hr)
cont.matrix <- array(0, c(length(design[1,]),
                          length(cellLabels) * length(timeSeries)))
rownames(cont.matrix) <- colnames(design)
colnames(cont.matrix) <- timeSeriesFull
nT <- length(timeSeries); nC <- length(cellLabels);
for(j in 1:nT) {
  cont.matrix[1:nC, 1 + (j-1)*nC:(j*nC)] <- -diag(nC)
  cont.matrix[(1 + j*nC):(nC + j*nC), 1 + (j-1)*nC:(j*nC)] <- diag(nC)
}

# fit contrasts for fold-change calculations (vs 0hr)
fittedContrastsBD <- contrasts.fit(lmAllFitBD, cont.matrix)
efittedContrastsBD <- eBayes(fittedContrastsBD)
# build contrast matrices for mutant vs wildtype (one per timepoint)
getVector <- function(mut, t) {
x <- rep(0,15); x[1 + 5*t + mut] <- 1;
x[1 + 5*t + mut] <- 1; x}
comparisons <- lapply(0:2, function(x) sapply(1:4, getVector, x))
for (i in 1:3) {
  colnames(comparisons[[i]]) <- c("Dim vs alpha", "AF1 vs alpha", "AF2 vs alpha", "Gamma vs alpha")
  rownames(comparisons[[i]]) <- colnames(cont.matrix)
}
fittedContrastsBD <- lapply(comparisons, function(x) contrasts.fit(fittedContrastsBD, x))
econtrastsBD <- lapply(fittedContrastsBD2, eBayes)

# Write results
write.fit(elmAllFitBD, 
  file=file.path(resultsDir, 'Expression_Levels_log2'),
  adjust='fdr')
write.fit(efittedContrastsBD, 
  file=file.path(resultsDir, 'Fold-changes_vs_0hr_log2'),
  results=decideTests(eBayes(fittedContrastsBD), method='global',
                      adjust='fdr')
write.fit(econtrastsBD[[1]], 
  file=file.path(resultsDir, 'Fold-difference_vs_GRalpha_2hr_log2'),
  adjust='fdr')
3.7.13 Additional file 17: R script for GBR analysis (text file)

Assembly of Additional files 13 – 15 and combination of gene expression data (Additional file 1) with occupancy data. See Materials and Methods, "Computational analysis of
# Written by Benjamin Schiller
# Keith Yamamoto lab, UC San Francisco 2013
# WT is GRalpha, Dim is A477T, AF1 is 30iiB, AF2 is E773R
# *summits.bed* files are from Additional file 2.tar.gz
# repeats.bed is rmsk table exported from RepeatMasker track
# in Variation and Repeats group on UCSC Table Browser
# Use bedtools to get information on which peaks are in repeats, as follows:
# bedtools intersect --a GR-WT/GR-WT_summits.bed --b repeats.bed --c > WT_repeat_anno.bed
# bedtools intersect --a GR-Dim/GR-Dim_summits.bed --b repeats.bed --c > Dim_repeat_anno.bed
# bedtools intersect --a GR-AF1/GR-AF1_summits.bed --b repeats.bed --c > AF1_repeat_anno.bed
# bedtools intersect --a GR-AF2/GR-AF2_summits.bed --b repeats.bed --c > AF2_repeat_anno.bed

```
WT.RepeatAnno <- read.table('WT_repeat_anno.bed', as.is=TRUE)
Dim.RepeatAnno <- read.table('Dim_repeat_anno.bed', as.is=TRUE)
AF1.RepeatAnno <- read.table('AF1_repeat_anno.bed', as.is=TRUE)
AF2.RepeatAnno <- read.table('AF2_repeat_anno.bed', as.is=TRUE)
```

# Read in differential occupancy data (from Additional files 4–6)
```
Dim.Dim <- read.table('GR-WT_vs_GR-Dim_diffpeaks_by_peaks1.xls',
                     header=TRUE, as.is=TRUE)
Dim.WT <- read.table('GR-WT_vs_GR-Dim_diffpeaks_by_peaks2.xls',
                     header=TRUE, as.is=TRUE)
AF1.AF1 <- read.table('GR-WT_vs_GR-AF1_diffpeaks_by_peaks1.xls',
                      header=TRUE, as.is=TRUE)
AF1.WT <- read.table('GR-WT_vs_GR-AF1_diffpeaks_by_peaks2.xls',
                      header=TRUE, as.is=TRUE)
AF2.AF2 <- read.table('GR-WT_vs_GR-AF2_diffpeaks_by_peaks1.xls',
                      header=TRUE, as.is=TRUE)
AF2.WT <- read.table('GR-WT_vs_GR-AF2_diffpeaks_by_peaks2.xls',
                      header=TRUE, as.is=TRUE)
```

# Add in repeat information
```
AF2.WT$is_repeat <- AF1.WT$is_repeat <- Dim.WT$is_repeat <- Dim.RepeatAnno$V6 > 0
Dim.Dim$is_repeat <- Dim.RepeatAnno$V6 > 0
AF1.AF1$is_repeat <- AF1.RepeatAnno$V6 > 0
AF2.AF2$is_repeat <- AF2.RepeatAnno$V6 > 0
```

# Here's how to produce peak regions for MEME analysis (Additional file 10)
```
get1000Peaks <- function(subset) {
    # select only a thousand from chr1–23 + X
    validChrs <- c("chr1", "chr2", "chr3", "chr4", "chr5", "chr6", "chr7"
```
```
chr8", "chr9", "chr10", "chr11", "chr12", "chr13", "chr14",
"chr15", "chr16", "chr17", "chr18", "chr19", "chr20",
"chr21",
"chr22", "chr23", "chrX")
summits <- tail(subset[which(subset$chr %in% validChrs),], 1000)$summit - 1
chr <- tail(subset[which(subset$chr %in% validChrs),], 1000)$chr
peaks <- data.frame(chr=chr, start = summits - 75, end = summits + 76)
return(peaks)
}
writePeaks <- function(peaks, filename) {
 write.table(peaks, file=filename, row.names=FALSE, col.names=FALSE, quote=FALSE)
}

# Get a group of peaks that don't change + reasonable length cutoff
somePeaks <- which(abs(Dim.WT$log2.fold.change.w.pseudocounts) < 0.5 & Dim.WT$length < 700 & !Dim.WT$is.repeat)
subsetNoChange <- Dim.WT[somePeaks, ][order(Dim.WT[somePeaks,]$treat2),]
# not differential, and is a peak in both samples
subsetNoChange <- subsetNoChange[which(is.na(subsetNoChange$diff.peakname) & !is.na(subsetNoChange$peakname1) & !is.na(subsetNoChange$peakname2)),]

# for GRalpha_and_A477T (WT & Dim)
writePeaks(get1000Peaks(subsetNoChange), 'wt_and_dim_same.bed')

# things that are WT-specific
# Get a group of peaks that do change at least 2-fold + reasonable length cutoff
# + not a repeat
somePeaks <- which(Dim.WT$log2.fold.change.w.pseudocounts < -1 & Dim.WT$length < 700 & !Dim.WT$is.repeat)
subsetWTonly <- Dim.WT[somePeaks, ][order(Dim.WT[somePeaks,]$treat2),]
# yes differential, and is a peak in 2
subsetWTonly <- subsetWTonly[which(!is.na(subsetWTonly$diff.peakname) & is.na(subsetWTonly$peakname1) & !is.na(subsetWTonly$peakname2)),]
writePeaks(get1000Peaks(subsetWTonly), 'wt_only.bed')

# things that are Dim-specific
# Get a group of peaks that do change at least 2-fold + reasonable length cutoff
```
somePeaks <- which(Dim.Dim$log2.fold.change.w.pseudocounts > 1 & Dim.Dim$length < 700 & !Dim.Dim$is.repeat)

# ordered by treatment1
subsetDimonly <- Dim.Dim[which(Dim.Dim$length <700 & !is.na(Dim.Dim$diff.peakname) & !is.na(Dim.Dim$peakname1) & is.na(Dim.Dim$peakname2) ), ]
writePeaks(get1000Peaks(subsetDimonly), 'dim_only.bed')

# Convert BED files to FASTA and then run MEME-ChIP

# How to make tables with all data merged (Additional files 13–15)
getMutantPeaksNearGenes <- function(mutantPeaks, mutantPeaksToGenesFile, removeWT=TRUE) {
  mutantPeaksNearGenes <- read.table(mutantPeaksToGenesFile, sep='\t', na.strings=c("NA", "NONE"))
  names(mutantPeaksNearGenes) <- c("peaks", "gene", "distance")
  # remove peaks that are not assigned to only one gene
  mutantPeaksNearGenes <- mutantPeaksNearGenes[!duplicated(mutantPeaksNearGenes$peaks), ]
  rownames(mutantPeaksNearGenes) <- mutantPeaksNearGenes$peaks
  mutantPeaksNearGenes2 <- merge(mutantPeaks, mutantPeaksNearGenes, by='row.names', all.x=TRUE, sort=FALSE)
  rownames(mutantPeaksNearGenes2) <- mutantPeaksNearGenes2$Row.names
  mutantPeaksNearGenes2$Row.names <- NULL
  mutantPeaksNearGenes2$intensity <- apply(mutantPeaksNearGenes2[, c("treat1", "treat2")], 1, max)
  mutantPeaksNearGenes2$size <- log10(mutantPeaksNearGenes2$intensity)
  mutantPeaksNearGenes2$dcategory <- factor("N.S.D.", levels=c("Mutant", "WT", "N.S.D."))
  mutantPeaksNearGenes2[which(mutantPeaksNearGenes2$log2.fold.change.w.pseudocounts > 0 & !is.na(mutantPeaksNearGenes2$diff.peakname)), "dcategory"] <- "Mutant"
  if (removeWT){
    return(mutantPeaksNearGenes2[which(is.na(mutantPeaksNearGenes2$peakname2)) , ])
  } else return(mutantPeaksNearGenes2)
}

getWildtypePeaksNearGenes <- function(wtPeaks, wtPeaksToGenesFile) {
  wtPeaksNearGenes <- read.table(wtPeaksToGenesFile, sep='\t', na.strings=c("NA", "NONE"))
  names(wtPeaksNearGenes) <- c("peaks", "gene", "distance")
  # remove peaks that are not assigned to only one gene

wtPeaksNearGenes <- wtPeaksNearGenes[which(!duplicated(wtPeaksNearGenes$peaks)),]
rownames(wtPeaksNearGenes) <- wtPeaksNearGenes$peaks
wtPeaksNearGenes2 <- merge(wtPeaks, wtPeaksNearGenes, 
by="row.names", all.x=TRUE, sort=FALSE)
rownames(wtPeaksNearGenes2) <- wtPeaksNearGenes2$Row.names
wtPeaksNearGenes2$intensity <- apply(wtPeaksNearGenes2[, c("treat1", "treat2")], 1, max)
wtnames <- NULL
wtPeaksNearGenes2$intensity <- apply(wtPeaksNearGenes2, 1, max)
wtnames <- apply(wtPeaksNearGenes2[, c("treat1", "treat2")], 1, max)
wtnames <- NULL
wtPeaksNearGenes2$size <- log10(wtPeaksNearGenes2$intensity)
wtnames <- apply(wtPeaksNearGenes2[, c("treat1", "treat2")], 1, max)
wtnames <- NULL
wtPeaksNearGenes2$dcategory <- factor("N.S.D.", levels=c("Mutant", "WT", "N.S.D."))
wtnames <- apply(wtPeaksNearGenes2, 1, max)
wtnames <- NULL
}
return(wtPeaksNearGenes2)

# Use GREAT to get genes near summits for each GR allele
# That data needs to be parsed into the format of peak, gene
# Settings: +/-20 kb basal domain, up to +/- 100 kb extension
# read results after parsing with python to "peakname\tgene\tcoordinate"
# format
# Here’s some Python code if you need help:
#listify = lambda x: x.rstrip().split()
#
#def printify(f):
#    ""
#    Lines as #peak, gene, distance"
#    ""
#out = []
#peak = ''
#for x in f:
#    L = [listify(x) for x in f]
#    if len(L) > 2: # ignore commented/blank lines
#        peak, rest = L[0], L[1:]
#        for i in range(0, len(rest), 2):
#            gene, dist = rest[i], rest[i+1].rstrip(',').split('-')[1:-1]
#            out.append('%s\t%s\t%0.2f\t%0.2f\n' % (peak, gene, dist))
#    return out
#
#open('newfile').writelines(printify(open('oldfile'))) # do the work
#
# Back to R code. Grabbing GREAT results
Dim.byWT.PNG <- getWildtypePeaksNearGenes(WTpeaks, '/data/WT_GREAT_peaks_to_genes.txt')
Dim.PNG <- getMutantPeaksNearGenes(Dim.Dim, '/data/Dim_GREAT_peaks_to_genes.txt')
AF1.byWT.PNG <- getWildtypePeaksNearGenes(AF1.WT, '/data/WT_GREAT_peaks_to_genes.txt')
AF1.PNG <- getMutantPeaksNearGenes(AF1.AF1, '/data/AF1_GREAT_peaks_to_genes.txt')
AF2.byWT.PNG <- getWildtypePeaksNearGenes(AF2.WT, '/data/WT_GREAT_peaks_to_genes.txt')
AF2.PNG <- getMutantPeaksNearGenes(AF2.AF2, '/data/AF2_GREAT_peaks_to_genes.txt')
Dim.byWT.PNG$origin <- AF1.byWT.PNG$origin <- AF2.byWT.PNG$origin <- "WT"
Dim.PNG$origin <- "Dim" AF1.PNG$origin <- "AF1" AF2.PNG$origin <- "AF2"

WT.WTmotifs <- read.table(file='GR-WT.full_sites.txt', header=TRUE, sep='\t')
Dim.WTmotifs <- read.table(file='GR-Dim.full_sites.txt', header=TRUE, sep='\t')
WT.Dimmotifs <- read.table(file='GR-WT.half_sites.txt', header=TRUE, sep='\t')
Dim.Dimmotifs <- read.table(file='GR-Dim.half_sites.txt', header=TRUE, sep='\t')

rownames(WT.WTmotifs) <- WT.WTmotifs$peak_ID
rownames(WT.Dimmotifs) <- WT.Dimmotifs$peak_ID
rownames(Dim.WTmotifs) <- Dim.WTmotifs$peak_ID
rownames(Dim.Dimmotifs) <- Dim.Dimmotifs$peak_ID

# merge full site / half site motif information for wt samples, dim samples
WT.merged <- merge(WT.WTmotifs, WT.Dimmotifs, by='row.names', suffixes=c("WT", "Dim"), all.x=T, sort=F)
Dim.merged <- merge(Dim.WTmotifs, Dim.Dimmotifs, by='row.names', suffixes=c("WT", "Dim"), all.x=T, sort=F)
rownames(WT.merged) <- WT.merged$Row.names
rownames(Dim.merged) <- Dim.merged$Row.names

# merge all motif information with main table
fullWT.Dim.PNG <- merge(WT.Dim.PNG, rbind(WT.merged, Dim.merged), by='row.names', suffixes=c("WT", "motif"), all.x=T, sort=F)
rownames(fullWT.Dim.PNG) <- fullWT.Dim.PNG$Row.names

# Write everything to files
write.table(fullWT.Dim.PNG, file='Additional file 13.txt', quote=FALSE, row.name=FALSE, sep='\t')
write.table(WT.AF1.PNG, file='Additional file 14.txt', quote=FALSE, row.name=FALSE, sep='\t')
write.table(WT.AF2.PNG, file='Additional file 15.txt', quote=FALSE, row.
name=FALSE, sep="\t")

# If you want gene classes too, you can get that from
geneResultsVs0 <- read.table("Fold-difference_vs_GRalpha_4hr_log2", sep="\t", header=TRUE, as.is=TRUE, quote="")
geneResultsVsWT <- read.table("Fold-changes_vs_0hr_log2", sep="\t", header=TRUE, as.is=TRUE, quote="")

# Note this is function actually grabs probes not genes
# you need to check for consistency across genes that multiple probes
getSelectiveGenes <- function(vs0=geneResultsVs0, vsWT=geneResultsVsWT, vsWT.col=2, vs0.cols=c(7,8)) {
  selective <- which(vsWT[,vsWT.col] != 0)
  regulationData <- cbind(vsWT[selective, vsWT.col],
                          vs0[selective, vs0.cols],
                          vs0[selective, "Genes.Symbol"])
  colnames(regulationData)[[1]] <- names(vsWT)[[vsWT.col]]
  colnames(regulationData)[[4]] <- "Gene.Symbol"
  return(as.data.frame(regulationData))
}

# if you want dim (A477T)
Dim.selective <- getSelectiveGenes(vs0=geneResultsVs0, vsWT=geneResultsVsWT,
                                   vsWT.col=20, vs0.cols=c(69,70))
# if you want AF1 (30iIB)
AF1.selective <- getSelectiveGenes(vs0=geneResultsVs0, vsWT=geneResultsVsWT,
                                    vsWT.col=21, vs0.cols=c(69,71))
# if you want AF2 (E773R)
AF2.selective <- getSelectiveGenes(vs0=geneResultsVs0, vsWT=geneResultsVsWT,
                                    vsWT.col=22, vs0.cols=c(69,72))
# if you want gamma
Gamma.selective <- getSelectiveGenes(vs0=geneResultsVs0, vsWT=geneResultsVsWT,
                                       vsWT.col=23, vs0.cols=c(69,73))

# Note Coef columns are like with limma
# +1 = up-regulated or greater in mutant
# -1 = down-regulated or greater in wt

# Assign phenotype. Dim as an example
Dim.selective$category <- "NA"
Dim.selective$category[which(dimSelective[,1]==1 & dimSelective[,2] !=
                           -1 &
                           dimSelective[,3] == 1)] <- "Gain of activation"
Dim.selective$category[which(dimSelective[,1]==-1 & dimSelective[,2] ==
                             1 &
                             dimSelective[,4] == 1)] <- "Loss of activation"
Dim.selective$Category [which(dimSelective[,1]==1 & dimSelective[,2]==
−1 &
   dimSelective[,3] == 1)] <- "Loss of activation"

Dim.selective$Category [which(dimSelective[,1]==−1 & dimSelective[,2]==
1 &
   dimSelective[,3] == −1)] <- "repression -> activation"

Dim.selective$Category [which(dimSelective[,1]==−1 & dimSelective[,2]==
1 &
   dimSelective[,3] == −1)] <- "activation -> repression"

Dim.selective$Category [which(dimSelective[,1]==1 & dimSelective[,2]==
−1 &
   dimSelective[,3] != 1)] <- "Gain of repression"

Dim.selective$Category [which(dimSelective[,1]==1 & dimSelective[,2]==
−1 &
   dimSelective[,3] != 1)] <- "Loss of repression"

# Note: Again, there may be multiple records per gene, and they may
conflict

Source Code 3.2: See Additional file 17 in Section 3.7.13

3.7.14 Additional file 18: Table of primers used for reporters (.xls)

3.8 Chapter Acknowledgements

The following people contributed to the manuscript contained in this chapter: Benjamin
J Schiller⁶, Rajas Chodankar⁷, Lisa C Watson⁶, Michael R Stallcup⁷, and Keith R Yamamoto⁶. BJS participated in the conception and design of the study, measurements of

⁶Department of Cellular and Molecular Pharmacology, University of California, 600 16th Street, S-222, San Francisco, CA 94158-2280
⁷Department of Biochemistry and Molecular Biology, University of Southern California, 1441 Eastlake Avenue, NOR 6316, Los Angeles, CA 90089-9176, USA
occupancy by GR, and reporter assays; carried out all computational and statistical analyses, prepared figures, and drafted the manuscript. RC participated in the conception and design of the study, confirmed the presence of GR mutations in all stable cell lines, carried out microarray and PCR gene expression measurements, and validated occupancy by GR at genomic regions used with the reporter assay. LCW carried nuclear magnetic resonance spectroscopy experiments, fitted electrophoretic mobility shift assay data, and participated in measurements of occupancy by GR and in reporter assays. MRS participated in the conception and design and coordination of the study. KRY participated in the conception and design and coordination of the study and helped to draft the manuscript. All authors read and approved the final manuscript.

Samantha Cooper was involved in the design of the study and optimizing chromatin immunoprecipitation sequencing protocols. Sheng-Hong Chen provided the luciferase reporter vectors pGL4.10-E4TATA and pGL4.70-MLE2. Clement Chu and Jessica Lund at the UCSF Center for Advanced Technology aided in the acquisition of sequencing data. Processing of RNA samples and gene expression microarrays were performed by the University of Southern California Epigenome Center. Joel Mefford suggested the statistical test used to identify nonselective, activated genes. Funding for this study was provided by grants to KRY (R01 CA020535) and to MRS (R01 DK043093) from the National Institute of Health and also by Cancer Center Support Grant P30CA014089 from the National Cancer Institute. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Cancer Institute or the National Institutes of Health.
Chapter 4

Perspective: Glucocorticoid receptor activity is modulated by a precise interplay between binding sequence and structural conformation, dynamics

In a series of studies (Meijsing et al. [5], Chapters 2 and 3), our group has shown that glucocorticoid receptor (GR) binds DNA in a diversity of sequence-specific conformations and with varied dynamics (association, dissociation, and inter-state mixing). Moreover, we have shown that degenerate full sites and consensus half sites differ not only in the conformational states of bound GR, but also in the utilization of a particular GR functional surface (i.e., the dimerization interface) and in the transcriptional regulatory activity these sequences induce via GR. Rather than a peculiarity of GR dimerization, the coupling of
distinct conformational states with GR functional surfaces, directed by binding to particular sequences, is likely a common theme in glucocorticoid-mediated gene regulation. Indeed, unpublished results (M Thomas-Chollier, LC Watson, SB Cooper, M Pufall, J Liu, K Borzym, E Einfeldt, M Vingron, SH Meijsing and KR Yamamoto) similarly define a distinct “sub-motif” corresponding to utilization of the lever arm, another GR functional surface. Collectively, these results begin to address a fundamental question regarding the complexity of gene regulation – namely, how does a single transcription factor produce diverse, yet reproducible gene-, cell- and tissue-specific outputs given only a limited number of inputs? In the case of human GR, only a single endogenous hormonal ligand (cortisol) is known. Additionally, GR is expressed in every known adult cell type, regulating a distinct set of genes in each. While the classical explanation that genes are regulated in a combinatorial manner by multiple transcription factors remains an important component of any working model, chromatin immunoprecipitation of GR [74] from distinct cell types shows that the genomic locations occupied by GR vary greatly among cell types. These results hint that varied GR occupancy may sufficient to confer differential (e.g., tissue-specific, cell-specific) regulation, independent of other factors.

The factor-intrinsic view of gene regulation that we have presented sharply contrasts with the between-factor combinatorial model of gene regulation. Nonetheless, it is important to note that regulation of each target gene may operate by a distinct mechanism. The regulation of any given gene may be modulated by GR alone or by multiple factors, or may even alternate between mechanisms depending on the set of conditions (cell types, for example) that are being considered for a given gene. Moreover, as we have seen, combinatorial control is equally important in the factor-intrinsic model – combinatorial utilization of GR functional surfaces [4] allows gene-specific regulation within a cell. GR functional surfaces may also play a more global role in cell type-specific regulation of genes;
one could easily imagine the global modulation of a particular surface by a variety of known (inhibitory or activating) post-translation modifications to GR. Such post-translational modifications could represent distinct states of the transcriptional regulatory network in the sense that the modifications are regulated by the expression of particular cell-type or condition-specific modifying enzymes.

Additionally, it is likely that combinatorial control acts at a factor-intrinsic level via the binding of particular factors to GR. Binding of such factors may be independent of GR (depending only on whether the factors are expressed in a given cell) or may be directed by the same structural changes that induce “utilization” of a particular surface (i.e., viz. allostery). By a similar allosteric mechanism, these factors may even inhibit or potentiate binding of GR or changes in target gene expression. Importantly, the aforementioned roles may apply together to one factor or across many factors. Indeed, one coregulatory transcription factor, Hypoxia Inducible Clone 5 (Hic-5), appears to block transcription at certain genes that are regulated by GR in its absence, as well as potentiating or modulating GR activity at other genes (unpublished results, R Chodankar, D-Y Wu, BJ Schiller, KR Yamamoto and MR Stallcup). Hic-5 binds to the tau2 activation domain in the hinge region of GR, suggesting the possibility that Hic-5 may inhibit GR binding to DNA via direct protein-protein interaction with GR. Collectively, these results highlight the remarkable complexity that can arise from only a single factor interacting with only a single surface of GR.

The complexity of gene regulation rests not only with a cell type-specific state, informed by the expression of particular transcription factors, but also in the molecular details of the interactions of each of these factors: via the DNA sequences they bind, the functional surfaces they utilize, and the factors with which they interact. While our recently increased understanding of the complexities of gene regulation may pessimistically be
viewed as a sign that we lack the wisdom to understand gene regulation, we are now aided in our studies by an unprecedented set of tools, including massively parallel DNA sequencing, biophysical approaches for directly characterizing macromolecular structure and conformation and bioinformatic approaches that allow enumeration on modeling of tens of thousands (and even millions) of components simultaneously. Further experiments will demand a more diverse set of skills than that of traditional biochemistry and molecular biology. In particular, a greater emphasis on quantitative and analytical methods will be required to understand the gene-specific and cell type-specific factors we have described.
Chapter 5

Stalled spliceosomes are a signal for RNAi-mediated genome defense

Abstract

Using the yeast Cryptococcus neoformans, we describe a mechanism by which transposons are initially targeted for RNAi-mediated genome defense. We show that intron-containing mRNA precursors template siRNA synthesis. We identify a Spliceosome-Coupled And Nuclear RNAi (SCANR) complex required for siRNA synthesis and demonstrate that it physically associates with the spliceosome. We find that RNAi target transcripts are distinguished by suboptimal introns and abnormally high occupancy on spliceosomes. Functional investigations demonstrate that the stalling of mRNA precursors on spliceosomes is required for siRNA accumulation. Lariat debranching enzyme is also necessary for siRNA production, suggesting a requirement for processing of stalled splicing intermediates. We propose that recognition of mRNA precursors by the SCANR complex is in kinetic competition with splicing, thereby promoting siRNA production from transposon transcripts stalled on spliceosomes. Disparity in the strength of expression signals encoded by transposons
versus host genes offers an avenue for the evolution of genome defense.

5.1 Introduction

RNAi-related RNA silencing pathways constitute a group of small RNA-based silencing mechanisms that antedate expansion of the eukaryotic lineage and function throughout this domain of life [107]. Enzymes required for RNA silencing are numerous and can differ between species, but universally include Argonaute or PIWI clade proteins, which bind small RNAs. Some RNA silencing pathways also utilize Dicer ribonucleases, which produce siRNA from double-stranded RNA (dsRNA) precursors, and RNA-dependent RNA polymerases, which produce dsRNA. Although RNAi-related systems perform disparate roles in different organisms – from histone modification to translational regulation – a deeply conserved and biologically critical function for these systems, observed from protists to man, is to defend genome integrity by silencing transposable elements [108, 109, 110]. Yet transposons occur in many families that bear little or no resemblance to each other [111], raising the question of how they are recognized as non-self DNA.

One RNAi-related system that suppresses transposon mobilization is the Piwi-interacting small RNA (piRNA) pathway, best understood in Drosophila. piRNAs derive from specific genomic clusters of transposon-related sequences and act with Argonaute proteins of the PIWI clade to silence homologous sequences throughout the genome [112, 111]. Such a mechanism constitutes an adaptive immunity to transposons, as it silences only transposon families that had previously been incorporated into a piRNA cluster [113]. These constraints raise the question of whether eukaryotes also demonstrate innate immunity to transposons, in which prior exposure to a transposon is not required for its recognition.

The processing of long dsRNA into siRNA can be viewed as an innate immune mecha-
anism for transposon defense, capable of recognizing even novel transposons by virtue of their tendency to generate dsRNA. For instance, transposons can produce dsRNA by mobilizing into an existing transcriptional unit, or by virtue of transposon-encoded inverted repeats and internal antisense promoters; such dsRNAs template the production of repressive endogenous siRNA in a manner that requires Dicer [114, 115, 116, 117, 118].

Mutations that block exogenous RNAi, which is triggered by long dsRNA, concomitantly increase endogenous transposon mobilization, providing additional evidence for the role of dsRNA processing in transposon recognition [119, 120]. Another class of RNAi-related system potentially involved in innate transposon immunity is thought to have evolved to recognize unusual DNA arrangements. The quelling pathway of *N. crassa* targets repetitive transgene arrays [121, 122], whereas meiotic silencing of unpaired DNA (MSUD) mechanisms silence transgenes that lack a partner during homolog pairing in meiosis I [123]. Unlike mechanisms of innate transposon immunity that rely on dsRNA recognition, quelling and MSUD can silence loci that do not naturally produce dsRNA, and both require RNA-dependent RNA polymerases [124, 125, 126]. Because transposons are particularly likely to form tandem arrays or occur asymmetrically on homologs during meiosis, quelling and MSUD may suppress transposon mobilization, but their underlying mechanisms remain poorly understood.

The human pathogenic yeast *Cryptococcus neoformans* offers a genetically tractable model system for approaching the mechanisms of small RNA-mediated transposon suppression. This organism displays an active RNAi pathway comprising one Argonaute clade member (Ago1), two redundant Dicer orthologs (Dcr1/2), and an RNA-dependent RNA polymerase ortholog (Rdp1). These factors play a key role in defending the *C. neoformans* genome: null mutations in their corresponding genes result in increased transposon expression, transposon mobilization, and transposon-induced drug resistance mutations [127, 128]. In this paper, we investigate the mechanism of siRNA biogenesis in *C. neoformans*. 
mans. Six observations indicate a key role for introns and the spliceosome in this process. First, small RNA analysis reveals that unspliced mRNA precursors are preferred substrates for siRNA production. Second, we describe a nuclear RNA-dependent RNA polymerase complex required for siRNA production, termed SCANR (Spliceosome-Coupled And Nuclear RNAi complex), and find that it physically associates with the spliceosome. Third, we observe that RNAi target transcripts encode suboptimal splicing signals and exhibit unusually high accumulation on spliceosomes. Fourth, we find that deletion of introns from a strong RNAi target blocks the accumulation of siRNA corresponding to this transcript. Fifth, we find that experimental stalling of a pre-mRNA’s splicing dramatically increases its siRNA production in a manner that requires entry into the splicing pathway. Sixth, we demonstrate that the lariat debranching enzyme is required for siRNA synthesis. These results indicate that stalled spliceosomes are a signal for RNAi and that splicing intermediates may be a favored substrate. We propose that a competition between the recognition of spliceosome-associated mRNA precursors by SCANR and the completion of their ongoing splicing plays an important role in specifying sequences from which repressive siRNA is produced.

5.2 Results

5.2.1 Endogenous siRNA of C. neoformans targets transposon mRNA precursors

To investigate the mechanism of small RNA biogenesis in C. neoformans genome defense, we used high-throughput sequencing to identify the siRNAs in haploid, vegetatively growing cells. Wild-type cells produced 21-23 nt siRNAs with a strong preference for a uridine residue at their 5’ termini, a characteristic feature of Argonaute-bound siRNAs in
other fungal systems (Figure 5.1A) [115]. These siRNAs were absent from cells lacking Ago1 or Rdpe1 (Figure 5.1B-C). siRNAs mapped most prominently to transposons: 22% of reads mapped to centromeres, which are composed primarily of transposons and transposon remnants [129], and 32% of reads mapped to intergenic regions that have sequence similarity to centromeres (Figure 5.1D). An additional 39% of siRNA reads mapped to mRNA-encoding genes, many of which had sequence similarity to centromeres. Genic siRNAs displayed a strand bias: 95% of reads had a polarity opposite that of the predicted transcript at their corresponding loci. These findings indicate that the previously-described broad targeting of transposon transcripts by siRNA that occurs during mating is also a feature of vegetative growth in *C. neoformans* [128].

Transcripts targeted by RNAi corresponded to loci on each of the 14 *C. neoformans* chromosomes (e.g., Figure 5.1E). These RNAi targets were not enriched for convergently transcribed genes, suggesting that dsRNA generated by bidirectional transcription is not a major driver of siRNA generation (Table 5.2). Strikingly, siRNA reads in genes mapped not only to exons but also to introns: over 15% of these siRNAs contained intronic sequence, whereas introns constitute 18% of all genic sequences. Moreover, the 50 genes targeted by the greatest number of siRNA reads exhibited not only intronic siRNAs (in 45 cases) but also siRNAs that spanned intron-exon junctions (in 42 cases) (Table 5.3). siRNA reads spanning exon-exon junctions were also observed within this gene set, at a frequency only 2.45-fold that of intron-exon junction reads, despite the fact that the vast majority of cellular RNAs are expected to be fully spliced (Table 5.3). The sequence features of siRNAs, together with their dependence on Rdpe1, are consistent with a model in which Rdpe1 acts preferentially upon intron-containing mRNA precursors to generate dsRNA, which is processed into siRNA to effect silencing of transposons and other RNAi targets. This model is further developed below.
Figure 5.1: Endogenous siRNA of *C. neoformans*. (A-C) Read counts for siRNAs based on length and 5’ nucleotide identity. (D) Genomic mapping of siRNA sequences. (E) Density plot of siRNAs mapping to an RNAi target locus, *CNAG_7721*, which comprises two exons and one intron. See also Tables 5.2 and 5.3.
5.2.2 Identification and characterization of proteins associated with RNA-dependent RNA polymerase

To gain further insights into how the RNAi machinery might be targeted to mRNA precursors in the nucleus, we used tandem affinity purification and mass spectrometry to identify proteins associated with Rdpl, the only component of the C. neoformans RNAi machinery that has been reported to be nuclear [128]. Purification of Rdpl-CBP-2xFLAG, expressed from its endogenous locus, yielded Rdpl itself as well as four additional proteins: Ago1 and three proteins we named Qip1, Gwc1, and Srr1 (Figure 5.2A). Qip1 is an ortholog of QIP, an N. crassa exonuclease that binds Argonaute and degrades the passenger strand of siRNA duplexes [130]. Gwc1 (GW-containing) contains 5 GW/WG dipeptides – a motif commonly found in Argonaute-binding proteins [131] – but no other obvious domain or homolog. Srr1 (Serine/Arginine-rich) has no clear ortholog in other organisms, but, intriguingly, its domain structure resembles those of several mammalian splicing factors. In particular, Srr1 contains 30 RS/SR dipeptides and three RNA recognition motifs.

To assess the potential function of each protein that co-purified with Rdpl, we attempted to create deletion mutations in their corresponding genes. We were successful for Ago1,
Qip1, and Gwc1, but unsuccessful for Srr1, suggesting it may be essential for viability. Strains lacking Ago1, Rdpl, Gwc1, or Qip1 exhibited increased levels of three transcripts highly targeted by siRNA in our sequencing experiments: CNAG_6757, an unannotated gene with homology to a transposon; CNAG_6844, a RecQ helicase; and CNAG_6705, an unannotated ORF (Figure 5.2B). The same strains also displayed a loss of siRNA corresponding to these transcripts, indicating a functional requirement for Ago1, Rdpl, Gwc1, and Qip1 in RNAi (Figure 5.2C).

The isolation of Ago1 as an Rdpl-associated protein was surprising since it had been reported to localize exclusively to cytoplasmic P bodies [128]. However, we found that although an mCherry-Ago1 fusion protein was cytoplasmic in most cells, a substantial fraction of cells (26%) displayed localization in both the nucleus and cytoplasm (Figure 5.2D). The cytoplasmic signal of mCherry-Ago1 localized both diffusely and in foci corresponding to P bodies (Figure 5.8A). GFP-Gwc1 and GFP-Qip1 fusion proteins displayed a nuclear and cytoplasmic pattern in all cells, consistent with a fraction of these proteins associating with Rdpl in the nucleus (Figure 5.2D).

To characterize the physical relationships among Rdpl-associated proteins, we performed tandem affinity purifications of Gwc1, Qip1, and Ago1. Protein identification by mass spectrometry revealed a dense network of interactions among these proteins, consistent with the existence of a protein complex (Figure 5.2E-F). Gwc1 and Qip1 purifications yielded only proteins that were also identified in an Rdpl purification, whereas an Ago1 purification yielded all of these proteins as well as six additional proteins, which we named Skp1, Gwo1, Aga1, Aga2, Bre1, and Aga3. These results can be most simply explained by proposing that Rdpl, Gwc1, Ago1, and Qip1 form a protein complex required for siRNA production, whereas Ago1 participates additionally in at least one other protein complex. For reasons described below, we refer to the Rdpl complex as SCANR (Spliceosomal-
Coupled And Nuclear RNAi complex) (Figure 5.2G). We note that the dual nuclear and cytoplasmic localization of Ago1, Gwc1, and Qip1 suggests the existence of a cytoplasmic subcomplex that lacks Rdp1. Yeast two-hybrid experiments using each SCANR component as bait and prey revealed interactions involving every subunit, further supporting the associations identified by tandem affinity purification (Figure 5.2H and Table 5.4).

We tested whether any of the proteins that co-purified with Ago1 but not with any other SCANR protein (Skp1, Gwo1, Aga1, Aga2, Bre1, and Aga3) is required for silencing of mRNAs targeted by RNAi. We were able to delete each corresponding gene except SKP1, which may be essential for viability, as it is in S. cerevisiae [132]. Of the five mutants, only the gwo1Δ strain exhibited derepression of RNAi target transcript levels, albeit to a lesser extent than that of strains lacking SCANR components (Figures 5.3A-B). Loss of Gwo1 did not affect siRNA accumulation, however, indicating that Gwo1 is not required for siRNA biogenesis (Figure 5.3C). Tandem affinity purification of Gwo1, followed by mass spectrometry, identified only Gwo1 itself and Ago1, suggesting that these two factors form a protein complex distinct from SCANR (Figures 5.3D-E). Indeed, yeast two-hybrid analysis detected an interaction between Gwo1 and Ago1, but not between Gwo1 and the SCANR components Rdp1, Qip1, or Gwc1 (Figure 5.3F and Table 5.4). Furthermore, analytical tandem affinity purification of Ago1 yielded Gwo1, whereas purification of the SCANR component Rdp1 did not yield Gwo1, as assessed by immunoblotting (Figure 5.3G). Finally, localization experiments indicated that a GFP-Gwo1 fusion protein resided in cytoplasmic foci that co-localized with Ago1 foci and with the P body marker Dcp1 (Figures 5.3H and 5.8B). We therefore refer to the Ago1-Gwo1 complex as PRSC (P body-associated RNA Silencing Complex) and conclude that it is distinct from the nuclear SCANR complex. Investigation of the function of PRSC is ongoing.
Figure 5.3: PRSC and SCANR subunits are physically and functionally distinct. (A-B) Transcript levels of three RNAi target genes, assessed by RT-qPCR and normalized to 18S rRNA levels. Error bars: SEM. *p < 0.05 by Student’s t-test. (C) sRNA Northern blot. Loading control: U18 snoRNA. (D) Proteins associated with Gwo1-CBP-2xFLAG. Assayed as in Figure 5.2A. (E) Predicted protein domains of PRSC subunits. (F) Protein-protein interactions among PRSC components, as assessed by yeast two-hybrid assay. No interactions were detected between Gwo1 and any exclusive member of SCANR. (G) Co-immunoprecipitation of Gwo1 with Ago1, but not with Rdp1. Strains expressing Gwo1-13xMyc and CBP-2xFLAG-Ago1 or Rdp1-CBP-2xFLAG were subjected to FLAG-CBP tandem purification. Input and purified material were analyzed by immunoblot using anti-FLAG, anti-Myc, or anti-PSTAIRE, which detects the negative control protein p31. (H) Co-localization of PRSC component Gwo1 with the P-body marker Dcp1. Unfixed cells were examined after incubation in YNB media. See also Figure 5.8 and Table 5.4.
5.2.3 SCANR associates with the spliceosome

Given that unspliced mRNA precursors appear to be preferred substrates for dsRNA synthesis, we were intrigued by the fact that Srr1, which was identified by mass spectrometry in purifications of two SCANR components, resembles mammalian splicing regulators (Figure 5.4A). To confirm a physical interaction between SCANR and Srr1, we performed analytical tandem affinity purifications of SCANR components and used immunoblotting to detect associated proteins. Purifications of each SCANR subunit yielded Srr1 (Figures 5.4B-D). In contrast, purification of the RNAi factor Dcr1, which has been reported to localize to P bodies [128], did not yield Srr1, nor was an unrelated control protein, p31, detected in any purification. Consistent with its SCANR association, an Srr1-mCherry fusion protein displayed a nuclear localization (Figure 5.4E).

Given its domain structure, we tested whether Srr1 physically associates with spliceosomal snRNAs by immunoprecipitating Srr1 and detecting bound RNAs by RT-qPCR. All four annotated *C. neoformans* spliceosomal snRNAs (U2, U4, U5, U6), but not a control snoRNA (U18), associated with Srr1 (Figure 5.4F). We next identified proteins associated with Srr1 using tandem affinity purification and mass spectrometry. Remarkably, of the 23 proteins that co-purified with Srr1, 17 were orthologs of known spliceosomal proteins.

Figure 5.4 (following page): SCANR physically associates with Srr1 and the spliceosome. (A) Predicted domains of Srr1. (B-D) Co-immunoprecipitation of Srr1 with SCANR components Ago1 (B), Gwc1 (C), Qip1 (C), and Rdp1 (D), but not with Dcr1 (B). Assayed as in Figure 5.3G. (E) Localization of Srr1. An Srr1-mCherry fusion protein was detected in fixed cells. Nuclei were stained using Hoechst dye. (F) Interaction of Srr1 and spliceosomal snRNAs. Levels of individual RNAs co-immunoprecipitated with Srr1 were assessed by RT-qPCR and normalized to their abundance in whole cell extract; transcript level is relative to that obtained in purifications from wild-type (untagged) lysates. Error bars: SD. (G) Proteins associated with CBP-2xFLAG-Srr1. Genes were named based on *S. cerevisiae* orthologs, or, in the absence of one, based on metazoan orthologs. Known spliceosome components are colored in red. (H) Co-immunoprecipitation of the spliceosome component Syf1 with Ago1, but not with Dcr1. Assayed as in Figure 5.3G. See also Figure 5.9.
These included components of spliceosomal snRNPs – such as Smb1, Prp4, and Msl1 – and other spliceosome-associated complexes – such as Prp19, Syf1, Cef1, Cwc2, and Isy1 of the NineTeen complex (NTC). Detection of SCANR components among Srr1-associated proteins was below our 10% coverage threshold, consistent with the lower sensitivity of this assay relative to immunoblotting. These data establish Srr1 as a nuclear protein that interacts with spliceosomal complexes.

The observed interaction between Srr1 and RNAi factors of SCANR, together with the associations between Srr1 and the spliceosome, suggested that SCANR might physically associate with the spliceosome, but too weakly to be detected by mass spectrometry methods. To test this hypothesis, we determined whether Ago1 and Rdpl physically interact with the NTC component Syf1, which is part of the spliceosome at multiple stages of its assembly and during the catalytic steps of splicing [133]. Using immunoblotting, we detected Syf1 in tandem affinity purifications of Ago1 or Rdpl, but not upon purification of Dcr1, which is not a SCANR component (Figures 5.4H and 5.9). These findings indicate that SCANR physically associates with at least a subset of spliceosomal complexes marked by Syf1.

5.2.4 RNAi target transcripts display high spliceosome occupancy

Our finding that mRNA precursors are a substrate not only for splicing, but also for siRNA production (Figure 5.1), led us to hypothesize that the spliceosome and RNAi may compete for their substrates in some way (Figure 5.5A). In this scenario, the slow splicing of a transcript would, by virtue of increasing its time spent as an mRNA precursor, increase its availability as a substrate for the RNAi pathway, thereby biasing RNAi targeting toward poorly spliced genes.
An important prediction of such a model is that genes targeted by RNAi should encode inefficiently spliced transcripts. As an initial test of this idea, we examined intron features likely to affect splicing efficiency. Unlike *S. cerevisiae*, in which most genes have no introns and genes with introns typically have only one, *C. neoformans* genes average 5 introns per gene, and it is thought that nearly all, if not all, genes harbor introns [129]. Furthermore, whereas *S. cerevisiae* intronic splicing signals conform closely to consensus, *C. neoformans* splice sites are considerably more degenerate, in this respect resembling those of metazoans [134, 129]. Although *C. neoformans* intron splice sites possess generally low information content, intron size is highly constrained: intron lengths are tightly distributed around a mode of 52 nt, and evolutionary studies of the *Cryptococcus* species complex suggest that an optimal intron size is under selection [135].

We examined the aforementioned splicing features of RNAi target and non-target transcripts to determine whether the RNAi pathway acts preferentially on transcripts with suboptimal introns. The vast majority of both gene classes contained introns: RNAi

---

**Figure 5.5 (following page):** Transcripts targeted by RNAi display suboptimal splicing features and are stalled on spliceosomes *in vivo.* (A) Model of competing pathways for mRNA precursor maturation. (B) Analysis of longest intron lengths of siRNA targets (>150 siRNA reads) versus all genes. Cumulative distribution functions of longest intron length for these two gene classes are plotted. (C) Analysis of weakest 5’ splice sites of siRNA targets (>150 siRNA reads) versus all genes. Cumulative distribution functions of weakest 5’ splice site strengths for these two gene classes are plotted. Inset: sequence logo generated from all *C. neoformans* 5’ splice sites. (D) Effect of intron deletion on the production of siRNA from the *CNAG_6705* transcript. Above: RNA was isolated from cells that expressed, from a *GAL7* promoter, either wild-type *CNAG_6705* or a mutated form of the gene in which one or both introns were deleted. Below: the transcript level of each *CNAG_6705* mutant was measured by RT-qPCR and normalized to levels of actin RNA. Error bars: SD. (E) Association of siRNA target transcripts and non-siRNA target transcripts with the spliceosome. Spliceosomes tagged with Prp19-CBP-2xFLAG were immunoprecipitated and co-purified RNAs were detected by RT-qPCR. Levels of individual RNAs were normalized to their abundance in whole cell extract; IP/WCE values are relative to those of purifications from wild-type (untagged) lysates. Error bars: SD. Bold: transposon-related transcripts. See also Figure 5.10.
A mRNA precursor → splicing → mRNA → RNAi → siRNA

B Median length of longest intron
- All genes: 73 nt, $p < 1 \times 10^{-8}$
- siRNA targets: 127.5 nt

C Median strength of weakest 5′ splice site
- All genes: 5.65 bits, $p = 1.3 \times 10^{-2}$
- siRNA targets: 4.72 bits

D RNAi
- Intron 1
- Intron 2
- U18 Northern
- RT-qPCR

E Spliceosome occupancy
- IP / WCE transcript abundance (relative to untagged)
- siRNA targets
- non-siRNA targets
target transcripts encode 3.9 introns per gene on average, as compared to 5.1 for all C. neoformans genes. Because a single poorly spliced intron in a multi-intron transcript would in principle be sufficient to stall maturation of the entire transcript, we classified each gene based on the properties of its weakest intron with respect to a given feature. We observed that the longest annotated introns of RNAi target genes were, on average, considerably longer than those of all genes (Figure 5.5B; $p < 1 \times 10^{-8}$ by two-tailed K-S test). In contrast, the strengths of the weakest 5’ splice sites of RNAi target genes were, on average, only modestly weaker than those of all C. neoformans genes (Figure 5.5C; $p = 0.013$ by two-tailed K-S test). Because the 5’ splice site is critical for initiation of spliceosome assembly, whereas experimentally increasing intron length can, depending on the site, block later steps of the splicing pathway [136, 137], these findings raise the possibility that siRNA target transcripts enter the splicing pathway, but become stalled at a later stage. Stalled splicing may thereby provide an opportunity for the spliceosome-associated SCANR complex to target mRNA precursors for RNAi.

As an initial test of this hypothesis, we examined whether causing an RNAi target transcript to bypass splicing would reduce its ability to template the production of siRNA. To do so, we generated mutants of an RNAi target transcript, CNAG_6705, in which its two introns were deleted, singly or together. Each of these mutations dramatically reduced the accumulation of siRNA corresponding to CNAG_6705 (Figure 5.5D). Intron deletion also decreased steady state levels of the CNAG_6705 transcript, but this effect appeared insufficient to explain the loss of CNAG_6705 siRNA caused by the mutations (Figure 5.5D). These results are consistent with a model – tested further below – in which spliceosome engagement is important for siRNA production.

To examine more directly the interactions between endogenous RNAi target transcripts and the splicing machinery, we sought to measure the in vivo occupancy of mRNA precursors
on the spliceosome. We immunoprecipitated NTC-containing spliceosomal complexes using an epitope-tagged Prp19 and examined levels of associated transcripts by RT-qPCR, normalizing the data for total transcript levels. Strikingly, the twelve transcripts most highly targeted by siRNA exhibited dramatically greater spliceosome occupancy than did six non-RNAi target genes that spanned a broad range of expression level, intron number, and gene size (Figure 5.5E). These twelve RNAi targets include seven transposon-related genes (indicated in bold, Figure 5.5E), whereas the non-RNAi targets include genes encoding actin (CNAG_0483) and GAPDH (CNAG_6699). We found that the accumulation of RNAi target transcripts on spliceosomes was maintained in the context of an rdp1Δ strain, demonstrating that this property is not caused by siRNAs themselves (Figure 5.10). Thus, transcripts targeted by RNAi exhibit intronic sequence features predictive of poor splicing and accumulate abnormally on spliceosomes in vivo.

5.2.5 Mutations that stall splicing trigger RNAi

Our finding that RNAi target transcripts accumulate on spliceosomes led us to hypothesize that these transcripts proceed inefficiently through the splicing pathway, during which some fraction is redirected to the RNAi pathway and converted into siRNA. This hypothesis predicts that the entry of a transcript into the RNAi pathway would be improved by intronic mutations that stall its splicing at an intermediate stage. To test this prediction, we introduced intronic mutations into a single-copy, endogenous gene that is very weakly targeted by RNAi: CNAG_7888 (Figure 5.11A). CNAG_7888 contains two predicted introns whose locations we verified by cDNA sequencing. To facilitate detection of siRNA generated from the CNAG_7888 locus by Northern hybridization, its promoter was replaced with the strong GAL7 promoter. We assessed two classes of intronic mutations: 5’ splice site mutations, which we predicted to block splicing prior to spliceosome assembly;
and 3’ splice site mutations, which we predicted to allow spliceosome assembly but block the second catalytic step of splicing.

Cells expressing wild-type CNAG_7888 exhibited a very low level of siRNA, comparable to the background observed in cells in which the CNAG_7888 locus was deleted (Figure 5.6A). No change in siRNA levels was detected in the context of CNAG_7888 alleles containing 5’ splice site mutations of either intron. In contrast, a 3’ splice site mutation – specifically, that of intron 2 – resulted in dramatically increased siRNA production from CNAG_7888. This siRNA production required Rdpl, confirming that it represented entry of CNAG_7888 into the RNAi pathway (Figure 5.11B). The inability of other CNAG_7888 mutant alleles to promote siRNA generation could not be explained by defects in their expression, as all mutants showed similar transcript levels (Figure 5.11C). Additionally, it was not the case that the sequence of intron 2 is uniquely capable of promoting RNAi: a CNAG_7888 construct in which the sequences of introns 1 and 2 had been exchanged similarly triggered siRNA synthesis only when the 3’ splice site of the downstream intron was mutated (Figure 5.11D). Evidently, the position of the mutated intron in CNAG_7888 (or perhaps its contiguous exonic sequence) influences its ability to stimulate siRNA production. The nature of the 3’ splice site mutation is unimportant: multiple, distinct 3’ splice site mutations in the second intron stalled CNAG_7888 splicing at the second catalytic step, as assessed by primer extension detection of lariat intermediate, and each strongly triggered siRNA production (Figure 5.6B).

Based on our finding that RNAi target transcripts are enriched for introns predicted to be longer than optimal (Figure 5.5B), we hypothesized that introns containing an increased distance from the branchpoint adenosine to the 3’ splice site would promote RNAi, because such introns stall splicing in other systems [136, 137]. We generated alleles of CNAG_7888 that contained 75 or 100 nt insertions of adenosine-free sequence between the mapped
Figure 5.6: Stalled spliceosomes promote RNAi in a manner that depends on entry into the splicing pathway.  (A) Effect of perturbed splicing on the production of siRNA from the CNAG_7888 transcript. RNA was isolated from cells that expressed, from a GAL7 promoter, either wild-type CNAG_7888 or a mutated form of the gene in which a single splice site was mutated. (B) Effect of 3′ splice site mutations on siRNA production from and splicing of the CNAG_7888 transcript. RNA was isolated from cells that expressed, from a GAL7 promoter, either wild-type CNAG_7888 or a mutated form of the gene in which the 3′ splice site of intron 2 was mutated. siRNA production was assessed by riboprobe hybridization and splicing was assessed by primer extension using a labeled primer complementary to CNAG_7888 exon 3. Loading control: U6 snRNA primer extension product. (C) Spliceosome-dependence of siRNA accumulation triggered by 3′ splice site mutations. RNA was isolated from cells that expressed, from a GAL7 promoter, either wild-type CNAG_7888 or a mutated form of the gene in which the 5′ and 3′ splice sites of intron 2 were mutated, singly or together, and assayed as in (B). (D) Association of the spliceosome with CNAG_7888 transcripts containing intron 2 splice site mutations, assayed as in Figure 5.5E. See also Figure 5.11.
branchpoint and 3’ splice site of intron 2, then examined their splicing efficiency and siRNA production. As predicted, both of these insertion mutations reduced intron 2 splicing efficiency, although not as severely as did mutation of the 3’ splice site, as assessed by level of lariat intermediate (Figure 5.11E). Accordingly, both insertion mutations triggered siRNA production from CNAG_7888, but to a lesser extent than that triggered by a 3’ splice site mutation. This inverse correspondence between splicing efficiency and siRNA accumulation across a range of splicing efficiencies is consistent with a kinetic competition between splicing and siRNA production and indicates that multiple types of suboptimal introns can promote RNAi.

5.2.6 Entry into the splicing pathway is required for RNAi

Our finding that the suboptimal introns that promote RNAi tend to cause stalling of the spliceosome, together with our observation that RNAi target transcripts accumulate on spliceosomes, led us to hypothesize that spliceosome assembly at an intron is required for that intron to promote RNAi. This hypothesis predicts that a 5’ splice site mutation, which disrupts the initial recognition of an intron by the spliceosome, should suppress the RNAi-promoting effects of 3’ splice site mutations in CNAG_7888. Strikingly, a 5’ splice site mutation of intron 2, which caused no effect on RNAi by itself, fully suppressed the siRNA production triggered by a 3’ splice site mutation (Figure 5.6C). As expected, the 5’ splice site mutation also suppressed the accumulation of intron 2 lariat intermediate observed in a 3’ splice site mutant.

To confirm that 3’ splice site mutations stall spliceosomes and that this effect is eliminated by a 5’ splice site mutation, we employed the spliceosome occupancy assay described above. As expected based on its exceedingly weak production of siRNA, wild-type CNAG_7888 exhibited low spliceosome occupancy, similar to that of non-RNAi target
transcripts (Figure 5.6D). A 5’ splice site mutation of intron 2 caused no change in spliceosome occupancy, whereas a 3’ splice site mutation dramatically increased spliceosome occupancy, consistent with our finding that this mutation blocks the second catalytic step of splicing. In accord with the well-established role of the 5’ splice site sequence in mediating entry into the splicing pathway, a 5’+3’ double splice site mutant displayed low spliceosome occupancy. The correlation between spliceosome occupancy and siRNA levels in this series of CNAG_7888 mutant transcripts mirrors our observation that RNAi target transcripts generally display greater spliceosome occupancy than do non-RNAi target transcripts, supporting the view that stalled spliceosomes are a signal for RNAi.

5.2.7 Splicing intermediates may be a preferred substrate for siRNA biogenesis

Our finding that inefficient progression of CNAG_7888 through the splicing pathway increases its targeting by RNAi raised the possibility that splicing intermediates of this transcript, such as the intron 2 lariat intermediate, represent preferred substrates for siRNA biogenesis. We therefore tested whether siRNA production from CNAG_7888 requires the lariat debranching enzyme (Dbr1), which debranches discarded lariat intermediates and excised lariats in order to allow processing of these RNA species [138, 139]. We found that Dbr1 was absolutely required for the siRNA production stimulated by a 3’ splice site mutation of CNAG_7888 intron 2 (Figure 5.7A). Strikingly, we also observed that debranchase was essential for accumulation of siRNA corresponding to the strong RNAi target transcripts CNAG_6757, CNAG_6844, and CNAG_6705 (Figure 5.7B). These results suggest that debranching of splicing intermediates may enable their use as preferred templates for siRNA production. Such a model predicts that the first exon of either a single- or multi-intron pre-mRNA would be an underrepresented substrate
for siRNA biogenesis, because this region cannot be encoded by any lariat intermediate. Consistent with this prediction, we found that, among the top 50 RNAi target transcripts, the siRNA read density in annotated first exons – relative to the overall exonic density in their corresponding genes – was significantly less than the same metric calculated for annotated last exons (Figure 5.7C). Together, these observations suggest that debranching of splicing intermediates stalled on spliceosomes may be a generally important step in siRNA production.

5.3 Discussion

RNAi-related systems that target transposons are important guardians of genome integrity, but our understanding of how self and non-self DNA are distinguished is incomplete. Whereas piRNA systems offer an adaptive mechanism for transposon recognition, the existence of innate mechanisms can also be inferred, particularly for cases in which siRNA is produced against transposons that form dsRNA. Our studies of the human pathogenic

---

Figure 5.7 (following page): Lariat debranchase is required for siRNA production. (A) siRNA accumulation triggered by a 3’ splice site mutation of CNAG_7888 requires the lariat debranching enzyme. RNA was isolated from wild-type or dbr1Δ cells that expressed, under the control of a GAL7 promoter, wild-type CNAG_7888 or a mutated form of the gene. (B) sRNA Northern blot. Loading control: U6 snRNA. (C) Relative siRNA read density in the first or last exon of RNAi target genes. For each of the top 50 RNAi target genes, the following ratio was calculated: siRNA read density in the first (or last) exon over siRNA read density across all exons of the corresponding ORF. These values were log10-transformed and are presented as boxplots; blue line indicates median. Values for first versus last exons were compared by the two-tailed Mann-Whitney U test. (D) Kinetic competition model. The utilization of particular mRNA precursors by SCANR is influenced by their splicing efficiency due to a kinetic competition between splicing and dsRNA synthesis: transcripts that are inefficiently spliced, such as foreign genetic elements, exhibit increased spliceosome association, thereby facilitating their conversion to dsRNA, and, ultimately, siRNA. In this hypothetical example, a lariat intermediate is produced by stalled splicing of a transcript’s first intron; downstream introns remain incompletely spliced. This intermediate is acted upon by SCANR and Dbr1 to produce a dsRNA substrate for Dcr1/2. See text for details.
A

Intron 1 | CNAG_7888 | Intron 2
---|---|---
GU | AG | WT
GU | AG | 3' ss CC
Δ | WT | 3' ss CC
WT | CC | 3' ss CC

sRNA Northern

B

WT | ago1 Δ | dbr1 Δ
CNAG 6757 | CNAG 6844 | CNAG 6705

sRNA Northern

C

First exon  All exons  Last exon  All exons

Log10(sRNA read density ratio)

p<1x10^-4

D

Exon intron

pre-mRNA

Spliceosome assembly

Transposon mRNA precursor

SCANR, Dbr1

Nucleus

Cytoplasm

DsRNA

Dcr1/2

siRNA

mRNA
yeast *Cryptococcus neoformans* provide multiple lines of evidence that the process of premRNA splicing by the spliceosome plays an important role in defining targets for the RNAi machinery.

Specifically, we find that transcripts targeted by RNAi are stalled in spliceosomes and that stalling promotes siRNA synthesis through a spliceosome-associated RNAi complex, SCANR. Our results are compatible with a competition between the SCANR-mediated recognition of unspliced mRNA precursors and the completion of their splicing, in which inefficiently spliced transcripts are channeled toward dsRNA synthesis. Although previous studies in *S. pombe*, *C. elegans*, and *A. thaliana* have demonstrated that mutation or knockdown of RNA processing factors, including essential splicing factors, can affect siRNA levels, in no case has it been clarified whether these effects are direct, in part because the underlying mechanisms have not been elucidated [140, 141, 142, 143, 144]. Below, we summarize evidence that intron-containing mRNA precursors are preferred substrates for siRNA synthesis. We discuss how this finding, together with the observation that transposon transcripts display unusually high spliceosome occupancy, suggests a kinetic competition model for RNAi targeting, and we describe potential advantages of such a mechanism. Finally, we speculate on the evolutionary forces that may have led to the high spliceosome occupancy observed for transposon RNAs.

### 5.3.1 Intron-containing mRNA precursors are a template for siRNA in *C. neoformans*

Our siRNA sequencing data demonstrate that siRNAs map to the introns and intron-exon junctions of most RNAi target genes, suggesting that either DNA or intron-containing mRNA precursors are a template for siRNA synthesis. Several lines of evidence point to mRNA precursors as relevant templates. First, our studies of SCANR show that this RNA-
dependent RNA polymerase complex is essential for siRNA production and associates with
the spliceosome, which itself assembles on pre-mRNA. More importantly, we show that
deletion of introns from a strong RNAi target gene reduces the level of its corresponding
siRNA, whereas introduction of a 3′ splice site mutation into a weak RNAi target gene,
which stalls its splicing, increases the level of its corresponding siRNA. Crucially, the latter
effect is blocked by a 5′ splice site mutation that prevents spliceosome association with the
transcript. Finally, we find that the lariat debranching enzyme, which acts on branched
RNA species generated by the action of the spliceosome (both lariat intermediate and
excised lariat), is required for siRNA accumulation. Taken together, these data argue
that intron-containing mRNA precursors are templates for dsRNA synthesis and siRNA
production.

5.3.2 Spliceosomal stalling of transposon transcripts suggests a kinetic competition

A second key finding of our studies is that RNAi target transcripts display much higher
occupancy on spliceosomal complexes than do transcripts that do not enter the RNAi
pathway. Importantly, RNAi itself is not responsible for this difference, as cells lacking
siRNA display the same behavior. These findings suggest a kinetic competition model
for the specification of transposon transcripts as substrates for siRNA synthesis (Figure
5.7D). In this model, mRNA precursors stalled on the spliceosome remain available for
recognition by SCANR, whereas transcripts that complete splicing undergo spliceosome
disassembly, intron degradation, and export out of the nucleus. The nuclear localization
of SCANR and its physical association with the spliceosome would facilitate the proposed
competition. Such a spliceosome-based mechanism is advantageous in that it does not
respond only to particular sequences, but rather to splicing efficiency itself, and therefore
could in principle recognize multiple types of suboptimal introns, including those of foreign genetic elements not previously encountered.

Although our experiments addressing transcript occupancy on Prp19-containing spliceosomal complexes do not speak to what specific kinetic steps are delayed during the splicing of transposon transcripts, other results suggest that spliceosome stalling events subsequent to the first catalytic step may be particularly capable of engaging SCANR. In fact, we find that a 3’ splice site mutation of CNAG_7888, which stalls splicing at the second catalytic step, promotes siRNA production from this locus. Furthermore, this siRNA production requires the lariat debranching enzyme, as does the production of siRNA corresponding to several other endogenous RNAi targets. Because loss of Dbr1 in other systems causes an accumulation of lariat RNAs but does not perturb splicing or spliceosome disassembly [139, 145], this finding suggests that lariat RNAs generated by the first step of splicing, once debranched, may be preferred substrates for Rdp1 and/or Dicer in C. neoformans.

5.3.3 The inefficient splicing of pre-mRNAs that contain transposons

The genome of C. neoformans is rich in introns: more than 97% of annotated genes contain at least one intron. In other organisms with intron-rich genomes, such as mammals, splicing enhances gene expression by promoting 3’ end formation, nuclear export of mRNA, and mRNA translation [146, 147]. Intron-dependent gene expression is also a feature of basidiomycetous yeast, raising the possibility that successful transposons in C. neoformans require introns [148, 149]. In fact, all of the 12 genes most targeted by siRNA in C. neoformans, the majority of which are related to transposons, contain introns. Yet all of these transcripts exhibit abnormally high spliceosome occupancy, suggesting that their introns may be poorly spliced. Furthermore, RNAi target transcripts generally exhibit
sequence features predictive of inefficient splicing, including relatively low 5’ splice site strength and increased intron length. But why would these foreign genetic elements tend to be poorly spliced as compared to endogenous genes? We suggest two non-mutually exclusive possibilities.

First, transposons that have entered the *C. neoformans* genome recently by horizontal transfer have had limited time to adapt to its specific splicing preferences. Efficient splicing in *C. neoformans* likely requires appropriate intron size, consensus-matched splice site sequences, and proper exonic splicing enhancer sequences [150]; these particular splicing preferences differ among organisms, creating a barrier to the efficient expression of horizontally transferred genes. Thus, the suboptimal 5’ splice sites and intron sizes of *C. neoformans* RNAi targets could be due in part to their limited co-evolution with endogenous genes.

A second possibility is based on an extensive literature describing cryptic introns in transposons that appear to minimize the impact of transposon insertion into host genes [151]. Transposon disruptions of essential host genes can result in the host cell’s death, which is deleterious for both the transposon and the host. These same transposon insertions, however, can be viable if transposon sequences are spliced out of the essential gene’s transcript. Therefore, splicing allows transposons to circumvent some negative fitness consequences associated with their propagation [152]. Consistent with this idea, multiple transposons – including *Ds* in maize, the 412 retrotransposon and *P* elements in *Drosophila*, and *Tc1* in *C. elegans* – contain splice sites near their termini such that the whole transposon can be spliced out of a larger transcription unit, thereby limiting disruption of endogenous loci [153, 154, 151]. Many other transposons, including Harbinger family DNA transposons and LINE elements, also contain introns, although these introns do not span the entire transposon [155, 156]. If a transposon, after its insertion into a
host gene, is spliced in such a way that mitigates its negative effects on host gene expression but also removes sequences important for mobilization, then weak splicing would be favored as a compromise between transposon expression and host organism health [157]. Such inefficiently spliced transcripts may become stalled in the spliceosome and recognized by SCANR, thereby contributing to the high spliceosome occupancy of transposon transcripts and to the production of transposon-specific siRNA for genome defense. Whether these principles apply to other organisms remains to be elucidated, but we are intrigued by the recent report of a class of C. elegans endo-siRNA that appears to be derived from intron-containing mRNA precursors [158].

5.4 Methods

5.4.1 Yeast strains

Yeast strains used in this study are listed in Table 5.5. All C. neoformans strains were derived from strain H99 using standard procedures [159].

5.4.2 Tandem affinity protein purification

C. neoformans cultures grown in YPAD media were harvested, snap frozen, then lysed using a coffee grinder and mortar and pestle. Proteins tagged with CBP-2xFLAG were purified using anti-FLAG M2 resin (Sigma) and calmodulin resin (Stratagene) according to manufacturers’ instructions and using buffers described in Supplemental Experimental Procedures.
5.4.3 RNA immunoprecipitation

Epitope-tagged proteins were purified using anti-FLAG resin as described above, and associated RNA was isolated by phenol-chloroform extraction, as described in Supplemental Experimental Procedures. Table 5.6 lists primers used for RT-qPCR.

5.4.4 RNA isolation and sRNA Northern blot

Total RNA was isolated using TRIzol (Invitrogen), whereas small RNA was isolated using a modified mirVana (Ambion) protocol, as described in Supplemental Experimental Procedures. Small RNA Northern blots were performed as described previously [160].

5.4.5 Accession Numbers

Small RNA sequencing data were deposited in the Gene Expression Omnibus (GSE43363).

5.4.6 Yeast media and techniques

Strains were grown in YPAD (1% yeast extract, 2% Bacto-peptone, 2% glucose, 0.015% L-tryptophan, 0.004% adenine), YPAG (1% yeast extract, 2% Bacto-peptone, 2% galactose, 0.015% L-tryptophan, 0.004% adenine), or YNB (1.5 g/L yeast nitrogen base, 5 g/L ammonium sulfate, 2% glucose) media at 30°C. Because *C. neoformans* can respond to light, strains were grown and harvested in darkness [161].
5.4.7 Gene nomenclature

*C. neoformans* genes were identified using Broad Institute (Cambridge, MA) annotations of the *var. grubii* H99 sequence \(^1\), in which genes are named “CNAG_#”.

5.4.8 Small RNA library preparation

cDNA libraries were prepared from small RNAs as described [162] and sequenced using the Illumina SBS platform.

5.4.9 siRNA read processing

Sequencing reads that passed a quality filter were truncated at the 3’ linker sequence (TCGTAT) and then mapped to the loci encoding rRNA and tRNA genes, allowing up to one mismatch and randomly sampling multiple alignments where applicable. Sequences that did not align to the rRNA or tRNA were then aligned against the full genome, allowing only perfect matches and randomly sampling multiple alignments where applicable. Sequence and feature files for *C. neoformans var. grubii* H99 were obtained from the Broad Institute (Cambridge, MA) on January 20, 2011. Mapping was done using Bowtie version 0.12.7 [163].

5.4.10 Scripts for siRNA read alignment

The following three Python scripts were used to prepare sequencing tagcounts for alignment to the *C. neoformans* genome such that reads mapping to multiple genomic locations could be randomly distributed.

\(^1\)http://www.broadinstitute.org/annotation/genome/cryptococcus_neoformans/MultiHome.html
import sys
counts = sys.argv[1]
bam = sys.argv[2]

L = []
for line in open(counts, 'rU'):
    n = int(line.split('\t')[1].rstrip())
    L.append(n)

import pysam
bfh = pysam.Samfile(bam, 'rb')
import os
try:
    os.makedirs('..//reconstructed')
except:
    pass
ofh = pysam.Samfile('..//reconstructed/%s' % bam, mode='wb', template=bfh)
for aread in bfh:
    x = aread.qname
    lineno = int(x[(x.find('_')+1):x.find(':')])
    for i in xrange(L[lineno]):
        ofh.write(aread)
ofh.close()
bfh.close()

import sys
import os.path
counts = sys.argv[1]
bam = sys.argv[2]
bampath, bamfile = os.path.split(bam)

L = []
for line in open(counts, 'rU'):
    n = int(line.split('\t')[1].rstrip())
    L.append(n)

import pysam
bfh = pysam.Samfile(bam, 'rb')
import os
ofh = pysam.Samfile('%s_renamed.bam' % bam, mode='wb', template=bfh)
for aread in bfh:
    x = aread.qname
    lineno = int(x[(x.find('_')+1):x.find(':')])
    count = L[lineno]
    if count == 0:
        continue
    aread.qname += 'x%d' % count
    ofh.write(aread)
bfh.close()
5.4.11 siRNA read classification

Genomic regions giving rise to siRNAs in wild-type cells were identified as follows. The genome of \textit{C. neoformans} was parsed into non-overlapping 100 bp windows. Windows with high levels of siRNA reads were selected by applying a read density cutoff of $\geq 10$ reads/window. Adjacent windows passing the cutoff were merged. siRNAs in these windows were then classified based on their genomic positions (Figure 5.1D). Centromeric sequences, which are known to consist of fragments of transposable elements, were used as queries to identify centromere-like sequence windows, which by definition align to
centromeric sequences with a tblastx E-value cutoff of 0.00001.

5.4.12 Informatic analysis of intron features and siRNA read density

Predicted intron lengths and 5’ splice site sequences were obtained from the current Broad annotation of the *C. neoformans var grubii* genome sequence. 5’ splice site sequences were converted into self-information (bits). Comparisons of siRNA targets to the genome overall was performed using a two-tailed Kolmogorov–Smirnov test.

Comparison of siRNA read density in first versus last exons was performed as follows. The siRNA read density in the first (or last) exon of each gene was normalized to the siRNA read density in the entire corresponding ORF. The values were then log$_{10}$-transformed (setting any 0 values equal to the minimum nonzero value for a given exon class) and compared using a two-tailed Mann-Whitney U test.

5.4.13 Tandem affinity protein purification

To purify proteins tagged with CBP-2xFLAG, *C. neoformans* cultures were grown to OD$_{600}$=2.0 in YPAD media, at which point they were harvested, resuspended in TAP buffer (25 mM HEPES-KOH pH7.9, 0.1 mM EDTA, 0.5 mM EGTA, 2 mM MgCl$_2$, 20% glycerol, 0.1% Tween-20, 300 mM KCl, 1x EDTA-free Complete protease inhibitor (CPI; Roche)), snap frozen, then lysed using a coffee grinder (3 min) and mortar and pestle (20 min). The frozen powder was resuspended in TAP buffer and cleared by 27,000 x g centrifugation for 40 min at 4°C. Anti-FLAG M2 affinity resin (Sigma) was incubated in cleared lysate for 2 hr at 4°C, at which point the resin was washed three times with TAP buffer. Tagged protein was eluted by three washes with FLAG elution buffer at 4°C (25
mM HEPES-KOH pH7.9, 2 mM MgCl₂, 20% glycerol, 300 mM KCl, 1x CPI, 0.4 mg/ml 3xFLAG peptide (Sigma)) totaling 1 hr. For the second purification step, 5 volumes of calmodulin binding buffer (10 mM Tris-HCl pH7.9, 10 mM β-mercaptoethanol, 2 mM CaCl₂, 0.1% Triton X-100, 300 mM NaCl, 1x CPI) were added to the anti-FLAG resin eluate; the resulting solution was incubated with calmodulin beads (Stratagene) at 4°C overnight. The beads were then washed once with calmodulin binding buffer and three times with calmodulin wash buffer (same as calmodulin binding buffer, except 0.1 mM CaCl₂) at 4°C. Protein was eluted by five washes with calmodulin elution buffer at 4°C (10 mM Tris-HCl, 10 mM β-mercaptoethanol, 3 mM EGTA, 0.1% Triton X-100, 300 mM NaCl) totaling 1 hr 45 min. Eluted protein was precipitated with 13% trichloroacetic acid, washed with acetone, and analyzed by mass spectrometry or immunoblot.

5.4.14 Mass spectrometry reagents and chemicals

Unless otherwise noted all chemicals were purchased from Thermo Fisher Scientific. Deionized water (18.2 MW, Barnstead) was used for all preparations. Buffer A was 5% acetonitrile 0.1% formic acid, B was 80% acetonitrile 0.1% formic acid, and C was 500 mM ammonium acetate.

5.4.15 Protein Digestion

Proteins were reduced with 5 mM Tris(2-carboxyethyl)phosphine hydrochloride (C4706, Sigma) and alkylated with 10 mM Iodoacetamide (Sigma). Proteins were digested for 18 hr at 37°C in 2 M urea, 100 mM Tris pH 8.5, 1 mM CaCl₂ with 1 µg trypsin (Promega). Digest was stopped with formic acid, 5% final concentration. Debris was removed by centrifugation, 30 min 18000 x g.
5.4.16 MudPIT Microcolumn

A MudPIT microcolumn [164, 165] was prepared by first creating a Kasil frit at one end of an undeactivated 250 mm ID/360 mm OD capillary (Agilent Technologies, Inc.). The Kasil frit was prepared by briefly dipping a 20-30 cm capillary in well-mixed 300 ml Kasil 1624 (PQ Corporation) and 100 mL formamide, curing at 100OC for 4 hrs, and cutting the frit to 2 mm in length. Strong cation exchange particles (SCX Luna, 5 mm dia., 125 Å pores, Phenomenex) were packed in-house from particle slurries in methanol to 2.5 cm. 2 cm reversed phase particles (C18 Aqua, 3 mm dia., 125 Å pores, Phenomenex) were then successively packed onto the capillary using the same method as SCX loading.

5.4.17 MudPIT analysis

An analytical RPLC column was generated by pulling a 100 mm ID/360 mm OD capillary (Polymicro Technologies) to 5 mm ID tip. Reversed phase particles (Luna C18, 3 mm dia., 125 Å pores, Phenomenex) were packed directly into the pulled column at 800 psi until 15 cm long. The column was further packed, washed, and equilibrated at 100 bar with buffer B followed by buffer A. MudPIT and analytical columns were assembled using a zero-dead volume union (Upchurch Scientific). LC-MS/MS analysis was performed using an Agilent 1100 HPLC pump and Finnigan LTQ using an in-house built electrospray stage. Electrospray was performed directly from the analytical column by applying the ESI voltage at a tee (150 mm ID, Upchurch Scientific) directly downstream of a 1:1000 split flow used to reduce the flow rate to 300 nl/min through the columns. 5-step MudPIT experiments were performed where each step corresponds to 0, 20, 50, 80, and 100% buffer C being run for 5 min at the beginning of a 110 min gradient. Precursor scanning was performed from 300 - 2000 m/z. Data-dependent acquisition of MS/MS spectra was performed with the following settings: MS/MS on the 5 most intense ions per precursor
scan. Dynamic exclusion settings used were as follows: repeat count, 1; repeat duration, 30 second; exclusion list size, 300; and exclusion duration, 180 seconds.

Protein and peptide identification and modified peptide analysis were done with Integrated Proteomics Pipeline - IP2 (Integrated Proteomics Applications, Inc.) using ProLuCID, DTASelect2. Spectrum raw files were extracted into ms2 files from raw files using RawExtract 1.9.9 (http://fields.scripps.edu/downloads.php) [166], and the tandem mass spectra were searched against a Cryptococcus proteins database (Broad Institute, Cambridge, MA). In order to accurately estimate peptide probabilities and false discovery rates, we used a decoy database containing the reversed sequences of all the proteins appended to the target database [167]. Tandem mass spectra were matched to sequences using the ProLuCID algorithm with 600 ppm peptide mass tolerance. ProLuCID searches were done on an Intel Xeon cluster running under the Linux operating system. The search space included all fully tryptic peptide candidates that fell within the mass tolerance window with no mis-cleavage constraint. Carbamidomethylation (+57.02146 Da) of cysteine was considered as a static modification. DTASelect parameters were -p 2 -y 0 --trypstat --dm -in.

5.4.18 Background filtering criteria for mass spectrometry analysis

To remove likely contaminants from the list of proteins identified by mass spectrometry, the dataset was filtered to remove proteins that were: (1) identified by less than 10% peptide coverage, (2) structural components of the ribosome, (3) proteins identified in untagged sample, or (4) other likely-abundant proteins such as cytoskeletal proteins, metabolic proteins, chaperones, and mitochondrial proteins. Filtered proteins are listed in Table 5.7.
5.4.19 Immunoblotting

Proteins were analyzed by SDS-PAGE and immunoblotting using primary antibodies at the following concentrations: mouse monoclonal anti-FLAG (Sigma F3165, 1:3,000), rabbit polyclonal anti-Myc (Abcam, 1:2,500), and rabbit polyclonal anti-PSTAIRES (Santa Cruz Biotechnology sc-53, 1:2,500). Secondary antibodies included HRP-conjugated goat anti-mouse (Bio-Rad, 1:20,000) and HRP-conjugated goat anti-rabbit (Bio-Rad, 1:20,000).

5.4.20 RNA immunoprecipitation

To isolate RNA associated with CBP-2xFLAG-tagged proteins, *C. neoformans* cultures were grown to OD$_{600}$=2.0 in YPAD or YPAG media, at which point they were harvested, resuspended in TAP buffer (supplemented with 100 U/ml RNase inhibitor (New England Biolabs)), snap frozen, then lysed using a coffee grinder (3 min) and mortar and pestle (15 min). Upon thawing, the lysate was cleared by centrifugation at 27,000 x g for 40 min at 4°C, after which it was incubated with anti-FLAG M2 affinity resin (Sigma) for 2 hr at 4°C. The resin was washed three times with TAP buffer, then bound proteins were eluted by three washes with FLAG elution buffer at 4°C, totaling 1 hr. To purify protein-associated RNA, the eluate was incubated with 0.17 mg/ml Proteinase K (Sigma) for 25 min at 37°C followed by an acid phenol-chloroform extraction and ethanol precipitation. RNA samples were treated with DNaseI (DNA-free, Ambion), and subsequent RT-qPCR analysis was carried out using primers listed in Table 5.6.

5.4.21 RNA isolation and sRNA Northern blot

To analyze gene expression or siRNA abundance, *C. neoformans* cultures were grown to OD$_{600}$=1.0 in YPAD or YPAG media, at which point they were harvested and snap
frozen. RNA was isolated using TRIzol (Invitrogen). To measure transcript abundance, total RNA was treated with DNaseI (Roche) followed by RT-qPCR or primer extension. To measure siRNA abundance, small RNAs were first enriched from total RNA samples by performing a modified mirVana (Ambion) small RNA isolation procedure, as described previously [168]. Next, 40 µg sRNA samples were resolved in a 15% polyacrylamide gel and transferred to a Hybond-NX membrane (Amersham), which was incubated in crosslinking solution (0.16 M N-(3-Dimethylaminopropyl)-N’-ethylcarbodiimide hydrochloride (Sigma) prepared in 0.13 M 1-methylimidazole at pH8) at 60°C for 1 hr [160]. The crosslinked membrane was washed with water and blocked with Ultrahyb solution (Ambion) for 30 min at 68°C. Radiolabeled riboprobes corresponding to the sense strand of particular gene loci were generated by in vitro transcription of a linearized plasmid (MAXIscript, Ambion). These riboprobes corresponded to the entire locus of each examined gene except for CNAG_6705, in which case only the non-repetitive regions were included. Riboprobes were fragmented by base hydrolysis to an average size of 100 nt, and hybridized to the membrane overnight at 48°C. The membrane was washed 5 min two times at 48°C (2x SSC, 0.1% SDS), then 15 min two times at 48°C (0.1x SSC, 0.1% SDS) and imaged using a storage phosphor screen (Amersham).

5.4.22 RT-qPCR

cDNA was generated by reverse transcription of 10 µg DNasel-treated, total RNA by SuperScript III reverse transcriptase (Invitrogen) using oligo-dT20N (38 ng/µl) and random 9-mers (10 ng/µl) as primers. The manufacturer’s standard reactions conditions were used.
5.4.23 Primer extension

For primer extension, $^{32}$P end-labeled primers were annealed to 15 μg total RNA and extended by AMV reverse transcriptase for 1 hr at 42°C (Primer Extension System, Promega), after which RNA was eliminated by base hydrolysis. Products were resolved in a 6% denaturing polyacrylamide gel and sized relative to a ΦX174 DNA/Hinf I ladder (Promega).

5.4.24 Fluorescence microscopy

Strains expressing mCherry and GFP fusion proteins were grown to saturation in YPAD (or YPAG) overnight, spotted on V8 mating medium (or V8 with 2% galactose) and incubated at 25°C for 48 hr. Cells were scraped off the plates, resuspended in water, and imaged immediately at 63x magnification in an Axiovert 200 M (Zeiss) microscope running Axiovision software. The images were pseudocolored and cropped using Photoshop software (Adobe). Subcellular localization was quantified by assessing cytoplasmic and nuclear signal in 100 cells per genotype. For nuclear staining, cells were fixed using 2% paraformaldehyde for 10 min at room temperature. After washing with 1x PBS, nuclei were stained using Hoechst 33342 (Invitrogen) at a concentration of 10 μg/ml. Cells were washed and imaged as described above. In order to visualize P-bodies, log phase cells grown in YNB with 2% galactose were spun down, washed, and incubated in YNB without galactose for 10 min. Cells were washed again, resuspended in media with galactose and imaged immediately.

mCherry-Ago1, Gwo1-mCherry, and Srr1-mCherry were expressed from their endogenous promoters, whereas GFP-Gwc1, GFP-Qip1, GFP-Dcp1, and GFP-Gwo1 were expressed from the GAL7 promoter to facilitate detection.
5.4.25 Yeast two-hybrid analysis

Plasmids were generated that encoded *C. neoformans* proteins (Rdp1, Ago1, Gwc1, Qip1, or Gwo1) fused to either a transcriptional activation domain (AD) or a LexA DNA-binding domain. Plasmids encoding AD fusion proteins, which were expressed from the GAL1 promoter, were used to transform the *S. cerevisiae* strain W303A, whereas plasmids encoding LexA fusion proteins, which were expressed from the *ADH* promoter, were used to transform EGY48. The EGY48 strain also carried the plasmid pSH18-34, which encodes 8 LexA operator sequences upstream of *LacZ*. To assess *LacZ* expression stimulated by fusion protein interaction, W303A- and EGY48-derived strains were mated and maintained as diploids. Saturated cultures in SC -his -trp -ura +2% raffinose media were diluted 1:20 in SC -his -trp -ura +1% raffinose +2% galactose media and incubated at 30°C with shaking until they reached log phase, at which point β-galactosidase activity was assessed in an Infinite M200 96-well plate reader (Tecan) as previously described [169]. The β-galactosidase activity stimulated by any given interaction between an AD fusion protein and a LexA fusion protein was normalized to a bait-only control in which the LexA fusion protein was expressed alongside an AD that was not fused to any additional sequence. Interactions that stimulated β-galactosidase activity at least 3-fold relative to the activity of a bait-only control were deemed positive.

5.5 Non-essential supplemental material

Certain supplementary tables deemed non-essential and their content was not included in the thesis. These tables correspond to supplementary tables available from [2], as follows:
Table 5.1: Non-essential Supplementary Tables

5.6 Chapter acknowledgements

The following people contributed to the manuscript contained in this chapter: Phillip A. Dumesic,1 Prashanthi Natarajan,1 Changbin Chen,1 Ines A. Drinnenberg,2 Benjamin J. Schiller,1 James Thompson,3 James J. Moresco,3 John R. Yates III,3 David P. Bartel,2 and Hiten D. Madhani.1 P.A.D. and H.D.M. designed the study. C.C., I.A.D., and D.P.B. performed high throughput sequencing shown in Figure 5.1. J.T., J.J.M., and J.R.Y. performed mass spectrometry analyses shown in Figures 5.2, 5.3 and 5.4. P.N. performed subcellular localization of tagged proteins shown in Figures 5.2, 5.3 and 5.4 and 5.8. B.J.S. performed bioinformatic analyses shown in Figures 5.1, 5.5, and 5.7, and Tables 5.2 and 5.3. P.A.D. performed experiments shown in Figures 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.9, 5.10, and 5.11. P.A.D. and H.D.M. wrote the manuscript. All authors contributed to editing the manuscript.

We thank members of the Madhani and Guthrie labs, P. O’Farrell, J. Reuter, and S. Coyle for critical reading of the manuscript and helpful discussions; N. Nguyen for media preparation; and X. Wang and J. Heitman for strains and plasmids. Mass spectrometry experiments were supported by the National Center for Research Resources (5P41RR011823-17), the National Institute of General Medical Sciences (8 P41 GM103533-17), and the

1Department of Biochemistry and Biophysics, University of California, San Francisco, CA 94158, USA
2Whitehead Institute for Biomedical Research, 9 Cambridge Center, Cambridge, MA 02142, USA
3Department of Chemical Physiology, The Scripps Research Institute, La Jolla, CA 92037, USA
National Institute on Aging (R01AG027463-04).
Figure 5.8: Characterization of Ago1 cytoplasmic foci, Related to Figure 5.2. (A) Co-localization of Ago1 and Dcp1. mCherry-Ago1 was expressed from its endogenous promoter, whereas GFP-Dcp1, which localizes to P-bodies, was expressed from a GAL7 promoter to facilitate detection. Unfixed cells expressing both fusion proteins were examined after incubation in YNB media. (B) Co-localization of Ago1 and Gwo1. mCherry-Ago1 was expressed from its endogenous promoter, whereas GFP-Gwo1 was expressed from the GAL7 promoter to facilitate detection. Unfixed cells expressing both fusion proteins were examined after incubation in YNB media.

<table>
<thead>
<tr>
<th>Gene orientation</th>
<th>Non-convergent</th>
<th>Convergent</th>
</tr>
</thead>
<tbody>
<tr>
<td>All genes</td>
<td>3514</td>
<td>3466</td>
</tr>
<tr>
<td>RNAi target genes</td>
<td>59</td>
<td>59</td>
</tr>
</tbody>
</table>

Table 5.2: Frequency of convergently oriented genes among RNAi target genes, Related to Figure 5.1. Values represent number of convergent or non-convergent genes among all C. neoformans genes (n=6980) or among RNAi target genes (>150 siRNA reads; n=118). p=1, Fisher’s exact test.

Table 5.3: siRNAs mapping to genes in wild-type C. neoformans, Related to Figure 5.1. See Section 5.5.
Figure 5.9: Co-immunoprecipitation of SCANR subunit Rdp1 and spliceosome component Syf1, Related to Figure 5.4. Co-immunoprecipitation of spliceosome component Syf1 with Rdp1. Strains expressing Syf1-13xMyc and Rdp1-CBP-2xFLAG were subjected to tandem affinity purification using anti-FLAG and calmodulin resins. Input and purified material were analyzed by immunoblot using anti-FLAG, anti-Myc, or anti-PSTAIRE antibody, which stains the negative control protein p31.

<table>
<thead>
<tr>
<th></th>
<th>Input</th>
<th>FLAG-CBP tandem purification</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+ +</td>
<td>+ + Syf1-13xMyc</td>
</tr>
<tr>
<td>Rdp1</td>
<td>- +</td>
<td>Rdp1-CBP-2xFLAG</td>
</tr>
</tbody>
</table>

- anti-Myc
- anti-PSTAIRE
- anti-FLAG

Syf1
p31
Rdp1

Table 5.4: Quantitative yeast two-hybrid results, Related to Figure 5.2. Values represent average β-galactosidase activity relative to that of a bait-only control. Standard deviation of two independent experiments is indicated in parentheses. Interactions that stimulated β-galactosidase activity at least 3-fold were deemed positive, and are indicated by bold and italic type.

<table>
<thead>
<tr>
<th>LexA-fusion (bait)</th>
<th>B42-fusion (prey)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Rdp1</td>
</tr>
<tr>
<td>Rdp1</td>
<td>0.7 (0.0)</td>
</tr>
<tr>
<td>Ago1</td>
<td>0.7 (0.2)</td>
</tr>
<tr>
<td>Gwo1</td>
<td>0.8 (0.0)</td>
</tr>
<tr>
<td>Gwc1</td>
<td>1.0 (0.1)</td>
</tr>
<tr>
<td>Qip1</td>
<td>2.7 (1.2)</td>
</tr>
</tbody>
</table>

Table 5.5: Strains used in this study, Related to Experimental Procedures. See Section 5.5.

Table 5.6: Primers used in this study, Related to Experimental Procedures. See Section 5.5.

Table 5.7: Proteins removed from mass spectrometry results by background filtering criteria, Related to Experimental Procedures. See Section 5.5.
Figure 5.10: Spliceosome occupancy of RNAi target transcripts in the absence of siRNA, Related to Figure 5.5. Association of siRNA target and non-siRNA target transcripts with the spliceosome in the absence of RNAi. Spliceosomes were purified from wild-type or *rdp1Δ* cells by immunoprecipitation of Prp19-CBP-2xFLAG and co-purified RNAs were detected by RT-qPCR. Levels of individual RNAs co-immunoprecipitated with Prp19 were normalized to their abundance in wild-type whole cell extract. IP/WCE values are relative to those of purifications from wild-type (untagged) lysates. Error bars: SD. Bold: transposon-related transcripts.
Stalled spliceosomes promote siRNA production by Rd1, Related to Figure 5.6. (A) Density plot of siRNAs mapping to the genomic sequence of an RNAi target transcript, *CNAG_7888*, which comprises three exons and two introns. (B) Rd1 dependence of siRNA generation triggered by a 3’ splice site mutation of *CNAG_7888* intron 2. RNA was isolated from wild-type or *rdp1Δ* cells that expressed, under the control of a *GAL7* promoter, wild-type *CNAG_7888* or a mutated form of the gene. siRNA derived from *CNAG_7888* was detected by riboprobe hybridization; U18 snoRNA served as loading control. (C) Transcript levels of wild-type *CNAG_7888* as well as mutated forms of the gene in which individual splice sites were mutated, as assessed by RT-qPCR. All *CNAG_7888* alleles were expressed from a *GAL7* promoter at the endogenous *CNAG_7888* locus. Expression levels were normalized to levels of actin transcript. Error bars: SD. (D) Effect of intron sequence on the siRNA production triggered by splice site mutations of *CNAG_7888*. RNA was isolated from cells that expressed, from a *GAL7* promoter, either wild-type *CNAG_7888* or a mutated form of the gene in which the sequences of its two introns were swapped and splice sites were mutated. siRNA derived from *CNAG_7888* was detected by riboprobe hybridization; staining of U18 snoRNA served as loading control. (E) Effect of suboptimal intron size on siRNA production and splicing of the *CNAG_7888* transcript. RNA was isolated from cells that expressed, from a *GAL7* promoter, either wild-type *CNAG_7888* or a mutated form of the gene containing a sequence insertion between the branchpoint adenine and 3’ splice site of intron 2. siRNA production was assessed by riboprobe hybridization and splicing was assessed by primer extension using a labeled primer complementary to *CNAG_7888* exon 3, which yielded a discrete product corresponding to the lariat intermediate of each mutant intron. Asterisk denotes a nonspecific primer extension product. Primer extension using a primer specific to U6 snRNA served as a loading control.
Chapter 6

twobitreader: a fast python package for reading .2bit files

6.1 License and Warranty

Licensed under Perl Artistic License 2.0

No warranty is provided, express or implied

6.2 Source Code

Source code is provided in Appendix A.

6.3 twobitreader module

twobitreader

Licensed under Perl Artistic License 2.0 No warranty is provided, express or implied
class twobitreader.TwoBitFile(foo)
    python-level reader for .2bit files (i.e., from UCSC genome browser) (note: no writing support)

    TwoBitFile inherits from dict You may access sequences by name, e.g. >>> genome = TwoBitFile('hg18.2bit') >>> chr20 = genome['chr20']
    Sequences are returned as TwoBitSequence objects You may access intervals by slicing or using str() to dump the entire entry e.g.
    >>> chr20[100100:100120]
    'ttttcctctaagataatattttgccttaaataactttttgttaataactaagaag taagataactttttttttgttaatttgcatgttaagttttttcc'
    >>> whole_chr20 = str(chr20)
    Fair warning: dumping the entire chromosome requires a lot of memory
    See TwoBitSequence for more info

    sequence_sizes()
        returns a dictionary with the sizes of each sequence

exception twobitreader.TwoBitFileError(msg)
    Base exception for TwoBit module

class twobitreader.TwoBitSequence(file_handle, off, file_size, byteswapped=False)
    A TwoBitSequence object refers to an entry in a TwoBitFile

    You may access intervals by slicing or using str() to dump the entire entry e.g.
    >>> genome = TwoBitFile('hg18.2bit')
    >>> chr20 = genome['chr20']
    >>> chr20[100100:100200] # slicing returns a string
    'ttttcctctaagataatattttgccttaaataactttttgttaataactaagaag taagataactttttttttgttaatttgcatgttaagttttttcc'
    >>> whole_chr20 = str(chr20) # get whole chr as string
    Fair warning: dumping the entire chromosome requires a lot of memory

    Note that we follow python/UCSC conventions: Coordinates are 0-based, end-open (Note: The UCSC web-based genome browser uses 1-based closed coordinates) If you attempt to access a slice past the end of the sequence, it will be truncated at the end.

    Your computer probably doesn’t have enough memory to load a whole genome but if you want to string-ize your TwoBitFile, here’s a recipe:
    x = TwoBitFile('my.2bit') d = x.dict() for k,v in d.iteritems(): d[k] = str(v)
get_slice(min_, max_=None)
    get_slice returns only a sub-sequence

twobitreader.base_to_bin(x)
    provided for user convenience convert a nucleotide to its bit representation

twobitreader.bits_to_base(x)
    convert integer representation of two bits to correct base

twobitreader.byte_to_bases(x)
    convert one byte to the four bases it encodes

twobitreader.cmdline_reader()
    cmdline_reader allows twobitreader module to be executed as a script accepts only
    one argument – the .2bit filename reads input (BED format) from stdin writes
    output (FASTA format) to stdout writes errors/warning to stderr
    Regions should be given in BED format on stdin chrom start(0-based) end(0-based,
    not included)
    To use a BED file of regions, do python -m twobitreader example.2bit < example.bed
    Non-regions will be skipped and warnings will be issued to logging (logging output
    to stderr by default)

twobitreader.create_byte_table()
    create BYTE_TABLE

twobitreader.create_twobyte_table()
    create TWOBYTE_TABLE

twobitreader.print_specification()
    Prints the twoBit file format specification I got from the Internet. This is only here
    for reference

twobitreader.split16(x)
    split a 16-bit number into integer representation of its course and fine parts in binary
    representation

twobitreader.true_long_type()
    OS X uses an 8-byte long, so make sure L (long) is the right size and switch to l
    (int) if needed

twobitreader.twobit_reader(twobit_file, input_stream=None, write=None)
twobit_reader takes a twobit_file (of class TwoBitFile) and an “input_stream” which can be any iterable (incl. file-like objects) writes output (FASTA format) using write (print if write=None) logs errors/warning to stderr

Regions should be given in BED format on stdin chrom start(0-based) end(0-based, not included)

To use a BED file of regions, do python -m twobitreader example.2bit < example.bed

Non-regions will be skipped and warnings will be issued to logging (logging output to stderr by default)

6.4 twobitreader.download

downloads a .2bit genome from UCSC .. note:: please comply with the restrictions of use at http://hgdownload.cse.ucsc.edu/downloads.html

and do not over-use this module

By default, genomes are saved to the current directory

twobitreader.download.save_genome(name, destdir=None, mode='ftp')

tries to download a genome from UCSC by name. for example, ‘hg19’ is at ftp://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/hg19.2bit

6.5 Run module as script

6.5.1 twobitreader

twobitreader can be imported as a module within python or run as a script from the command line, for example

$ python -m twobitreader example.2bit < example.bed

When run in this mode, twobitreader accepts only one argument, the filename of the .2bit genome, and reads coordinates from stdin in BED format
Output is given in FASTA format to stdout (warnings and errors are issued on stderr)

6.5.2  twobitreader.download

twobitreader.download can also be run as a script to fetch a genome by name, for example

$ python -m twobitreader.download hg19

which will save hg19.2bit to your current directory
Chapter 7

scripter: a tool for parallel execution of functions on many files

7.1 License and Warranty

Licensed under Perl Artistic License 2.0

No warranty is provided, express or implied

7.2 Source Code

Source code is provided in Appendix B.
7.3 Philosophy

scripter tries to make it easy to write scripts that parallelize tasks by first parsing filenames and options, and then executing an action (function) on the parsed filename objects.

7.4 Setting up the Environment

Its critical class is Environment which will generally be imported by

```python
import scripter
e = scripter.Environment(version=VERSION, doc=__doc__)
```

Passing the version and documentation (usually __doc__) is recommended so that users can use the expected “–help” and “–version” options

7.5 Attaching a FilenameParser

It is usually necessary to set a FilenameParser, which acts on the filenames given at the command. Your FilenameParser should inherit the class FilenameParser from scripter and should usually execute its __init__() method (either before or after yours, see the example below). It is possible to use scripter’s FilenameParser directly if you don’t need to customize it much. It is important that you allow **kwargs in __init__() for your custom FilenameParser or it will almost certainly fail to work. All options given at the command line and by scripter are passed to both FilenameParser and the action. Here is an example FilenameParser hooked to Environment
class ExampleFilenameParser(FilenameParser):
    def __init__(self, filename, number_of_apples=5, **kwargs):
        super(ExampleFilenameParser, self).__init__(self, **kwargs)
        self.tree = [filename + '_%d.txt' % num for num in range(number_of_apples)]

e.set_filename_parser(ExampleFilenameParser)

7.6 Defining an action

The last thing you must for the script to run is to define the action and tell the Environment to execute it. Like FilenameParser, the action should accept **kwargs or it will probably fail. Here is an example action which makes files in the output directory for the specified number_of_apples:

import os.path

def example_function(filename_obj, **kwargs):
    tree = filename_obj.tree
    input = filename_obj.input_file
    output_dir = filename_obj.output_dir
    for f in tree:
        output_filename = os.path.join(output_dir, f)
        fh = open(output_filename, 'wb')
        fh.write('This apple came from %s' % input)

e.do_action(example_function) # this actually starts the script
7.7 Modifying the script options

You probably want to specify additional values at the command line besides the scripter defaults. You can import the argument parser and modify it, see argparse for more information:

```python
def main():
    parser = e.argument_parser
    parser.add_argument("--number-of-apples", type=int, nargs='?')
```

Options from the parser are converted into keywords that can be accessed either from the kwarg dictionary or by including the kwarg directly in the action definition or the `__init__` method of the `FilenameParser` (we did that above by include number_of_apples in our custom `__init__` method)

scripter includes a number of default options that set various script parameters

7.8 Dealing with errors/exceptions

If something goes wrong with scripter, it will usually raise a `scripter.Usage` exception.

You may want to raise these too to avoid users seeing python error codes.

The decorator `scripter.exit_on_Usage()` is provided for allowing scripts to exit gracefully when errors occur. It is often a good idea to decorate the main action for this purpose:

```python
from scripter import Usage, exit_on_Usage

@exit_on_Usage

def action(filename_obj, **kwargs):
    try:
```
do_something()

ext except NameError:
    raise Usage, 'could not do something'

action()

This will cause the program to exit and tell the user “could not do something”

7.9 Logging

Use scripter.get_logger() inside the action for parallel-processing-safe logging

7.10 Default command-line options

A number of default command-line options are included with the default argument parser in Environment.

See Also:

__init__()

Most importantly, we take a list of filenames to be acted on. This list accepts wildcards using glob

Additionally, there are a number of optional arguments:

- **-h, --help** show this help message and exit
- **-v, --version** show version info and exit
- **-p NUM_CPUS, --num-cpus NUM_CPUS** specify the number of maximum # CPUs to use
--debug
Sets logging level to DEBUG

--info
Sets logging level to INFO [default]

--quiet
Sets logging level to WARNING

--silent
Sets logging level to ERROR

--target TARGET
Specify the target directory

--no-target
Write new files in the current directory / do
not preserve directory structure

--recursive RECURSIVE, -r RECURSIVE
Recurse through
any directories listed looking for valid files

--no-action, --do-nothing, --dry-run
Don’t act on files

--config CONFIG
Use configuration in file foo

7.11 scripter API

Warning: Do not use private methods, they may change or disappear in future
releases.

7.11.1 Classes

class scripter.Environment(doc=None, version='', handle_files=True)
the base class for scripter
provides an execution environment for jobs

    __init__(doc=None, version='', handle_files=True)
_build_default_parser(doc=None, version='', add_help=True)
build the default ArgumentParser

_construct_target(name=None)

static _is_valid_dir(f)
checks if a directory is valid

_is_valid_file(f)
checks if a file is valid for processing

_update_sequence(files=[], recursive=False, **kwargs)
updates _sequence with files specified at command line (wildcards ok)

do_action(action, stay_open=False)
exectues an action
actions should be functions that at least take FilenameParser objects

execute_next_script()
execute the next script

get_context(force_new=False)

get_filename_parser(**kwargs)
returns the class being used as the filename parser
if more kwargs are supplied, then partial is used to apply arguments as appropriate

get_sequence(**kwargs)
returns the sequence of FilenameParser objects for action
Running this more than once will not do anything

get_target_dir(name=None)

override_num_cpus(num)
override the number of processes we’re going to start

set_config_reader(reader)
will be called as reader(vars(parser.parse_args())['config'])

set_config_writer(writer)
will be called as writer(**parser.parse__args())

set_filename_parser(filename_parser)
use the provided filename parser instead of the default one

update_context(update_dict)

class scripter.FilenameParser(filename, drop_parent_name=True, target_dir=None, no_target=False, *args, **kwargs)
The default FilenameParser class included with scripter
its one mandatory argument is a filename it must accept arbitrary **kwargs or it
will be very unhappy
It is recommend you customize this class for parsing filenames as needed

__init__(filename, drop_parent_name=True, target_dir=None, no_target=False, *args, **kwargs)
check_output_dir(output_dir)
set_input_file(filename)
with_extension(ext)
    Path to output file with extension

class scripter.AnnounceExitFilter(announce_exit)
rejests messages announcing thread exit iff the initial condition is False
looks for specific messages hardcoded into multiprocessing/pool.py see source for
more details

__init__(announce_exit)
filter(record)

7.11.2 Exceptions

class scripter.InvalidFileException(arg=None)
    Bases: exceptions.ValueError
Exception for files that do not return a valid FilenameParser object

```python
class scripter.Usage(*args):
    Bases: exceptions.Exception
```

### 7.11.3 Decorators

```python
scripter.exit_on_Usage(func)
    exit_on_Usage is a decorator that cause functions raising Usage to exit nicely
```

### 7.11.4 Functions

```python
scripter.assert_path(path)
    if path does not exist, raise IOError

scripter.extend_buffer(b, x, spacerlines=0)
    extends buffer b with string x, ignores if x is None

scripter.get_logger(level=WARNING) wraps multiprocessing.get_logger()
    adds an AnnounceExitFilter to prevent output from getting very garbled at program exit

scripter.is_valid_executable(filename)
    checks if a filename is a valid executable

scripter.leaves(dir_or_file, allow_symlinks=True, ignore_hidden_files=True, max_depth=None)
    takes as input a VALID path and descends into all directories
    WARNING: this will get caught in an infinite loop if you have a symlink which references a node above itself in tree

scripter.path_to_executable(name, directories=None, max_depth=2, envi-
    ron=None)
    construct the path to the executable, search in order
    **the directory specified (or any directory that matches with Unix style pathname pattern expansion*)**
    then env PATH then the current directory then give up
we reverse the order so that we will usually get the newest version

scripter.pformat_list(L)
Takes a list and turns each item into a str then returns the pretty-printed version of that list

scripter.valid_directories(directory)
wrapper for glob.glob, enforces that output must be a valid directory

scripter.valid_int(thing, msg, vmin, vmax)
checks if something is a valid integer and thing >= vmin and thing <= vmax
returns the thing as an integer
Chapter 8

seriesoftubes: An extended pipeline for Solexa ChIP-seq data

8.1 License and Warranty

Licensed under Perl Artistic License 2.0

No warranty is provided, express or implied

8.2 Source Code

Source code is provided in Appendix C.
8.3 Scripts

8.3.1 Overview

seriesoftubes is organized as a set of scripts, built on top of scripter (see Chapter 7). preprocess_reads.py (Section 8.3.2) is used to extract sequences from raw reads (if necessary); align2.py (Section 8.3.3) or align.py (not described here) is used to align sequences to the reference genome; call_peaks.py (Section 8.3.4) is used to find genomic regions that are enriched in an aligned ChIP-seq library; and analyze.py (Section 8.3.5) will find occurrences of a specified motif in a set of genomic regions. The scripts and their usage are described in detail in this section.

8.3.2 preprocess_reads.py

preprocess_reads.py is meant to be run on GERALD output although it will run on any FASTQ files. It can perform the following tasks:

- Separate reads by a set of variable-length barcodes
- Cleave linker/adaptor sequence from the 3’ ends of reads
- Cleave adapter sequence from the 5’ end and/or before barcode and/or after barcode
- Discard sequences that are less than 4 nucleotides in length
- Produce gzipped FASTQ sequence files ready for immediate alignment

Note: TRAILING Ns ARE NO LONGER TRIMMED (per NCBI guidelines). A configuration file 'preprocess_reads.cfg' is saved in target directory (unless one is provided by the user). The script expects that files are named s_?_sequence.* (single-end reads) or s_?_[12]_sequence.* (paired-end reads)
usage: preprocess_reads.py [-h] [-v] [-p NUM_CPUS]
            [--debug | --info | --quiet | --silent]
            [--target TARGET] [--no-target] [--recursive]
            [--no-action] [--config CONFIG] [--no-clipping]
            [--strip-after-barcode STRIP_AFTER_BARCODE]
            [--strip-before-barcode STRIP_BEFORE_BARCODE]
            [--min-length MIN_LENGTH] [--max-length MAX_LENGTH]
            [--no-gzip] [-b BARCODES] [--kry-barcodes]
            [--linker LINKER]
            [files [files ...]]

position arguments:
    files          A list of files to act upon (wildcards ok)

optional arguments:
    -h, --help     show this help message and exit
    -v, --version  show version info and exit
    -p NUM_CPUS, --num-cpus NUM_CPUS
                    specify the number of maximum \# CPUs to use
    --debug        Sets logging level to DEBUG
    --info         Sets logging level to INFO [default]
    --quiet        Sets logging level to WARNING
    --silent       Sets logging level to ERROR
    --target TARGET
    --no-target    Write new files in the current directory / do not
                   preserve directory structure
    --recursive, -r Recurse through any directories listed looking for
                   valid files
    --no-action, --do-nothing, --dry-run
                    Don’t act on files
    --config CONFIG Use configuration in file foo
    --no-clipping  Do not clip barcodes from reads when assigning to
                   barcode. OVERIDES ALL OTHER CLIPPING OPTIONS
    --strip-after-barcode STRIP_AFTER_BARCODE
                    strip n bases after the barcode is removed (5’ end)
                    (by default this 1 now, and is ignored if GERALD
                     handled the barcoding)
    --strip-before-barcode STRIP_BEFORE_BARCODE

171
strip n bases before the barcode is removed (5' end)
(by default this 0 now, and is ignored if GERALD handled the barcoding)

--min-length MIN_LENGTH
require sequences to be at least n total bases of
non-N sequence (default: ignore)

--max-length MAX_LENGTH
truncate final sequences to n bases (default: ignore)

--no-gzip
Do not gzip output files

--linker LINKER
Specify a 3' adaptor/linker sequence that we should
clip off of each read

barcodes:
Specify sequence barcodes in the sample(s)

-b BARCODES, --barcodes BARCODES
Specify a barcode sequence. May be invoked multiple
times

--kry-barcodes
Alias for -bTCAT -bGACG -bAGTC -bCTGA

8.3.3  align2.py

align FASTQ, SAM, or BAM file (gzip and bzip2 supported) with bowtie2, produces BAM
files (sorted and indexed) output will be in ./align2 In align2, there will be a folder for
each reference genome e.g. align2/ref1, align2/ref2, align2/ref1, align2/ref2

usage: align2.py [-h] [-v] [-p [NUM_CPUS]]
[--debug | --info | --quiet | --silent] [--target [TARGET]]
[--path-to-bowtie2 [PATH_TO_BOWTIE2]]
[--path-to-samtools [PATH_TO_SAMTOOLS]]
[--reference REFERENCES] [--ignore-quality]
[--counter-reference COUNTER_REFERENCES]
[--passthru-args [PASSTHRU_ARGS [PASSTHRU_ARGS ...]]]
[files [files ...]]

positional arguments:
files A list of files to act upon (wildcards ok)
optional arguments:
-h, --help show this help message and exit
-v, --version show version info and exit
-p [NUM_CPUS], --num-cpus [NUM_CPUS]
    specify the number of maximum # CPUs to use
--debug Sets logging level to DEBUG
--info Sets logging level to INFO [default]
--quiet Sets logging level to WARNING
--silent Sets logging level to ERROR
--target [TARGET]
--no-target Write new files in the current directory / do not
    preserve directory structure
--recursive, -r Recurse through any directories listed looking for
    valid files
--no-action, --do-nothing, --dry-run
    Don't act on files
--config CONFIG Use configuration in file foo
--path-to-bowtie2 [PATH_TO_BOWTIE2]
    The path to the bowtie2 executable
--path-to-samtools [PATH_TO_SAMTOOLS]
    The path to the samtools executable
--reference REFERENCES
    Reference genome to align against (either a bowtie2
    index name or file, or a fasta file). This flag may be
    called multiple times (which will cause each reference
    to be aligned to separately). If no references are
    specified, we'll look the for environment variable
    SOT_DEFAULT_REFERENCES, which should be given as a
    list, e.g. "foo foo2 foo3"
--ignore-quality Ignore quality scores if available. Also applies to
    counter-references if any are called
--passthru-args [PASSTHRU_ARGS [PASSTHRU_ARGS ...]]
    A list of arguments to be passed through to bowtie2
    [alignment and counter-alignment]. Substitute + for -
    (e.g., --passthru-args +m 4 50

counter-alignments:
    specify counter-reference genome(s)/sequence(s) to use for filtering out
unwanted reads.

Optional counter-reference genome/sequences to align against (either a bowtie2 index name or file, or a fasta file). This flag may be called multiple times. All counter-references will be concatenated into one index, and reads will be aligned in --fast mode. Any reads which align will be saved in a separate directory called 'counteraligned' and not aligned against the reference genomes/sequences. If no counter-references are specified, we'll look the for environment variable SOT_DEFAULT_COUTNER_REFERENCES, which should be given as a list, e.g. "foo foo2 foo3"

## 8.3.4 `call_peaks.py`

calls peaks using MACS v2 use –config to specify file with matched sample/controls.

Additional flags can passed to MACS v2 (callpeak) using –passthru-args

usage: call_peaks.py [-h] [-v] [-p [NUM_CPUS]]

    [--debug | --info | --quiet | --silent]

    [--target [TARGET]] [--no-target] [--recursive]

    [--no-action] [--config CONFIG] [-g USER_GSIZE]

    [--path-to-macs PATH_TO_MACS] [--no-subpeaks] [-q QVALUE]

    [--passthru-args [PASSTHRU_ARGS [PASSTHRU_ARGS ...]]]

    [files [files ...]]

positional arguments:

files A list of files to act upon (wildcards ok)

optional arguments:

-h, --help show this help message and exit

-v, --version show version info and exit

-p [NUM_CPUS], --num-cpus [NUM_CPUS]

    specify the number of maximum # CPUs to use

--debug Sets logging level to DEBUG

--info Sets logging level to INFO [default]
--quiet          Sets logging level to WARNING
--silent         Sets logging level to ERROR
--target [TARGET]
--no-target      Write new files in the current directory / do not
                 preserve directory structure
--recursive, -r  Recurse through any directories listed looking for
                 valid files
--no-action, --do-nothing, --dry-run
                 Don’t act on files
--config CONFIG  Use configuration in file foo
-g USER_GSIZE, --genome-size USER_GSIZE
                 Optional user-specified genome size (DEFAULT: script
                 will try to auto-detect the genome)
--path-to-macs PATH_TO_MACS
                 optional path to macs2 executable
--no-subpeaks    do not call subpeaks with --call-summits
-q QVALUE, --q-value QVALUE
                 FDR/q-value cutoff (default is 0.01)
--passthru-args [PASSTHRUARGS [PASSTHRUARGS ...]]
                 A list of arguments to be passed through to MACS2.
                 Substitute + for - (e.g., --passthru-args -m 4 50

8.3.5  analyze.py

analyze.py searches a pairs of BED (or MACS xls) and FASTA files for matches to a TFBS
motif

usage: analyze.py [-h] [-v] [-p [NUM_CPUS]]
                 [-d | --info | --quiet | --silent] [--target [TARGET]]
                 [-bysummit] --motif MOTIF_FILE
                 [--motif-number MOTIF_NUMBER] [--motif-type MOTIF_TYPE]
                 [-g GENOME]
                 [files [files ...]]

positional arguments:
files    A list of files to act upon (wildcards ok)
optional arguments:

- h, --help show this help message and exit
- v, --version show version info and exit
- p [NUM_CPUS], --num-cpus [NUM_CPUS]
    specify the number of maximum # CPUs to use
--debug Sets logging level to DEBUG
--info Sets logging level to INFO [default]
--quiet Sets logging level to WARNING
--silent Sets logging level to ERROR
--target [TARGET]
--no-target Write new files in the current directory / do not
    preserve directory structure
--recursive, -r Recurse through any directories listed looking for
    valid files
--no-action, --do-nothing, --dry-run
    Don’t act on files
--config CONFIG Use configuration in file foo
--bysummit Assume FASTA files are centered at the summit, rather
    than spanning start to end.
--motif MOTIF_FILE Path to file containing motif
--motif-number MOTIF_NUMBER
    Motif number within file (e.g. 1, 2, 3) [Default is
    1st]
--motif-type MOTIF_TYPE
    motif type (see Bio.Motif for more info)
--genome GENOME Reference genome (path to 2bit file)

8.4 Installation of seriesoftubes

Installing seriesoftubes is easy if you already have Python and setuptools installed:

easy_install seriesoftubes

However, in order to use the scripts, you will need to manually install the
external dependencies too.
8.5 Requirements

8.5.1 Python 2.7

seriesoftubes is written in and requires Python. Currently version >= 2.7 is required (Python 2.7.3 is recommended)

8.5.2 Setuptools

Setuptools (or distribute) is now required for installation. It is probably already installed, but if not, you must download and install it. It will install all the other python packages for you.

8.5.3 Operating System

Several parts of seriesoftubes will not function properly on Windows, if at all. Mac OS X and Linux/*nix should work, and the pipeline has been most recently tested on Ubuntu 12.04 LTS.

8.5.4 Python Packages (installed with easy_install)

If you install seriesoftubes using easy_install from setuptools (or an equivalent package), then it will automatically install most of the dependencies. If not, then you must install the following packages

- Biopython
- pysam
- scripter
8.5.5 External software dependencies

Bowtie2 (recommended) or Bowtie

Bowtie2 is used by align2.py (recommended) to align reads to the reference genome. Legacy support for bowtie is provided by align.py. align2.py expects bowtie2 to be in your PATH or in /usr/local/bowtie2(-*). Similarly, align.py expects bowtie to be in your PATH or in /usr/local/bowtie(-*).

Samtools

Samtools is required for preprocess_read.py, align.py, and align2.py. seriesoftubes expects samtools to be in your PATH or in /usr/local/samtools(-*).

8.6 seriesoftubes API

Contents:

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>seriesoftubes.converters</td>
<td>conversion/discover tools for file formats</td>
</tr>
<tr>
<td>seriesoftubes.fnparsers</td>
<td>implements FilenameParsers for scripter</td>
</tr>
<tr>
<td>seriesoftubes.tubes</td>
<td>contains useful functions and classes for stringing together input and output</td>
</tr>
<tr>
<td>seriesoftubes.bedgraph</td>
<td></td>
</tr>
</tbody>
</table>
8.6.1 seriesoftubes.converters

conversion/discover tools for file formats most of these can be run as scripts

seriesoftubes.converters.discover

...discover.discover_file_format(filename)

  discover the format of a file returns a tuple (open_function, ‘FORMAT’)

  open_function will either be open, gzip.GzipFile, bz2.BZ2File, or None

  FORMAT can be ‘BAM’, ‘SAM’, ‘FASTQ’, ‘FASTA’, or None

seriesoftubes.converters.cat

Writes file(s) to stdout, decompress them if needed supports gzip, bzip2

Output is always to stdout (err goes to stderr, redirect it if you need to)

seriesoftubes.converters.cat.main()

  what to do if we execute the module as a script

     cat

seriesoftubes.converters.anytotab

Convert single-end or paired-end sequencing files to tab-format NAME SEQ1 QUAL1

  (SEQ2 QUAL2)

Supports FASTQ (plaintext, gzip and bz2), SAM, BAM Output is always to stdout (err
goes to stderr, redirect it if you need to)

seriesoftubes.converters.anytotab.main()

  what to do if we execute the module as a script
seriesoftubes.converters.bamtofastq

Convert BAM/SAM to FASTQ format* @name sequence +name quality score (phred33)
files may be SAM or BAM (autodetected) If the file(s) contain paired-end sequences, we
will write to two files (in the current working directory) If the files contain single end
sequences, we will write to stdout by default Output is always to stdout (err goes to
stderr, redirect it if you need to)

class ...bamtofastq.UnpairedBAMToFastqConverter(file_, wd=None,
            stderr=None, log-
ger=None)

    Works with unpaired SAM/BAM file

seriesoftubes.converters.bamtofastq.main()
    what to do if we execute the module as a script
    bamtofastq can only convert files (not stdin) because of the paired-end problem

seriesoftubes.converters.bamtofastq.read_files(files=None,
                               no_gzip=False, sing-
gle_stdout=<open file
       ‘<stdout>’, mode ‘w’ at
          0x7fb1ce09b1e0>)

    actually reads the SAM/BAM files

seriesoftubes.converters.bamtofastq2

Another module for converting bamtofastq, defaults to writing gzfiles Accepts only one
input file

Includes class for converting paired-end sequencing files to two fastq pipes

Supports SAM, BAM

class ...bamtofastq2.PairedBAMToFastqConverter(file_, wd=None,
                                                   stderr=None, log-
ger=None)

    Works with any SAM/BAM file
seriesoftubes.converters.bamtofastq2.main()
    what to do if we execute the module as a script (not intended for user by user)

seriesoftubes.converters.bamtotab

Convert BAM/SAM to tab-format: NAME SEQ1 QUAL1 SEQ2 QUAL2

Files may be SAM or BAM (autodetected). Output is always to stdout (err goes to stderr, redirect it if you need to)

seriesoftubes.converters.bamtotab.main()
    what to do if we execute the module as a script bamtotab can only convert files because of the paired-end problem

seriesoftubes.converters.fastqtotab

Convert single or pairs of FASTQ files to tab-format

NAME SEQ1 QUAL1 (SEQ2 QUAL2)

Output is always to stdout (err goes to stderr, redirect it if you need to)

seriesoftubes.converters.fastqtotab.main()
    what to do if we execute the module as a script fastqtotab can only convert files because of the paired-end problem

seriesoftubes.converters.fastqtotab.read_files(file1, file2, open_func)
    We do not validate records here

8.6.2 seriesoftubes.fnparsers

implements FilenameParsers for scripter

class seriesoftubes.fnparsers.BowtieFilenameParser(filename, *args,
    **kwargs)

    check_paired_end()
tmp_filename(ref, match_type=None)

class seriesoftubes.fnparsers.BarcodeFilenameParser(filename, verbose=False, *args, **kwargs)

output_filename(barcode, is_barcode=True, no_gzip=False)

output_filename2(barcode, is_barcode=True, no_gzip=False)

class seriesoftubes.fnparsers.BAMFilenameParser(filename, controls={}, *args, **kwargs)

for alignments produced by align.py

class seriesoftubes.fnparsers.PeaksFilenameParser(filename, include_width_in_name=False, target=None, motif_file='unknown_motif', genome=None, *args, **kwargs)

8.6.3 seriesoftubes.tubes

contains useful functions and classes for stringing together input and output via pipes and converting data formats on the fly

seriesoftubes.tubes.wait_for_job(job, logs=[], logger=None)

wait for a job (Popen instance) to complete

class seriesoftubes.tubes.polledpipe.PolledPipe(logger=None, level=None)

A PolledPipe object has two attributes:

r - pipe read file descriptor, w - pipe write file descriptor

and three methods:

poll() – poll the read end of the pipe, readlines() – read available lines if the poll says the pipe is ready, log(level=logging.error) – emit the result of readline (if not None)

a select.poll() object has r registered to it
log(level=None)
    emit the results of readlines

8.6.4 seriesoftubes.bedgraph

seriesoftubes.bedgraph.in_window_factory(chrom, start, end)
    function factory
    returns a function that checks if a bedgraph line overlaps a given interval. along
    with UCSC convention, end coordinate is open and start coordinate is 0-based (also
    compatible with BED files)

seriesoftubes.bedgraph.in_windows_factory(intervals)
    function factory
    returns a function that checks if a bedgraph line overlaps any given interval. assumes
    we already filtered by chrom
    intervals must be given as tuples (int start, int end)
    along with UCSC convention, end coordinate is open and start coordinate is 0-based
    (also compatible with BED files)

seriesoftubes.bedgraph.window_to_iter(chrom, start, end, filename,
my_type=<type 'int'>)
    generator that returns the values in a region [start,end) on chrom
    from a bedgraph file called filename
    note: bedgraph values are assumed to be ints unless my_type is set to something
    else

8.6.5 bioplus API (Part of seriesoftubes)

<table>
<thead>
<tr>
<th>bioplus.fasta</th>
<th>meta-tools for large FASTA files</th>
</tr>
</thead>
<tbody>
<tr>
<td>bioplus.genometools</td>
<td>requires genome.db from bioplus, an sqlite3 database w/</td>
</tr>
<tr>
<td>bioplus.motif</td>
<td></td>
</tr>
<tr>
<td>bioplus.peaktools</td>
<td></td>
</tr>
</tbody>
</table>

Continued on next page
8.6.6 bioplus.fasta

meta-tools for large FASTA files

bioplus.fasta.count(\texttt{foo})
  takes a file named \texttt{foo} returns the number lines

bioplus.fasta.pad(\texttt{x}, \texttt{y})
  takes two integers \texttt{x} and \texttt{y}, and returns \texttt{str(x)} with enough 0s to match the length of \texttt{str(y)}

bioplus.fasta.permute fasta(\texttt{f})
  takes a FASTA file and returns a new FASTA file with each sequence randomly permuted (separately, such that its % A,T,G,C doesn't change)

bioplus.fasta.random_files(\texttt{foo}, \texttt{n}, \texttt{R})
  takes a FASTA file \texttt{foo} and creates (in the current directory) \texttt{R} random files each containing \texttt{n} random sequences from \texttt{foo}, named \texttt{foo\_random[0-R].fa}

bioplus.fasta.random_seq(\texttt{foo}, \texttt{n})
  takes a file \texttt{foo} and returns \texttt{n} random sequences from it

bioplus.fasta.reader(\texttt{foo})
  generator yielding Bio.Seq.Seq objects from a FASTA file

bioplus.fasta.truncate_lines(\texttt{f}, \texttt{n})
  \texttt{truncate\_lines(f,n)} truncates lines in a file to at most \texttt{n} characters See \texttt{truncate\_seqs} to truncate sequences instead of lines

bioplus.fasta.truncate_seq(\texttt{f}, \texttt{n})
  \texttt{truncate\_seqs(f,n)} truncates sequences in a FASTA file named \texttt{f} to at most \texttt{n}
bases, writing a new FASTA file named f.n.fa

bioplus.fasta.writer(\texttt{foo, iterable})
writes SeqRecord objects from iterable to FASTA file \texttt{foo}. Warning: overwrites \texttt{foo}, does not append

### 8.6.7 \texttt{bioplus.genometools}

requires genome.db from bioplus, an sqlite3 database w/ the following structure:

- tables represent genomes and are named according to genome name
- each table has fields VARCHAR `name` and INT UNSIGNED `length`
- each entry describes a chromosome/scaffold in a genome

\textbf{exception} \texttt{bioplus.genometools.GenomeNotAvailableError}

\textbf{exception} \texttt{bioplus.genometools.NoMatchFoundError}

\texttt{bioplus.genometools.TemporaryGenomeFile(\texttt{genome\_name})}
returns a file-like object pointing to a temporary file containing the chromosome names and sizes

- the current file position will be 0
- it will be deleted when the object is garbage collected

\texttt{bioplus.genometools.add\_genome(\texttt{genome\_name, genome\_dict, replace=False})}

\textbf{WARNING: THIS WILL PERMANENTLY ADD A GENOME}

- make a call to pybedtool and tries to add to genome registry then reloads module and repopulates globals
- note: this may not be persistent if your egg is zipped

\texttt{bioplus.genometools.genome(\texttt{genome\_name})}
if available, returns a dict of \{‘chr\_name’: length\} for all chromosomes in that genome

\texttt{bioplus.genometools.genome\_from\_bam(\texttt{bam\_file\_or\_filename})}

\texttt{bioplus.genometools.guess\_bam\_genome(\texttt{bam\_file\_or\_filename})}
bioplus.genometools.guess_genome(genome1)
    expects a dictionary or iterable of ('name', length)
    checks for a match in the database

bioplus.genometools.import_db(db, replace=False, ignore_warnings=False)
    imports a db like genome.db into the existing database
    note: this may not be persistent if your egg is zipped

bioplus.genometools.matches_genome(genome1, genome2, symmetric=False)
    expects two dictionaries or iterables of ('name', length) tells you if genome1 matches
    genome2 (not all the entries in genome2 need to be matched, but all the entries in
    genome1 must be)
    symmetric=True means matches(genome1,genome2) and
    matches(genome2,genome1)

8.6.8 bioplus.motif

class bioplus.motif.CharDict(dict={})
    CharDict is a generalized dictionary that inherits defaultdict. default values are 0.
    Accepts any keys.

    uncertainty()
    calculates the uncertainty H in this position (specified by CharDict). reats
    0*log(0) as 0

bioplus.motif.KL(p, q)
    returns a list of the KL divergence (relative entropy) at each position from position-
    matrix p to positionmatrix q. use sum() for the sum

class bioplus.motif.PositionWeightMatrix(n=None)
    Stores counts of nucleotide bases at each position. objects are immutable. se-
    quences may be added to the counts, but the object may not be modified in situ

    Rs()
    returns the Schneider Rs value, which is the expectation of Ri over all possible
    sequences, calculated as the sum of 2-uncertainty.

    count_file(seqsFile, n=0)
    uses a tabFile with a list of sequences, in column n (by default n=0, the first

186
column) and extracts counts

count_seqs(L, debug=False)
    adds a list of sequences to the counts

import_from_MEME(filename, n=1, mode='biotools')
    imports a motif from the output of MEME (meme.txt)
    if there are multiple motifs in the output, we will use motif n (the first is n=1, which is also the default)

make_Ri()
    changes from counts or probabilities to Ri, information content

make_probs(trueProbs=False)
    normalizes everything to 1

rc()
    returns the reverse complement of this object

seq_Ri(s)
    seqRi returns the information content Ri in bits of a sequences, as measured with the given positionmatrix

uncertainty()
    returns the uncertainty H(l) of the matrix as a list. Use sum() for the total uncertainty.
    Note: this function calls uncertainty() from the baseDict instance, and as such it can be overwritten implicitly. baseDict.uncertainty() treats 0*log(0) as 0

bioplus.motif.center_region(f, max_dist=75, motif_length=17)
    returns a function that specifies whether a given motif is in +/- x bp from the peak_center
    requires the tabFile object f to determine the indices properly

bioplus.motif.count_letters(L)

bioplus.motif.count_spacers_from_info(foo, cutoff=None, re-
    gion_rule=None, region_width=None,
    spacer_offset=8, spacer_length=3, output_file=None)
    count spacers from a .sites.info or .peaks.info file
optionally you may supply cutoff, a minimum cutoff (float or int) region_rule, a function that selects the column

bioplus.motif.joint_matrix(sites)  
  takes as input a filename and returns the joint Rate matrix for the list of sequences contained in that file
  Joint rates $R(X; Y)$ are defined as $R(X; Y) = - \sum X, Y p(x, y) * I(X; Y) I(X; y)$  
  $= - \sum X, Y p(x, y) * \log_2[p(x, y)/(p(x)p(y))]$

bioplus.motif.spacerGC(L, spacerOffset=6, spacerLength=3)  
  spacerGC takes as input a list of [15 bp GBSs (strings)] and returns the number of sequences that have 0,1,2,3 G/Cs in the 3 bp spacer as an array in that order

bioplus.peaktools

bioplus.peaktools.MACS_track_to_iter(track)

bioplus.peaktools.array_to_bedgraph(a, chrom, write_zero_values=False, precision=0)  
  generator that takes a numpy array and yields tuples bedGraph file
  note: the array must be compatible with enumerate() chrom should be specified too
  precision specifies how many places past the decimal to retain (default 1)
  To coerce into a string use something like
  for line in array_to_bedgraph(a, chrom): '%s%d%d%.1f' % line

8.6.9 bioplus.seqtools

bioplus.seqtools.analyze_sites(some_list)  
  requires the following input format: seq score cdist cons cdist = distance to center, cons = conservation value

bioplus.seqtools.complement(nucleotide)  
  returns the complement of a single nucleotide
bioplus.seqtools.count_compare(a, b, rc=False)

count_compare takes two default dictionaries (defaultdict objects), a and b, each with default factory "lambda: 0" and key-value pairs that specify the counts (value) of each sequence (key), and returns a new dictionary whose values are tuples (value_a,value_b), the values from a and b.

Reverse complements: if rc=True, count_seqs will count reverse complements as the same sequence and use the alphabetically prior sequence as the key. if rc=False, reverse complements will be treated as different sequences. If you have already ensured that the keys meet this condition, you should use rc=False, but rc=True is also safe.

note: defaultdict is in the collections module of the standard library

bioplus.seqtools.count_seqs(iterable, reverse_complement=True, ignore_case=True)

count_seqs takes as input an iterable that yields sequences, and then returns the counts of each sequence as a defaultdict (similar to dict, can be recast as dict type).

Sorting: Use dictSort to sort, if needed.

Reverse complements: if rc=True, count_seqs will count reverse complements as the same sequence and use the alphabetically prior sequence as the key. if rc=False, reverse complements will be treated as different sequences.

bioplus.seqtools.get_first(seq1, seq2, ignore_case=True)

returns the alphabetically prior of seq1 and seq2 if they are the same or we fail to order, return seq1

if ignore_case=True, we apply str.upper to each letter

bioplus.seqtools.rc(seq)

returns the reverse complement of a sequence

bioplus.seqtools.reversecomplement(seq)

returns the reverse complement of a sequence

8.6.10 bioplus.sitefinder

tools for dealing with binding sites (instances of sequence motifs)

bioplus.sitefinder.MOODS_search(seq, motif, thresholds=0)

an equivalent to Motif.search_pwm()
bioplus.sitefinder.find_sites(peaks_file, fasta_file, motif, bed=True, xls=False, output_dir=None, motif_type='MEME', src_fnc='find_sites', bysummit=False, **kwargs)

findSites(peaks_file, FASTAFile, motif) takes the NAME_peaks.xls file outputed by MACS, as well as a FASTAfile, and finds instances of the motif specified by motif (a Bio.Motif object). It will output two new files for peaks and sites called NAME.peaks.info and NAME.sites.info. It will also create files called NAMES.peaks.bed and NAME.sites.bed which are proper BED files (scores are tag density, and information content, respectively). All files are 0-based, half-open in line with the BED convention. MACS coordinates are corrected accordingly.

f.peaks.info contains Peak (1) chr, (2) start (3) end (4) Peak ID (5) Relative summit (6) Number of unique tags in peak region (7) -10*log10(pvalue) (8) fold_enrichment (9) FDR (10) # motif instances found (11) Total Ri for discovered motif instances (12) Greatest Ri of any motif in peak region (13) Sequence of that motif instance (14) Position (offset) of that motif (left-end)

f.peaks.bed contains Peak (1) chr, (2) start (3) end (4) Peak ID (5) Number of unique tags in peak region (6) Strand . (7) Summit position (absolute) (8) Summit position + 1

f.sites.info contains Site (1) chr (2) start (3) end (4) Unique Site ID (internally generated) (5) The motif information content Ri, in bits (6) motif orientation, best score (+) or (-) —— BED file ends here —— (7) the motif sequence (e.g., ACAACA) (8) Position (offset) of that motif (left-end) (9) peak ID, fetched from MACS (10) used peak length (11) true peak length (11) peak summit offset

bioplus.sitefinder.search_peak(peak_ID, peak, peakseq, motif, bysummit=False)

provide information about matches to a motif in a peak region, and about the region peak MUST provide EITHER (1) the following public methods chrom = reference (e.g. chr1, chrX) chromStart = start coordinate, 0-based chromEnd = end coordinate, open or (2) the following public method coordinates = a tuple containing (chrom, chromStart, chromEnd)

peak may optionally provide the following methods tags (if not found, we will replace with 'NA') summit (if not found, we will use the peak center) misc (a list of anything else)

For each peak, the best motif hit is returned where best is defined as the motif hit with the most information and closest to the center (in the case of ties)

Note site position is 0-based, in contrast with earlier versions of biotools returns a tuple of four things: peak info peak BED row a list of info about sites (motif matches) a list of BED rows for sites
8.6.11 bioplus.tabfile

tools for dealing with files that have tabular data

class bioplus.tabfile.BedFile(f, additional_comments=[], **kwargs)
A BED file is a type of TabFile, but also defines a method for working with rows.
rows are given as instances of BedRow, instead of lists. BEDrows inherit all list
methods and therefore are compatible with write_row. BedRow has additional
methods for chrom, chromStart, chromEnd, etc. For more info, see BedRow
track, browser lines are treated as comments
Assumes track row is a comment. Use getTrackLine to see the track info

DEFAULT_BED_COMMENTS = ['(?i)track', '(?i)browser']

get_track_line()
  returns the current track line, if any

class bioplus.tabfile.BedRow

BEDrows are list, but you can access their chromStart(), chromEnd(),
etc. use help for a full list. Uses the same conventions as
http://genome.ucsc.edu/FAQ/FAQformat#format1. Note that only the first three
entries (chrom, chromStart, chromEnd) are required, so the others may not be
defined.

blockCount()
  return the number of blocks (exons) in the BED line

blockSizes()
  returns a comma-separated list of the block sizes. The number of items in this
  list should correspond to blockCount.

blockStart()
  returns a comma-separated list of block starts. All of the blockStart positions
  should be calculated relative to chromStart. The number of items in this list
  should correspond to blockCount.

chrom()
  returns the name of the chromosome (e.g. chr3, chrY, chr2_random) or
scaffold (e.g. scaffold10671)

chromEnd()

chromStart()

chrom_end()
returns the ending position of the feature in the chromosome or scaffold. The chromEnd base is not included in the display of the feature. For example, the first 100 bases of a chromosome are defined as chromStart=0, chromEnd=100, and span the bases numbered 0-99

chrom_start()
returns the starting position of the feature in the chromosome or scaffold. The first base in a chromosome is numbered 0

itemRgb()
returns itemRgb, An RGB value of the form R,G,B (e.g. 255,0,0). If the track line itemRgb attribute is set to “On”, this RBG value will determine the display color of the data contained in this BED line. NOTE: It is recommended that a simple color scheme (eight colors or less) be used with this attribute to avoid overwhelming the color resources of the Genome Browser and your Internet browser

name()
returns the name of the BED line. This label is displayed to the left of the BED line in the Genome Browser window when the track is open to full display mode or directly to the left of the item in pack mode.

score()
returns the score, a number between 0 and 1000. If the track line useScore attribute is set to 1 for this annotation data set, the score value will determine the level of gray in which this feature is displayed (higher numbers = darker gray)

strand()
returns the strand, either ‘+’ or ‘-’

thickEnd()
returns the ending position at which the feature is drawn thickly (for example, the stop codon in gene displays).
thickStart()
returns the starting position at which the feature is drawn thickly (for example, the start codon in gene displays)

class bioplus.tabfile.Bzip2TabFile(*args, **kwargs)
For bzip2-compressed tab-delimited files
See Tabfile for usage info

exception bioplus.tabfile.DetectCommentsError(*args)

class bioplus.tabfile.GzipTabFile(*args, **kwargs)
For gzip-compressed tab-delimited files
See Tabfile for usage info

class bioplus.tabfile.Macs2Row

FDR()

name()

qvalue()

class bioplus.tabfile.MacsFile(f, convert_spaces=True, **kwargs)
A MACS file is a type of TabFile, but also defines a method for working with rows. rows are given as instances of MACSRow, instead of lists. MACSrows inherit all list methods and therefore are compatible with write_row. MacsRow has additional methods for chrom, chromStart, chromEnd, etc. For more info, see MacsRow

class bioplus.tabfile.MacsRow
MACSrows are list, but you can access their features as follows:

chr() or chrom() – chromosome name start() or chromStart() – start position, start() is 1-based, chromStart is 0-based (BED) end() or chromEnd() – end position, equivalent but chromEnd (BED) is defined as 0-based, exclusive length() – length summit() – position of summit tags() – number of unique tags in the peak region pvalue() – returns the -10*log10(pvalue) fold_enrichment – returns the fold enrichment FDR – returns the FDR in %
FDR(<tt>type_={'str'}<tt>)
returns the FDR (%). preserves the str to eliminate rounding error. use
type=float to get a decimal value.

chr()
returns the name of the chromosome (e.g. chr3, chrY, chr2_random) or
scaffold (e.g. scaffold10671)

chrom()
returns the name of the chromosome (e.g. chr3, chrY, chr2_random) or
scaffold (e.g. scaffold10671)

chromEnd()

chromStart()

chrom_end()
returns the ending position of the feature in the chromosome or scaffold. The
chromEnd base is not included in the display of the feature. For example, the
first 100 bases of a chromosome are defined as chromStart=0, chromEnd=100,
and span the bases numbered 0-99

chrom_start()
returns the starting position of the feature in the chromosome or scaffold. The
first base in a chromosome is numbered 0

end()
returns the end position, 1-based, inclusive

fold_enrichment(<tt>type={'str'}<tt>)
returns the fold_enrichment vs control. preserves the str to eliminate rounding
error. use type=float to get a decimal value.

length()
returns the length

pvalue(<tt>type_={'str'}<tt>)
returns the -10*log10(pvalue). preserves the str to eliminate rounding error.
use type=float to get a decimal value.

start()
returns the starting position of the feature in the chromosome or scaffold. The
first base in a chromosome is numbered 1

summit()
    returns the position of the summit

tags(type_=<type 'int'>)
    returns the number of unique tags in the peak region

tagsv1()

tagsv2()

class bioplus.tabfile.TabFile(filename, mode='r', convert_spaces=True, compression=None, comments=[], column_names=False)

Usage: f = TabFile('filename', convert_spaces=True, comments=[], column_names = False)

TabFile is a class for handling tab-delimited files.
Use convert_spaces=False if you’re file is tab-delimited and you wish to preserve other whitespace.
TabFile suports commented lines. Commented lines are not recognized as part of the table

By default, only lines beginning with '#' will be recognized as comments (not part of the table). You may specify a list of additional keywords using comments=['keyword1','keyword2',etc.]. All lines containing that keyword will be recognized as a comment. keywords may be regular expressions.

['(?i)track','(?i)browser'] if column_names = True, the first properly formatted row will be treated as column names (i.e. ignored as a comment)

close()
    works just like the built-in close method in the file class

column_dict()
    returns a dictionary which gives the index corresponding to a particular column name

comment_line_contents()
    returns the list of lines that are comments

comment_line_numbers()
returns the list of lines that are comments

get_column_names()
  returns the column names

mergesort(f, n, numerical=False)

open(mode=None)
  mode can be overridden here but defaults to TabFile.mode
  acts just like the built-in open method in the file class. use write=True to
  write to a file, otherwise it will be opened in read-only mode

previous_line()
  returns the line number of the last line read

process_table(output_filename, fnc, column_names=None)
  process_tables2 writes a new file (name is specified with new_file), which
  applies a user-defined function fnc to each row of data in the original file. fnc
  should yield a row (i.e. a list, array, or something else finitely iterable).
  process_tables2 preserves all commented lines and also the line which col-
  umn names, if applicable. The user may specify new column names using
  column_names (a list or other finite iterable), or we will use the old col-
  umn_names, which might not preserve the column labels if columns were
  inserted in the middle of the table

read_col(n)
  returns a list of items in column n (numbering starts at 0) as items rather than
  lists. Unlike read_cols and read_table, elements of the read_col list are not
  lists, but strings

read_cols(L)
  read_cols behaves like read_col but instead of taking a single column number
  (numbering starts at 0), it takes a list of column numbers and returns a list of
  partial rows, where each partial row is a list with entries from the appropriate
  columns IN THE ORDER SPECIFIED.
  tip: use range() to create lists of ordered integers. e.g., range(2,6)=[2,3,4,5]

read_first_col()
  return a list of items in the first column

read_last_col()
returns a list of items in the last column

**read_row**()
returns the next (or first) line that is not a comment, parsed. uses `__iter__` as a generator, and simply returns the next value
read_row is deprecated. Use `x = self.__iter__() and x.next()`

**read_table**(override=False)
returns the contents of a file as a list of rows (with each row as a list). will ignore any lines that begin with a “#” symbol and truncate any lines that contain a “#” symbol

**readline**()
reads one line and returns it. uses a generator, and will raise StopIteration if it reaches the EOF.
readline is deprecated. use `x = self.__rawiter__() and x.next()`

**set_column_names**(L)

**write**(s)
writes a string directly to a file, without modification (user must supply n if desired)

**write_column_names**()
writes the stored column names

**write_row**(row, separator='t')
Writes a list to the file as a line (Tab-delimited). A different separator may also be specified with separator='x'. (Note: uses file writelines method)

**write_rows**(iterable)

**write_table**(table, separator='t', override=False, column_names=True)
Writes a table to a file (tab-delimited). An alternative separator may be specified with separator='x'. if column_names = True, column_names will be included as the first line unless they do not exist.

**zap**()
forces status to not open. use with caution. this may destroy data

**exception** bioplus.tabfile.TabFileError(*args)
bioplus.tabfile.merge_files(\texttt{left, right, output, comments='left'})
merge\_files merges the tab-delimited files named \texttt{left} and \texttt{right}, which may have commented lines. the output is directed to the file named \texttt{output}.

There are few modes. If comments='\texttt{left}', comments in \texttt{left} are preserved. If comments='\texttt{right}', comments in \texttt{right} are preserved. If comments='none', no comments are preserved. If comments='all', all comments in \texttt{left} and \texttt{right} are appended to the beginning of output, although they may previously have been contained within the data in \texttt{left} or \texttt{right}.

Use \texttt{merge\_tab\_files} if you need to pass custom parameters to TabFile.

bioplus.tabfile.merge_tab_files(\texttt{file1, file2, output\_filename, comments='left'})
merge\_tab\_files merges the tab-delimited files represented by TabFile objects \texttt{left} and \texttt{right}, which may have commented lines. the output is directed to the file named \texttt{output}.

There are few modes. If comments='\texttt{left}', comments in \texttt{left} are preserved. If comments='\texttt{right}', comments in \texttt{right} are preserved. If comments='none', no comments are preserved. If comments='all', all comments in \texttt{left} and \texttt{right} are appended to the beginning of output, although they may previously have been contained within the data in \texttt{left} or \texttt{right}.

See also \texttt{merge\_files}

bioplus.tabfile.shift_peaks(\texttt{f, peak\_lengths=2})
shift\_peaks takes a file \texttt{f} (foo.bed) and produces a new file (foo\_shifted.bed) with all the sequences shifted (left) by \texttt{peak\_lengths} times their length. If \texttt{peak\_lengths}, is negative they are shifted to the right. comments are stripped

\textbf{8.6.12 bioplus.wrappers}

bioplus.wrappers.random_seq(\texttt{n=1, GC=0.5})
random\_seq provides a random nucleotide (A, T, G, or C). You may optionally provide \texttt{n}, a positive integer, which will cause random\_seq to return a string of \texttt{n} nucleotides. You may also optionally provide \texttt{GC}, the probability of encountering a G or C, which must be on the closed interval \([0,1]\). The probability of encountering an A or T is calculated as \(1 - \text{GC}\).

bioplus.wrappers.random_seq_generator(\texttt{n=1, GC=0.5})
random_seq_generator acts like random_seq, but returns a generator that returns the nucleotides one by one.
Appendix A

twobitreader Source Code

A.1 twobitreader module

```python
###
twobitreader

Licensed under Perl Artistic License 2.0
No warranty is provided, express or implied
###

from array import array
from bisect import bisect_right
from errno import ENOENT, EACCES
from os import R_OK, access
try:
    from os import strerror
except ImportError:
    strerror = lambda x: 'strerror not supported'
from os.path import exists, getsize
from itertools import izip
import logging
import textwrap
import sys

def true_long_type():
    """
    OS X uses an 8-byte long, so make sure L (long) is the right size
    and switch to I (int) if needed
    """
```
for type_ in ['L', 'I']:
    test_array = array(type_, [0])
    long_size = test_array.itemsize
    if long_size == 4:
        return type_
    raise ImportError("Couldn't determine a valid 4-byte long type to use \ as equivalent to LONG")

LONG = true_long_type()

def byte_to_bases(x):
    """convert one byte to the four bases it encodes""
    c = (x >> 4) & 0xf
    f = x & 0xf
    cc = (c >> 2) & 0x3
    cf = c & 0x3
    fc = (f >> 2) & 0x3
    ff = f & 0x3
    return map(bits_to_base, (cc, cf, fc, ff))

def bits_to_base(x):
    """convert integer representation of two bits to correct base""
    if x is 0:
        return 'T'
    elif x is 1:
        return 'C'
    elif x is 2:
        return 'A'
    elif x is 3:
        return 'G'
    else:
        raise ValueError('Only integers 0–3 are valid inputs')

def base_to_bin(x):
    """provided for user convenience convert a nucleotide to its bit representation ""
    if x == 'T':
        return '00'
    elif x == 'C':
        return '01'
    elif x == 'A':
        return '10'
    elif x == 'G':
        return '11'
    else:
        raise ValueError('Only characters \'ATGC\' are valid inputs')
def create_byte_table():
    """ create BYTE_TABLE""
    d = {}
    for x in xrange(2**8):
        d[x] = byte_to_bases(x)
    return d

def split16(x):
    """ split a 16-bit number into integer representation
    of its course and fine parts in binary representation
    ""
    c = (x >> 8) & 0xff
    f = x & 0xff
    return c, f

def create_twobyte_table():
    """ create TWOBYTE_TABLE""
    d = {}
    for x in xrange(2**16):
        c, f = split16(x)
        d[x] = byte_to_bases(c) + byte_to_bases(f)
    return d

BYTE_TABLE = create_byte_table()
TWOBYTE_TABLE = create_twobyte_table()

def longs_to_char_array(longs, first_base_offset, last_base_offset, array_size, more_bytes=None):
    if more_bytes is not None: print array('B', more_bytes)
    """ takes in an array of longs (4 bytes) and converts them to bases in
    a char array
    you must also provide the offset in the first and last block
    (note these offsets are pythonic. last_offset is not included)
    and the desired array_size
    If you have less than a long worth of bases at the end, you can
    provide
    them as a string with more_bytes=
    NOTE: last_base_offset is inside more_bytes not the last long, if
    more_bytes
    is not None
    returns the correct subset of the array based on provided offsets
    """
if array_size == 0:
    return array('c')
elif array_size < 0:
    raise ValueError('array_size must be at least 0')

if not first_base_offset in range(16):
    raise ValueError('first_base_offset must be in range(16)')
if not last_base_offset in range(1, 17):
    raise ValueError('last_base_offset must be in range(1, 17)')

longs_len = len(longs)
if more_bytes is None:
    shorts_length = 0
else:
    shorts_length = len(more_bytes)
if array_size > longs_len * 16 + 4 * shorts_length:
    raise ValueError('array_size exceeds maximum possible for input')

dna = array('c', 'N' * (longs_len * 16 + 4 * shorts_length))
# translate from 32-bit blocks to bytes
# this method ensures correct endianness (byteswap as needed)
bytes_ = array('B')
bytes_.fromstring(longs.tostring())
# first block
first_block = ''.join([''.join(BYTE_TABLE[byte]) for x in range(4)])
i = 16 - first_base_offset
if array_size < i:
    i = array_size
dna[0:i] = array('c', first_block[first_base_offset:first_base_offset + i])
if longs_len > 1:
    # middle blocks (implicitly skipped if they don't exist)
    for byte in bytes_[4:-4]:
        dna[i:i + 4] = array('c', BYTE_TABLE[byte])
        i += 4
    # last block
last_block = array('c', ''.join([''.join(BYTE_TABLE[byte])
                                 for x in range(-4, 0)]))
if more_bytes is None:
    dna[i:i + last_base_offset] = last_block[0:last_base_offset]
else:
    # if there are more bytes, we need the whole last block
    dna[i:i + 16] = last_block[0:16]
i += 16
if more_bytes is not None:
    bytes_ = array('B')
    bytes_.fromstring(more_bytes)
j = i
    for byte in bytes_:
        j = i + 4
if j > array_size:
    dnabytes = array('c', BYTE_TABLE[byte])[0:(array_size -
    i)]
    dna[i:i+array_size] = dnabytes
    break
    dna[i:i+last_base_offset] = array('c', BYTE_TABLE[byte])
    i += 4
return dna[0:array_size]

class TwoBitFile(dict):
    ""
    python-level reader for .2bit files (i.e., from UCSC genome browser)
    (note: no writing support)
    TwoBitFile inherits from dict
    You may access sequences by name, e.g.
    >>> genome = TwoBitFile('hg18.2bit')
    >>> chr20 = genome['chr20']
    Sequences are returned as TwoBitSequence objects
    You may access intervals by slicing or using str() to dump the entire
    entry
e.g.
    >>> chr20[100100:100120]  
    'ttttccttaagataatatttttgccttaaatctatcatttgaatataag
    aagtaagataacttcttttttgggtattttgtcattgaagtttttttccc'
    >>> whole_chr20 = str(chr20)
    Fair warning: dumping the entire chromosome requires a lot of memory
    See TwoBitSequence for more info
    ""
    def __init__(self, foo):
        super(TwoBitFile, self).__init__()
        if not exists(foo):
            raise IOError(ENOENT, strerror(ENOENT), foo)
        if not access(foo, R_OK):
            raise IOError(EACCES, strerror(EACCES), foo)
        self._filename = foo
        self._file_size = getsize(foo)
        self._file_handle = open(foo, 'rb')
        self._load_header()
        self._load_index()
        for name, offset in self._offset_dict.iteritems():
            self[name] = TwoBitSequence(self._file_handle, offset,
                self._file_size,
                self._byteswapped)
        return
def _load_header(self):
    file_handle = self._file_handle
    header = array('LONG')
    header.fromfile(file_handle, 4)
    # check signature --- must be 0x1A412743
    # if not, swap bytes
    byteswapped = False
    (signature, version, sequence_count, reserved) = header
    if not signature == 0x1A412743:
        byteswapped = True
        header.byteswap()
        (signature2, version, sequence_count, reserved) = header
    if not signature2 == 0x1A412743:
        raise TwoBitFileError('Signature in header should be ' +
                               '0x1A412743, instead found 0x%X' % signature)
    if not version == 0:
        raise TwoBitFileError('File version in header should be 0.
                 ')
    if not reserved == 0:
        raise TwoBitFileError('Reserved field in header should be 0.
                 ')
    self._byteswapped = byteswapped
    self._sequence_count = sequence_count

def _load_index(self):
    file_handle = self._file_handle
    byteswapped = self._byteswapped
    remaining = self._sequence_count
    sequence_offsets = []
    file_handle.seek(16)
    while remaining > 0:
        name_size = array('B')
        name_size.fromfile(file_handle, 1)
        if byteswapped:
            name_size.byteswap()
        name = array('c')
        name.fromfile(file_handle, name_size[0])
        if byteswapped:
            name.byteswap()
        offset = array('LONG')
        offset.fromfile(file_handle, 1)
        if byteswapped:
            offset.byteswap()
        sequence_offsets.append((name.tostring(), offset[0]))
        remaining -= 1
    self._sequence_offsets = sequence_offsets
    self._offset_dict = dict(sequence_offsets)

def sequence_sizes(self):
    """returns a dictionary with the sizes of each sequence""
    d = {}
file_handle = self._file_handle
byteswapped = self._byteswapped
for name, offset in self._offset_dict.iteritems():
    file_handle.seek(offset)
    dna_size = array(LONG)
    dna_size.fromfile(file_handle, 1)
    if byteswapped:
        dna_size.byteswap()
    d[name] = dna_size[0]
return d

class TwoBitSequence(object):
    
    A TwoBitSequence object refers to an entry in a TwoBitFile
    
    You may access intervals by slicing or using str() to dump the entire
    entry
    e.g.
    >>> genome = TwoBitFile('hg18.2bit')
    >>> chr20 = genome['chr20']
    >>> chr20[100100:100200] # slicing returns a string
    'ttttccccagagagttttttgcctttaatatgtttgtttttccttaaaatattttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttttt
if byteswapped:
    header.byteswap()
 dna_size, n_block_count = header
 self._dna_size = dna_size  # number of characters, 2 bits each
 self._n_bytes = (dna_size + 3) / 4  # number of bytes
 # number of 32-bit fragments
 self._packed_dna_size = (dna_size + 15) / 16
 n_block_starts = array(LONG)
 n_block_sizes = array(LONG)
 n_block_starts.fromfile(file_handle, n_block_count)
 if byteswapped:
    n_block_starts.byteswap()
 n_block_sizes.fromfile(file_handle, n_block_count)
 if byteswapped:
    n_block_sizes.byteswap()
 self._n_block_starts = n_block_starts
 self._n_block_sizes = n_block_sizes
 mask_rawc = array(LONG)
 mask_rawc.fromfile(file_handle, 1)
 if byteswapped:
    mask_rawc.byteswap()
 mask_block_count = mask_rawc[0]
 mask_block_starts = array(LONG)
 mask_block_starts.fromfile(file_handle, mask_block_count)
 if byteswapped:
    mask_block_starts.byteswap()
 mask_block_sizes = array(LONG)
 mask_block_sizes.fromfile(file_handle, mask_block_count)
 if byteswapped:
    mask_block_sizes.byteswap()
 self._mask_block_starts = mask_block_starts
 self._mask_block_sizes = mask_block_sizes
 file_handle.read(4)
 self._offset = file_handle.tell()

def __len__(self):
    return self._dna_size

def __getslice__(self, min_, max_=None):
    return self.get_slice(min_, max_)

def get_slice(self, min_, max_=None):
    # handle negative coordinates
    dna_size = self._dna_size
    if max_ is not None and max_ < 0:
        if max_ < -dna_size:
            raise IndexError('index out of range')
        max_ = dna_size + 1 + max_
if min_ < 0:
    if max_ < -dna_size:
        raise IndexError('index out of range')
    min_ = dna_size + 1 + min_
    # make sure there's a proper range
    if max_ is not None and min_ > max_:
        return ''
    if max_ == 0 or max_ == min_:
        return ''
# load all the data
if max_ is None or max_ > dna_size:
    max_ = dna_size
file_handle = self._file_handle
byteswapped = self._byteswapped
n_block_starts = self._n_block_starts
n_block_sizes = self._n_block_sizes
mask_block_starts = self._mask_block_starts
mask_block_sizes = self._mask_block_sizes
offset = self._offset
packed_dna_size = self._packed_dna_size
#n_bytes = self._n_bytes

# region_size is how many bases the region is
if max_ is None:
    region_size = dna_size - min_
else:
    region_size = max_ - min_

# start_block, end_block are the first/last 32-bit blocks we need
# blocks start at 0
start_block = min_ / 16
# jump directly to desired file location
local_offset = offset + (start_block * 4)
end_block = (max_ - 1 + 16) / 16
# don't read past seq end

file_handle.seek(local_offset)

# note we won't actually read the last base
# this is a python slice first_base_offset:16*blocks+
    last_base_offset
first_base_offset = min_ % 16
last_base_offset = max_ % 16
if last_base_offset == 0:
    last_base_offset = 16
# +1 we still need to read end_block maybe
blocks_to_read = end_block - start_block
if (blocks_to_read + start_block) > packed_dna_size:
    blocks_to_read = packed_dna_size - start_block
fourbyte_dna = array('L')
# remainder_seq = None
if (blocks_to_read * 4 + local_offset) > self._file_size:
    fourbyte_dna.fromfile(file_handle, blocks_to_read - 1)
morebytes = file_handle.read()  # read the remaining characters
    if byteswapped:
        morebytes = ''.join(reversed(morebytes))
else:
    fourbyte_dna.fromfile(file_handle, blocks_to_read)
morebytes = None
if byteswapped:
    fourbyte_dna.byteswap()
str_as_array = longs_to_char_array(fourbyte_dna,
    first_base_offset,
    last_base_offset, region_size,
    more_bytes=morebytes)
for start, size in izip(n_block_starts, n_block_sizes):
    end = start + size
    if end <= min_
        continue
    if start > max_
        break
    if start < min_
        start = min_
    if end > max_
        end = max_
    start -= min_
    end -= min_
    # this should actually be decoded, 00=N, 01=n
    str_as_array[start:end] = array('c', 'N' * (end-start))
lower = str.lower
first_masked_region = max(0,
    bisect_right(mask_block_starts, min_) - 1)
last_masked_region = min(len(mask_block_starts),
    1 + bisect_right(mask_block_starts,
    max_,
    lo=first_masked_region))
for start, size in izip(mask_block_starts[first_masked_region:
    last_masked_region],
    mask_block_sizes[first_masked_region:
    last_masked_region]):
    end = start + size
    if end <= min_
        continue
    if start > max_
        break
    if start < min_:
    start = min_
    if end > max_:
        end = max_
    start = min_
    end = min_
    str_as_array[start:end] = array('c',
        lower(str_as_array[start:end]).
        tostring())
    if not len(str_as_array) == max_ - min_:
        raise RuntimeError("Sequence was the wrong size")
    return str_as_array.tostring()

def __str__(self):
    ""
    returns the entire chromosome
    ""
    return self.__getslice__(0, None)

class TwoBitFileError(StandardError):
    ""
    Base exception for TwoBit module
    ""
    def __init__(self, msg):
        errtext = 'Invalid 2-bit file.' + msg
        return super(TwoBitFileError, self).__init__(errtext)

def print_specification():
    ""
    Prints the twoBit file format specification I got from the Internet.
    This is only here for reference
    ""
    return """From http://www.its.caltech.edu/~alok/reviews/blatSpecs.html .2 bit files
    A .2 bit file can store multiple DNA sequence (up to 4 gig total) in a
    compact randomly accessible format. The two bit files contain masking
    information as well as the DNA itself. The file begins with a 16 byte header containing
    the following fields:
    signature – the number 0x1A412743 in the architecture of the machine
    that created the file."""
version — zero for now. Readers should abort if they see a version
number higher than 0.

sequenceCount — the number of sequences in the file
reserved — always zero for now.
All fields are 32 bits unless noted. If the signature value is not as
given, the reader program should byte swap the signature and see if the swapped
version matches. If so all multiple-byte entities in the file will need
to be byte-swapped. This enables these binary files to be used unchanged on different architectures.

The header is followed by a file index. There is one entry in the index for each sequence. Each index entry contains three fields:

nameSize — a byte containing the length of the name field
name — this contains the sequence name itself, and is variable length depending on nameSize.
offset — 32 bit offset of the sequence data relative to the start of the file

The index is followed by the sequence records. These contain 9 fields:

dnaSize — number of bases of DNA in the sequence.
nBlockCount — the number of blocks of N’s in the file (representing unknown sequence).
nBlockStarts — a starting position for each block of N’s
nBlockSizes — the size of each block of N’s
maskBlockCount — the number of masked (lower case) blocks
maskBlockStarts — starting position for each masked block
maskBlockSizes — the size of each masked block
packedDna — the dna packed to two bits per base as so: 00 — T, 01 — C, 10 — A, 11 — G. The first base is in the most significant 2 bits byte, and the last base in the least significant 2 bits, so that the sequence TCAG would be represented as 00011011. The packedDna field will be padded with 0 bits as necessary so that it takes an even multiple of 32 bit in the file, as this improves i/o performance on some machines.

***

211


def cmdline_reader():
    """
    cmdline_reader allows twobitreader module to be executed as a script
    accepts only one argument — the .2 bit filename
    reads input (BED format) from stdin
    writes output (FASTA format) to stdout
    writes errors/warning to stderr

    Regions should be given in BED format on stdin
    chrom  start(0−based)   end(0−based, not included)

    To use a BED file of regions, do
    python −m twobitreader example.2 bit < example.bed

    Non−regions will be skipped and warnings will be issued to logging
    (logging output to stderr by default)
    """
    # if no argument provided, print docstring
    argv = sys.argv
    if len(argv) == 1:
        print argv[0] + "::
        print cmdline_reader.__doc__
        sys.exit()
    return
    # if user is trying to get help, print docstring
    elif len(argv) == 2 and argv[1] in ['−−help', '−h', '−help']:
        print argv[0] + "::
        print cmdline_reader.__doc__
        sys.exit()
    return
    # if user specified multiple files, exit with error
    elif len(argv) > 2:
        sys.exit("Too many files specified")
    return
    # otherwise proceed with opening the .2 bit file
    twobit_file = TwoBitFile(argv[1])
    # print error/warning messages as we go
    twobit_reader(twobit_file, input_stream=sys.stdin)

    def twobit_reader(twobit_file, input_stream=None, write=None):
        """
        twobit_reader takes a twobit_file (of class TwoBitFile)
        and an "input_stream" which can be any iterable (incl. file−like
        objects)
        writes output (FASTA format) using write (print if write=None)
        logs errors/warning to stderr

        Regions should be given in BED format on stdin
        chrom  start(0−based)   end(0−based, not included)
To use a BED file of regions, do

```python
python -m twobitreader example.2bit < example.bed
```

Non-regions will be skipped and warnings will be issued to logging
(logging output to stderr by default)

```python
warning_msg = 'Invalid %s at line %d\n\t"%s"
if input_stream is None:
    return
for i, line in enumerate((line.rstrip(\'\n\r\') for line in
    input_stream)):
    fields = line.split()
    if not len(fields) >= 3:
        logging.warn(warning_msg, 'start', i, line)
        continue
    chrom = fields[0]
    if not chrom in twobit_file:
        logging.warn(warning_msg, 'chrom', i, line)
        continue
    try:
        start = long(fields[1])
    except ValueError:
        logging.warn(warning_msg, 'start', i, line)
        continue
    except ValueError:
        logging.warn(warning_msg, 'start', i, line)
        continue
    try:
        end = long(fields[2])
    except ValueError:
        logging.warn(warning_msg, 'end', i, line)
        continue
    chrom_len = len(twobit_file[chrom])
    if end > len(twobit_file[chrom]):
        logging.warn('At line %d, end is greater than chrom length %d\n\n%s', i, chrom_len, line)
        logging.warn('Sequence will be truncated at chrom' +
            'length for line %d', i)
        end = chrom_len
    seq = twobit_file[chrom][start:end]
    if write is not None:
        write('>%s:%d-%d
' % (chrom, start, end))
        write(textwrap.fill(seq, 60))
    else:
        print '>%s:%d-%d' % ( chrom, start, end)
        print textwrap.fill(seq, 60)
return
```

if __name__ == '__main__':
    cmdline_reader()
Source Code A.1: twobitreader module

A.2 twobitreader.download module

```python
###
downloads a .2bit genome from UCSC
.. note:: please comply with the restrictions of use at
http://hgdownload.cse.ucsc.edu/downloads.html

and do not over-use this module

By default, genomes are saved to the current directory
###
from urllib2 import urlopen
from shutil import copyfileobj
from os.path import exists, join
from os import getcwd

def save_genome(name, destdir=None, mode='ftp'):
    
    '''
    tries to download a genome from UCSC by name
    
    for example, 'hg19' is at
    ftp://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/hg19.2bit
    '''
    urlpath = "%s://hgdownload.cse.ucsc.edu/goldenPath/%s/bigZips/%s.2bit" %
        (mode, name, name)
    if destdir is None:
        destdir = getcwd()
    remotefile = urlopen(urlpath)
    assert exists(destdir), 'Desination directory %s does not exist' %
        destdir
    with open(join(destdir, "%s.2bit" % name), 'wb') as destfile:
        copyfileobj(remotefile, destfile)
    return

def main():
    import sys
    if len(sys.argv) != 2:
        sys.exit('Example: python -m twobitreader.download hg19')
    else:
        save_genome(sys.argv[1])
```
A.3 twobitreader Unit Tests

```python
# Some of this test code is specific to 2.7
import unittest
twobitreader
os

class HasLongTypeTestCase(unittest.TestCase):
    def test_has_long_type(self):
        self.assertIn(twobitreader.true_long_type(), ['L', 'I'])

class ByteTableTestCase(unittest.TestCase):
    def setUp(self):
        self.byte_table = twobitreader.create_byte_table()
    
    def test_module_table(self):
        self.assertTrue(type(twobitreader.BYTE_TABLE) is dict)
        self.assertEqual(len(twobitreader.BYTE_TABLE), 256)
        self.assertEqual(twobitreader.BYTE_TABLE, self.byte_table)
    
    def test_has_all_keys(self):
        bt = self.byte_table
        for i in range(256):
            self.assertTrue(i in bt)

    def test_polyNs(self):
        bt = self.byte_table
        self.assertEqual(bt[0], ['T', 'T', 'T', 'T'])
        self.assertEqual(bt[85], ['C', 'C', 'C', 'C'])
        self.assertEqual(bt[170], ['A', 'A', 'A', 'A'])
        self.assertEqual(bt[255], ['G', 'G', 'G', 'G'])

class SimpleBytesTestCase(unittest.TestCase):
    def test_bits_to_base(self):
        self.assertEqual(twobitreader.bits_to_base(0), 'T')
        self.assertEqual(twobitreader.bits_to_base(1), 'C')
        self.assertEqual(twobitreader.bits_to_base(2), 'A')
```
self.assertEqual(twobitreader.bits_to_base(3), 'G')
self.assertRaises(ValueError, twobitreader.bits_to_base, 4)

def test_base_to_bin(self):
    self.assertEqual(twobitreader.base_to_bin('T'), '00')
    self.assertEqual(twobitreader.base_to_bin('C'), '01')
    self.assertEqual(twobitreader.base_to_bin('A'), '10')
    self.assertEqual(twobitreader.base_to_bin('G'), '11')

class SimpleLongsToCharTest(unittest.TestCase):
def setUp(self):
    from array import array
    self.longs_array = array(twobitreader.LONG, [683102738,
                                                3396552641,
                                                3797081033,
                                                1243780212])
    self.as_string = \'TCTACCTAAGCGTAATGTTCCTCGCGTGGTAAGTACGCAGCCTAGATACGCTACCTTATACTAA\'
    self.chars_array = array(\'c\', \'TCTACCTAAGCGTAATGTTCCTCGCGTGGTAAGTACGCAGCCTAGATACGCTACCTTATACTAA\')

def test_longs_to_char(self):
    self.assertEqual(twobitreader.longs_to_char_array(self.longs_array, 0, 16, 64),
                     self.chars_array)

def test_longs_to_string(self):
    as_string = twobitreader.longs_to_char_array(self.longs_array, 0, 16, 64).tostring()
    self.assertEqual(as_string, self.as_string)

def test_string_length(self):
    for length in range(65):
        char_array = twobitreader.longs_to_char_array(self.longs_array, 0, 16, length)
        self.assertEqual(len(char_array), length,
                         'Longs to character array conversion failed at length %d' % length)

def test_first_base_with_offsets(self):
    for offset in range(16):
        first_base = twobitreader.longs_to_char_array(self.longs_array,
                                                    offset, 16, 1)
        self.assertEqual(first_base, self.chars_array[offset])
def test_first_base_with_offsets(self):
    for offset in range(16):
        first_base = twobitreader.longs_to_char_array(self.longs_array, 
                                                    offset, 16, 1)[0]
        self.assertEqual(first_base, self.chars_array[offset], 
                         "Failed at offset %d" % offset)
        self.assertEqual(first_base, self.as_string[offset], 
                         "Failed at offset %d" % offset)

def test_last_base_with_offsets(self):
    for offset in reversed(range(1, 17)):
        last_base = twobitreader.longs_to_char_array(self.longs_array, 
                                                   0, offset, 64 - (16 - offset))[-1]
        self.assertEqual(last_base, self.chars_array[-1 + (offset - 16)], 
                         "Failed at offset %d" % offset)
        self.assertEqual(last_base, self.as_string[-1 + (offset - 16)], 
                         "Failed at offset %d" % offset)

class BadTwoBitFileTest(unittest.TestCase):
    def setUp(self):
        import tempfile
        self.t = tempfile.mkstemp()
        self.file_handle = os.fdopen(self.t[0], 'w')

    def tearDown(self):
        self.file_handle.close()
        os.remove(self.t[1])

    def test_open_bad_file(self):
        self.file_handle.write('Writing some garbage to a file\n' * 10)
        self.file_handle.flush()
        self.file_handle.close()
        self.assertRaises(twobitreader.TwoBitFileError, 
                          twobitreader.TwoBitFile, 
                          self.t[1])

    def test_open_empty_file(self):
        self.assertRaises(EOFError, 
                          twobitreader.TwoBitFile, 
                          self.t[1])

    def test_open_not_a_file(self):
        self.assertRaises(IOError, 
                          twobitreader.TwoBitFile, 
                          self.t[1])
```python
class CheckTestTwoBitFileTest(unittest.TestCase):

def setUp(self):
    # not sure how to get the path more robustly
    this_dir = os.path.join(os.path.dirname(__file__))
    self.filename = os.path.join(this_dir, 'test.2bit')

def test_testfile_can_be_opened(self):
    with open(self.filename) as f:
        pass

def test_open_two_bit_file_and_delete_object(self):
    t = twobitreader.TwoBitFile(self.filename)
    self.assertIsInstance(t, twobitreader.TwoBitFile)
    del t

def test_two_bit_file_has_chrs(self):
    '''make sure file has chr1 - chr10'''
    t = twobitreader.TwoBitFile(self.filename)
    self.assertEqual(set(t.keys()),
                     set(['chr%d' % x for x in range(1,11)]))

def test_two_bit_file_has_sequences(self):
    '''make sure file has chr1 - chr10'''
    t = twobitreader.TwoBitFile(self.filename)
    for sequence in t.values():
        self.assertIsInstance(sequence, twobitreader.TwoBitSequence)

def test_two_bit_sequence_lengths(self):
    t = twobitreader.TwoBitFile(self.filename)
    self.assertEqual(t.sequence_sizes()['chr1'], 75)
    for i in range(2, 11):
      self.assertEqual(t.sequence_sizes()['chr%d' % i], 50)

def test_two_bit_chr1_sequence(self):
    t = twobitreader.TwoBitFile(self.filename)
    chr1 = str(t['chr1'])
    self.assertEqual(chr1,
                     'GAACATGTACAAACCTGACCTCCACgaacatgtacaacctg\accttccacNNNNATGTACAAACCTGACCTCCAC')

def test_two_bit_chr10_sequence(self):
    t = twobitreader.TwoBitFile(self.filename)
    chr10 = str(t['chr10'])
    self.assertEqual(chr10,
                     'gaaagggaactccctgacccttgtgaaggggaactccctgacccttgt')
```

Source Code A.3: twobitreader Unit Tests
Appendix B

scripter Source Code

B.1 scripter module

```python
###
twobitreader

Licensed under Perl Artistic License 2.0
No warranty is provided, express or implied
###
from array import array
from bisect import bisect_right
from errno import ENOENT, EACCESS
from os import R_OK, access
try:
    from os import strerror
except ImportError:
    strerror = lambda x: 'strerror not supported'
from os.path import exists, getsize
from itertools import izip
import logging
import textwrap
import sys

def true_long_type():
    ###
    OS X uses an 8-byte long, so make sure L (long) is the right size
    and switch to I (int) if needed
    ###
```
for type_ in ['L', 'l']:
    test_array = array(type_, [0])
    long_size = test_array.itemsize
    if long_size == 4:
        return type_
    raise ImportError("Couldn't determine a valid 4-byte long type to use \n as equivalent to LONG")
LONG = true_long_type()

def byte_to_bases(x):
    """convert one byte to the four bases it encodes""
    c = (x >> 4) & 0xf
    f = x & 0xf
    cc = (c >> 2) & 0x3
    cf = c & 0x3
    fc = (f >> 2) & 0x3
    ff = f & 0x3
    return map(bits_to_base, (cc, cf, fc, ff))

def bits_to_base(x):
    """convert integer representation of two bits to correct base""
    if x is 0:
        return 'T'
    elif x is 1:
        return 'C'
    elif x is 2:
        return 'A'
    elif x is 3:
        return 'G'
    else:
        raise ValueError("Only integers 0–3 are valid inputs")

def base_to_bin(x):
    """provided for user convenience
    convert a nucleotide to its bit representation""
    if x == 'T':
        return '00'
    elif x == 'C':
        return '01'
    elif x == 'A':
        return '10'
    elif x == 'G':
        return '11'
    else:
        raise ValueError("Only characters 'ATGC' are valid inputs")
```python
def create_byte_table():
    """create BYTE_TABLE""
    d = {}
    for x in xrange(2**8):
        d[x] = byte_to_bases(x)
    return d

def split16(x):
    """split a 16-bit number into integer representation of its course and fine parts in binary representation""
    c = (x >> 8) & 0xff
    f = x & 0xff
    return c, f

def create_twobyte_table():
    """create TWobyte TABLE""
    d = {}
    for x in xrange(2**16):
        c, f = split16(x)
        d[x] = byte_to_bases(c) + byte_to_bases(f)
    return d

BYTE_TABLE = create_byte_table()
TWOBYTE_TABLE = create_twobyte_table()

def longs_to_char_array(longs, first_base_offset, last_base_offset, array_size, more_bytes=None):
    if more_bytes is not None: print(array('B', more_bytes)
    """takes in an array of longs (4 bytes) and converts them to bases in a char array
    you must also provide the offset in the first and last block
    (note these offsets are pythonic. last_offset is not included)
    and the desired array size
    If you have less than a long worth of bases at the end, you can provide
    them as a string with more_bytes=
    NOTE: last_base_offset is inside more_bytes not the last long, if
    more_bytes is not None
    returns the correct subset of the array based on provided offsets""
```

```
if array_size == 0:
    return array('c')
elif array_size < 0:
    raise ValueError('array_size must be at least 0')

if not first_base_offset in range(16):
    raise ValueError('first_base_offset must be in range(16)')
if not last_base_offset in range(1, 17):
    raise ValueError('last_base_offset must be in range(1, 17)')

longs_len = len(longs)
if more_bytes is None:
    shorts_length = 0
else:
    shorts_length = len(more_bytes)
if array_size > longs_len * 16 + 4 * shorts_length:
    raise ValueError('array_size exceeds maximum possible for input')

dna = array('c', 'N' * (longs_len * 16 + 4 * shorts_length))
# translate from 32-bit blocks to bytes
# this method ensures correct endianess (byteswap as needed)
bytes_ = array('B')
bytes_.fromstring(longs.tostring())
# first block
first_block = ''.join([''.join(BYTE_TABLE[byte]) for x in range(4)])
i = 16 - first_base_offset
if array_size < i:
    i = array_size
dna[0:i] = array('c', first_block[first_base_offset:first_base_offset + i])

if longs_len > 1:
    # middle blocks (implicitly skipped if they don't exist)
    for byte in bytes_[:-4):
        dna[i:i + 4] = array('c', BYTE_TABLE[byte])
i += 4
# last block
last_block = array('c', ''.join([''.join(BYTE_TABLE[byte])
                                for x in range(-4, 0)]))
if more_bytes is None:
    dna[i:i + last_base_offset] = last_block[0:last_base_offset]
else:
    # if there are more bytes, we need the whole last block
    dna[i:i + 16] = last_block[0:16]
i += 16
if more_bytes is not None:
    bytes_ = array('B')
    bytes_.fromstring(more_bytes)
j = i
    for byte in bytes_:
        j = i + 4
if j > array_size:
    dnabytes = array(‘c’, BYTE_TABLE[byte])[0:(array_size - i)]
    dna[i:array_size] = dnabytes
    break
    dna[i:i + last_base_offset] = array(‘c’, BYTE_TABLE[byte])
    i += 4
    return dna[0:array_size]

class TwoBitFile(dict):
    ""
    python-level reader for .2bit files (i.e., from UCSC genome browser)
    (note: no writing support)
    TwoBitFile inherits from dict
    You may access sequences by name, e.g.
    >>> genome = TwoBitFile(‘hg18.2bit’)
    >>> chr20 = genome[‘chr20’]
    Sequences are returned as TwoBitSequence objects
    You may access intervals by slicing or using str() to dump the entire
    entry
    e.g.
    >>> chr20[100100:100120]
    ‘ttttcttcctaagataatttttgccttaaatatatgttttcaataactaag
    aagtaagataactctcttttgggtattttgcatgttaagtttttttttttcc’
    >>> whole_chr20 = str(chr20)
    Fair warning: dumping the entire chromosome requires a lot of memory
    See TwoBitSequence for more info
    ""
    def __init__(self, foo):
        super(TwoBitFile, self).__init__()
        if not exists(foo):
            raise IOError(ENOENT, strerror(ENOENT), foo)
        if not access(foo, R_OK):
            raise IOError(EACCES, strerror(EACCES), foo)
        self._filename = foo
        self._file_size = getsize(foo)
        self._file_handle = open(foo, ‘rb’)
        self._load_header()
        self._load_index()
        for name, offset in self._offset_dict.iteritems():
            self[name] = TwoBitSequence(self._file_handle, offset,
                                          self._file_size,
                                          self._byteswapped)
        return
def _load_header(self):
    file_handle = self._file_handle
    header = array(LONG)
    header.fromfile(file_handle, 4)
    # check signature — must be 0x1A412743
    # if not, swap bytes
    byteswapped = False
    (signature, version, sequence_count, reserved) = header
    if not signature == 0x1A412743:
        byteswapped = True
        header.byteswap()
        (signature2, version, sequence_count, reserved) = header
        if not signature2 == 0x1A412743:
            raise TwoBitFileError('Signature in header should be ' +
                                   '0x1A412743, instead found 0x%X' % signature)

    if not version == 0:
        raise TwoBitFileError('File version in header should be 0. ')
    if not reserved == 0:
        raise TwoBitFileError('Reserved field in header should be 0. ')

    self._byteswapped = byteswapped
    self._sequence_count = sequence_count

def _load_index(self):
    file_handle = self._file_handle
    byteswapped = self._byteswapped
    remaining = self._sequence_count
    sequence_offsets = []
    file_handle.seek(16)
    while remaining > 0:
        name_size = array('B')
        name_size.fromfile(file_handle, 1)
        if byteswapped:
            name_size.byteswap()
        name = array('c')
        name.fromfile(file_handle, name_size[0])
        if byteswapped:
            name.byteswap()
        offset = array(LONG)
        offset.fromfile(file_handle, 1)
        if byteswapped:
            offset.byteswap()
        sequence_offsets.append((name.tostring(), offset[0]))
        remaining -= 1
    self._sequence_offsets = sequence_offsets
    self._offset_dict = dict(sequence_offsets)

def sequence_sizes(self):
    """returns a dictionary with the sizes of each sequence"
    d = {}
272    file_handle = self._file_handle
273    byteswapped = self._byteswapped
274    for name, offset in self._offset_dict.items():
275        file_handle.seek(offset)
276        dna_size = array('LONG')
277        dna_size.fromfile(file_handle, 1)
278        if byteswapped:
279            dna_size.byteswap()
280        d[name] = dna_size[0]
281    return d

282
283class TwoBitSequence(object):
    
    A TwoBitSequence object refers to an entry in a TwoBitFile
    You may access intervals by slicing or using str() to dump the entire
    entry
    e.g.
    >>> genome = TwoBitFile('hg18.2bit')
    >>> chr20 = genome['chr20']
    >>> chr20[100100:100200] # slicing returns a string
    'ttttccttaagataatttttgctttaatactatattttgttcaataactaaga
    agtaagataacttccctttggttatatttcgcatgtaagttttttttcc'
    >>> whole_chr20 = str(chr20) # get whole chr as string
    
    Fair warning: dumping the entire chromosome requires a lot of memory
    Note that we follow python/UCSC conventions:
    Coordinates are 0-based, end-open
    (Note: The UCSC web-based genome browser uses 1-based closed coordinates)
    If you attempt to access a slice past the end of the sequence,
    it will be truncated at the end.
    Your computer probably doesn't have enough memory to load a whole genome
    but if you want to string-ize your TwoBitFile, here's a recipe:
    x = TwoBitFile('my.2bit')
    d = x.dict()
    for k,v in d.items(): d[k] = str(v)

    def __init__(self, file_handle, offset, file_size, byteswapped=False):
        self._file_size = file_size
        self._file_handle = file_handle
        self._original_offset = offset
        self._byteswapped = byteswapped
        file_handle.seek(offset)
        header = array('LONG')
        header.fromfile(file_handle, 2)
if byteswapped:
    header.byteswap()

DNA Size, n_block_count = header
self._DNA_size = DNA_size # Number of characters, 2 bits each
self._n_bytes = (DNA_size + 3) / 4 # Number of bytes
# Number of 32-bit fragments
self._packed_dna_size = (DNA_size + 15) / 16
n_block_starts = array(LONG)
n_block_sizes = array(LONG)
n_block_starts.fromfile(file_handle, n_block_count)
if byteswapped:
    n_block_starts.byteswap()

n_block_sizes.fromfile(file_handle, n_block_count)
if byteswapped:
    n_block_sizes.byteswap()

self._n_block_starts = n_block_starts
self._n_block_sizes = n_block_sizes
mask_rawc = array(LONG)
mask_rawc.fromfile(file_handle, 1)
if byteswapped:
    mask_rawc.byteswap()

mask_block_count = mask_rawc[0]
mask_block_starts = array(LONG)
mask_block_starts.fromfile(file_handle, mask_block_count)
if byteswapped:
    mask_block_starts.byteswap()

mask_block_sizes = array(LONG)
mask_block_sizes.fromfile(file_handle, mask_block_count)
if byteswapped:
    mask_block_sizes.byteswap()

self._mask_block_starts = mask_block_starts
self._mask_block_sizes = mask_block_sizes
file_handle.read(4)
self._offset = file_handle.tell()

def __len__(self):
    return self._DNA_size

def __getslice__(self, min_, max_=None):
    return self.get_slice(min_, max_)

def get_slice(self, min_, max_=None):
    """Get slice returns only a sub-sequence"""
    # Handle negative coordinates
    DNA_size = self._DNA_size
    if max_ is not None and max_ < 0:
        if max_ < -DNA_size:
            raise IndexError('Index out of range')
        max_ = DNA_size + 1 + max_
if min_ < 0:
    if max_ < -dna_size:
        raise IndexError('index out of range')
    min_ = dna_size + 1 + min_
# make sure there's a proper range
if max_ is not None and min_ > max_:
    return ''
if max_ == 0 or max_ == min_:
    return ''
# load all the data
if max_ is None or max_ > dna_size:
    max_ = dna_size
file_handle = self._file_handle
byteswapped = self._byteswapped
n_block_starts = self._n_block_starts
n_block_sizes = self._n_block_sizes
mask_block_starts = self._mask_block_starts
mask_block_sizes = self._mask_block_sizes
offset = self._offset
packed_dna_size = self._packed_dna_size
# n_bytes = self._n_bytes

# region_size is how many bases the region is
if max_ is None:
    region_size = dna_size - min_
else:
    region_size = max_ - min_

# start_block, end_block are the first/last 32-bit blocks we need
# blocks start at 0
start_block = min_ / 16
# jump directly to desired file location
local_offset = offset + (start_block * 4)
end_block = (max_ - 1 + 16) / 16
# don't read past seq end
file_handle.seek(local_offset)

# note we won't actually read the last base
# this is a python slice first_base_offset:16*blocks+
# last_base_offset
first_base_offset = min_ % 16
last_base_offset = max_ % 16
if last_base_offset == 0:
    last_base_offset = 16
# +1 we still need to read end_block maybe
blocks_to_read = end_block - start_block
if (blocks_to_read + start_block) > packed_dna_size:
    blocks_to_read = packed_dna_size - start_block
fourbyte_dna = array(LONG)
# remainder_seq = None
if (blocks_to_read * 4 + local_offset) > self._file_size:
    fourbyte_dna.fromfile(file_handle, blocks_to_read - 1)
    morebytes = file_handle.read()  # read the remaining characters
    if byteswapped:
        morebytes = ''.join(reversed(morebytes))
else:
    fourbyte_dna.fromfile(file_handle, blocks_to_read)
morebytes = None
if byteswapped:
    fourbyte_dna.byteswap()
str_as_array = longs_to_char_array(fourbyte_dna,
    first_base_offset,
    last_base_offset, region_size,
    more_bytes=morebytes)
for start, size in izip(n_block_starts, n_block_sizes):
    end = start + size
    if end <= min_:
        continue
    if start > max_:
        break
    if start < min_:
        start = min_
    if end > max_:
        end = max_
    start -= min_
    end -= min_
    # this should actually be decoded; 00=N, 01=n
    str_as_array[start:end] = array('c', 'N'*(end-start))
lower = str.lower
first_masked_region = max(0,
    bisect_right(mask_block_starts, min_ - 1))
last_masked_region = min(len(mask_block_starts),
    1 + bisect_right(mask_block_starts, max_,
    lo=first_masked_region
))
for start, size in izip(mask_block_starts[first_masked_region:
    last_masked_region],
    mask_block_sizes[first_masked_region:
    last_masked_region]):
    end = start + size
    if end <= min_:
        continue
    if start > max_:
        break
    if start < min_:
```python
start = min_
if end > max_
    end = max_
start = -- min_
end = -- min_
str_as_array[start:end] = array('c',
    lower(str_as_array[start:end]
    toString())
if not len(str_as_array) == max_ - min_
    raise RuntimeError("Sequence was the wrong size")
return str_as_array.toString()

def __str__(self):
    """returns the entire chromosome"""
    return self.__getslice__(0, None)

class TwoBitFileError(StandardError):
    """Base exception for TwoBit module"""
def __init__(self, msg):
    errtext = 'Invalid 2-bit file.' + msg
    return super(TwoBitFileError, self).__init__(errtext)

def print_specification():
    """Prints the twoBit file format specification I got from the Internet.
    This is only here for reference"""
    return "From http://www.its.caltech.edu/~alok/reviews/blatSpecs.html"

    .2 bit files

    A .2 bit file can store multiple DNA sequence (up to 4 gig total) in a
    compact \n    randomly accessible format. The two bit files contain masking
    information as \n    well as the DNA itself. The file begins with a 16 byte header containing
    the \n    following fields:

    signature – the number 0x1A412743 in the architecture of the machine
    that \n    created the file.
```
version — zero for now. Readers should abort if they see a version number \ higher than 0.
sequenceCount — the number of sequences in the file reserved — always zero for now.
All fields are 32 bits unless noted. If the signature value is not as given, \ the reader program should byte swap the signature and see if the swapped \ version matches. If so all multiple-byte entities in the file will need to be \ byte-swapped. This enables these binary files to be used unchanged on \ different architectures.

The header is followed by a file index. There is one entry in the index for \ each sequence. Each index entry contains three fields:

nameSize — a byte containing the length of the name field name — this contains the sequence name itself, and is variable length depending on nameSize.
offset — 32 bit offset of the sequence data relative to the start of the file

The index is followed by the sequence records. These contain 9 fields:
dnaSize — number of bases of DNA in the sequence.
nBlockCount — the number of blocks of N’s in the file (representing unknown sequence).
nBlockStarts — a starting position for each block of N’s nBlockSizes — the size of each block of N’s maskBlockCount — the number of masked (lower case) blocks maskBlockStarts — starting position for each masked block maskBlockSizes — the size of each masked block packedDna — the dna packed to two bits per base as so: 00 — T, 01 — C, 10 — A, 11 — G. The first base is in the most significant 2 bits byte, and the last base in the least significant 2 bits, so that the sequence TCAG would be represented as 00011011. The packedDna field will be padded with 0 bits as necessary so that it takes an even multiple of 32 bit in the file, as this \ improves i/o performance on some machines.

***
def cmdline_reader():
    """
    cmdline_reader allows twobitreader module to be executed as a script
    accepts only one argument — the .2 bit filename
    reads input (BED format) from stdin
    writes output (FASTA format) to stdout
    writes errors/warning to stderr
    
    Regions should be given in BED format on stdin
    chrom start(0-based) end(0-based, not included)
    To use a BED file of regions, do
    python -m twobitreader example.2 bit < example.bed
    Non-regions will be skipped and warnings will be issued to logging
    (logging output to stderr by default)
    """
    # if no argument provided, print docstring
    argv = sys.argv
    if len(argv) == 1:
        print argv[0] + "::"
        print cmdline_reader.__doc__
        sys.exit()
    # if user is trying to get help, print docstring
    elif len(argv) == 2 and argv[1] in ['--help', '-h', 'help']:
        print argv[0] + "::"
        print cmdline_reader.__doc__
        sys.exit()
    # if user specified multiple files, exit with error
    elif len(argv) > 2:
        sys.exit("Too many files specified")
        return
    # otherwise proceed with opening the .2bit file
    twobit_file = TwoBitFile(argv[1])
    # print error/warning messages as we go
    twobit_reader(twobit_file, input_stream=sys.stdin)

def twobit_reader(twobit_file, input_stream=None, write=None):
    """
    twobit_reader takes a twobit_file (of class TwoBitFile)
    and an "input_stream" which can be any iterable (incl. file-like
    objects)
    writes output (FASTA format) using write (print if write=None)
    logs errors/warning to stderr
    
    Regions should be given in BED format on stdin
    chrom start(0-based) end(0-based, not included)
To use a BED file of regions, do

d python -m twobitreader example.2bit < example.bed

Non-regions will be skipped and warnings will be issued to logging
(logging output to stderr by default)

```
warning_msg = 'Invalid %s at line %d\n\t"%s"
if input_stream is None:
    return
for i, line in enumerate((line.rstrip('\n\r') for line in input_stream)):
    fields = line.split()
    if not len(fields) >= 3:
        logging.warn(warning_msg, 'start', i, line)
    continue
    chrom = fields[0]
    if not chrom in twobit_file:
        logging.warn(warning_msg, 'chrom', i, line)
    continue
    try:
        start = long(fields[1])
    except ValueError:
        logging.warn(warning_msg, 'start', i, line)
    if start < 0:
        logging.warn(warning_msg, 'start', i, line)
        logging.warn('Using 0 as start instead for line %d', i)
    start = 0
    try:
        end = long(fields[2])
    except ValueError:
        logging.warn(warning_msg, 'end', i, line)
    if end > len(twobit_file[chrom]):
        logging.warn('At line %d, end is greater than chrom length %d\n\n%s',
                      i, chrom_len, line)
        logging.warn('Sequence will be truncated at chrom' +
                      'length for line %d', i)
        end = chrom_len
    seq = twobit_file[chrom][start:end]
    if write is not None:
        write(">%s.%d-%d" % (chrom, start, end))
        write(textwrap.fill(seq, 60))
    else:
        print "%s.%d-%d" % (chrom, start, end)
        print textwrap.fill(seq, 60)
    return
```

if __name__ == '__main__':
    cmdline_reader()
Source Code B.1: scripter module
Appendix C

seriesoftubes Source Code

C.1 seriesoftubes scripts

C.1.1 preprocess_reads.py

```
#!/usr/bin/env python
#
preprocess_reads.py is meant to be run on GERALD output
    although it will run on any FASTQ files*

We can
+ Separate reads by a set of variable-length barcodes
+ Cleave linker/adaptor sequence from the 3' ends of reads
+ Cleave adaptor sequence from the 5' end
    + before barcode
    + after barcode
+ Discard sequences that are less than 4 nucleotides in length
+ Produce gzipped FASTQ sequence files ready for immediate alignment

TRAILING Ns ARE NO LONGER TRIMMED (per NCBI guidelines)

A configuration file 'preprocess_reads.cfg' is saved in target
directory (unless one is provided by the user).

*it expects that files are named s_?_sequence.* (single-end reads) or
  s_?[12]_sequence.* (paired-end reads)
```
import os
from itertools import imap, izip, chain
from functools import partial
from scripter import Environment, get_logger
from seriesoftubes.converters.discover import discover_file_format,
    PATH_TO_GZIP, gzip_class_factory
from seriesoftubes.fnparsers import BarcodeFilenameParser
from seriesoftubes.cPreprocess import *
from Bio.SeqIO.QualityIO import FastqGeneralIterator
from ConfigParser import SafeConfigParser
from errno import ENOENT, EACCES
from os import access, strerror, R_OK
from os.path import exists
from argparse import ArgumentTypeError
from gzip import GzipFile
from pkg_resources import get_distribution
__version__ = get_distribution('seriesoftubes').version
VERSION = __version__

def valid_seq(s):
    if not s.isalpha() and len(s)>0:
        msg = '%s is not a valid sequence'
        raise ArgumentTypeError(msg)
    else:
        return s.upper()

def main():
    e = Environment(version=VERSION, doc=__doc__)
    parser = e.argument_parser
    parser.add_argument('---no-clipping', default=False, action='store_true',
        help='Do not clip barcodes from reads when assigning to barcode. OVERRIDES ALL OTHER CLIPPING OPTIONS')
    parser.add_argument('---strip-after-barcode', default=1, type=int,
        help='strip n bases after the barcode is removed (5' end)
    (by default this 1 now, and is ignored if GERALD handled the barcoding)
    ''
    parser.add_argument('---strip-before-barcode', default=0, type=int,
        help='strip n bases before the barcode is removed (5' end)
    (by default this 0 now, and is ignored if GERALD handled the barcoding)
    ''
    parser.add_argument('---min-length', type=int, default=4,
        help='require sequences to be at least n total bases of non-N sequence (default: ignore)')
    parser.add_argument('---max-length', type=int, default=-1,
        help='truncate final sequences to n bases (default: ignore)')
    parser.add_argument('---no-gzip', default=False, action='store_true',
help = 'Do not gzip output files')
bgroup = parser.add_argument_group('barcodes',
    'Specify sequence barcodes in the sample(s)')
bgroup.add_argument('-b', '--barcodes', action='append',
    help='Specify a barcode sequence. May be invoked multiple times')
bgroup.add_argument('--kry-barcodes', dest='barcodes', action='store_const',
    help='Alias for --TCAT --GACG --AGTC --CTGA',
    const=['TCAT', 'GACG', 'AGTC', 'CTGA'])
parser.add_argument('--linker', default='',
    help='Specify a 3’ adaptor/linker sequence that we should clip off of each read')
parser.set_defaults(**{ 'target': 'processed' })
e.set_filename_parser(BarcodeFilenameParser)
e.set_config_reader(read_config)
e.set_config_writer(write_config)
e.do_action(splitter)

def write_config(barcodes= None, min_length= None, max_length= None,
    strip_after_barcode=0, strip_before_barcode=0, linker= None, target_dir = os.
curdir,
    target= None, *
args, **kwargs):
    config = SafeConfigParser()
    config.add_section('main')
    if barcodes is not None:
        config.set('main', 'barcodes', ', '.join(barcodes))
    config.set('main', 'linker', linker)
    config.set('main', 'min-length', str(min_length))
    config.set('main', 'max-length', str(max_length))
    config.set('main', 'strip-after-barcode', str(strip_after_barcode))
    config.set('main', 'strip-before-barcode', str(strip_before_barcode))
    config_f = os.path.join(target_dir, 'preprocess_reads.cfg')
    if not os.path.exists(target_dir):
        os.makedirs(target_dir, mode=0755)
    with open(config_f, 'wb') as configfile:
        config.write(configfile)

def read_config(setup_file):
    if not os.path.exists(setup_file):
        raise IOError(ENOENT, strerror(ENOENT), setup_file)
    if not access(setup_file, R_OK):
        raise IOError(EACCES, strerror(EACCES), setup_file)
    config = SafeConfigParser()
    config.readfp(open(setup_file, 'rU'))
    context = { }
if config.has_option('main', 'barcodes'):
    context['barcodes'] = config.get('main', 'barcodes').split(',')
if config.has_option('main', 'max-length'):
    context['max_length'] = int(config.get('main', 'max-length'))
if config.has_option('main', 'min-length'):
    context['min_length'] = int(config.get('main', 'min-length'))
if config.has_option('main', 'linker'):
    context['linker'] = config.get('main', 'linker')
context['strip_after_barcode'] = int(config.get('main', 'strip-after-barcode'))
context['strip_before_barcode'] = int(config.get('main', 'strip-before-barcode'))
return context

def splitter(pf, **kwargs):
    logger = get_logger()
    if pf.paired_end:
        return split_paired_files(pf, logger=logger, **kwargs)
    else:
        return split_file(pf, logger=logger, **kwargs)

def split_file(fp_obj, no_gzip=False, barcodes=[], linker='', min_length=4, max_length=-1, logger=None, strip_before_barcode=0, strip_after_barcode=0, no_clipping=False, **kwargs):
    if logger is None: logger = get_logger()
    filename = fp_obj.input_file
    open_func, format_ = discover_file_format(filename)
    if not format_ == 'FASTQ':
        logger.error('Only FASTQ files are supported at this time')
        return
    f = open_func(filename, "rU")
    records = FasterFastqIterator(f)

    barcoded_files = {}
    filenames = []
    output_filename = partial(fp_obj.output_filename, no_gzip=no_gzip)
    if no_gzip: open_func = open
    elif PATH_TO_GZIP is not None:
        open_func = gzip_class_factory(PATH_TO_GZIP)
    else: open_func = GzipFile
    if barcodes is None: barcodes = []
    if len(barcodes) > 0:
        processed_file = None
        for barcode in barcodes:
            fname = output_filename(barcode)
filenames.append(fname)
barcoded_files[barcode] = open_func(fname, 'w')

# and make a unmatched file
unmatched_filename = output_filename("unmatched")
filenames.append(unmatched_filename)
unmatched_file = open_func(unmatched_filename, 'w')

else:
    barcoded_files = None
    unmatched_file = None
process_filename = output_filename("processed", is_barcode=False)
filenames.append(process_filename)
processed_file = open_func(process_filename, 'w')

writer_args = {
    'barcoded_files': barcoded_files,
    'unmatched_file': unmatched_file,
    'processed_file': processed_file
}
results = apply_plan(records, writer_args, barcodes=barcodes, linker=linker,
    min_length=min_length, max_length=max_length,
    strip_after_barcode=strip_after_barcode,
    strip_before_barcode=strip_before_barcode,
    no_clipping=no_clipping,
    logger=logger)

linker_only = results['linker']
too_short = results['short']
record_count = results['all']

# close and exit 
#
f.close()

if barcoded_files is not None:
    logger.debug('closing barcoded files')
    for f in barcoded_files.values(): f_.close()
if unmatched_file is not None:
    logger.debug('closing unmatched file')
    unmatched_file.close()
if processed_file is not None:
    logger.debug('closing output file')
    processed_file.close()

logger.info('Split %s as %s', fp_obj.input_file, ', '.join(filenames))
logger.info('Processed %s records', record_count)
logger.info('%s linker only dimers', linker_only)
logger.info('%s sequences too short (1-3 bp)', too_short)

def split_paired_files(fp_obj, no_gzip=False,
    barcodes=None, linker='',
    min_length=4,
max_length = -1,
strip_before_barcode=0,
strip_after_barcode=0,
logger=None,
no_clipping=False,
**kwargs):
    filename = fp_obj.input_file
    filename2 = fp_obj.second_file
    open_func, format1 = discover_file_format(filename)
    open_func2, format2 = discover_file_format(filename2)
    if not (format1 == 'FASTQ' and format2 == 'FASTQ'):
        logger.error('Only FASTQ files are supported at this time')
        return
    f = open_func(filename, "rU")
    f2 = open_func2(filename2, "rU")
    records = FasterFastqIterator(f)
    records2 = FasterFastqIterator(f2)

    barcoded_file_pairs = {}
    filenames = []
    if no_gzip:
        open_func = open
    elif PATH_TO_GZIP is not None:
        open_func = gzip_class_factory(PATH_TO_GZIP)
    else:
        open_func = GzipFile
    output_filename = partial(fp_obj.output_filename, no_gzip=no_gzip)
    output_filename2 = partial(fp_obj.output_filename2, no_gzip=no_gzip)
    if barcodes is None: barcodes = []
    if len(barcodes) > 0:
        processed_files = None
        orphaned_read_files = None
        for barcode in barcodes:
            fname = output_filename(barcode)
            fname2 = output_filename2(barcode)
            filenames.extend((fname, fname2))
            barcoded_file_pairs[barcode] = (open_func(fname, 'w'),
                                           open_func(fname2, 'w'))

        # and make a unmatched file
        unmatched_filename = output_filename("unmatched")
        unmatched_filename2 = output_filename2("unmatched")
        unmatched_files = (open_func(unmatched_filename, 'w'),
                           open_func(unmatched_filename2, 'w'))

        mismatched_filename = output_filename("mismatched")
        mismatched_filename2 = output_filename2("mismatched")
        mismatched_files = (open_func(mismatched_filename, 'w'),
                           open_func(mismatched_filename2, 'w'))
    filenames.extend((unmatched_filename, unmatched_filename2,
                      mismatched_filename, mismatched_filename2))
else:
    barcoded_file_pairs = None
    unmatched_files = None
    mismatched_files = None
    orphaned_read_filename = output_filename("orphaned",
            is_barcoded=False)
    orphaned_read_filename2 = output_filename2("mismatched",
            is_barcoded=False)
    orphaned_read_files = (open_func(orphaned_read_filename, 'w'),
                           open_func(orphaned_read_filename2, 'w'))
    processed_filename = output_filename("processed", is_barcoded=False)
    processed_filename2 = output_filename2("processed", is_barcoded=False)
    processed_files = (open_func(processed_filename, 'w'),
                       open_func(processed_filename2, 'w'))
    filenames.extend((orphaned_read_filename,
                       orphaned_read_filename2),
                      (processed_filename, processed_filename2))
    writer_args = {
                           'barcoded_file_pairs': barcoded_file_pairs,
                           'unmatched_files': unmatched_files,
                           'mismatched_files': mismatched_files,
                           'processed_files': processed_files,
                           'orphaned_read_files': orphaned_read_files,
                           'linker': linker,
                           'min_length': min_length
                        }

    results = apply_plan_pe(records, records2, writer_args,
                           barcodes=barcodes, linker=linker,
                           min_length=min_length,
                           max_length=max_length,
                           strip_after_barcode=strip_after_barcode,
                           strip_before_barcode=strip_before_barcode,
                           no_clipping=no_clipping,
                           logger=logger)

    linker_only = results['linker']
    too_short = results['short']
    record_count = results['all']

    # close and exit #
    f.close()
    for f, f2 in barcoded_file_pairs.values():
        f.close()
        f2.close()
    unmatched_files[0].close()
    unmatched_files[1].close()
    mismatched_files[0].close()
    mismatched_files[1].close()
logger.info('Split %s, %s as %s', fp_obj.input_file, fp_obj.second_file, ', '.join(filenames))
logger.info('Processed %s records', record_count)
logger.info('%s linker only dimers', linker_only)
logger.info('%s sequences too short (1-3 bp)', too_short)

if __name__ == '__main__': main()

Source Code C.1: preprocess_reads.py

C.1.2 align2.py

#!/usr/bin/env python
##
# align FASTQ, SAM, or BAM file (gzip and bzip2 supported) with bowtie2, produces BAM files (sorted and indexed)
# output will be in ./align2
# In align2, there will be a folder for each reference genome e.g. align2/ref1, align2/ref2, align2/ref1, align2/ref2
##
import pysam
import logging
import os
from os import getcwd, environ, curdir, makedirs, devnull
from os.path import join, exists, splitext
from textwrap import dedent
import sys
from tempfile import mkdtemp
from subprocess import Popen, PIPE, STDOUT
import scripter
from scripter import assert_path, path_to_executable, Usage, \
    exit_on_Usage, InvalidFileException, get_logger, \
    Environment, critical, debug
from seriesoftubes.tubes.polledpipe import PolledPipe
from seriesoftubes.tubes import wait_for_job
from seriesoftubes.fnparsers import BowtieFilenameParser

from pkg_resources import get_distribution
__version__ = get_distribution('seriesoftubes').version
VERSION = __version__

def main():
    e = Environment(version=VERSION, doc=__doc__)
    e.set_filename_parser(BowtieFilenameParser)
    # let bowtie2 do the multiprocessing
    e.override_num_cpus(1)
parser = e.argument_parser
parser.add_argument('−−path-to-bowtie2', nargs='?',
                  default=path_to_executable('bowtie2',
'/usr/local/bowtie2∗',
environ='SOT_PATH_TO_BOWTIE2'),
                  help='The path to the bowtie2 executable')
parser.add_argument('−−path-to-samtools', nargs='?',
                  default=path_to_executable('samtools',
'/usr/local/samtools∗',
environ='SOT_PATH_TO_SAMTOOLS'),
                  help='The path to the samtools executable')
# fix aliases, should be --ref too
parser.add_argument('−−reference', dest='references', action='append',
                  help=dedent('''
Reference genome to align against (either a bowtie2 index name or
file, or a fasta file). This flag may be called multiple times (which will
cause each reference to be aligned to separately). If no references are
specified, we'll look for environment variable SOT_DEFAULT_REFERENCES,
which should be given as a list, e.g. "foo foo2 foo3"
'''),
parser.add_argument('−−ignore-quality', dest='use_quality',
                  action='store_false',
                  help=dedent('''
Ignore quality scores if available. Also applies to
counter-references if any are called'''))
cparser = parser.add_argument_group('counter-alignments',
description=dedent('''
specify counter-reference genome(s)/sequence(s) to use for filtering out
unwanted reads.'''))
cparser.add_argument('−−counter-reference', dest='counter_references',
                  action='append',
                  help=dedent('''
Optional counter-reference genome/sequences to align against (either
a bowtie2 index name or file, or a fasta file). This flag may be called
multiple times.
All counter-references will be concatenated into one index, and
reads will be aligned in −−fast mode. Any reads which align will be saved
in a separate directory called 'counteraligned' and not aligned
against the reference genomes/sequences. If no counter-references are specified,
we'll look the for environment variable SOT_DEFAULT_COUNTER_REFERENCES,
''')
which should be given as a list, e.g. "foo foo2 foo3"
)
parser.add_argument('--passthru-args', nargs='*','
        help='A list of arguments to be passed through
to bowtie2 [alignment and counter-alignment ]. Substitute + for − (e.g., −−passthru-args
-im 4 50 )

context = e.get_context()
new_references = validate_references(**context)
new_counter_references = cat_counter_references(**context)
e.update_context({'references': new_references,
        'counter_references': new_counter_references})
sequence = e.get_sequence(**context)
e._sequence = merge_pairs(sequence)
e.do_action(align2)

def merge_pairs(old_sequence):
    """ pair sequence files that differ only by 1/2 in one position
    """
    new_sequence = []
    sequence = []
    for fp_obj in old_sequence:
        if fp_obj.paired_end: new_sequence.append(fp_obj)
        else: sequence.append(fp_obj)
    input_files = [fp_obj.input_file for fp_obj in sequence]
    mates = []
    for i in xrange(len(input_files)):
        if i in mates: continue
        these_mates = []
        for j in xrange(i, len(input_files)):
            identity = [(not x==y) for x,y in zip(input_files[i],
            input_files[j])]
            if sum(identity) == 1:
                index = identity.index(True)
                if (input_files[i][index] == '1' and
                    input_files[j][index] == '2') or
                (input_files[i][index] == '2' and
                    input_files[j][index] == '1'):
                    these_mates.append(j)
                    print i,j,input_files[i],input_files[j]
            if len(these_mates) == 0: continue
            elif len(these_mates) == 1:
                j = these_mates[0]
            else:
                # User input required
                print "Ambiguous filename pairing"
                print "Please select the correct mate pair for %s:" %
                    input_files[i]
                while True:
                    for j in these_mates:
                        print "[%d] %s" % (j, input_files[j])
                        input_file = input()
                        if input_file != '1' and input_file != '2':
                            continue
                        if input_file == 1:
                            j = these_mates[0]
                        else:
                            j = input()
                            continue
            print 'Selected: %s' % input_files[j]
            mates.append(j)
            break
        else:
            print 'Mates not found for %s' % input_files[i]
    return new_sequence

"""
rawinput = raw_input("Enter your choice: ")

try:
    choice = int(rawinput.strip())
except ValueError:
    continue

if not choice in these_mates:
    print "%d is not a valid choice" % choice
else:
    j = choice
    break

identity = [(not x==y) for x,y in zip(input_files[i], input_files[j])]
index = identity.index(True)
if input_files[i][index] == '1' and input_files[j][index] == '2':
    sequence[i].second_file = sequence[j].input_file
    sequence[i].paired_end = True
    mates.append(j)
elif input_files[j][index] == '1' and input_files[i][index] == '2':
    sequence[j].second_file = sequence[i].input_file
    sequence[j].paired_end = True
    mates.append(i)

new_sequence.extend([x for i, x in enumerate(sequence) if not i in mates])
return new_sequence

def fasta_to_bowtie2(fasta_file, target_dir=curdir, path_to_bowtie2='bowtie2'):
    """given a filename, makes a bowtie2 index if that file is a FASTA file"""
    if exists(fasta_file):
        f = open(fasta_file, 'rU')
        for line in f:
            if line.startswith('#'): continue
            elif line.startswith('> '):
                args = [path_to_bowtie2 + '−build', fasta_file,
                        join(target_dir, fasta_file)]
                debug(''.join(args))
                P = Popen(args, stdout=open(devnull, 'w'), stderr=PIPE)
                stderr = P.communicate()[1]
                if stderr.splitlines()[0].startswith('Error'): return None
                else: return join(getcwd(), target_dir, fasta_file)
        return None

def find_bowtie2_index(r, path_to_bowtie2='bowtie2'):
    """check for bowtie2 index as given. return True if found, else return False"""
    args = [path_to_bowtie2 + '−inspect', '−v', '−s', r]
    debug(''.join(args))
P = Popen(args, stdout=open(devnull, 'w'), stderr=PIPE, cwd=mkdtemp())
stderr = P.communicate()[1].splitlines()
if not stderr[0].startswith('Could not locate '):
    for line in stderr:
        if line.startswith('Opening '):
            index_bt2 = line[(1 + line.find('"')):line.rfind('"')]
            index_basename = index_bt2[0:index_bt2.find('.1.bt2')]
            return index_basename
for d in [getcwd(), os.path.split(path_to_bowtie2)[0],
            join(os.path.split(path_to_bowtie2)[0], 'indexes')]:
    rprime = join(d, r)
    args = [path_to_bowtie2 + '−inspect', '−v', '−s', rprime]
    debug(join(args))
    P = Popen(args, stdout=open(devnull, 'w'), stderr=PIPE, cwd=mkdtemp())
stderr = P.communicate()[1].splitlines()
if not stderr[0].startswith('Could not locate '):
    for line in stderr:
        if line.startswith('Opening '):
            index_bt2 = line[(1 + line.find('"')):line.rfind('"')]
            index_basename = index_bt2[0:index_bt2.find('.1.bt2')]
            return index_basename
return None

def cat_counter_references(counter_references=None, target_dir=curdir,
                            path_to_bowtie2='bowtie2',
                            logger=None, **kwargs):
    try:
        makedirs(target_dir, mode=0755)
    except OSError:
        pass
    debug('Validating counter−references and building counter−reference index')
    valid_references = validate_references(
        references=counter_references,
        target_dir=target_dir,
        path_to_bowtie2=path_to_bowtie2,
        logger=logger,
        environ_key='SOT_DEFAULT_COUNTER_REFERENCES'
    )
    crefs_fa = open(join(target_dir, 'counter_references.fa'), 'w')
    for ref in counter_references:
        Popen([path_to_bowtie2 + '−inspect', ref], stdout=crefs_fa).wait()
    crefs_index = join(target_dir, counter_references)
    args = [path_to_bowtie2 + '−build', crefs_fa, crefs_index]
    P = Popen(args, stderr=PIPE)
    stderr = P.communicate()[1]
    if stderr.startswith('Error '):

critical(stderr)
critical('No counter−references will be used.')
return crefs_index

def validate_references(references= None, path_to_bowtie2='bowtie2',
logger=None, environ_key='SOT_DEFAULT_REFERENCES',
target_dir=curdir, **kwargs):
makedirs(target_dir, mode=0755)
debug('Validating references')
new_references = []
if references is None:
    if environ.has_key(environ_key):
        references = environ[environ_key].split()
    else:
        critical('no reference genomes specified')
        return []
for r in references:
    bowtie2_index = find_bowtie2_index(r, path_to_bowtie2=path_to_bowtie2)
    if bowtie2_index is None:
        if exists(r):
            debug('Attempting to build bowtie2 index f r o m %s' %r)
            new_index = fasta_to_bowtie2(r, target_dir=target_dir,
path_to_bowtie2=path_to_bowtie2)
            if new_index is not None:
                new_references.append(new_index)
                continue
            else:
                critical('Failed to build bowtie2 index.')
                critical('bowtie2 could not find the index for %s', r)
                critical('we will not align to %s', r)
        else:
            new_references.append(bowtie2_index)
return new_references

def convert_to_fastq(fp_obj, logger=None):
    """Convert a SAM or BAM file to FASTQ file(s) for alignment
    """
    input_file = fp_obj.input_file
    output_dir = fp_obj.output_dir
    fastq_dir = join(output_dir, 'fastq_input')
    fp_obj.check_output_dir(fastq_dir)
    protoname = fp_obj.protoname
    if fp_obj.paired_end:
        fastq_filenames = (join(fastq_dir, '%s.1.txt.gz' % protoname),
join(fastq_dir, '%s.2.txt.gz' % protoname))
        logger.info('Converting file %s to FASTQ files %s, %s',
            %s, %s,
in_args = [sys.executable, '−m', 'seriesoftubes.converters.
    bamtofastq2',
    '−−gzip', input_file, fastq_filenames[0],
    fastq_filenames[1]]
else:
    fastq_filename = join(fastq_dir, '%s.txt.gz' % protoname)
    logger.info('Converting file %s to FASTQ file %s',
    input_file, fastq_filename)
in_args = [sys.executable, '−m', 'seriesoftubes.converters.
    bamtofastq2',
    input_file, fastq_filename]
logger.debug('Launching %s' % join(in_args))
polledpipe = PolledPipe(logger=logger, level=logging.ERROR)
job = Popen(in_args, stdout=polledpipe.w, stderr=STDOUT)
wait_for_job(job, [polledpipe], logger)
if fp_obj.paired_end:
    logger.debug('Settings input_file to %s', fastq_filenames[0])
    fp_obj.input_file = fastq_filenames[0]
    logger.debug('Settings second_file to %s', fastq_filenames[1])
    fp_obj.second_file = fastq_filenames[1]
else:
    logger.debug('Settings input_file to %s', fastq_filename)
    fp_obj.input_file = fastq_filenames[0]
    logger.debug('Setting use_pysam to False')
    fp_obj.use_pysam = False
    logger.debug('Setting format to FASTQ')
    fp_obj.format = 'FASTQ'
    logger.debug('Ignoring open_func, it will not be used')
    if not job.returncode == 0:
        logger.critical('Conversion FAILED!')
    else:
        logger.info('Conversion successful')
return

@exit_on_Usage
def align2(fp_obj, references=[], counter_references=None,
    unique=True, seed_len='28',
    use_quality=True, logging_level=10, num_cpus=1,
    passthru_args=None,
    **kwargs):
    common_flags = ['−−time', '−p', str(num_cpus), '−L', seed_len]
    if fp_obj.paired_end: common_flags.extend(['−X', '600'])

    logger = get_logger(logging_level)
    if references is None or references == []:
        logger.critical('Nothing to do')
    return
stdout_buffer = []
if fp_obj.format == 'BAM' or fp_obj.format == 'SAM':
convert_to_fastq(fp_obj, logger=logger)

if not fp_obj.format == 'FASTQ':
    logger.critical('Files only supports FASTQ files % _ file_ )
    return

if passthru_args is not None:
    for i in range(len(passthru_args)):
        passthru_args[i] = passthru_args[i].replace('+', '-')
    logger.debug('Passing thru arguments %s', ' '.join(passthru_args))
kwgs['passthru_args'] = passthru_args

for ref in references:
    s = align_once(fp_obj, flags, ref, logger=logger, **kwgs)
    stdout_buffer.append(s)
return '
'.join([s for s in stdout_buffer if s is not None])

def make_paired_name(name1, name2):
    identity = [(not x==y) for x,y in zip(name1, name2)]
    if not sum(identity) == 1: raise ValueError('Not valid names')
    else:
        index = identity.index(True)
        return identity[0:index] + '%' + identity[(index + 1):]

def counteralign_once(fp_obj, flags, ref, use_quality=False, path_to_bowtie2=None, path_to_samtools=None, logger=None, passthru_args=None, **kwgs):
    """Produce counter-alignments""
    if use_quality:
        if fp_obj.use_pysam: flags.append('--phred33')
        else: flags.append('--phred64')
    refname = os.path.basename(ref)
    output_dir, output_file = os.path.split(fp_obj.tmp_filename(refname))
fp_obj.check_output_dir(output_dir)
fp_obj.check_output_dir(join(output_dir, 'counteraligned'))
filename1 = os.path.abspath(fp_obj.input_file)
second_file = fp_obj.second_file
if second_file is not None: filename2 = os.path.abspath(second_file)
else: filename2 = None

if fp_obj.paired_end:
    try:
        paired_file = make_paired_name(input_file, second_file)
counteraligned = os.path.abspath(join(output_dir, 'counteraligned', paired_file))
    except ValueError:
        counteraligned = os.path.abspath(join(output_dir, 'counteraligned', input_file))
    file_args = ['-x', ref, '-1', filename1, '-2', filename2, '--al-conc-gz', counteraligned, '--un-conc-gz', join(output_dir, paired_file)]
    new_filenames = (join(output_dir, input_file), join(output_dir, second_file))
else:
    file_args = ['-x', ref, '-U', filename1, '--al-gz', join(output_dir, 'counteraligned', input_file), '--un-gz', join(output_dir, input_file)]
    new_filenames = (join(output_dir, input_file), None)

if passthru_args is not None:
    bowtie2_args = [path_to_bowtie2] + flags + passthru_args + file_args
else:
    bowtie2_args = [path_to_bowtie2] + flags + file_args

# finish parsing input here
bowtie2_stdout = PolledPipe(logger=logger, level=logging.ERROR)
bowtie2_stderr = PolledPipe(logger=logger, level=logging.ERROR)
logger.info("Launching bowtie2 (output will be piped to samtools for BAM encoding)")
logger.info("", join(bowtie2_args))
bowtie2_aligner = Popen(bowtie2_args, stdout=open(devnull, 'w'), stderr=bowtie2_stderr.w,
    bufsize=-1)
logger.info("", join(in_args))
logger.info("counteraligned reads will be saved as GZIPed FASTQ files in counteraligned/")
logger.debug("Waiting for bowtie2 to finish")
pollables = [bowtie2/stderr]
wait_for_job(bowtie2_aligner, pollables, logger)
if not bowtie2_aligner.returncode == 0:
    logger.critical("bowtie2 did not run properly [%d]",
                   bowtie2_aligner.returncode)
    return

logger.debug('Alignment successfully completed')

return new_filenames

def align_once(fp_obj, flags, ref, use_quality=False,
              path_to_bowtie2=None, path_to_samtools=None, logger=None,
              passthru_args=None,**kwargs):
    if use_quality:
        if fp_obj.use_pysam: flags.append('−−phred33')
        else: flags.append('−−phred64')

    refname = os.path.basename(ref)
    path_to_unsorted = fp_obj.tmp_filename(refname)
    output_dir = os.path.split(path_to_unsorted)[0]
    fp_obj.check_output_dir(output_dir)
    filename1 = os.path.abspath(fp_obj.input_file)
    second_file = fp_obj.second_file
    if second_file is not None: filename2 = os.path.abspath(second_file)
    else: filename2 = None

    if fp_obj.paired_end:
        file_args = ['-x', ref, '−1', filename1, '−2', filename2]
    else:
        file_args = ['-x', ref, '−U', filename1]

    if passthru_args:
        bowtie2_args = [path_to_bowtie2] + flags + passthru_args +
                        file_args
    else:
        bowtie2_args = [path_to_bowtie2] + flags + file_args

# finish parsing input here
bowtie2_stderr = PolledPipe(logger=logger, level=logging.ERROR)
logger.info('Launching bowtie2 (output will be piped to
            samtools for
            BAM encoding)')
logger.info(' '.join(bowtie2_args))
bowtie2_aligner = Popen(bowtie2_args, stdout=PIPE, stderr=bowtie2_stderr.w,
                         bufsize=-1)

samtools_args = [path_to_samtools, 'view', '−b', '−S', '−o',
                  path_to_unsorted, '−']
logger.info('Launching samtools to encode bowtie2 output as BAM')
logger.info(' '.join(samtools_args))
samtools_stdout = PolledPipe(logger=logger, level=logging.WARN)
samtools_stderr = PolledPipe(logger=logger, level=logging.ERROR)
samtools_viewer = Popen(samtools_args, stdin=bowtie2_aligner.stdout, stdout=samtools_stdout.w, stderr=samtools_stderr.w, bufsize=-1)

logger.debug('Waiting for bowtie2 to finish')
pollables = [bowtie2_stderr, samtools_stdout, samtools_stderr]
wait_for_job(bowtie2_aligner, pollables, logger)

if not bowtie2_aligner.returncode == 0:
    logger.critical("bowtie2 did not run properly [%d]", bowtie2_aligner.returncode)
    samtools_viewer.terminate()
    samtools_viewer.poll()
    logger.critical("samtools terminated")
    return

logger.debug('Alignment successfully completed')
logger.debug('Waiting for samtools to finish')
wait_for_job(samtools_viewer, [samtools_stdout, samtools_stderr], logger)
if not samtools_viewer.returncode == 0:
    logger.critical("samtools view did not run properly [%d]", samtools_viewer.returncode)
    return

logger.debug('Unsorted BAM file successfully written')

logger.info('Launching samtools again to sort BAM output')
output_dir, output_file = os.path.split(path_to_unsorted)
bam_file = os.path.splitext(output_file)[0]
sorter_args = [path_to_samtools, 'sort', output_file, bam_file]
logger.info(''.join(sorter_args))
samtools_stdout = PolledPipe(logger=logger, level=logging.WARN)
samtools_stderr = PolledPipe(logger=logger, level=logging.ERROR)
samtools_sorter = Popen(sorter_args, stdin=samtools_stdout.w, stderr=samtools_stderr.w, cwd=output_dir)
wait_for_job(samtools_sorter, [samtools_stdout, samtools_stderr], logger)
if not samtools_sorter.returncode == 0:
    logger.critical("samtools sort did not run properly [%d]", samtools_sorter.returncode)
    return

# don't destroy the files until we're sure we succeeded!
assert_path(os.path.join(output_dir, bam_file + '.bam'))
logger.debug('Removing unsorted file %s', path_to_unsorted)
o.remove(path_to_unsorted)

logger.debug('Launching samtools again to index sorted BAM output')
from align2 import main

# Make sure indexing succeeds
assert_path(os.path.join(output_dir, bam_file + '.bam.bai'))
return

if __name__ == '__main__':
    main()
```python
__version__ = get_distribution('seriesoftubes').version

VERSION = __version__

MACS_VERSION = get_distribution('MACS2>=2.0.10').version

def main():
    e = Environment(doc=_doc_, version=VERSION)
    parser = e.argument_parser
    parser.add_argument('-g', '--genome-size', dest='user_gsize',
                        default=None,
                        help='Optional user-specified genome size (DEFAULT: script will try to auto-detect the genome)')
    parser.add_argument('--path-to-macs',
                        default=path_to_executable('macs2'),
                        help='optional path to macs2 executable')
    parser.add_argument('--no-subpeaks', dest='subpeaks', action='store_false',
                        default=True,
                        help='do not call subpeaks with --call-summits')
    parser.add_argument('-q', '--q-value', dest='qvalue', default='0.01',
                        help='FDR/q-value cutoff (default is 0.01)')
    parser.add_argument('--passthru-args', nargs='*',
                        help='A list of arguments to be passed through to MACS2. Substitute + for -(e.g., --passthru-args +m 4 50)')
    parser.set_defaults(**{'target': 'peaks'})
    e.set_filename_parser(BAMFilenameParser)
    e.set_config_reader(read_setup_file)
    e.set_config_writer(write_setup_file)
    e.do_action(run_macs)

def write_setup_file(controls=None, target_dir=curdir, *args, **kwargs):
    makedirs(target_dir, mode=0755)
    setup_file = open(join(target_dir, 'setup.txt'), 'w')
    setup_file.write('#%s
' % ' '.join(sys.argv))
    if controls is None: return
    for sample, value in controls.items():
        name, control = value
        if control == None: control = 'None'
        record = '%s\nsample = %s\ncontrol = %s\n' % (name, sample, control)
        setup_file.write(record)
    setup_file.close()

def read_setup_file(setup_file):
    if not exists(setup_file):
        raise IOError(ENOENT, strerror(ENOENT), setup_file)
    if not access(setup_file, R_OK):
        raise IOError(EACCES, strerror(EACCES), setup_file)
    config_parser = ConfigParser()
config_parser.readfp(open(setup_file, 'rU'))
files = []
controls = {}  # {sample: (name, control)}
for section in config_parser.sections():
    name = section
    sample = config_parser.get(section, 'sample')
    if config_parser.has_option(section, 'control'):
        control = config_parser.get(section, 'control')
        if control.strip() == 'None': control = None
    else:
        control = None
    controls[sample] = (name, control)
    files.append(sample)
    files.append(control)
return {'controls': controls, 'files': files}

def decide_format(input_file, control_file, logger=None):
    # See if we have paired-end files
    s = pysam.Samfile(input_file)
    try:
        is_paired = [s.next().is_paired for i in xrange(100000)]
    except StopIteration:
        is_paired = [r.is_paired for r in s]
    if control_file is not None:
        t = pysam.Samfile(control_file)
        try:
            is_paired_control = [s.next().is_paired for i in xrange(100000)]
        except StopIteration:
            is_paired_control = [r.is_paired for r in s]
        else:
            is_paired_control = [True]
    if all(is_paired) and all(is_paired_control):
        if logger is not None:
            logger.warn('Detected paired end files')
            logger.warn('Using new BAMPEParser instead of BAMParser')
    return 'BAMPE'
    else:
        return 'BAM'

def run_macs(f, subpeaks=True, path_to_macs=None, logging_level=10, user_gsize=None, qvalue=0.01, passthru_args=None, **kwargs):
    """Run MACS on a BAM file
    """
    logger = get_logger(logging_level)
    if path_to_macs is None:
        path_to_macs = path_to_executable("macs2")
    input_file = f.input_file
    control_file = f.control_file
logger.debug('Processing %s', input_file)
if control_file is not None:
    logger.debug('with control %s', control_file)

    # determine genome name and size
if user_gsize:
    genome_size = user_gsize
    try:
        genome_build = guess_bam_genome(input_file)
    except NoMatchFoundError:
        genome_build = None
else:
    try:
        genome_build = guess_bam_genome(input_file)
    except NoMatchFoundError:
        raise Usage('Could not determine genome / genome size for file %s' % input_file)

    gname = ''.join([x for x in genome_build if x.isalpha()])
    if gname == "hg": genome_size = 'hs'
    elif gname in ['mm', 'ce', 'dm']:
        genome_size = gname
    else:
        genome_size = '%.1e' % sum(generate_genome(genome_build).itervalues())

fmt = decide_format(input_file, control_file, logger)
name = f.sample_name.replace(' ', '_')
if passthru_args is not None:
    for i in range(len(passthru_args)):
        passthru_args[i] = passthru_args[i].replace('+', '-
    logger.debug('Passing thru arguments %s', ' '.join(passthru_args))

macs_options = ['--trackline',
                '-f', fmt, # correct file format BAM or BAMPE
                '-B', '--SPMR', # bedgraphs, SPMR
                '-g', genome_size,
                '-q', qvalue,
                '-n', name, # run name
                '-t', join(getcwd(), input_file)] # treatment

if control_file is not None:
    macs_options.extend(['-c', join(getcwd(), control_file)])
if subpeaks: macs_options.append('--call-summits')
if passthru_args is not None:
    macs_options.extend(passthru_args)

step = [path_to_macs, 'callpeak'] + macs_options
if platform.system() is 'Windows': step.insert(0, sys.executable)

macs_stdout = PollledPipe(logger=logger, level=WARNING)
macs_stderr = PollledPipe(logger=logger, level=ERROR)
logger.debug('Launching %s', ' '.join(step))
job = Popen(step, stdout=macs_stdout.w, stderr=macs_stderr.w, cwd=f.output_dir)
pollables = [macs_stdout, macs_stderr]
wait_for_job(job, pollables, logger)

return '\n'.join(step)

if __name__=='__main__': main()
```python
def get_motif(foo, motif_number, motif_type):
    """
    get motif_number motif from file foo
    """
    if not exists(foo):
        raise Usage(foo, "not found")
    motifs = Bio.Motif.parse(open(foo), motif_type)
    for i in xrange(motif_number):
        try:
            motif = motifs.next()
        except StopIteration:
            raise Usage("%s only contains %d motifs") % (foo, i)
    return motif

def action(fp_obj, motif_file=None, motif_type=None, motif_number=1,**kwargs):
    logger = get_logger()
    logger.debug("trying to find sites for %s", fp_obj.input_file)
    motif = get_motif(motif_file, motif_number, motif_type)
    stdout_buffer = find_sites(fp_obj.input_file,
                                fp_obj.fasta_file,
                                motif, bed=fp_obj.is_bed, xls=fp_obj.is_xls,
                                output_dir=fp_obj.output_dir,
                                src_fnc = __file__, **kwargs)

    return stdout_buffer

if __name__ == '__main__': main()
```

### C.1.5 align.py (deprecated)

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

Important Notice:
--references has been removed, please use --reference (multiple uses are ok)

align FASTQ, SAM, or BAM file (gzip and bzip2 supported)
with bowtie, produces BAM files (sorted and indexed)
output will be in ./align
In align, there will be a folder for each reference genome
e.g. align/ref1, align/ref2, align/ref1, align/ref2
In reference folder, there will be a folder for unique or random
alignments
(unique means only maps to one spot, random is maq-like behavior)
e.g. align/ref1/random, align/ref1/unique, etc.
```
import pysam
import logging
import os
from os import getcwd, environ, curdir, makedirs, devnull
from os.path import join, exists, splitext
from textwrap import dedent
import sys
from tempfile import mkdtemp
from subprocess import Popen, PIPE
import scripter
from scripter import assert_path, path_to_executable, Usage, exit_on_Usage, InvalidFileException, get_logger,
    Environment, critical, debug
from seriesoftubes.tubes.polledpipe import PolledPipe
from seriesoftubes.tubes import wait_for_job
from seriesoftubes.fnparsers import BowtieFilenameParser

from pkg_resources import get_distribution
__version__ = get_distribution('seriesoftubes').version

VERSION = __version__

def main():
    e = Environment(version=VERSION, doc=_doc_)
    e.set_filename_parser(BowtieFilenameParser)
    # let bowtie do the multiprocessing
    e.override_num_cpus(1)
    parser = e.argument_parser
    parser.add_argument('−−path−to−bowtie', nargs='?',
        default=path_to_executable('bowtie', '/usr/local
             /bowtie∗',
             environ='SOT_PATH_TO_BOWTIE
             '),
        help='The path to the bowtie executable')
    parser.add_argument('−−path−to−samtools', nargs='?',
        default=path_to_executable('samtools', '/usr/local/samtools∗',
             environ='SOT_PATH_TO_SAMTOOLS'),
        help='The path to the samtools executable')
    # fix aliases, should be −−ref too
    parser.add_argument('−−reference', dest='references', action='append '
        ,
        help=dedent(''
            Reference genome to align against (either a bowtie index name or
            file, or a
            fasta file). This flag may be called multiple times (which will
            cause each
            reference to be aligned to separately). If no references are
            specified,
            '))
we'll look for environment variable SOT_DEFAULT_REFERENCES, which
should be given as a list, e.g. "foo foo2 foo3"
)
parser.add_argument('—no—unique', dest='unique', action='store_false',
    help='do not produce unique/ alignment folder')
parser.add_argument('—no—random', dest='random', action='store_false',
    help='do not produce random/ alignment folder')
parser.add_argument('—ignore—quality', dest='use_quality',
    action='store_false',
    help=dedent('')
Use —v mode with bowtie, allows only n mismatches total. Also applies to
counter—references if any are called
)
parser.add_argument('—mismatches', default='2',
    help=dedent('')
allow n mismatches, in the seed (default) or total if —ignore—quality

(—v mode))
parser.add_argument('—quals—type', default='solexa1.3',
    choices=['solexa', 'solexa1.3', 'phred64', 'phred33',
        'integer'],
    help='Valid options are integer, solexa1.3, solexa, phred33, or phred64 (see bowtie for more info)')
parser.add_argument('—max—quality', default='70',
    help=dedent('')
specify maximum quality scores of all mismatched positions (default is 70),
ignored in —ignore—quality (—v mode))
parser.add_argument('—seed—length', dest='seed_len', default='28',
    help='use seed length of m (default is 28)')
cparser = parser.add_argument_group('counter—alignments',
    description=dedent('')
specify counter—reference genome(s)/sequence(s) to use for filtering out
unwanted reads)
parser.add_argument('—counter—reference', dest='counter_references',
    action='append',
    help=dedent('')
Optional counter—reference genome/sequences to align against (either a bowtie
index name or file, or a fasta file). This flag may be called multiple times.
All counter—references will be concatenated into one index, and reads will
be aligned in —no—unique (—M 1) mode. Any reads which align will be saved
in a separate directory called 'bad_reads' and not aligned against
the
reference genomes/sequences. If no counter-references are specified,
we'll
look for environment variable SOT_DEFAULT_COUNTER_REFERENCES,
which should be given as a list, e.g. "foo foo2 foo3"
)
cparser.add_argument('−−counter−mismatches', default=0,
    help=dedent('''
allow n mismatches to counter-reference(s), in the seed (default) or
total
if −−ignore−quality (−v mode). Default: same as references'''))
cparser.add_argument('−−counter−max−quality', default='70',
    help=dedent('''
specify maximum quality scores of all mismatched positions when
aligning to
counter-reference(s) (default is 70), ignored in −−ignore−quality (−
v) mode'''))
context = e. get_context()
new_references = validate_references(**context)
new_counter_references = cat_counter_references(**context)
e.update_context({'references': new_references,
    'counter_references': new_counter_references})
e.do_action(align)
def fastq_to_bowtie(fasta_file, target_dir=curdir, path_to_bowtie='bowtie'):
    """ given a filename, makes a bowtie index
    if that file is a FASTA file
    """
    if exists(fasta_file):
        f = open(fasta_file, 'rU')
        for line in f:
            if line.startswith('#'): continue
            elif line.startswith('>'):  
                args = [path_to_bowtie + '−−build', fasta_file,
                       join(target_dir, fasta_file)]
                debug(''.join(args))
                P = Popen(args, stdout=open(devnull, 'w'), stderr=PIPE)
                stdout = P.communicate()[1]
                if len(stderr.splitlines()) == 0:
                    return join(getcwd(), target_dir, fasta_file)
                else: return join(getcwd(), target_dir, fasta_file)
        return None
    def find_bowtie_index(r, path_to_bowtie='bowtie'):
        """ check for bowtie index as given. return True if found, else
        return False """
        args = [path_to_bowtie + '−−inspect', '−v', '−s', r]
        if path_to_bowtie not in args:
            return False
        f = open(fasta_file, 'rU')
        for line in f:
            if line.startswith('#') or line.startswith('>'):
                continue
            elif line.startswith('>'):
                args = [path_to_bowtie + '−−inspect', '−v', '−s', r]
                debug(''.join(args))
                P = Popen(args, stdout=open(devnull, 'w'), stderr=PIPE)
                stdout = P.communicate()[1]
                if len(stderr.splitlines()) == 0:
                    return join(getcwd(), target_dir, fasta_file)
                else: return join(getcwd(), target_dir, fasta_file)
        return None
debug(' '.join(args))
P = Popen(args, stdout=open(devnull, 'w'), stderr=PIPE, cwd=mkdtemp())
stderr = P.communicate()[1].splitlines()
if not stderr[0].startswith('Could not locate '):
    for line in stderr:
        if line.startswith('Opening '):
            index_ebwt1 = line[(1 + line.find('"')): line.rfind('"')]
            index_basename = index_ebwt1[0: index_ebwt1.find('.1.ebwt ')]
            return index_basename
rprime = join(getcwd(), r)
args = [path_to_bowtie + '-inspect', '-v', '-s', rprime]
debug(' '.join(args))
P = Popen(args, stdout=open(devnull, 'w'), stderr=PIPE, cwd=mkdtemp())
stderr = P.communicate()[1].splitlines()
if not stderr[0].startswith('Could not locate '):
    for line in stderr:
        if line.startswith('Opening '):
            index_ebwt1 = line[(1 + line.find('"')): line.rfind('"')]
            index_basename = index_ebwt1[0: index_ebwt1.find('.1.ebwt ')]
            return index_basename
return None

def cat_counter_references(counter_references=None, target_dir=curdir, path_to_bowtie='bowtie', logger=None, **kwargs):
    if counter_references is None: return try: makedirs(target_dir, mode=0755) except OSError: pass
debug('Validating counter-references and building counter-reference index')
valid_references = validate_references(
    references=counter_references,
    target_dir=target_dir,
    path_to_bowtie=path_to_bowtie,
    logger=logger,
    environ_key='SOT_DEFAULT_COUNTER_REFERENCES')
crefs_fa = open(join(target_dir, 'counter_references.fa'), 'w')
for ref in counter_references:
    Popen([path_to_bowtie + '-inspect', ref], stdout=crefs_fa).wait()
crefs_index = join(target_dir, counter_references)
args = [path_to_bowtie + '-build', crefs_fa, crefs_index]
P = Popen(args, stderr=PIPE)
stderr = P.communicate()[1]
if stderr.startswith('Error '):
critical(stderr)
critical('No counter-references will be used.')
return crefs_index

def validate_references(references=None, path_to_bowtie='bowtie',
                        logger=None, environ_key='SOT_DEFAULT_REFERENCES',
                        target_dir=curdir,
                        **kwargs):
makedirs(target_dir, mode=0755)
debug('Validating references')
new_references = []
if references is None:
    if environ.has_key(environ_key):
        references = environ[environ_key].split()
    else:
        critical('no reference genomes specified')
        return []
for r in references:
bowtie_index = find_bowtie_index(r, path_to_bowtie=path_to_bowtie)
if bowtie_index is None:
    if exists(r):
        debug('Attempting to build bowtie index from %s' % r)
        new_index = fastq_to_bowtie(r, target_dir=target_dir,
                                    path_to_bowtie=path_to_bowtie)
        if new_index is not None:
            new_references.append(new_index)
            continue
    else:
        critical('Failed to build bowtie index.')
        critical('bowtie could not find the index for %s', r)
    else:
        new_references.append(bowtie_index)
        return new_references
@exit_on_Usage
def align(fp_obj, references=[], counter_references=None,
          random=True, unique=True, max_quality='70',
          quals_type='solexa1.3', mismatches='2', seed_len='28',
          counter_mismatches=None,
          use_quality=True, logging_level=10, num_cpus=1, **kwargs):
    if not unique and not random: raise Usage('Nothing to do')
    common_flags = [ '-y', '-a', '--time', '--best', '--chunkmbs', '1024',
                    '--strata', '--sam']
    common_flags.extend([('-p', str(num_cpus))])
    logger = get_logger(logging_level)
    uniqueness = {}
    if unique: uniqueness.update({'unique': ['--m', '1']})
if random: uniqueness.update({'random': ['-M', '1']})

stdout_buffer = []
common_flags.extend(['-l', seed_len])

new_sources = []
if counter_references is not None:
    #counter_align_first
    flags = [item for item in common_flags]
    flags.extend(uniqueness['random'])
    if counter_mismatches is None: counter_mismatches = mismatches
    if use_quality:
        flags.extend(['-e', max_quality])
        flags.extend(['-n', counter_mismatches])
    else:
        flags.extend(['-v', counter_mismatches])

if fp_obj.paired_end: flags.extend(['-X', '600'])
new_filenames = counteralign_once(fp_obj, flags, counter_references, logger=logger, **kwargs)

# after alignment
fp_obj.input_file = new_filenames[0]
fp_obj.second_file = new_filenames[1]
fp_obj.use_pysam = True
fp_obj.format = 'BAM'

for match_type, match_flag in uniqueness.items():
    flags = [item for item in common_flags]
    flags.extend(match_flag)

    source = fp_obj.fastq_source
    # In case we're from that bad day...
    if source == '081124_HWI-EAS355_0001_Meghan' and match_type == 'unique':
        flags.extend(['-v', '3'])
    # elif use_quality:
    #    if use_quality:
    #        flags.extend(['-e', max_quality])
    #        flags.extend(['-n', mismatches])
    #    else:
    #        flags.extend(['-v', mismatches])

    if fp_obj.paired_end: flags.extend(['-X', '600'])

    for ref in references:
        s = align_once(fp_obj, flags, ref, match_type, logger=logger , **kwargs)
        stdout_buffer.append(s)

    return '
'.join([s for s in stdout_buffer if s is not None])
def counteralign_once(fp_obj, flags, ref, **kwargs):
    """Produce counter-alignments""
    refname = os.path.basename(ref)
    output_dir, output_file = os.path.split(fp_obj.tmp_filename(refname))

    fp_obj.check_output_dir(output_dir)
    filename1 = os.path.abspath(fp_obj.input_file)
    second_file = fp_obj.second_file
    if second_file is not None: filename2 = os.path.abspath(second_file)
    else: filename2 = None
    if use_quality:
        if fp_obj.use_pysam: flags.append('--phred33quals')
        else: flags.append(''.join([f'--{quals_type}', f'--quals']))
    if fp_obj.paired_end:
        file_args = [ref, '--12', '-']
    else:
        file_args = [ref, '-']
    bowtie_args = [path_to_bowtie] + flags + file_args

    # finish parsing input here
    input_stderr = PolledPipe(logger=logger, level=logging.ERROR)
    bowtie_stderr = PolledPipe(logger=logger, level=logging.ERROR)
    if fp_obj.use_pysam:
        if fp_obj.paired_end:
            in_args = [sys.executable, '-m', 'seriesoftubes.converters.bamtotab',
                       filename1]
        else:
            in_args = [sys.executable, '-m', 'seriesoftubes.converters.bamtofastq',
                       '--no-gzip',
                       filename1]
    elif fp_obj.paired_end and fp_obj.format == 'FASTQ':
        in_args = [sys.executable, '-m', 'seriesoftubes.converters.fastqtotab',
                   filename1, filename2]
    elif fp_obj.format == 'FASTQ':
        in_args = [sys.executable, '-m', 'seriesoftubes.converters.cat',
                   filename1]
    else:
        logger.critical("Couldn't figure out what to do with file %s of format %s",
                         fp_obj.input_file, fp_obj.format)
    log.info(' '.join(in_args))
    input_reader = Popen(in_args, stdout=PIPE, stderr=input_stderr.w,
                         bufsize=-1)
    log.info('Launching bowtie (output will be piped to samtools)')
    log.info(' '.join(bowtie_args))
bowtie_aligner = Popen(bowtie_args, stdin=input_reader.stdout, stdout=PIPE, stderr=bowtie_stderr.w, bufsize=-1)
logger.info("Only unaligned reads will be saved.")

samtools_args = [path_to_samtools, 'view', '-b', '-S', '-o', '-f', '0x4', '# ONLY SAVE UNALIGNED READS
join(output_dir, output_file), ' - ']
logger.info(\'Launching samtools to encode bowtie output as BAM\')
samtools_viewer = Popen(samtools_args, stdin=bowtie_aligner.stdout, stdout=samtools.stdout.w, stderr=samtools.stderr.w, bufsize=-1)

logger.debug("Waiting for bowtie to finish")
pollables = [input_stderr, bowtie_stderr, samtools.stdout, samtools.stderr]
wait_for_job(bowtie_aligner, pollables, logger)

if not bowtie_aligner.returncode == 0:
llogger.critical("bowtie did not run properly [%d]", bowtie_aligner.returncode)
samtools_viewer.terminate()
samtools_viewer.poll()
logger.critical("samtools terminated")
return

logger.debug("Alignment successfully completed")
logger.debug("Waiting for samtools to finish")
wait_for_job(samtools_viewer, [samtools.stdout, samtools.stderr], logger)

if not samtools_viewer.returncode == 0:
logger.critical("samtools view did not run properly [%d]", samtools_viewer.returncode)
return

logger.debug("Unsorted BAM file successfully written")

return (join(output_dir, output_file), None)

align_once(fp_obj, flags, ref, match_type, use_quality=False, quals_type='solexa1.3', path_to_bowtie=None, path_to_samtools=None, logger=None, **kwargs):
refname = os.path.basename(ref)
path_to_unsorted = fp_obj.tmp_filename(refname, match_type)
output_dir = os.path.split(path_to_unsorted)[0]
fp_obj.check_output_dir(output_dir)
filename1 = os.path.abspath(fp_obj.input_file)
second_file = fp_obj.second_file
if second_file is not None: filename2 = os.path.abspath(second_file)
else: filename2 = None
if use_quality:
    if fp_obj.use_pysam: flags.append('--phred33-quals')
    else: flags.append(''.join(['--', quals_type, '-quals']))
if fp_obj.paired_end:
    file_args = [ref, '--12', '-1']
    logger.info('Automagically interpreting %s files', fp_obj.format)
else:
    logger.info('Automagically interpreting %s file', fp_obj.format)
    file_args = [ref, '-1']
bowtie_args = [path_to_bowtie] + flags + file_args

# finish parsing input here
input_stderr = PolledPipe(logger=logger, level=logging.ERROR)
bowtie_stderr = PolledPipe(logger=logger, level=logging.ERROR)
if fp_obj.use_pysam:
    if fp_obj.paired_end:
        in_args = [sys.executable, '-m', 'seriesoftubes.converters.bamtotab',
                   filename1]
    else:
        in_args = [sys.executable, '-m', 'seriesoftubes.converters.bamtofastq',
                   '--no-gzip',
                   filename1]
else:
    in_args = [sys.executable, '-m', 'seriesoftubes.converters.fastqtotab',
               filename1, filename2]
elif fp_obj.paired_end and fp_obj.format == 'FASTQ':
    in_args = [sys.executable, '-m', 'seriesoftubes.converters.fastqtotab',
               filename1, filename2]
elif fp_obj.format == 'FASTQ':
    in_args = [sys.executable, '-m', 'seriesoftubes.converters.cat',
               filename1]
else:
    logger.critical("Couldn’t figure out what to do with file %s of format %s",
                    fp_obj.input_file, fp_obj.format)
    logger.info(''.join(in_args))
    input_reader = Popen(in_args, stdout=PIPE, stderr=input_stderr.w, bufsize=-1)
    logger.info('Launching bowtie (output will be piped to samtools)')
    logger.info(''.join(bowtie_args))
bowtie_aligner = Popen(bowtie_args, stdin=input_reader.stdout, stdout=PIPE, stderr=bowtie_stderr.w,
                          bufsize=-1)
    samtools_args = [path_to_samtools, 'view', '-b', '-S', '-o',
                     path_to_unsorted, '--']
    logger.info('Launching samtools to encode bowtie output as BAM')
    logger.info(''.join(samtools_args))
samtools_stdout = PolledPipe(logger=logger, level=logging.WARN)
samtools_stderr = PolledPipe(logger=logger, level=logging.ERROR)
samtools_viewer = Popen(samtools_args, stdin=bowtie_aligner.stdout, stdout=samtools_stdout.w, stderr=samtools_stderr.w, bufsize=-1)

logger.debug('Waiting for bowtie to finish')
pollables = [input_stderr, bowtie_stderr, samtools_stdout, samtools_stderr]
wait_for_job(bowtie_aligner, pollables, logger)

if not bowtie_aligner.returncode == 0:
    logger.critical("bowtie did not run properly [%d]", bowtie_aligner.returncode)
samtools_viewer.terminate()
samtools_viewer.poll()
    logger.critical("samtools terminated")
    return

logger.debug('Alignment successfully completed')
logger.debug('Waiting for samtools to finish')
wait_for_job(samtools_viewer, [samtools_stdout, samtools_stderr], logger)
if not samtools_viewer.returncode == 0:
    logger.critical("samtools view did not run properly [%d]", samtools_viewer.returncode)
    return

logger.debug('Unsorted BAM file successfully written')

logger.info('Launching samtools again to sort BAM output')
output_dir, output_file = os.path.split(path_to_unsorted)
bam_file = os.path.splitext(output_file)[0]
sorter_args = [path_to_samtools, 'sort', output_file, bam_file]
logger.info(''.join(sorter_args))
samtools_stdout = PolledPipe(logger=logger, level=logging.WARN)
samtools_stderr = PolledPipe(logger=logger, level=logging.ERROR)
samtools_sorter = Popen(sorter_args, stdout=samtools_stdout.w, stderr=samtools_stderr.w, cwd=output_dir)
wait_for_job(samtools_sorter, [samtools_stdout, samtools_stderr, samtools_sorter], logger)
if not samtools_sorter.returncode == 0:
    logger.critical("samtools sort did not run properly [%d]", samtools_sorter.returncode)
    return

# don't destroy the files until we're sure we succeeded!
assert_path(os.path.join(output_dir, bam_file + '.bam'))
logger.debug('Removing unsorted file %s', path_to_unsorted)
os.remove(path_to_unsorted)
logger.debug('Launching samtools again to index sorted BAM output')

samtools_stdout = PolledPipe(logger=logger, level=logging.WARN)
samtools_stderr = PolledPipe(logger=logger, level=logging.ERROR)

index_args = [path_to_samtools, 'index', bam_file + '.bam']
samtools_indexer = Popen(index_args, stdout=samtools_stdout.w, stderr=samtools_stderr.w, cwd=output_dir)

wait_for_job(samtools_indexer, [samtools_stdout, samtools_stderr], logger)

if not samtools_indexer.returncode == 0:
    logger.critical("samtools index did not run properly [%d]", samtools_indexer.returncode)
    return

# Make sure indexing succeeds
assert_path(os.path.join(output_dir, bam_file + '.bam.bai'))

if __name__ == '__main__':
    main()
```python
def main():
    e = scripter.Environment(version=VERSION, doc=__doc__)
    parser = e.argument_parser
    parser.add_argument('--sam-out', action='store_true',
                        help='Output SAM files (not BAM)')
    parser.add_argument('--remove-all', action='store_true',
                        help='Remove all reads in mapped BAM/SAM file, not just mapped ones')
    parser.set_defaults(**{'target': 'filtered'})
    e.set_filename_parser(SubtractBamFilenameParser)
    e.do_action(remove_reads)

def is_mapped(read):
    # true if a pysam.AlignedRead is properly mapped, otherwise false
    # (checks if read is paired end; if so, returns if the pair is mapped)
    if read.is_paired:
        return read.is_proper_pair
    else:
        return not read.is_unmapped

def remove_reads(parsed_filename, remove_all=False, sam_out=False,
                  debug=False, **kwargs):
    # note: you must be looking at a sorted file, or this won't work
    if sam_out: write_opts = 'w'
    else: write_opts = 'wb'

    with pysam.Samfile(parsed_filename.mapped_file) as mapped:
        if remove_all:
            reads_to_remove = set([read.qname for read in mapped])
        else:
            reads_to_remove = set([read.qname for read in mapped
                                    if is_mapped(read)])

        if debug:
            scripter.debug('Found {ls} reads in {ls}'.format(len(reads_to_remove),
                                                             parsed_filename.mapped_file))

        with pysam.Samfile(parsed_filename.input_file) as bam_file:
            with pysam.Samfile(parsed_filename.output_file, write_opts, 
                               template=bam_file) as out_bam_file:
                for read in bam_file:
                    if read.qname not in reads_to_remove:
                        out_bam_file.write(read)

    return
```

269
class SubtractBamFilenameParser(FilenameParser):
    def __init__(self, filename, sam_out=False, *args, **kwargs):
        super(SubtractBamFilenameParser, self).__init__(filename, 
            sam_out=sam_out, 
            *args, 
            **kwargs)
        fext = os.path.splitext(filename)[1].rstrip(os.extsep)
        if not (fext == 'sam' or fext == 'bam'): raise InvalidFileException
        if not self.is_dummy_file:
            # check for the mapped file
            input_dir_parts = self.input_dir.split(os.sep)
            glob_path = ['mapped', input_dir_parts[0], '*'] + 
                     input_dir_parts[2:] + 
                     [os.path.basename(self.input_file)]
            potential_filenames = glob.glob(os.sep.join(glob_path))
            if len(potential_filenames) == 1:
                self.mapped_file = potential_filenames[0]
            elif len(potential_filenames) == 0:
                raise scripter.Usage('Could not find mapped file')
            else:
                raise scripter.Usage('Ambiguous mapped file', * 
                    potential_filenames)
            scripter.debug('Mapped file will be', self.mapped_file)
            if sam_out:
                self.output_file = os.path.join(
                    self.output_dir, 
                    self.with_extension('sam'))
            else:
                self.output_file = os.path.join(
                    self.output_dir, 
                    self.with_extension('bam'))
            scripter.debug('Output file will be', self.output_file)

if __name__ == '__main__': main()
from itertools import repeat

def in_window_factory(chrom, start, end):
    """
    function factory
    returns a function that checks if a bedgraph line overlaps a given
    interval. along with UCSC convention, end coordinate is open and
    start coordinate is 0-based
    (also compatible with BED files)
    """
    def in_window(line):
        parts = line.rstrip('
').split()
        if len(parts) < 3: return False
        this_chrom, this_start, this_end = parts[0:3]
        if not this_chrom == chrom: return False
        elif not int(this_start) < end: return False
        elif not int(this_end) > start: return False
        else: return '\t'.join(parts)
    return in_window

def in_windows_factory(intervals):
    """
    function factory
    returns a function that checks if a bedgraph line overlaps any given
    interval. assumes we already filtered by chrom
    intervals must be given as tuples (int start, int end)
    along with UCSC convention, end coordinate is open and start
    coordinate is 0-based
    (also compatible with BED files)
    """
    for rec in intervals:
        rec = 0
    return in_window

def window_to_iter(chrom, start, end, filename, my_type=int):
    """
    generator that returns the values in a region [start,end) on chrom
    from a bedgraph file called filename
    note: bedgraph values are assumed to be ints unless my_type is set
    to something else
    """
    in_window = in_window_factory(chrom, start, end)
    output = filter(in_window, open(filename, 'rU'))
50 if len(output) == 0:
51     for i in repeat(0, end - start):
52         yield i

53 first_range = output[0].rstrip('\n').split('	')
54 last_range = output[0].rstrip('\n').split('	')

56 previous_end = start
57 for line in output:
58     this_range = line.rstrip('\n').split()
59     this_start = int(this_range[1])
60     for i in xrange(previous_end, this_start): yield 0
61     this_end = int(this_range[2])
62     this_value = my_type(this_range[3])
63     for i in xrange(this_start, this_end): yield this_value
64     previous_end = this_end
65
67 for i in xrange(last_end, end): yield 0
raise StopIteration

Source Code C.7: seriesoftubes.bedgraph module

C.2.2 seriesoftubes.cPreprocess module (Cython extension)

# cython: profile=True
from cpython cimport bool
from libc.stdlib cimport malloc, free

DEF ERR_TOO_SHORT = 1
DEF ERR_LINKER = 2
DEF MAX_FASTQ_SIZE = 131072

cdef struct int_pair:
    int left
    int right

def FasterFastqIterator(fastq_handle):
    """Cython-ized version of FastqGeneralIterator""

cdef:
    str line, title_line, seq_string, second_title, quality_string
    int seq_len
    #We need to call handle.readline() at least four times per record,
    #so we’ll save a property look up each time:
    #Skip any text before the first record (e.g. blank lines, comments?)
    handle_readline = fastq_handle.readline
    while True:
        line = handle_readline()
# if line == "": return #Premature end of file, or just empty?
    if line[0] == "@":
        break

while True:
    if line[0] != "@":
        raise ValueError("Records in Fastq files should start with ' @' character")
    title_line = line[1:].rstrip()
    #Will now be at least one line of quality data — in most FASTQ files
    #just one line! We therefore use string concatenation (if needed)
    #rather using than the "".join(...) trick just in case it is multiline:
    seq_string = handle_readline().rstrip()
    #There may now be more sequence lines, or the "+" quality marker line:
    while True:
        line = handle_readline()
        if line == "":
            raise ValueError("End of file without quality information.")
        ## disable second title checking
        if line[0] == "+":
            #The title here is optional, but if present must match!
            second_title = line[1:].rstrip()
            # if second_title and second_title != title_line:
                raise ValueError("Sequence and quality captions differ.")
            break
        seq_string += line.rstrip() #removes trailing newlines
        #This is going to slow things down a little, but assuming
        #this isn't allowed we should try and catch it here:
        ## disable character checking
        if " " in seq_string or "\t" in seq_string:
            raise ValueError("Whitespace is not allowed in the sequence.")
    seq_len = len(seq_string)

    #Will now be at least one line of quality data...
    quality_string = handle_readline().rstrip()
    #There may now be more quality data, or another sequence, or EOF
    while True:
        line = handle_readline()
        if line == "": break #end of file
        if line[0] == "@":
            #This COULD be the start of a new sequence. However, it MAY just
            #be a line of quality data which starts with a "@" character. We
#should be able to check this by looking at the sequence length
#and the amount of quality data found so far.
if len(quality_string) >= seq_len:
    #We expect it to be equal if this is the start of a new record.
    #If the quality data is longer, we'll raise an error below.
    break
    #Continue - its just some (more) quality data.
quality_string += line.rstrip()

if seq_len != len(quality_string):
    raise ValueError("Lengths of sequence and quality values differs 
                 " for %s (%i and %i)." \ % (title_line, seq_len, len(quality_string))
         )

    #Return the record and then continue...
yield (title_line, seq_string, quality_string)
if line == '': return  #StopIteration at end of file
#    assert False, "Should not reach this line"

cdef struct Read:
    char *title
    char *seq
    char *qual

    #cdef Read asRead(str title, str seq, str qual):
    #    cdef Read read
    #    read.title = <bytes>title
    #    read.seq = <bytes>seq
    #    read.qual = <bytes>qual
    #    return read

cdef struct Record:
    char *barcode
    Read *read

    #cdef Record asRecord(str barcode, str title, str seq, str qual):
    #    cdef Record record
    #    record.barcode = <bytes>barcode
    #    record.read = asRead(title, seq, qual)
    #    return record

    #cdef struct s_RecordPair:
    #    Record *first
    #    Record *second
    #
    #ctypedef s_RecordPair RecordPair
#cdef RecordPair pair(Record record1, Record record2):
#    cdef RecordPair rp
#    rp.first = record1
#    rp.second = record2
#    return rp

cdef bytes match_barcode(bytes seq, list barcodes, int mismatches=1):
    """
    try to match seq to a list of barcodes
    allow mismatches (default 1)
    returns the match (from barcodes) or None
    """
    cdef:
        list barcode_lengths = map(len, barcodes)
        int n_barcodes = len(barcodes)
        int max_barcode_length = max(barcode_lengths)
        int seq_len = len(seq)
        int barcode_length
        bytes accepted = b''
        bytes barcode
        int hamming, i, j
        for i in range(n_barcodes):
            barcode = barcodes[i]
            barcode_length = barcode_lengths[i]
            if seq_len < barcode_length: continue
            hamming = 0
            for j in range(barcode_length):
                if barcode[j] != seq[j]: hamming += 1
                if mismatches > hamming:
                    if accepted == b'': accepted = barcode
                    else:
                        accepted = b''
                    break
        return accepted

cdef void pretrim_read_5prime(Read *read, int trim_length):
    """
    trim an assigned record from the 5' end
    expects (barcode, (title, seq, qual))
    """
    read[0].seq += trim_length
    read[0].qual += trim_length

cdef void trim_read_5prime(Read *read, int trim_length):
    """
    trim an assigned record from the 5' end
    expects (barcode, (title, seq, qual))
    """
    read[0].seq += trim_length
    read[0].qual += trim_length
cdef void truncate_read(Read *read, int max_length):
    
    truncate a record so that it is at most max_length
    starting at the 5' end
    expects (barcode, (title, seq, qual))
    
    read[0].seq[max_length] = b'\0'
    read[0].qual[max_length] = b'\0'

cdef void trim_read_3prime(Read *read, int trim_length):
    
    trim a record from the 3' end
    expects (barcode, (title, seq, qual))
    
    if trim_length != 0:
        read[0].seq[-trim_length] = b'\0'
        read[0].qual[-trim_length] = b'\0'

# Violates NCBI SRA requirements

#cdef Record trim_trailing_Ns(Record record):
    
    # trim any trailing 3' 'N's
    # expects record is (barcode, (title, seq, qual))
    # returns truncated (barcode, (title, seq, qual))
    
    # cdef int end = len(str.rstrip(record.read.seq, 'N'))
    # record.read.seq[end] = b'\0'
    # record.read.qual[end] = b'\0'
    # return record

cdef void cleave_linker(Record *record, char *linker):
    cdef:
        Read *read = record[0].read
        int i
        i = str.find(read.seq, linker)
        if i != -1:
            record[0].barcode = linker
            read[0].seq[i] = b'\0'
            read[0].qual[i] = b'\0'

cdef Read as_read(bytes title, bytes seq, bytes qual):
    cdef Read read
    read.title = title
    read.seq = seq
    read.qual = qual
    return read

cdef Read *as_read2(tuple t):
    cdef Read read
    read.title = <bytes>t[0]
    read.seq = <bytes>t[1]
cdef bytes assign_read(Read *read, list barcodes):
    r""
    Assign a record to a barcode
    returns (barcode, new_record)
    if unmatched, returns (None, record)
    r""
    title, seq, qual = record/
    title_head, last_part = title.rsplit(':', 1)
    cdef:
        bytes title_head, last_part
        int pound_loc, slash_loc
        bytes barcode
        int barcode_len
        int n_barcodes = len(barcodes)
        if n_barcodes == 0:
            barcode = b''
        else:
            title_head, last_part = read.title.rsplit(':', 1)
            if last_part.isalpha():
                # CASAVA 1.8 file
                barcode = match_barcode(<bytes>last_part.rstrip(), barcodes)
            else:
                pound_loc = last_part.find('#')
                slash_loc = last_part[pound_loc:].find('/')
                if slash_loc != -1:
                    barcode = <bytes>last_part[(pound_loc+1):(pound_loc+slash_loc)]
                else:
                    barcode = <bytes>last_part[(pound_loc+1):]
                    if barcode==b'0':
                        barcode = match_barcode(<bytes>read.seq, barcodes)
                        barcode_len = len(barcode)
                        read.seq += barcode_len
                        read.qual += barcode_len
                elif barcode.isdigit():
                    # then we have a numbered index from Illumina, just use it as-is
                    #record.barcode = barcode
                    pass
                elif barcode.isalpha():
                    # then we already extracted the barcode at some point, try to match it
                    barcode = match_barcode(barcode, barcodes)
            else:
                barcode = b''
        return barcode

cdef bytes assign_read_no_clip(Read *read, list barcodes, int offset):
###
Assign a record to a barcode
returns (barcode, new_record)
if unmatched, returns (None, record)
###
# title, seq, qual = record/
# title_head, last_part = title.rsplit(':', 1)
cdef:
    bytes title_head, last_part
    int pound_loc, slash_loc
    bytes barcode = b''
    int barcode_len
    int n_barcodes = len(barcodes)

if n_barcodes == 0:
    barcode = b''
else:
    title_head, last_part = read[0].title.rsplit(':', 1)
    if last_part.isalpha():
        # CASAVA 1.8 file
        barcode = match_barcode(last_part.rstrip(), barcodes)
    else:
        pound_loc = last_part.find('#')
        slash_loc = last_part[pound_loc:].find('/')
        if slash_loc != -1:
            barcode = last_part[(pound_loc+1):(pound_loc+slash_loc)]
        else:
            barcode = last_part[(pound_loc+1):]
    if barcode == b'0':
        barcode = match_barcode(read.seq + offset, barcodes)
        barcode_len = len(barcode)
    elif barcode.isdigit():
        # then we have a numbered index from Illumina, just use it as-is
        pass
    elif barcode.isalpha():
        # then we already extracted the barcode at some point,
        # try to match it
        barcode = match_barcode(barcode, barcodes)
    else:
        barcode = b''
return barcode

cdef bool too_short(Read *read, int min_length):
    cdef char c
    cdef int i = 0, L = 0
    for c in read.seq:
        L += 1
        if c == b'N': i += 1
    return L - i < min_length
cdef int write_record(Record *record, barcoded_files=None,
unmatched_file=None,
processed_file=None,
str linker='', int min_length=4):
    
    write the assigned_record to the correct barcode file
an assigned_record is a (barcode, (title, seq, qual)) (all strs)

will not work unless you provide a dictionary of barcoded_files and
an unmatched_file object

    cdef:
        str barcode = record.barcode
        str title = record.read.title
        str seq = record.read.seq
        str qual = record.read.qual
        str line = "@%s
        +%s
        %s
        \n"
    %(title, seq, title, qual)

    bool is_linker = (len(seq) == 0 and
        barcode == linker
        and not barcode == '')
    bool is_too_short = too_short(record.read, min_length)

    if is_linker: return ERR_LINKER
    elif is_too_short: return ERR_TOO_SHORT

    # produce lines
    if is_processed: processed_file.write(line)
    elif no_barcode: unmatched_file.write(line)
    else: barcoded_files[barcode].write(line)
    return 0

cdef int_pair write_record_pair(Record *record, Record *record2,
object barcoded_file_pairs=None,
object unmatched_files=None,
object processed_files=None,
object orphaned_read_files=None,
object mismatched_files=None,
str linker='', int min_length=4):

    write the assigned_record to the correct barcode file
an assigned_record is a (barcode, (title, seq, qual)) (all strs)

will not work unless you provide a dictionary of barcoded_file_pairs
and
unmatched_files and mismatched_files (2-tuples of file object)

    cdef:
int_pair ret

str barcode, title, seq, qual, line
int problem = 0, problem2 = 0
bool is_linker1, is_too_short1

str barcode2, title2, seq2, qual2, line2
bool is_linker2, is_too_short2
bool is_processed = (barcoded_file_pairs is None)

barcode = record.barcode
title = record.read.title
seq = record.read.seq
qual = record.read.qual
line = "@%s
%s
+%s
%s"
%(title, seq, title, qual)
is_linker1 = len(seq) == 0 and (not linker == '') and barcode == linker
is_too_short1 = too_short(record.read, min_length)

barcode2 = record2.barcode
title2 = record2.read.title
seq2 = record2.read.seq
qual2 = record2.read.qual
line2 = "@%s
%s
+%s
%s"
%(title2, seq2, title2, qual2)
is_linker2 = len(seq2) == 0 and (not linker == '') and barcode2 == linker
is_too_short2 = too_short(record2.read, min_length)

# check if record 1 is valid
if is_linker1: problem = ERR_LINKER
elif is_too_short1: problem = ERR_TOO_SHORT

# check if record 2 is valid
if is_linker2: problem2 = ERR_LINKER
elif is_too_short2: problem2 = ERR_TOO_SHORT
ret.left = problem
ret.right = problem2

# abort write if both reads have problems
if (not problem == 0) or (not problem2 == 0):
    # if only one read has a problem, use write_record instead
    if not problem2 == 0:
        if is_processed: orphaned_read_files[0].write(line)
        else: mismatched_files[0].write(line)
    else:
        if is_processed: orphaned_read_files[1].write(line2)
        else: mismatched_files[1].write(line2)
    return ret

# select output files
if barcoded_file_pairs is None:
```python
output_file = processed_files[0]
output_file2 = processed_files[1]

elif barcode == '' and barcode2 == '':
    output_file = unmatched_files[0]
    output_file2 = unmatched_files[1]

elif not barcode == barcode2:
    output_file = mismatched_files[0]
    output_file2 = mismatched_files[1]

else:
    output_file = barcoded_file_pairs[barcode][0]
    output_file2 = barcoded_file_pairs[barcode][1]

# write and return
output_file.write(line)
output_file2.write(line2)
return ret
```

```python
cpdef dict apply_plan(reads, writer_args, list barcodes=[], str linker=' ',
                        int min_length=4, int max_length = -1,
                        int strip_after_barcode = 1, int strip_before_barcode = 0,
                        bool no_clipping=False,
                        logger=None):

cdef:
    int i=0, n_short=0, n_linker=0
    char *llinker = <bytes>linker
    int result
    Record *record = <Record *>malloc(sizeof(Record))
    Read *read = <Read *>malloc(sizeof(Read))

record.read = read
while True:
    i += 1
    try: t = reads.next()
    except StopIteration: break
    read.title = <char *>malloc(len(t[0]))
    read.seq = <char *>malloc(len(t[1]))
    record.barcode = <char *>malloc(len(t[1]))
    read.qual = <char *>malloc(len(t[2]))
    read.title = <bytes>(t[0])
    read.seq = <bytes>(t[1])
    read.qual = <bytes>(t[2])
    apply_plan_to_read(record,
                        read, barcodes, llinker,
                        min_length, max_length,
                        strip_after_barcode, strip_before_barcode,
                        no_clipping)
    result = write_record(record,
                           writer_args['barcoded_files'],
                           writer_args['unmatched_file'],
                           writer_args['mismatched_file'],
```

281
writer_args['processed_file'],
    linker,
    min_length)

if result == ERR_TOO_SHORT:
    n_short += 1
elif result == ERR_LINKER:
    n_linker += 1
return {'all': i, 'short': n_short, 'linker': n_linker}

cdef dict apply_plan_pe(reads, reads2, writer_args, list barcodes=[],
    str linker='',
    int min_length=4, int max_length = -1,
    int strip_afterBarcode = 1, int strip_beforeBarcode = 0,
    bool no_clipping=False, logger=None):

cdef:
    int i = 0, n_short = 0, n_linker = 0
    char *llinker = <bytes>linker
    int_pair result
    Record *record = <Record *>malloc(sizeof(Record))
    Record *record2 = <Record *>malloc(sizeof(Record))
    Read *read = <Read *>malloc(sizeof(Read))
    Read *read2 = <Read *>malloc(sizeof(Read))

tuple t, t2
    record.read = read
    record2.read = read2
    # Allocate ~1 MB of memory each
    # Should be enough for any sequence???
    read.title = <char *>malloc(MAX_FASTQ_SIZE)
    read.seq = <char *>malloc(MAX_FASTQ_SIZE)
    read.barcode = <char *>malloc(MAX_FASTQ_SIZE)
    read.qual = <char *>malloc(MAX_FASTQ_SIZE)
    read2.title = <char *>malloc(MAX_FASTQ_SIZE)
    read2.seq = <char *>malloc(MAX_FASTQ_SIZE)
    read2.barcode = <char *>malloc(MAX_FASTQ_SIZE)
    read2.qual = <char *>malloc(MAX_FASTQ_SIZE)

while True:
    i += 1
    #
    read = as_read(t[0], t[1], t[2])
    try:
        t = reads.next()
    except StopIteration:
        try: reads2.next()
        except StopIteration: pass
        else: raise SyntaxWarning('More reads left in second file')
        finally: break

    t2 = reads2.next()
    read.title = <bytes>(t[0])
    read.seq = <bytes>(t[1])
    read.qual = <bytes>(t[2])
read2.title = <bytes>(t2[0])
read2.seq = <bytes>(t2[1])
read2.qual = <bytes>(t2[2])

apply_plan_to_read(record, read, barcodes, llinker, min_length, max_length, strip_after_barcode, strip_before_barcode, no_clipping)
apply_plan_to_read(record2, read2, barcodes, llinker, min_length, max_length, strip_after_barcode, strip_before_barcode, no_clipping)

result = write_record_pair(record, record2, writer_args['barcoded_file_pairs'], writer_args['unmatched_files'], writer_args['processed_files'], writer_args['orphaned_read_files'], linker, min_length)
if result.left == ERR_LINKER or result.right == ERR_LINKER:
    n_linker += 1
else:
    if result.left == ERR_TOO_SHORT: n_short += 1
    if result.right == ERR_TOO_SHORT: n_short += 1
    return {'all': i, 'short': n_short, 'linker': n_linker}

cdef void apply_plan_to_read(Record *record, Read *read, list barcodes, char *linker, int min_length, int max_length, int strip_after_barcode, int strip_before_barcode, bool no_clipping):

cdef bytes barcode
if no_clipping:
    barcode = assign_read_no_clip(read, barcodes, strip_before_barcode)
else:
    if strip_before_barcode > 0:
        pretrim_read_5prime(read, strip_before_barcode)
    barcode = assign_read(read, barcodes)
    if strip_after_barcode > 0: trim_read_5prime(read, strip_after_barcode)
    if not linker == b'': cleave_linker(record, linker)
    # record = trim_trailing_Ns(record)
    if max_length >= 0: truncate_read(read, max_length)
    record.barcode = barcode

Source Code C.8: seriesoftubes.cPreprocess module (Cython extension)
C.3 seriesoftubes.converters package

Description

conversion/discover tools for file formats. Most can be run as scripts

C.3.1 seriesoftubes.converters.discover module

```python
from gzip import GzipFile
from bz2 import BZ2File
try:
    from scripter import path_to_executable, Usage
    from subprocess import Popen, PIPE
    from os import pipe
    from io import TextIOBase
    try:
        PATH_TO_GZIP = path_to_executable('gzip')
    except Usage:
        pass
    except ImportError:
        pass
    from sys import stderr

# slow for reading, fast for writing
def gzip_class_factory(path_to_gzip='gzip'):
    class gzip_open_func(object):
        """gzip open func
        modes:
        (r) read using gzip.GzipFile
        (w) write using system gzip
        (P) PIPE from 'gzip -d' for
        """
        def __init__(self, filename, mode='r'):
            self._mode = mode[0]
            if mode[0] == 'w':
                args = [path_to_gzip, '-f', '-c', '-d']
                stdout = open(filename, 'wb')
                self._stdout = stdout
                self.proc = Popen(args, stdin=PIPE, bufsize=0,
                                  stderr=stderr,
                                  stdout=stdout)
                self.write = self.proc.stdin.write
                #
                self.write = self._fakewrite
                self.close = self._close_w
                self.filename = filename
            elif mode[0] == 'r':
                self._file = GzipFile(filename, 'rb')
```
self.read = self._file.read
self.readline = self._file.readline
self.close = self._file.close
else:
    raise NotImplementedError

# def _fakewrite(self, x):
#     print x
#     if x.count(‘\x1a’) > 0:
#         raise ValueError(x)
#     self.proc.stdin.write(x)
def __iter__(self):
    if not self._mode == ‘r’:
        raise IOError(‘cannot iterate over file in write mode’)
    return self._file.__iter__()

def _close_w(self):
    """ returns returncode ""
    self.proc.stdin.flush()
    self._stdout.flush()
    # self.proc.stdout.flush()
    self._stdout.close()
    # self.proc.communicate(‘\x1a’)
    return

def discover_file_format(filename):
    """ discover the format of a file
    returns a tuple (open_function, 'FORMAT')
    open_function will either be open, gzip.GzipFile, bz2.BZ2File, or None
    FORMAT can be 'BAM', 'SAM', 'FASTQ', 'FASTA', or None
    ""
    f = open(filename, ‘rb’)  
    head = f.read(3) 
    f.close()  
    # check magic words for compression
    if head == ‘\x1f\x8b\x08’:
        if PATH_TO_GZIP is None:
            open_func = GzipFile
        else:
            open_func = gzip_class_factory(PATH_TO_GZIP)
    elif head == ‘\x42\x5a\x68’:
        open_func = BZ2File
    else:
        open_func = open
    uncompressed = open_func(filename)
    head2 = uncompressed.read(4)
def main():
    # what to do if we execute the module as a script
    cat
    #
    parser = ArgumentParser(description=\_\_doc\_\_
    parser.add_argument('files', nargs='*', help='files')
    args = parser.parse_args()
    context = vars(args)
    files = context['files']
    for file in files:
        open_func = discover_file_format(file)[0]
for line in open_func(file):
    print line
exit(0)

if __name__ == '__main__': main()
C.3.4 seriesoftubes.converters.bamtofastq module

```python
### Convert BAM/SAM to FASTQ format

@name sequence
+name quality score (phred33)

files may be SAM or BAM (autodetected)
If the file(s) contain paired-end sequences, we will write to two files
(in the current working directory)
If the files contain single end sequences, we will write to stdout by
default
Output is always to stdout (err goes to stderr, redirect it if you need to)

```import pysam
import os
import select
from scripter import path_to_executable
from argparse import ArgumentParser
from copy import copy
from subprocess import Popen, PIPE
from os import mkfifo, getcwd, devnull
from os.path import join, exists, abspath
import subprocess
from sys import argv, stdin, stdout, stderr, exit, executable
from gzip import GzipFile
from .discover import PATH_TO_GZIP, gzip_class_factory

class UnpairedBAMToFastqConverter(object):
    """Works with unpaired SAM/BAM file"""
    def __init__(self, file_, wd=None, stderr=None, logger=None):
        self.require_bam(file_)
        if wd is None: wd = getcwd()
        fifofile = join(wd, '.sot fifo')
        if exists(fifofile):
            raise RuntimeError('%s already exists' % fifofile)
        mkfifo(fifofile)
        self.fifo = abspath(fifofile)
        self._args = [executable, '-m', 'seriesoftubes.converters.bamtofastq',
                      file_, '--single-stdout', fifofile]
        self._logger = logger
        self._stderr = stderr

    def launch(self):
        if self._logger is not None:
            self._logger.info('Launching %s', ' '.join(self._args))
```

288
self.subprocess = Popen(self._args, stdout=open(devnull, 'w'),
                          stderr=self._stderr, bufsize=-1)

def get_fifo_readers(self):
    return (open(self.fifos[0], 'r'), open(self.fifos[1], 'r'))

def require_bam(self, filename):
    with open(filename, 'rb') as f:
        head = f.read(3)
        # check magic words for compression
        if head == b'\x1f\x8b\x08 ':
            open_func = GzipFile
        else:
            open_func = open
        uncompressed = open_func(filename)
        head2 = uncompressed.read(4)
        # check for BAM
        if head2 == b'BAM\x01 ': return
        # check for SAM
        if head2 == b'@HD\t ': return
        else: raise ValueError('Not a SAM/BAM file ')

def main():
    """
    what to do if we execute the module as a script
    """
    parser = ArgumentParser(description=__doc__)
    parser.add_argument('files', nargs='*', help='List of input files ')
    parser.add_argument('--no-gzip', action='store_true', default=False,
                        help='Do not compress output ')
    parser.add_argument('--single-stdout',
                        help='Save single-end reads to here (default: stdout)',
                        default=None)
    args = parser.parse_args()
    context = vars(args)
    read_files(**context)

def pair_writer(out1, out2):
    def writer(read1, read2):
        out1.write('@ %s
 %s
+
 %s
 ' % (read1.qname, read1.seq, read1.qual))
        out2.write('@ %s
 %s
+
 %s
 ' % (read2.qname, read2.seq, read2.qual))
    return writer

def read_files(files=None, no_gzip=False, single_stdout=stdout):
```python
actually reads the SAM/BAM files
for file in files:
    if file is None: continue
    f = pysam.Samfile(file)
    # check if first read is paired
    aread = f.next()
    f.close()
    f = pysam.Samfile(file)
    if aread.is_paired:
        if no_gzip:
            print 'Detected paired-end reads, redirecting output to text files'
            file1 = file + '_1.txt'
            file2 = file + '_2.txt'
            fh1 = open(file1, 'w')
            fh2 = open(file2, 'w')
        elif PATH_TO_GZIP is not None:
            print 'Detected paired-end reads, redirecting output to.gz text files (using system gzip)'
            file1 = file + '_1.txt.gz'
            file2 = file + '_2.txt.gz'
            open_func = gzip_class_factory(PATH_TO_GZIP)
            fh1 = open_func(file1, 'wb')
            fh2 = open_func(file2, 'wb')
        else:
            print 'Detected paired-end reads, redirecting output to.gz text files'
            file1 = file + '_1.txt.gz'
            file2 = file + '_2.txt.gz'
            fh1 = GzipFile(file1, 'wb')
            fh2 = GzipFile(file2, 'wb')
        is_paired = False
        write = pair_writer(fh1, fh2)
        incomplete_pairs = []
        for aread in f:
            is_paired = False
            qname = aread.qname
            for i in xrange(len(incomplete_pairs)):
                if incomplete_pairs[i].qname == qname:
                    mate_read = incomplete_pairs.pop(i)
                    # figure out order
                    if aread.flag & 0x4 == 0x4:
                        write(aread, mate_read)
                    else:
                        write(mate_read, aread)
                    is_paired = True
                    break
            if not is_paired: incomplete_pairs.append(aread)
        unpaired = len(incomplete_pairs)
        if not unpaired == 0:
```

raise RuntimeError('%d unpaired reads remaining' % unpaired)

else:
    if no_gzip:
        open_func = open
    elif PATH_TO_GZIP is not None:
        open_func = gzip_class_factor(PATH_TO_GZIP)
    else:
        open_func = GzipFile
    if single_stdout is None:
        fh1 = stdout
    else:
        fh1 = open_func(single_stdout, 'wb')
    for aread in f:
        qname = aread.qname or ''
        seq = aread.seq or ''
        qual = aread.qual or ''
        rec = '@%s
%sn+%s
' % (qname, seq, qual)
        fh1.write(rec)
    fh1.close()
    exit(0)

if __name__ == '__main__': main()
class PairedBAMToFastqConverter(object):
    """Works with any SAM/BAM file"""
    def __init__(self, file_, wd=None, stderr=None, logger=None):
        self.require_bam(file_)
        if wd is None: wd = getcwd()
        fifofile1 = join(wd, '__sot_fifo1')
        fifofile2 = join(wd, '__sot_fifo2')
        if exists(fifofile1):
            raise RuntimeError('%s already exists' % fifofile1)
        if exists(fifofile2):
            raise RuntimeError('%s already exists' % fifofile2)
        mkfifo(fifofile1)
        mkfifo(fifofile2)
        self.fifos = (abspath(fifofile1), abspath(fifofile2))
        self._args = [executable, '-m', '__module__', file_, fifofile1, fifofile2]
        self._logger = logger
        self._stderr = stderr

    def launch(self):
        if self._logger is not None:
            self._logger.info('Launching %s' % ' '.join(self._args))
        self.subprocess = Popen(self._args, stdout=open(devnull, 'w'),
                                 stderr=self._stderr, bufsize=-1)

    def get_fifo_readers(self):
        return (open(self.fifos[0], 'r'), open(self.fifos[1], 'r'))

    def require_bam(self, filename):
        with open(filename, 'rb') as f:
            head = f.read(3)
            # check magic words for compression
            if head == '\x1f\x8b\x08':
                open_func = GzipFile
            else:
                open_func = open
        uncompressed = open_func(filename)
        head2 = uncompressed.read(4)
        # check for BAM
        if head2 == 'BAM\x01': return
        # check for SAM
        if head2 == '@HD\t': return
        else: raise ValueError('Not a SAM/BAM file')

    def pair_writer(out1, out2):
        def writer(read1, read2):
            POLLOUT = select.POLLOUT
            rec = '@%s
%+ %s
' % (read1.qname, read1.seq, read1.qual)
            N = len(rec) / PIPE_BUF + 1 - (len(rec) % PIPE_BUF == 0)
            i = 0
while True:
    if i == N:
        break
    while True:
        x = select.select([], [out1], [], 2)
        if x[1] == [out1]:
            break
        y = len(rec[i*PIPE_BUF:((i+1)*PIPE_BUF)])
        x = os.write(out1, rec[i*PIPE_BUF:((i+1)*PIPE_BUF)])
        if not x == y:
            raise IOError('partial write')
        i += 1

rec = '@%s
%s
+
%s
' % (read2.qname, read2.seq, read2.qual)
N = len(rec) / PIPE_BUF + 1 - (len(rec) == PIPE_BUF)
i = 0
while True:
    if i == N:
        break
    while True:
        x = select.select([], [out2], [], 2)
        if x[1] == [out2]:
            break
        y = len(rec[i*PIPE_BUF:((i+1)*PIPE_BUF)])
        x = os.write(out2, rec[i*PIPE_BUF:((i+1)*PIPE_BUF)])
        if not x == y:
            raise IOError('partial write')
        i += 1
return writer
def gzwriter(out1, out2):
    def writer(read1, read2):
        out1.write('@%s
%s
+
%s
' % (read1.qname, read1.seq, read1.qual))
        out2.write('@%s
%s
+
%s
' % (read2.qname, read2.seq, read2.qual))
    return writer
def main():
    """what to do if we execute the module as a script
    (not intended for user by user)
    """
    parser = ArgumentParser(description=__doc__)
    parser.add_argument('-i', default=stdin,
                        help='BAM/SAM input file (default: stdin)')
    parser.add_argument('gzip', action='store_true', default=False,
                        help='Compress paired-end output files')
    parser.add_argument('outfile1', help='Output file for first mate / single reads (default: stdout)')
    parser.add_argument('outfile2', help='(required for paired-end files) output filename for second mate')
    args = parser.parse_args()
    context = vars(args)
    outfile1 = context['outfile1']
    outfile2 = context['outfile2']
f = Samfile(context['infile'])
incomplete_pairs = []
if context['gzip']:
    if PATH_TO_GZIP is not None:
        open_func = gzip_class_factory(PATH_TO_GZIP)
        fh1 = open_func(outfile1, 'w')
        fh2 = open_func(outfile2, 'w')
    else:
        fh1 = GzipFile(outfile1, 'wb')
        fh2 = GzipFile(outfile2, 'wb')
    is_paired = False
    gzwrite = gzwriter(fh1, fh2)
    for aread in f:
        is_paired = False
        qname = aread.qname
        for i in range(len(incomplete_pairs)):
            if incomplete_pairs[i].qname == qname:
                # figure out order
                if aread.flag & 0x4 == 0x4:
                    gzwrite(aread, mate_read)
                else:
                    gzwrite(mate_read, aread)
                is_paired = True
                break
        if not is_paired: incomplete_pairs.append(aread)
    unpaired = len(incomplete_pairs)
    out1.close()
    out2.close()
    f.close()
else:
    if not exists(outfile1): os.mknod(outfile1)
    if outfile2 is not None:
        if not exists(outfile2): os.mknod(outfile2)
        out1 = os.open(outfile1, os.O_WRONLY|os.O_NONBLOCK)
        out2 = os.open(outfile2, os.O_WRONLY|os.O_NONBLOCK)
        is_paired = False
        write = pair_writer(out1, out2)
        for aread in f:
            is_paired = False
            qname = aread.qname
            for i in range(len(incomplete_pairs)):
                if incomplete_pairs[i].qname == qname:
                    # figure out order
                    if aread.flag & 0x4 == 0x4:
                        write(aread, mate_read)
                    else:
                        write(mate_read, aread)
                    is_paired = True
                    break
            if not is_paired: incomplete_pairs.append(aread)
unpaired = len(incomplete_pairs)
os.close(out1)
os.close(out2)
f.close()
if not unpaired == 0:
    raise RuntimeError('%d unpaired reads remaining' % unpaired)
if __name__ == '__main__':
    main()
fh = open_func(file1)
i = 0
if file2 is not None:
    fh2 = open_func(file2)
# rec = [title, seq, qual, seq2, qual]
rec = [None, None, None, None, None]
line_fmt = '{0!s}\t{1!s}\t{2!s}\t{3!s}\t{4!s}'
for line, line2 in izip(fh, fh2):
    if i == 0: rec[0] = line[1:].rstrip()
    elif i == 1:
        rec[1] = line.rstrip()
        rec[3] = line2.rstrip()
    elif i == 2: pass
    elif i == 3:
        rec[2] = line.rstrip()
        rec[4] = line2.rstrip()
        print line_fmt.format(*rec)
i = -1
i += 1
else:
    # rec = [title, seq, qual]
    rec = [None, None, None]
    line_fmt = '{0!s}\t{1!s}\t{2!s}'
    for line in fh:
        if i == 0: rec[0] = line[1:].rstrip()
        elif i == 1: rec[1] = line.rstrip()
        elif i == 2: pass
        elif i == 3:
            rec[2] = line.rstrip()
            print line_fmt.format(*rec)
i = -1
        i += 1
        exit(0)
if __name__ == '__main__': main()
from bowtie import BowtieFilenameParser
from illumina import BarcodeFilenameParser
from aligned import BAMFilenameParser
from peaks import PeaksFilenameParser
__all__ = ['BowtieFilenameParser', 'BarcodeFilenameParser', 'BAMFilenameParser', 'PeaksFilenameParser']

Source Code C.15: seriesoftubes.fnparsers module

C.4.2 seriesoftubes.fnparsers.bowtie module

import pysam
import os
import os.path
import gzip
import bz2
import scripter
from ..converters.discover import discover_file_format

class BowtieFilenameParser(scripter.FilenameParser):
    def __init__(self, filename, *args, **kwargs):
        super(BowtieFilenameParser, self).__init__(filename, *args, **kwargs)
        open_func, format = discover_file_format(filename)
        # self.split_file = False
        self.format = format
        self.open_func = open_func
        self.second_file = None
        if format == 'SAM' or format == 'BAM':
            self.use_pysam = True
            # try to open the file so we're sure it works
            f = pysam.Samfile(filename)
            aread = f.next()
            self.paired_end = aread.is_paired
            del f, aread
            self.fastq_source = 'Unknown'
        elif format == 'FASTQ':
            self.use_pysam = False
            self.check_paired_end()
            if len(self.protoname.split('.')) > 6:
                self.fastq_source = self.protoname.split('.')[6]
            else:
                self.fastq_source = 'Unknown'
        else:
            raise RuntimeError('Dubious file format')
    def check_paired_end(self):
# check if this is a paired-end file
# if so, grab its partner
seqfile_name = os.path.basename(self.input_file)
pair_info = get_pair_info(seqfile_name)
if pair_info is None:
    pair_info = get_new_pair_info(seqfile_name)
# if pair_info is not None: self.split_file = True
if pair_info is not None:
    pair_index = pair_info[0]
    second_name = pair_info[1]
    new_name = pair_info[2]
scripter.debug('NOTICE: Detected paired read file."
if pair_index == '1':
    scripter.debug('Attempting to find second file."
    self.second_file = os.sep.join([self.input_dir, second_name])
    self.protoname = os.path.splitext(new_name)[0]
    try:
        scripter.debug('Found %s', self.second_file)
        # self.paired_end = True
    except IOError:
        scripter.debug('Failed to find paired end file')
        self.paired_end = False
    # elif pair_index == '2':
    #     scripter.debug('This is the second file, ignoring it."
    #     raise scripter.InvalidFileException
    else:
        scripter.debug('Failed to find paired end')
        self.paired_end = False
else:
    scripter.debug('This file contains single-end reads."
    self.paired_end = False

def tmp_filename(self, ref, match_type=None):
    if match_type is None:
        return os.path.join(self.output_dir, ref,
            self.with_extension('tmp'))
    else:
        return os.path.join(self.output_dir, ref, match_type,
            self.with_extension('tmp'))

def get_new_pair_info(illumina_name):
    """
    take a filename from CASAVA 1.8 output and figure out whether it's the
    first or second read (or single-end read) and return a tuple
    (pair_index, second_file_name, new_output_name).
    """
    end = illumina_name.find('.fastq')
if end == -1: return None
name = illumina_name[0:end]
parts = name.split('_')
if len(parts) < 3: return None

# align split files one at a time
# if not parts[-1].split(':')[0] == '001':
#   scripter.debug('Not the first of a split file, ignoring it')
#   raise scripter.InvalidFileException
read = parts[-2]
if read == 'R1' or read == 'R2':
    num = parts[-1]
    second_file = '_'.join(parts[0:-2]) + '_R2_' + num +illumina_name[end:]
    output_name = '_'.join(parts[0:-2]) + '_' + num +'.fastq'
    return (read[1], second_file, output_name)
return None

def get_pair_info(illumina_name):
    """
take a filename from GERALD output and figure out whether it’s the first
or second read (or single-end read) and return a tuple
(pair_index, second_file_name, new_output_name)
    """
name_parts = illumina_name.split('.')
for i in range(len(name_parts)):
    part = name_parts[i]
    subparts = part.split('_')
    if len(subparts) > 0:
        if subparts[0] == 's':
            if len(subparts) == 1: continue
            pair_index = subparts[2]
            # lane = subparts[1]
            join_part = '_'.join(subparts[0:2] + subparts[3:])
            new_output_name = '.join(name_parts[0:i] + [join_part] +
\name_parts[i+1:])
            second_part = '_'.join(subparts[0:2] + ['2'] + subparts[3:])
            second_name = '.join(name_parts[0:i] + [second_part] +
name_parts[i+1:])
            return (pair_index, second_name, new_output_name)
    return None

Source Code C.16: seriesoftubes.fnparsers.bowtie module

C.4.3 seriesoftubes.fnparsers.illumina module
import scripter
import os

class BarcodeFilenameParser(scripter.FilenameParser):
    def __init__(self, filename, verbose=False, *args, **kwargs):
        super(BarcodeFilenameParser, self).__init__(filename, *args, **kwargs)
        protoname = self.protoname
        # check for old-style
        if os.path.splitext(protoname)[-3:] == 'all':
            protoname = protoname[0:-4]
        if kwargs['no_target']:
            self.output_dir = self.input_dir

        # check if this is a paired-end file
        # if so, grab its partner
        input_file = self.input_file
        illumina_name = os.path.basename(input_file)
        if illumina_name.count('_') >= 3:
            scripter.debug('NOTICE: Detected paired read file. ')
            iln_parts = illumina_name.split('_')
            if iln_parts[2] == '1':
                scripter.debug('Attempting to find second file. ')
                second_file = os.sep.join([self.input_dir, '_'.join(iln_parts[0:2] + ['2'] + iln_parts[3:])])
                self.protoname2 = os.path.splitext(os.path.basename(second_file))[0]
                try:
                    scripter.assert_path(second_file)
                    scripter.debug('Found %s', second_file)
                    self.second_file = second_file
                    paired_end = True
                except IOError:
                    scripter.debug('Failed to find paired end file ')
                    paired_end = False
                elif iln_parts[2] == '2':
                    scripter.debug('This is the second file, ignoring it. ')
                    raise scripter.InvalidFileException(input_file)
                else:
                    scripter.debug('Failed to find paired end ')
                    paired_end = False
                else:
                    paired_end = False
            self.paired_end = paired_end

    def output_filename(self, barcode, is_barcode=True, no_gzip=False):
        file_ext = self.file_extension
        if is_barcode:
            ret = os.path.join(self.output_dir,
os.extsep.join([self.protoname,
               'barcode_' + barcode,
               file_ext]))
else:
    ret = os.path.join(self.output_dir,
                       os.extsep.join([self.protoname,
                                       barcode, file_ext]))
if ret.endswith("gz") or no_gzip: return ret
else: return os.extsep.join([ret, "gz"])

def output_filename2(self, barcode, is_barcode=True, no_gzip=False):
    file_ext = self.file_extension
    if is_barcode:
        ret = os.path.join(self.output_dir,
                           os.extsep.join([self.protoname2,
                                           'barcode_' + barcode,
                                           file_ext]))
    else:
        ret = os.path.join(self.output_dir,
                           os.extsep.join([self.protoname2,
                                           barcode, file_ext]))
    if ret.endswith("gz") or no_gzip: return ret
    else: return os.extsep.join([ret, "gz"])

Source Code C.17: seriesoftubes.fnparsers.illumina module

C.4.4 seriesoftubes.fnparsers.aligned module

import scripter
import os.path
from bioplus.genometools import guess_bam_genome, NoMatchFoundError

class BAMFilenameParser(scripter.FilenameParser):
    """
    for alignments produced by align.py
    """
    def __init__(self, filename, controls = {}, *args, **kwargs):
        if not os.path.splitext(filename)[1] == ".bam":
            raise scripter.InvalidFileException(filename)
        super(BAMFilenameParser, self).__init__(filename, *args, **
kwargs)

        sample = self.protoname
        control_files = [v[1] for v in controls.values()]
        # check controls
        if controls.has_key(sample):
            sample_name, control = controls[sample]
scripter.debug('%s has control %s', sample, control)
if control is None:
    self.control_file = None
else:
    self.control_file = os.path.join(self.input_dir, control + '.bam')
if controls.has_key(sample):
    sample_name, control = controls[sample]
    scripter.debug('%s has control %s', sample, control)
    if control is None:
        self.control_file = None
    else:
        self.control_file = control
elif sample in control_files or self.input_file in control_files:
    scripter.debug('%s is a control, aborting', sample)
    raise scripter.InvalidFileException
else:
    scripter.debug('%s has no control indicated, continuing anyway', sample)
    # not in setup.txt, make an entry in controls
    self.control_file = None
    sample_name = sample
    controls[sample] = (sample, None)

self.sample_name = sample_name
self.output_dir = os.path.join(self.output_dir, sample_name)

Source Code C.18: seriesoftubes.fnparsers.aligned module

C.4.5 seriesoftubes.fnparsers.peaks module

from twobitreader import TwoBitFile, TwoBitFileError, twobit_reader
from twobitreader.download import save_genome
from urllib2 import URLError
from scripter import Usage, InvalidFileException, FilenameParser, \
    debug, warning, error
from os.path import abspath, exists, extsep, join, splitext
from re import sub
from os import sep

def try_to_find_genome(genome):
    ""
    takes a string that may refer to a file or genome hosted on UCSC
    and tries to open the 2bit file or download it if needed
    """
if exists(genome):
    try:
        t = TwoBitFile(genome)
        return t
    except TwoBitFileError:
        warning('%s is not a valid .2bit file', genome)
        return None
else:
    try:
        save_genome(genome)
        try:
            t = TwoBitFile('%s.2bit % genome)
            return t
        except TwoBitFileError:
            error('Downloaded %s.2bit but %s.2bit is not a valid .2bit file', genome)
        except URLError:
            warning('Download of %s.2bit failed', genome)
        return None

class PeaksFilenameParser(FilenameParser):
    def __init__(self, filename, include_width_in_name=False, target=None, motif_file='unknown_motif', genome=None, *args, **kwargs):
        fext = splitext(filename)[1].lstrip(extsep)
        if fext == 'bed':
            self.is_bed = True
            self.is_xls = False
        elif fext == 'xls':
            self.is_bed = False
            self.is_xls = True
        else:
            raise InvalidFileException
        motif_name = sub(r'\W+', '_', abspath(motif_file))
        target = target + sep + motif_name
        super(PeaksFilenameParser, self).__init__(filename, target=target, *args, **kwargs)
        self.fasta_file = None
        for file_extension in ['.fa', '.fasta', 'FA', 'FASTA']:
            fasta_file = join(self.input_dir, extsep.join([self.protoname, file_extension]))
            debug("Trying", fasta_file)
            if exists(fasta_file):
                self.fasta_file = fasta_file
                debug("Using", fasta_file)
                break
        if self.fasta_file is None:
            warning('Could not find the FASTA file for %s', self.input_file)
if genome is None:
    raise Usage("Could not find the FASTA file for ", self.
        input_file,
        " and no genome was specified")
else:
    t = try_to_find_genome(genome)
    if t is None:
        raise Usage("Could not find the FASTA file for ",
            self.input_file,
            " and failed to use %s" % genome)
    else:
        fasta_file = join(self.input_dir, '%%s.fa' % self.
            protoname)
        debug('Creating FASTA file %s for %s using %s',
            fasta_file, self.input_file, genome)
        input_fhd = open(self.input_file, 'rU')
        fasta_fhd = open(fasta_file, 'w')
        twobit_reader(t, input_stream=input_fhd,
            write=fasta_fhd.write)
        fasta_fhd.close()
        self.fasta_file = fasta_file

Source Code C.19: seriesoftubes.fnparsers.peaks module

C.5 seriesoftubes.tubes package

C.5.1 seriesoftubes.tubes module

def wait_for_job(job, logs=[], logger=None):
    """wait for a job (Popen instance) to complete""
    otime = time()
    poll = job.poll
    while True:
        atime = time()
if otime - atime > 60 and logger is not None:
    logger.debug('Still running')
    otime = atime
sleep(3)
if poll() is not None:
    break
for log in logs:
    log.log()
for log in logs:
    log.log()

Source Code C.20: seriesoftubes.tubes module

C.5.2 seriesoftubes.tubes.polledpipe module

import select
import os
import logging
import platform

if platform.system() == 'Windows':
    raise RuntimeError("Microsoft Windows is not and will not be supported. This is due to an underlying design problem in the OS. Use cygwin if you need to run this on Windows.""")

class PolledPipe(object):
    
    A PolledPipe object has two attributes:
    
r - pipe read file descriptor,
w - pipe write file descriptor

    and three methods:

    poll() — poll the read end of the pipe,
    readlines() — read available lines if the poll says the pipe is ready,
    log(level=logging.error) — emit the result of readline (if not None)

    a select.poll() object has r registered to it

    def __init__(self, logger=None, level=None):
        self._logger = logger
        self._level = level
        r, w = os.pipe()
        self.r = r
        self._r_file = os.fdopen(r, 'r', 0)
```python
self.w = w
self._poll = select.poll()
self._poll.register(r)

def poll(self, timeout=0):
    return self._poll.poll(timeout)

def readlines(self, timeout=0):
    results = self.poll(timeout)
    if results is None: raise StopIteration
    for result in results:
        event = result[1]
        if (event & select.POLLERR) == select.POLLERR:
            raise IOError('Something went wrong trying to read from a pipe')
        elif (event & select.POLLNVAL) == select.POLLNVAL:
            raise IOError('Invalid request: descriptor for pipe not open')
        elif (event & select.POLLPRI) == select.POLLPRI:
            raise IOError('Pipe file descriptor already hung up')
        elif (event & select.POLLIN) == select.POLLIN or
            (event & select.POLLPRI) == select.POLLPRI:
            yield self._r_file.readline()
        raise StopIteration

def log(self, level=None):
    """
    emit the results of readlines
    """
    if self._logger is None:
        raise RuntimeWarning('PolledPipe did not have a logger attached')
    if level is None: level = self._level
    if level is None: level = logging.ERROR
    while True:
        for line in self.readlines():
            self._logger.log(level, line.rstrip())
        results = self.poll()
        if results is None: break
        if len(results) == 0: break
        event = results[0][1]
        if (event & select.POLLIN) == select.POLLIN or
            (event & select.POLLPRI) == select.POLLPRI: continue
        else: break
    return
```

Source Code C.21: seriesoftubes.tubes.polledpipe module
C.6 bioplus package

C.6.1 bioplus.fasta module

```python
'''meta−tools for large FASTA files'''
import random
import itertools
import os.path
from Bio.SeqIO import parse, write
import Bio.Seq

def count(foo):
    '''takes a file named foo returns the number lines'''
    f = parse(open(foo,'rU'),'fasta')
    n = 0
    for dummyX in f:
        n += 1
    return n

def random_seq(foo, n):
    '''takes a file foo and returns n random sequences from it'''
    max_n = count(foo)
    record_numbers = itertools.repeat(random.randint(1,max_n),times=n)
    seq_recs = parse(open(foo,'rU'),'fasta')
    i = 0
    seqs = []
    for rec in seq_recs:
        i += 1
        if i in record_numbers: seqs.append(rec)
    return seqs

def reader(foo):
    '''
generator yielding Bio.Seq.Seq objects from a FASTA file
'''
    for record in parse(open(foo, 'rU'), 'fasta'):
        yield record.seq

def writer(foo, iterable):
    '''
writes SeqRecord objects from iterable to FASTA file foo.
Warning: overwrites foo, does not append
'''
    write(iterable, open(foo, 'w'), 'fasta')

def random_files(foo, n, R):
    '''
```
```
takes a FASTA file foo and creates (in the current directory) R
random files each containing n random sequences from foo, named foo_random
[0−R].fa

for filenumber in R:
    prefix = foo.split(os.path.sep)[-1]
    boo = prefix + '_random' + pad(filenumber, R) + '.fa'
    writer(boo, random(foo, n))

def pad(x, y):
    '''
takes two integers x and y, and returns str(x) with enough 0s to
match the length of str(y)
'''
    return str(x).zfill(len(str(y)))

def truncate_lines(f, n):
    '''
truncate_lines(f,n) truncates lines in a file to at most n
characters See truncate_seqs to truncate sequences instead of lines
'''
    with open(f, 'r') as seqs, open('{!s}.{!s}'.format(f, n), 'w') as tseqs:
        tseqs.writelines((seq[0:n] + '
' for seq in seqs))

def truncate_seqs(f, n):
    '''
truncate FASTA seqs truncates sequences in a FASTA file named f to
at most n bases, writing a new FASTA file named f.n.fa
'''
    seq_recs = parse(open(f, 'rU'), 'fasta')
    # USE FILENAME CORRECTION SCHEME
    foo = f+str(n)+'_fa'
    writer(foo, (rec[0:n] for rec in seq_recs))

def permute_fasta(f):
    '''
takes a FASTA file and returns a new FASTA file with each sequence
randomly permuted (separately, such that its % A,T,G,C doesn’t change)
'''
    mute = Bio.Seq.Seq.tomutable
    shuffle = random.shuffle
    with open(f + '.permuted.fa', 'w') as output:
        with open(f, 'rU') as fobj:
            for seq_rec in parse(fobj, 'fasta'):
                seq_rec.seq = mute(seq_rec.seq)
shuffle(seq_rec.seq)
write(seq_rec, output, 'fasta')

Source Code C.22: bioplus.fasta module

C.6.2 bioplus.genometools module

def TemporaryGenomeFile(genome_name):
    
    returns a file-like object pointing to a temporary file containing
    the chromosome names and sizes

    the current file position will be 0

    it will be deleted when the object is garbage collected
    
    f = NamedTemporaryFile()
    genome_rows = genome(genome_name).iteritems()
    f.writelines(( '%s	%d
' for chrom, size in genome_rows))
    f.flush()
    f.seek(0)
    return f

def _populate_available_genomes():
    name_tuples = _CONNECTION.execute("select name from sqlite_master
WHERE" +
      " type='table'").fetchall()
    return map(itemgetter(0), name_tuples)
AVAILABLE_GENOMES = _populate_available_genomes()

def import_db(db, replace=False, ignore_warnings=False):
    
    """imports a db like genome.db into the existing database
    note: this may not be persistent if your egg is zipped
    """
    connection = sqlite3.connect(db)
    name_tuples = connection.execute("select name from sqlite_master WHERE \n    type='table'").fetchall()
    available_genomes = map(itemgetter(0), name_tuples)
    for genome_name in available_genomes:
        if not ignore_warnings:
            assert genome_name in AVAILABLE_GENOMES and not replace,
            "Genome %s already in database" % genome_name
        _CONNECTION.execute("create table %s (name VARCHAR, length INT UNSIGNED)" % genome_name)
        genome_dict = dict(connection.execute("select * from %s" % genome_name))
        _CONNECTION.executemany("insert into %s values (?,?)" % genome_name, dict(genome_dict).items())
    return

def add_genome(genome_name, genome_dict, replace=False):
    """WARNING: THIS WILL PERMANENTLY ADD A GENOME
    make a call to pybedtool and tries to add to genome registry
    then reloads module and repopulates globals
    note: this may not be persistent if your egg is zipped
    """
    assert genome_name in AVAILABLE_GENOMES and not replace,
    "Genome %s already in database" % genome_name
    _CONNECTION.execute("create table %s (name VARCHAR, length INT UNSIGNED)" % genome_name)
    _CONNECTION.executemany("insert into %s values (?,?)" % genome_name, dict(genome_dict).items())
    return

def genome(genome_name):
    """if available,
    returns a dict of {'chr_name': length} for all chromosomes in that genome
    """
if not genome_name in AVAILABLE_GENOMES:
    raise GenomeNotAvailableError
else:
    try:
        return dict(_CONNECTION.execute("select * from %s" %
                                             genome_name))
    except sqlite3.OperationalError:
        raise GenomeNotAvailableError(genome_name)

class GenomeNotAvailableError(ValueError):
    pass

class NoMatchFoundError(LookupError):
    pass

def guess_bam_genome(bam_file_or_filename):
    return guess_genome(genome_from_bam(bam_file_or_filename))

def genome_from_bam(bam_file_or_filename):
    if isinstance(bam_file_or_filename, Samfile):
        bam_file = bam_file_or_filename
    else:
        bam_file = Samfile(bam_file_or_filename)
    return dict(zip(bam_file.references, bam_file.lengths))

def guess_genome(genome1):
    
    expects a dictionary or iterable of (name, length) 
    checks for a match in the database 
    
    for name in AVAILABLE_GENOMES:
        genome2 = genome(name)
        if matches_genome(genome1, genome2):
            return name
    raise NoMatchFoundError

def matches_genome(genome1, genome2, symmetric=False):
    
    expects two dictionaries or iterables of (name, length) 
    tells you if genome1 matches genome2 
    (not all the entries in genome2 need to be matched, but all the 
    entries in 
    genome1 must be)
symmetric=True means matches(genome1, genome2) and matches(genome2, genome1)

```python
symmetric=True means matches(genome1, genome2) and matches(genome2, genome1)

g1 = dict(genome1)
g2 = dict(genome2)
for k in g1.keys():
    if not k in g2:
        return False
    elif not g2[k] == g1[k]:
        return False
    # else: continue
if symmetric:
    return matches_genome(genome2, genome1)
else:
    return True
```

Source Code C.23: bioplus.genometools module

### C.6.3 bioplus.motif module

```python
import math
import operator
from collections import defaultdict
from numpy.core.multiarray import zeros
from tabfile import TabFile

class CharDict(defaultdict):
    """CharDict is a generalized dictionary that inherits defaultdict. default values are 0. Accepts any keys. """

    def __init__(self, dict={}):
        def returnZero():
            return 0
        super(CharDict, self).__init__(returnZero, dict)

    def __add__(self, x, y):
        z = CharDict()
        for k in set(x.keys() + y.keys()):
            z[k] = x[k] + y[k]
        return z

    def uncertainty(self):
        """calculates the uncertainty H in this position (specified by CharDict).
```
reats 0*log(0) as 0

```
H = 0
for pN in self.itervalues():
    if pN==0: H = 0
    else: H = -pN * math.log(pN, 2)
return H
```

```python
class PositionWeightMatrix(object):
    """
    Stores counts of nucleotide bases at each position. objects are immutable.
    sequences may be added to the counts, but the object may not be modified
    in situ
    """
    def __init__(self, n=None):
        raise DeprecationWarning("Use Bio.Motif.PositionWeightMatrix will be removed soon")
        self._errN = "n must be a positive integer"
        if not n==None: self._initialize_positions(n)
        self._is_probs=False
        self._is_Ri=False
    def _initialize_positions(self, n):
        self._L = []
        if type(n)==int:
            if n > 0:
                self._n = n
                for dummyVariable in range(self._n):
                    self._L.append(CharDict())
            else: raise ValueError(self._errN)
        else: raise TypeError(self._errN)
    def __add__(self, x, y):
        n = x._n
        if n == y._n:
            z = PositionWeightMatrix(n)
            for i in range(n):
                z[i] = x[i] + y[i]
        else: raise ValueError("PositionWeightMatrix objects are not the same length (number of positions)\")
        return z
    def __getitem__(self, y):
        return self._L[y]
    def __len__(self):
        return 313
return len(self._L)

def count_file(self, seqsFile, n=0):
    """uses a tabFile with a list of sequences, in column n (by
default n=0, the first column) and extracts counts"""
    if self._is_probs: raise UserWarning('Already converted to
    probabilities')
    # open file
    seqsFile.open()
    # read first sequence and set siteLength
    rows = (row for row in seqsFile)
    row = rows.next()
    site = row[n]
    siteLength = len(site)
    self._initialize_positions(siteLength)
    # initialize the object
    for i in range(self._n): self._L[i][site[i].upper()] += 1
    # read remaining sequences
    while True:
        try:
            row = rows.next()
            site = row[n]
        except StopIteration: break
        if len(site) == siteLength:
            for i in range(self._n): self._L[i][site[i].upper()] += 1
        else:
            # clean up
            del self._L
            del self._n
            seqsFile.close()
            raise ValueError('One of the sequences you are trying to
            add is not the correct length (' + str(self._n) + '): ' + site)
    self._n = siteLength

def count_seqs(self, L, debug=False):
    """adds a list of sequences to the counts"""
    if self._is_probs: raise UserWarning('Already converted to
    probabilities')
    firstSite = True
    n = 0
    m = 0
    for site in L:
        n += 1
        if n%(10**6) == 0:
            m += 1
            if debug: print str(n)
        if firstSite:
            siteLength = len(site)
            self._initialize_positions(siteLength)
firstSite = False
for i in range(self._n):
    if len(site)==siteLength: self._L[i][site[i].upper()] += 1
    else:
        # clean up
        del self._L
        del self._n
        raise ValueError('One of the sequences you are trying to add is not the correct length ('+str(self._n)+'):'+'site)

def import_from_MEME(self,filename,n=1,mode='biotools'):
    """imports a motif from the output of MEME (meme.txt)"
    if there are multiple motifs in the output, we will use motif n (the first is n=1, which is also the default)"
    import Bio.Motif.Parsers.MEME as MEME
    f = open(filename)
    MEME_object = MEME.read(f)
    motif_name = 'Motif ' + str(n)
    biopython_motif = MEME_object.get_motif_by_name(motif_name)
    if mode=='biopython': return biopython_motif
    if mode=='biotools':
        internal_n = len(biopython_motif)
        # this next line is instead of initializePositions
        biotools_motif = [CharDict(biopython_motif[i]) for i in range(internal_n)]
        self._L = biotools_motif
        self._n = internal_n
    else: raise UserWarning('Not a valid mode. ')

def rc(self):
    """returns the reverse complement of this object""
    new = PositionWeightMatrix(self._n)
    # complement the object
    for i in range(self._n):
        for base in self._L[i].keys():
            complement = {'A': 'T', 'C': 'G', 'G': 'C', 'T': 'A'}
            if complement.has_key(base): new[i][complement[base]] = self._L[i][base]
            else: new[i][base] = self._L[i][base]
    new._L.reverse() # reverse the object
    return new

def __repr__(self):
    return self._L._repr()

def make_probs(self, trueProbs=False):
    """normalizes everything to 1"""
if self._is_Ri:
  for x in self._L:
    for k in x.keys():
      x[k] = 2**(x[k]-2)
  self._is_Ri = False
else:
  for x in self._L:
    total = sum(x.values())
    zeros = x.values().count(0)
    if trueProbs or zeros==0:
      for k in x.keys():
        x[k] = float(x[k]) / total
    else:
      # fake one occurrence
      total += 1
      for k in x.keys():
        if x[k]==0: x[k]=1
        x[k] = float(x[k]) / total

self._is_probs = True

def make_Ri(self):
  """changes from counts or probabilities to Ri, information content""
  if not self._is_Ri:
    if not self._is_probs: self.make_probs()
    for p in self._L:
      for k in p.keys():
        p[k] = 2+math.log(p[k],2)
    self._is_probs = False
    self._is_Ri = True
  else:
    print 'Already Ri'

def seq_Ri(self,s):
  """seqRi returns the information content Ri in bits of a sequences , as measured with the given positionmatrix""
  if not self._is_Ri: self.makeRi()
  Ri = 0
  if len(s) != self._n:
    raise UserWarning('Cannot evaluate a sequence which is not the exact length of the position matrix')
  for x in range(self._n): Ri += self[x][s[x].upper()]
  return Ri

def uncertainty(self):
  """returns the uncertainty H(l) of the matrix as a list. Use sum () for the total uncertainty.

Note: this function calls uncertainty() from the baseDict instance , and as such it can be overwritten implicitly. baseDict.uncertainty() treats 0*log(0) as 0"""
if not self._is_probs: self.make_probs()
return [position.uncertainty() for position in self]

def Rs(self):
    """returns the Schneider Rs value, which is the expectation of Ri over all possible sequences, calculated as the sum of 2−uncertainty."""
    if not self._is_probs: self.make_probs()
    return sum([2−position.uncertainty() for position in self])

def KL(p,q):
    """returns a list of the KL divergence (relative entropy) at each position from positionmatrix p to positionmatrix q. use sum() for the sum"""
    if not len(p)==len(q): raise SyntaxError('Length of p and q must be the same, instead length of p is ' + len(p) + ' and length of q is ' + len(q))
    else: n = len(p)
    for i in xrange(n):
        KLi = 0
        for j in ['A', 'G', 'T', 'C']:
            KLi += p[i][j] * math.log(p[i][j] / q[i][j], 2)
        KL.append(KLi)
    return KL

# requires numpy, maybe relax requierment?
# needs to return an mmMatrix object
# needs to be able to save to file
# needs to be able to make and save image

def joint_matrix(sites):
    """takes as input a filename and returns the joint Rate matrix for the list of sequences contained in that file
    Joint rates R(X;Y) are defined as R(X;Y) = − sum over X,Y p(x,y) ∗ I(X;Y)
    I(X;y) = − sum over X,Y p(x,y) ∗ log2[p(x,y)/(p(x)p(y))] """
    bases = ['A', 'C', 'G', 'T']
    indexDictionary = {}  # the index dictionary
    for i in range(4):
        for j in range(4):
            ssPair = bases[i] + bases[j]
            indexDictionary[ssPair]=i,j
    site_length = len(sites[0])
    # initialize the matrix
    di_counts = zeros([site_length,site_length],dtype='(4,4)int')

    def add_seq(m,s,n,b):
        """adds the dinucleotide counts from one sequence to the mm_matrix (an array, passed by reference). requires the length
        """
        for i in range(4):
            for j in range(4):
                ssPair = bases[i] + bases[j]
                indexDictionary[ssPair]=i,j
                di_counts共生 = zeros([site_length,site_length])
                for x in range(4):
                    for y in range(4):
                        di_counts共生[x,y] = p共生[x,y] * math.log(p共生[x,y] / q共生[x,y], 2)
```python
for i in range(n):
    for j in range(n):
        m[i, j][b[s[i]+s[j]] ] += 1

# count pairs over every sequence
for site in sites:
    add_seq(di_counts, site.upper(), site_length, indexDictionary)

# convert to probabilities
di_probs = zeros([site_length, site_length], dtype='(4,4) float')
total_seqs = di_counts[0,0].sum()
for i in range(site_length):
    for j in range(site_length):
        for ii in range(4):
            for jj in range(4):
                di_probs[i, j][ii, jj] = di_counts[i, j][ii, jj] / float(total_seqs)

mm_matrix = zeros([site_length, site_length], dtype='float')
for i in range(site_length):
    for j in range(site_length):
        # sum over all dinucleotide combinations
        pM = di_probs[i, j]

        # Determine lij
        lij = 0.0
        for x in range(4):
            for y in range(4):
                px = pM[x,:].sum()
                py = pM[:,y].sum()
                pxy = pM[x,y]
                if any([pxy==0, py==0, px==0]): continue
                lij += pxy * math.log(pxy/px/py, 2)

        # Determine Rij
        Rij = 0.0
        for x in range(4):
            for y in range(4):
                pxy = pM[x, y]
                Rij -= pxy * lij

        mm_matrix[i][j] = Rij
return (di_counts, di_probs, mm_matrix)

def spacerGC(L, spacerOffset=6, spacerLength=3):
    """
    spacerGC takes as input a list of [15 bp GBSs (strings)] and
    returns the number of sequences that have 0,1,2,3 G/Cs in the 3 bp spacer
    as an array in that order
    """
```
# gc counts of 0, 1, 2, 3
GCcounts = [0, 0, 0, 0]
for s in L:
    spacer = s[0][spacerOffset:spacerOffset+spacerLength].upper()
    GCs = spacer.count('C') + spacer.count('G')
    GCcounts[GCs] += 1
return GCcounts

def center_region(f, max_dist=75, motif_length=17):
    """returns a function that specifies whether a given motif is in +/- x bp from the peak_center
    requires the tabFile object f to determine the indices properly
    ""
    column_dict = f.column_dict()
    peak_summit = column_dict['peak_summit']
    for offset_name in ('motif_offset', 'offset', 'max_offset'):
        if column_dict.has_key(offset_name):
            site_offset = column_dict[offset_name]
            break
    return lambda x: int(x[peak_summit]) > (int(x[site_offset]) -
    max_dist) and \
    int(x[peak_summit]) < (int(x[site_offset]) +
    max_dist - motif_length)

def count_spacers_from_info(foo, cutoff=None, region_rule=None,
region_width=None, spacer_offset=8, spacer_length=3, output_file=None):
    """count spacers from a .sites.info or .peaks.info file
    optionally you may supply
cutoff, a minimum cutoff (float or int)
region_rule, a function that selects the column
    ""
    input_file = TabFile(foo, colNames=True)
    rows = (x for x in input_file)
    conditions = [lambda x: x[7].isalpha, # col 7 is a sequence
    lambda x: x[7] is not '-', # col 7 is not -
    lambda x: x[7] is not 'NA', # col 7 is not NA
    lambda x: x[7].strip() is not '-' # col 7 is not missing
    if cutoff is not None:
        conditions.append(lambda x: float(x[4])>cutoff)
    if region_rule is 'center_region':
        if region_width is not None:
            conditions.append(center_region(input_file,
            max_dist=region_width/2))
```python
else:
    conditions.append(center_region(input_file, max_dist=75))
elif region_rule is not None:
    conditions.append(lambda x: region_rule(x))

selected_rows = (x[7].upper() for x in rows if all([f(x) for f in conditions]))

spacers = CharDict(dict)
for s in selected_rows:
    if not s == '-' and not s == 'NA':
        spacer = s[spacer_offset:spacer_offset + spacer_length].upper()
        spacers[spacer] += 1

if output_file is None:
    output_file = raw_input('Output file name: ')
with TabFile(output_file, 'w') as f:
    f.write_table(sorted(spacers.items(),
                          key=operator.itemgetter(1), reverse=True))

return spacers

def count_letters(L):
    n = range(len(L[0]))
    counts = []
    for j in n:
        counts.append(CharDict())
    for x in L:
        for j in n:
            counts[j][x[j]] += 1
    return counts
```

Source Code C.24: bioplus.motif module

### C.6.4 bioplus.peaktools module

```python
from itertools import izip
import numpy

def MACS_track_to_iter(track):
    i = 0
    for end_pos, value in izip(*track):
        while i < end_pos:
            i += 1
        yield value
```
def array_to_bedgraph(a, chrom, write_zero_values=False, precision=0):
    ""
    generator that takes a numpy array and yields tuples bedGraph file
    note: the array must be compatible with enumerate()
    chrom should be specified too

    precision specifies how many places past the decimal to retain (default 1)

    To coerce into a string use something like ::

    for line in array_to_bedgraph(a, chrom): '%s	%d	%d	%.1f
' % line
    ""
    last_pos = 0
    last_value = round(a[0], precision)
    for pos, value in enumerate(numpy.around(a, decimals=precision)):
        if value == last_value:
            continue
        else:
            if last_value != 0 or write_zero_values:
                yield (chrom, last_pos, pos, last_value)
            last_pos = pos
            last_value = value
    if last_value != 0:
        yield (chrom, last_pos, pos, last_value)

Source Code C.25: bioplus.peaktools module

C.6.5 bioplus.seqtools module

import collections
import operator
from itertools import izip, imap
from numpy.core.fromnumeric import std

def get_first(seq1, seq2, ignore_case=True):
    ""
    returns the alphabetically prior of seq1 and seq2
    if they are the same or we fail to order, return seq1

    if ignore_case=True, we apply str.upper to each letter
    ""
    if ignore_case:
        upper = str.upper
        for a, b in izip(seq1, seq2):
            A = upper(a)
B = upper(b)
    if A == B: continue
    elif A > B: return seq1
    elif B > A: return seq2
    else:
        for a, b in izip(seq1, seq2):
            if a == b: continue
            elif a > b: return seq1
            elif b > a: return seq2
        return seq1

def count_seqs( iterable, reverse_complement=True, ignore_case=True):
    '''
count_seqs takes as input an iterable that yields sequences, and then returns the counts of each sequence as a defaultdict (similar to dict, can be recast as dict type).
    Sorting: Use dictSort to sort, if needed.
    
    Reverse complements: if rc=True, count_seqs will count reverse complements as the same sequence and use the alphabetically prior sequence as the key.
    if rc=False, reverse complements will be treated as different sequences.
    '''
    seq_counts = collections.defaultdict(lambda: 0)
    if reverse_complement:
        orient_seq = lambda s: get_first(s, rc(s), ignore_case=ignore_case)
        oriented_seqs = imap(orient_seq, iterable)
        for seq in oriented_seqs: seq_counts[seq] += 1
    else: # if not rc
        for seq in iterable: seq_counts[seq] += 1
    return seq_counts

def count_compare(a, b, rc=False):
    """
count_compare takes two default dictionaries (defaultdict objects), a and b, each with default factory "lambda: 0" and key-value pairs that specify the counts (value) of each sequence (key), and returns a new dictionary whose values are tuples (value_a, value_b), the values from a and b.
    Reverse complements: if rc=True, count_seqs will count reverse complements as the same sequence and use the alphabetically prior sequence as the key.
    if rc=False, reverse complements will be treated as different sequences.
    """
If you have already ensured that the keys meet this condition, you should use rc=False, but rc=True is also safe.

Note: defaultdict is in the collections module of the standard library

```python
if not (type(a) == collections.defaultdict and type(b) == collections.defaultdict):
    raise ValueError('a and b must be default dictionaries (collections.defaultdict)')
```

# proceed
comparedCounts = {}
for k in set(a.keys(), b.keys()):
    comparedCounts[k] = (a[k], b[k])
return comparedCounts

def analyze_sites(some_list):
    
    requires the following input format:
    seq score cdist cons
    cdist = distance to center, cons = conservation value
    
    # column definitions for reference / calling
    x0 = 0  # x0 is sequence
    x1 = 1  # x1 is score (Ri)
    x2 = 2  # x2 is peak length
    x3 = 3  # x3 is cdist
    x4 = 4  # x4 is conservation score

    # columns are seq, score, peak length, cdist, cons
    # build an array scalar with the correct column types
    for x in some_list:
        # forces sequences to uppercase
        x[x0] = x[x0].upper()
        # forces length to int
        x[x2] = int(x[x2])
        # forces cdist to int
        x[x3] = int(x[x3])
        # forces cons score to floating point
        # and correct nan
        if x[x4] == 'nan': x[x4] = 0.0
        else: x[x4] = float(x[x4])

    # sort the list by sequence (arbitrarily)
    sorted_list = sorted(some_list, key=operator.itemgetter(x0))

    # Part 3: continue with analysis
    analyzedSeqs = []
    counter = len(sorted_list)
    while counter > 0:
        site = sorted_list.pop()
        counter -= 1
        seq = site[x0]
        score = site[x1]
```
# start a list of dists, absdists, cscores, we'll sum later
i = 1
plengths = [ site[x2] ]
dists = [ site[x3] ]
absdists = [ abs(site[x3]) ]
cscores = [ site[x4] ]
while True:  # grab sequences that match
    try:
y = sorted_list.pop()
    except IndexError:
        break
    counter -= 1
    # look if we match the last entry
    if y[x0] == seq:
        i += 1
        plengths.append(y[x2])
        dists.append(y[x3])
        absdists.append(abs(y[x3]))
        cscores.append(y[x4])
    # if we don't, escape to previous loop
    else:
        # append back
        sorted_list.append(y)
        break
# append seq, sum of scores, # of instances
avgPlength = float(sum(plengths))/i
avgDist = float(sum(dists))/i
avgAbsdist = float(sum(absdists))/i
distErr = std(absdists)
analyzedSeqs.append([seq, score, i, sum(cscores), avgPlength, avgDist, avgAbsdist, distErr])
analyzedSeqs.reverse()
return sorted(analyzedSeqs, key=operator.itemgetter(x2))
def complement(nucleotide):
    '''returns the complement of a single nucleotide'''
    complement = {'A': 'T', 'C': 'G', 'G': 'C', 'T': 'A', 'a': 't', 'c': 'g', 'g': 'c', 't': 'a'}
    return complement.setdefault(nucleotide, 'N')

def rc(seq):
    '''returns the reverse complement of a sequence'''
    L = [complement(s) for s in seq]
    L.reverse()
    return ''.join(L)
reversecomplement = rc

Source Code C.26: bioplus.seqtools module
tools for dealing with binding sites (instances of sequence motifs)

import os.path
from seqtools import rc
from tabfile import TabFile, MacsFile, BedFile
import operator
import Bio.Motif
import Bio.SeqIO

try:
impo
results_sorted_like_Bio_Motif = sorted(search_results,
    key=operator.itemgetter(0),
    cmp=lambda x,y: cmp(abs(x), abs(y)))
return results_sorted_like_Bio_Motif

def search_peak(peak_ID, peak, peakseq, motif, bysummit=False):
    '''provide information about matches to a motif in a
    peak region, and about the region

    peak MUST provide EITHER
    (1) the following public methods
    chrom = reference (e.g. chr1, chrX)
    chromStart = start coordinate, 0-based
    chromEnd = end coordinate, open
    or (2) the following public method
    coordinates = a tuple containing (chrom, chromStart, chromEnd)

    peak may optionally provide the following methods
    tags (if not found, we will replace with 'NA')
    summit (if not found, we will use the peak center)
    misc (a list of anything else)

    For each peak, the best motif hit is returned where
    best is defined as the motif hit with the most information
    and closest to the center (in the case of ties)

    Note site position is 0-based, in contrast with
    earlier versions of biotools

    returns a tuple of four things:
    peak info
    peak BED row
    a list of info about sites (motif matches)
    a list of BED rows for sites
    '''
    peak_length = len(peakseq)
clean_peak_seq = _clean_sequence(peakseq)
clean_peak_length = len(clean_peak_seq)
clean_offset = peakseq.find(clean_peak_seq)

    site_count = 0
total_Ri = 0
best_Ri = 0
best_seq = 'NA'
best_position = 'NA'
best_strand = 'NA'
sites_info_rows = []
sites_bed_rows = []

    motif_length = len(motif)
try:
    peak_coordinates = peak.coordinates()
except AttributeError:
    peak_coordinates = [peak.chrom(),
                       peak.chrom_start(),
                       peak.chrom_end()]

try:
    peak_center = peak.summit()
    peak_summit = peak.summit()
    if int(peak_center) > int(peak_coordinates[1]):
        peak_center -= int(peak_coordinates[1])
    else:
        peak_summit += int(peak_coordinates[1])
except AttributeError:
    peak_center = clean_peak_length / 2 + clean_offset
    peak_summit = int(peak_coordinates[1]) + peak_center

try:
    peak_intensity = peak.tags()
except (ValueError, AttributeError):
    try:
        peak_intensity = peak.tagsv2()
    except AttributeError:
        peak_intensity = 'NA'

try:
    peak_misc = peak.misc()
except AttributeError:
    peak_misc = []

if USE_MOODS:
    search_results = MOODS_search(clean_peak_seq, motif)
else:
    search_results = motif.search_pwm(clean_peak_seq)

for position, Ri in search_results:
    site_count += 1
    total_Ri += Ri

    if position > 0:
        offset = position + clean_offset
        site_seq = clean_peak_seq[offset:offset + motif_length]
        strand = '+'
    else:
        offset = clean_offset + clean_peak_length + position - motif_length
        site_seq = _reverse_complement(clean_peak_seq[offset:
                                                   offset + motif_length])
        strand = '-'

    site_ID = '_'.join([''.join([str(x) for x in
                                  [peak_ID, 'motif', site_count]])
                        for _ in range(len(search_results))])
chrom = peak_coordinates[0]
chrom_start = int(peak_coordinates[1])
if bysummit:
    site_BED_row = [chrom, peak_summit - peak_length/2 + offset, 
                peak_summit - peak_length/2 + offset + 
                motif_length, 
                site_ID, Ri, strand]
else:
    site_BED_row = [chrom, chrom_start + offset, 
                chrom_start + offset + motif_length, 
                site_ID, Ri, strand]

site_info_row = site_BED_row + [offset, site_seq, 
                peak_ID, clean_peak_length, 
                len(peak), peak_center]
sites_bed_rows.append(site_BED_row)
sites_info_rows.append(site_info_row)

# check if this is the best site
if Ri > best_Ri or \
    Ri == best_Ri and (abs(peak_center - offset) - motif_length 
        /2) < \
    best_position:
    best_Ri = Ri
    best_seq = str(site_seq)
    best_position = offset
    best_strand = strand

peak_info_row = peak_coordinates + [peak_ID, peak_intensity, 
                site_count, total_Ri, 
                best_Ri, best_seq, best_position, best_strand, 
                clean_peak_length, peak_center, ', '.join(peak_misc)]

peaks_BED_row = peak_coordinates + [peak_ID, 1000, '+', 
                int(peak_coordinates[1]) + peak_center, 
                int(peak_coordinates[1]) + peak_center + 1]

return (peak_info_row, peaks_BED_row, sites_info_rows, 
    sites_bed_rows)

def find_sites(peaks_file, fasta_file, motif, bed=True, xls=False, 
    output_dir=None, motif_type='MEME',  
    src_fnc="find_sites", bysummit=False, **kwargs):
    ...
motif (a Bio.Motif object). It will output two new files for peaks and
sites called NAME.peaks.info and NAME.sites.info. It will also create files
called NAMES.peaks.bed and NAME.sites.bed which are proper BED files (scores
are tag
density, and information content, respectively). All files are 0–based,
half–open in line with the BED convention. MACS coordinates are
corrected
accordingly.

f.peaks.info contains
Peak (1) chr, (2) start (3) end
(4) Peak ID
(5) Relative summit
(6) Number of unique tags in peak region
(7) −10•log10(pvalue)
(8) fold_enrichment
(9) FDR
(10) # motif instances found
(11) Total Ri for discovered motif instances
(12) Greatest Ri of any motif in peak region
(13) Sequence of that motif instance
(14) Position (offset) of that motif (left–end)

f.peaks.bed contains
Peak (1) chr, (2) start (3) end
(4) Peak ID
(5) Number of unique tags in peak region
(6) Strand
(7) Summit position (absolute)
(8) Summit position + 1

f.sites.info contains
Site (1) chr (2) start (3) end
(4) Unique Site ID (internally generated)
(5) The motif information content Ri, in bits
(6) motif orientation, best score (+) or (−)

BED file ends here
(7) the motif sequence (e.g., ACAACA)
(8) Position (offset) of that motif (left–end)
(9) peak ID, fetched from MACS
(10) used peak length
(11) true peak length
(11) peak summit offset

if type(motif) is str:
    motif = Bio.Motif.read(open(motif), motif_type)

# start the output file
prefix = os.path.splitext(os.path.basename(peaks_file))[0]
if output_dir is not None: prefix = os.path.join(output_dir, prefix)
sites_info = TabFile(os.extsep.join([prefix, 'sites', 'info']), 'w')
sites_bed = TabFile(os.extsep.join([prefix, 'sites', 'bed']), 'w')
peaks_info = TabFile(os.extsep.join([prefix, 'peaks', 'info']), 'w')
peaks_bed = TabFile(os.extsep.join([prefix, 'peaks', 'bed']), 'w')

peaks_cols = ['chr', 'start', 'end', 'peak_ID', 'peak_intensity', 'site_count', 'total_Ri', 'best_Ri', 'best_seq', 'best_offset', 'best_strand', 'clean_peak_length', 'peak_summit', 'peak_misc']
peaks_msg = os.linesep.join([f'# This file was generated by {src_fnc}', '	'.join(peaks_cols), ''])
peaks_info.write(peaks_msg)
sites_cols = ['chr', 'start', 'end', 'site_ID', 'Ri', 'strand', 'offset', 'motif_seq', 'peak_ID', 'peak_length', 'reported_peak_length', 'peak_summit']
sites_msg = os.linesep.join([f'# This file was generated by {src_fnc}', '	'.join(sites_cols), ''])
sites_info.write(sites_msg)

if bed: peak_generator = BedFile(peaks_file)
elif xls:
    peak_generator = MacsFile(peaks_file)
else: raise ValueError('Neither bed nor xls')

# peakSeqs is a generator
peak_seqs = (r.seq for r in Bio.SeqIO.parse(open(fasta_file, 'rU'), 'fasta'))
nosites = 0
peaknumber = 0

for peak in iter(peak_generator):
    # if peaknumber%10000 is 0: print peaknumber
    peaknumber += 1
    seq = peak_seqs.next()
    # Generate a peak ID
    try: peak_ID = peak.name()
    except NameError: peak_ID = '{!s}_{!s}'.format(prefix, peaknumber)
    # Change behavior to use sequences centered at summit
    (peak_info, peak_bed, sites_info_rows, sites_bed_rows) = search_peak(peak_ID, peak, seq, motif, bysummit=бysummit)
peaks_info.write_row(peak_info)
peaks_bed.write_row(peak_bed)
sites_info.write_rows(sites_info_rows)
sites_bed.write_rows(sites_bed_rows)
if len(sites_info_rows) == 0:
    nosites += 1

sites_info.close()
sites_bed.close()
peaks_info.close()
peaks_bed.close()
message = f"There were {nosites} of {peak_num} peaks with no identifiable sites in {fasta_file} using a cutoff of 0 bits".
stdout_buffer = message
# get the motif
motif_str = ''
try:
    motif_str = os.linesep.join(["",".".join([str(base), ":", str(ods)])
    for base, ods in position.items()])
    for position in motif.log_odds()]
except AttributeError:
    motif_str = str(motif)
message = os.linesep.join([message,"The following motif was used",motif_str])
# print message and write it to a log
f = open(prefix + ".log", 'w')
g.write(message)
g.close()
return stdout_buffer

Source Code C.27: bioplus.sitefinder module

C.6.7 bioplus.tabfile module

'''tools for dealing with files that have tabular data'''
import os
import itertools
import re
import gzip
import bz2

def merge_files(left, right, output, comments="left"):  
    """
    merge_files merges the tab-delimited files named left and right, which may have commented lines. the output is directed to the file named output.
    """

"""
There are few modes. If comments='left', comments in left are preserved.
If comments='right', comments in right are preserved. If comments='none', no comments are preserved. If comments='all', all comments in left and right are appended to the beginning of output, although they may previously have been contained within the data in left or right.

Use merge_tab_files if you need to pass custom parameters to TabFile.

```python
def merge_tab_files(file1, file2, output_filename, comments='left'):
    file1 = TabFile(left)
    file2 = TabFile(right)
    return merge_tab_files(file1, file2, output, comments)
```

merge_tab_files merges the tab-delimited files represented by TabFile objects left and right, which may have commented lines. The output is directed to the file named output.

There are few modes. If comments='left', comments in left are preserved.
If comments='right', comments in right are preserved.
If comments='none', no comments are preserved.
If comments='all', all comments in left and right are appended to the beginning of output, although they may previously have been contained within the data in left or right.

See also merge_files.

```python
if comments == 'all':
    file1.process_table(file1, lambda x: x + file2.read_row())
elif comments == 'right':
    file2.process_table(file2, lambda x: x + file1.read_row())
elif comments == 'none':
    outfile = TabFile(output_filename, write=True)
    for x in file1: outfile.write_row(x + file2.read_row())
elif comments == 'all':
    # write comments first
    for x in file1.comment_line_contents(): outfile.write_row(x)
    for x in file2.comment_line_contents(): outfile.write_row(x)
```
for x in file1: outfile.write_row(x + file2.read_row())

else:
    raise ValueError("comments must be one of 'left', 'right', 'none', 'all'")

class TabFileError(Exception):
    def __init__(self, *args):
        self.msg = ''.join(args)
        return

    def __str__(self):
        return repr(self.msg)

class DetectCommentsError(TabFileError):
    def __init__(self, *args):
        super(DetectCommentsError, self).__init__(*args)
        return

class TabFile(object):
    '''Usage: f = TabFile('filename', convert_spaces=True,
    comments=[], column_names=False)

    TabFile is a class for handling tab-delimited files.
    Use convert_spaces=False if you're file is tab-delimited and you
    wish to preserve other whitespace.

    TabFile suports commented lines. Commented lines are not
    recognized as part of the table

    By default, only lines beginning with '#' will be recognized as
    comments (not part of the table). You may specify a list of additional
    keywords using comments=['keyword1','keyword2',etc.]. All lines containing
    that keyword will be recognized as a comment. keywords may be regular
    expressions.
    ['(?!track)', '(?!browser)']
    if column_names = True, the first properly formatted row will be
    treated as column names (i.e. ignored as a comment)
    '''

    def __init__(self, filename, mode='r', convert_spaces=True,
                 compression=None, comments=[], column_names=False):
        self._filename = filename
self._file_extension = os.path.splitext(filename)[1].lstrip(os.extsep)
self.mode = mode
self.open(mode)
self._previous_line = 0
#_detect_comments sets _comment_line_numbers,
#_comment_line_contents
if 'r' in mode:
    try:
        self._detect_comments(comments, column_names)
    except:
        raise DetectCommentsError("Failed with comments=", *comments)

def previous_line(self):
    '''returns the line number of the last line read'''
    return self._previous_line

def _detect_comments(self, comments, column_names=False):
    self.close()
    self.open()
    if not os.path.isfile(self._filename):
        self._column_names = None
        self._comment_line_numbers = []
        self._comment_line_contents = []
        return

    number_keywords = len(comments)
    if number_keywords is not 0:
        searchables = [re.compile(keyword) for keyword in comments]

    if column_names:
        first_valid_line = True
    else:
        first_valid_line = False
        self._column_names = None

    line_number = 0
    comment_line_numbers = []
    comment_line_contents = []
    for line in self._rawiter__():
        line_number += 1
        # blank lines are comments
        if line.lstrip() == '':
            comment_line_numbers.append(line_number)
            comment_line_contents.append(line)
        # lines starting with # are comments
        elif line.lstrip()[0] == '#':
            comment_line_numbers.append(line_number)
            comment_line_contents.append(line)
        elif number_keywords is not 0:
for searchable in searchables:
    if searchable.search(line) is not None:
        comment_line_numbers.append(line_number)
        comment_line_contents.append(line)
        break
    elif first_valid_line:
        first_valid_line = False
        self._column_names = self._parse_line(line)
        comment_line_numbers.append(line_number)
        comment_line_contents.append(line)

    self._comment_line_numbers = comment_line_numbers
    self._comment_line_contents = comment_line_contents
    self.close()
    self.open()

def _parse_line(self, input_string, convert_spaces=True):
    '''parses tab-separated elements in a string and returns a list.
    if convert_spaces=True, _parse_line will treat contiguous
    whitespace as a tab
    '''
    if convert_spaces:
        return input_string.split()
    else:
        return input_string.split('	 ')

def _make_line(self, input_array):
    '''takes a list or array and returns a tab-delimited string'''
    return '\t'.join([str(item) for item in input_array]) + os.linesep

def __enter__(self):
    return self

def __exit__(self, exc_type, exc_val, exc_tb):
    self._file_pointer.close()

def open(self, mode=None):
    '''
    mode can be overridden here but defaults to TabFile.mode
    acts just like the built-in open method in the file class. use
    write=True to write to a file, otherwise it will be opened in
    read-only
    '''
    if mode is None: mode = self.mode
    self._file_pointer = open(self._filename, mode)
def zap(self):
    '''Forces status to not open. Use with caution. This may destroy data.'''
    self._file_pointer = None
    self._previous_line = 0

def close(self):
    '''Works just like the built-in close method in the file class.'''
    if self._file_pointer is None:
        raise IOError('{!s} not open'.format(self._filename))
    else:
        self._file_pointer.close()
    self._previous_line = 0

def read_table(self, override=False):
    '''Returns the contents of a file as a list of rows (with each row as a list). Will ignore any lines that begin with a "#" symbol and truncate any lines that contain a "#" symbol.'''
    if self._file_pointer is None: self.open()
    elif not override:
        raise UserWarning('File already opened and must be closed and re-opened to read whole table. ')
    return [row for row in self]

def read_cols(self, L):
    '''read_cols behaves like read_col but instead of taking a single column number (numbering starts at 0), it takes a list of column numbers and returns a list of partial rows, where each partial row is a list with entries from the appropriate columns in the order specified.

    Tip: use range() to create lists of ordered integers. E.g., range(2, 6) = [2, 3, 4, 5]
    '''
    if self._file_pointer is None: self.open()
    elif type(L) == list:
        raise UserWarning('File already opened and must be closed and re-opened to read all rows of the column.')
    else: raise TypeError('read_cols requires a list of column numbers (0-based) as input')

    my_array = []
    for row in self:
        my_array.append([])
        for n in L: my_array[-1].append(row[n])
    return my_array

def read_col(self, n):
    '''...
returns a list of items in column n (numbering starts at 0) as items
rather than lists. Unlike read_cols and read_table, elements of the
read_col list are not lists, but strings

if self._file_pointer is None: self.open()
elif type(n) is int:
    raise UserWarning('File already opened and must be close and
    re-open to read all rows of the column')
else:
    raise TypeError('read_col requires an integer \n\n(column number, 0-based) as input')

return [row[n] for row in self]

def read_first_col(self):
    '''return a list of items in the first column'''
    return self.read_col(0)

def read_last_col(self):
    '''returns a list of items in the last column'''
    return self.read_col(-1)

def readline(self):
    '''reads one line and returns it. uses a generator, and will raise
    StopIteration if it reaches the EOF.

    readline is deprecated. use x = self.__rawiter__() and x.next()
    
    lines = self.__rawiter__()
    return lines.next()

__rawiter__(self):
    '''like iter, but does not process rows'''
    if self._file_pointer is None: self.open()
    while True:
        self._previous_line += 1
        thisline = self._file_pointer.readline()
        if thisline=='':
            self.close()
            raise StopIteration
        else:
            yield thisline

__iter__(self):
    '''returns the next (or first) line that is not a comment, parsed'''
    for line in self.__rawiter__():
if not self.previous_line() in self.comment_line_numbers():
    yield self._parse_line(line)

def comment_line_contents(self):
    '''returns the list of lines that are comments'''
    return self._comment_line_contents

def comment_line_numbers(self):
    '''returns the list of lines that are comments'''
    return self._comment_line_numbers

def read_row(self):
    '''returns the next (or first) line that is not a comment,
parsed. uses __iter__ as a generator, and simply returns the
next value

    read_row is deprecated. Use x = self.__iter__() and x.next()
    '''
    gen = self.__iter__()
    return gen.next()

def write(self, s):
    '''writes a string directly to a file,
without modification (user must supply \n if desired)
    '''
    if self._file_pointer is None:
        raise IOError('File not open for writing.
    else:
        self._file_pointer.write(s)

def write_rows(self, iterable):
    for row in iterable: self.write_row(row)
    return

def write_row(self, row, separator='\t '):
    '''Writes a list to the file as a line (Tab-delimited). A
different separator may also be specified with separator='x'
    (Note: uses file writelines method)'''
    if self._file_pointer is None:
        raise IOError('File not open for writing.
    else:
        self._file_pointer.writelines([self._make_line(row)])

def write_table(self, table, separator='\t ', override=False,
                column_names=True):
    ''' Writes a table to a file (tab-delimited).
An alternative separator may be specified with separator='x'.
    if column_names = True, column_names will be included as the
    first line

    '''
unless they do not exist.

if self._file_pointer is None:
    raise IOError('File not open for writing.')
elif not override:
    raise UserWarning("File already opened and must be closed and \nre-opened to write the table. You may override this behavior by passing \noverride=True to the method")
try:
    # write column_names if applicable
    if column_names and not self._column_names==None:
        self._file_pointer.writelines(
            self._make_line(self._column_names))
    write_list = itertools.imap(self._make_line, table)
    self._file_pointer.writelines(write_list)
finally:
    self.close()

def process_table(self, output_filename, fnc, column_names=None):

    process_tables2 writes a new file (name is specified with new_file),
    which applies a user-defined function fnc to each row of data in the
    original file.  fnc should yield a row (i.e. a list, array, or
    something else finitely iterable).

    process_tables2 preserves all commented lines and also the line
    which column names, if applicable. The user may specify new column
    names using column_names (a list or other finite iterable), or we will
    use the old column_names, which might not preserve the column labels
    if columns were inserted in the middle of the table

    if self._file_pointer is None: self.open()
    output_file = TabFile(output_filename, 'w')
    if not column_names == None:
        first_valid_line = True
        output_file.set_column_names(column_names)
    elif not self._column_names==None:
        first_valid_line = True
        output_file.set_column_names(column_names)
    else: first_valid_line = False
# treat rawriter as a generator
# we’re going to thread it
lines = self._rawriter__()

# make sure we set column_names correctly
if first_valid_line:
    while True:
        line = lines.next()
        if self.previous_line() in self.comment_line_numbers():
            output_file.write(line)
        else:
            first_valid_line = False
            output_file.write_row(column_names)
            break

# proceed with the remainder
while True:
    line = lines.next()
    if self.previous_line() in self.comment_line_numbers():
        output_file.write(line)
    else:
        output_file.write_row(fnc(self._parse_line(line)))
        output_file.close()

def get_column_names(self):
    '''returns the column names'''
    return self._column_names

def column_dict(self):
    '''returns a dictionary which gives the index corresponding to a
    particular column name'''
    d = {}
    for i in range(len(self._column_names)):
        d[self._column_names[i]] = i
    return d

def set_column_names(self, L):
    self._column_names = L

def write_column_names(self):
    '''writes the stored column names'''
    self.write_row(self._column_names)

def mergesort(self, f, n, numerical=False):
    raise NotImplemented

    # if self._file_pointer is None:
    #     raise UserWarning("File already opened and must be closed
    # and re-opened to mergesort the file. This behavior may not be
    # overridden.
    #")

    # fcontents = self.read_table(f)
    # if numerical==True: fcontents = self._numerize(fcontents, n)
```python
## newcontents = merge, mergesort(fcontents, n)
## writeFile(newcontents, f)

class BedRow(list):  
    '''BEDrows are list, but you can access their chromStart(), chromEnd(), etc. use help for a full list. Uses the same conventions as http://genome.ucsc.edu/FAQ/FAQformat#format1. Note that only the first three entries (chrom, chromStart, chromEnd) are required, so the others may not be defined.'''
    def chrom(self):
        '''returns the name of the chromosome (e.g. chr3, chrY, chr2_random) or scaffold (e.g. scaffold10671)'''
        return self[0]

    def chrom_start(self):
        '''returns the starting position of the feature in the chromosome or scaffold. The first base in a chromosome is numbered 0'''
        return self[1]

    def chromStart(self):
        raise DeprecationWarning()
    return self.chrom_start()

    def chrom_end(self):
        '''returns the ending position of the feature in the chromosome or scaffold. The chromEnd base is not included in the display of the feature. For example, the first 100 bases of a chromosome are defined as chromStart=0, chromEnd=100, and span the bases numbered 0-99'''
        return self[2]

    def chromEnd(self):
        raise DeprecationWarning()
    return self.chrom_end()

    def name(self):
        '''returns the name of the BED line. This label is displayed to the left of the BED line in the Genome Browser window when the track is open to full display mode or directly to the left of the item in pack mode.'''
        return self[3]

    def score(self):
        '''returns the score, a number between 0 and 1000. If the track line useScore attribute is set to 1 for this annotation data set, the score value will determine the level of gray in which this feature is displayed (higher numbers = darker gray)'''
        return self[4]
```

def strand(self):
    '''returns the strand, either '+' or '-' '''
    return self[5]

def thickStart(self):
    '''returns the starting position at which the feature is drawn thickly (for example, the start codon in gene displays)'''
    return self[6]

def thickEnd(self):
    '''returns the ending position at which the feature is drawn thickly (for example, the stop codon in gene displays). '''
    return self[7]

def itemRgb(self):
    '''returns itemRgb, An RGB value of the form R,G,B (e.g. 255,0,0). If the track line itemRgb attribute is set to "On ", this RGB value will determine the display color of the data contained in this BED line. NOTE: It is recommended that a simple color scheme (eight colors or less) be used with this attribute to avoid overwhelming the color resources of the Genome Browser and your Internet browser '''
    return self[8]

def blockCount(self):
    '''return the number of blocks (exons) in the BED line '''
    return self[9]

def blockSizes(self):
    '''returns a comma-separated list of the block sizes. The number of items in this list should correspond to blockCount. '''
    return self[10]

def blockStart(self):
    '''returns a comma-separated list of block starts. All of the blockStart positions should be calculated relative to chromStart. The number of items in this list should correspond to blockCount. '''
    return self[11]

class GzipTabFile(TabFile):
    '''
    For gzip-compressed tab-delimited files
    See Tabfile for usage info
    '''
    def __init__(self, *args, **kwargs):
        super(TabFile, self).__init__(*args, **kwargs)
def _open(self, mode=None):
    '''
    mode can be overridden here but defaults to TabFile.mode
    acts just like the built-in open method in the file class. use
    write=True to write to a file, otherwise it will be opened in
    read-only
    mode
    '''
    if mode is None: mode = self.mode
    if not 'b' in mode: mode += 'b'
    self._file_pointer = gzip.open(self._filename, mode)

class Bzip2TabFile(TabFile):
    '''
    For bzip2-compressed tab-delimited files
    See Tabfile for usage info
    '''
    def __init__(self, *args, **kwargs):
        super(TabFile, self).__init__(*args, **kwargs)
        def _open(self, mode=None):
            '''
            mode can be overridden here but defaults to TabFile.mode
            acts just like the built-in open method in the file class. use
            write=True to write to a file, otherwise it will be opened in
            read-only
            mode
            '''
            if mode is None: mode = self.mode
            if not 'b' in mode: mode += 'b'
            self._file_pointer = bzip.BZ2File(self._filename, mode)

class BedFile(TabFile):
    '''
    A BED file is a type of TabFile, but also defines a method for
    working with
    rows. rows are given as instances of BedRow, instead of lists.
    BEDRows
    inherit all list methods and therefore are compatible with write_row
    .
    BedRow has additional methods for chrom, chromStart, chromEnd, etc.
    For more info, see BedRow
    track, browser lines are treated as comments
    Assumes track row is a comment. Use getTrackLine to see the track
    info
    '''
DEFAULT_BED_COMMENTS = ['(?i)track', '(?i)browser']

def __init__(self, f, additional_comments=[], **kwargs):
    comments = self.DEFAULT_BED_COMMENTS + additional_comments
    TabFile.__init__(self, f, comments=comments, **kwargs)
    self._track_line = None
    # check again for track line, and save it#
    for x in self._comment_line_contents:
        expr = re.compile('(?!track)')
        if expr.search(x) is not None:
            self._track_line = x
        break

def get_track_line(self):
    '''returns the current track line, if any'''
    return self._track_line

def __iter__(self):
    '''returns the next (or first) line that is not a comment, parsed'''
    for line in self._rawiter():
        if not self.previous_line() in self.comment_line_numbers():
            yield BedRow(self._parse_line(line))

class MacsRow(list):
    '''MACS rows are list, but you can access their features as follows:

    chr() or chrom() — chromosome name
    start() or chromStart() — start position, start() is 1-based, chromStart is 0-based (BED)
    end() or chromEnd() — end position, equivalent but chromEnd (BED) is defined as 0-based, exclusive
    length() — length
    summit() — position of summit
    tags() — number of unique tags in the peak region
    pvalue() — returns the -10*log10(pvalue)
    fold_enrichment — returns the fold enrichment
    FDR — returns the FDR in %
    '''

    def chr(self):
        '''returns the name of the chromosome (e.g. chr3, chrY, chr2_random) or scaffold (e.g. scaffold10671)'''
        return self[0]

    def chrom(self):
        '''returns the name of the chromosome (e.g. chr3, chrY, chr2_random) or scaffold (e.g. scaffold10671)'''
        return self[0]
def start(self):
    '''returns the starting position of the feature in the
    chromosome or scaffold. The first base in a chromosome is
    numbered 1'''
    return int(self[1])

def chrom_start(self):
    '''returns the starting position of the feature in the
    chromosome or scaffold. The first base in a chromosome is
    numbered 0'''
    return int(self[1]) - 1

def chromStart(self):
    raise DeprecationWarning()
    return self.chrom_start()

def end(self):
    '''returns the end position, 1-based, inclusive'''
    return int(self[2])

def chrom_end(self):
    '''returns the ending position of the feature in the chromosome
    or scaffold. The chromEnd base is not included in the
display of the feature. For example, the first 100 bases of
    a chromosome are defined as chromStart=0, chromEnd=100, and
    span the bases numbered 0–99'''
    return int(self[2])

def chromEnd(self):
    raise DeprecationWarning()
    return self.chrom_end()

def length(self):
    '''returns the length'''
    return int(self[3])

def summit(self):
    '''returns the position of the summit'''
    return int(self[4]) - 1

def tags(self, type_=int):
    '''returns the number of unique tags in the peak region'''
    return type_(self[5])

def tagsv1(self):
    return self.tags(int)

def tagsv2(self):
    return self.tags(lambda x: int(float(x)))

def pvalue(self, type_=str):
'''returns the $-10\log_{10}(p\text{value})$. preserves the str to eliminate rounding error. use type=float to get a decimal value'''

return type_(self[6])

def fold_enrichment(self, type=str):
    '''returns the fold_enrichment vs control. preserves the str to eliminate rounding error. use type=float to get a decimal value'''
    return type_(self[7])

def FDR(self, type_=str):
    '''returns the FDR (%). preserves the str to eliminate rounding error. use type=float to get a decimal value'''
    return type_(self[8])

class Macs2Row(MacsRow):
    def FDR():
        raise NotImplementedError

def qvalue(self):
    return float_(self[8])

def name(self):
    return self[9]

class MacsFile(TabFile):
    '''A MACS file is a type of TabFile, but also defines a method for working with rows. rows are given as instances of MACSRow, instead of lists. MACSRows inherit all list methods and therefore are compatible with write_row. MacsRow has additional methods for chrom, chromStart, chromEnd, etc. For more info, see MacsRow'''

def __init__(self, f, convert_spaces=True,**kwargs):
    super(MacsFile, self).__init__(f, column_names=True,**kwargs)
    if self.column_dict().has_key('name):
        # MACS2 file
        self.Row = Macs2Row
        self.MACs_version = 2
    else:
        self.Row = MacsRow
        self.MACs_version = 1

def __iter__(self):
    '''returns the next (or first) line that is not a comment, parsed'''
    for line in self._rawiter__():
        if not self.previous_line() in self.comment_line_numbers():
            yield self.Row(self._parse_line(line))

def shift_peaks(f, peak_lengths=2):
shift_peaks takes a file \( f \) (foo.bed) and produces a new file (foo_shifted.bed) with all the sequences shifted (left) by peak_lengths times their length. If peak_lengths is negative they are shifted to the right. Comments are stripped.

```python
x = BedFile(f)
# CHANGE TO USE FILENAME CORRECTION SCHEME
y = BedFile(f[0:-4] + '_shifted.bed')
y.open(write=True)
for peak in x:
    peak_start, peak_end = int(peak[1]), int(peak[2])
    new_peak = peak.copy()
    peak_shift = peak_lengths * (peak_end - peak_start)
    # update start, end
    new_peak[1] = peak_start - peak_shift
    new_peak[2] = peak_end - peak_shift
    if int(new_peak[1]) < 0: new_peak[1] = 0
    y.write_row(new_peak)
```

def _quote(s):
    return "".join(['"', s, '"'])
```

C.6.8 bioplus.wrappers module

---

random_seq(n=1, GC=0.5):

```
return ''.join(["".join(["","", s,"",""])
```
random_seq_generator acts like random_seq, but returns a generator that returns the nucleotides one by one

myError = ValueError(‘randomN requires a positive integer n (default = 1) \ and a probability GC (float 0.0 to 1.0)’)

#AT = 1 − GC
if not type(GC)==float or GC == 0 or GC == 1: raise myError
elif not GC >= 0 and GC <= 1: raise myError
elif not type(n)==int: raise myError
elif n < 1: raise myError
else:
    randomGenerator = itertools.repeat(random.random())
gc_or_at = lambda x: random.choice([‘G’, ‘C’]) \ if x < GC else random.choice([‘A’, ‘T’])
return itertools.imap(gc_or_at, randomGenerator)

Source Code C.29: bioplus.wrappers module
References


[31] Neil A Farrow, Ouwen Zhang, Julie D Forman-Kay, and Lewis E Kay. A heteronuclear correlation experiment for simultaneous determination of 15n longitudinal decay and chemical exchange rates of systems in slow equilibrium. *Journal


[42] Jun Zhang, Michael J Chalmers, Keith R Stayrook, Lorri L Burris, Yongjun Wang, Scott A Busby, Bruce D Pascal, Ruben D Garcia-Ordonez, John B Bruning, Mon-


[48] Kathleen M Scully, Eric M Jacobson, Kristen Jepsen, Victoria Lunyak, Hector Via-
diu, Catherine Carrière, David W Rose, Farideh Hooshmand, Aneel K Aggarwal, and
Michael G Rosenfeld. Allosteric effects of pit-1 dna sites on long-term repression
sciencemag.org/content/290/5494/1127.short.

Hager. The glucocorticoid receptor: rapid exchange with regulatory sites in living
org/cgi/content/abstract/sci;287/5456/1262.

[50] Diana A Stavreva, Waltraud G Müller, Gordon L Hager, Carolyn L Smith, and
James G McNally. Rapid glucocorticoid receptor exchange at a promoter is coupled
to transcription and regulated by chaperones and proteasomes. Molecular and cel-
lular biology, 24(7):2682–2697, 2004. URL http://mcb.asm.org/content/24/7/
2682.short.

[51] Signe Holmbeck, H Jane Dyson, and Peter E Wright. Dna-induced conformational
changes are the basis for cooperative dimerization by the dna binding domain of

gene regulation by lambda phage repressor. Proceedings of the National Academy
1129.short.

[53] James P Robblee, Michael T Miura, and David L Bain. Glucocorticoid receptor–
promoter interactions: Energetic dissection suggests a framework for the specificity


[59] Frank Delaglio, Stephan Grzesiek, Geerten W Vuister, Guang Zhu, John Pfeifer, and AD Bax. Nmrpipe: a multidimensional spectral processing system based on


[65] Yiwen Chen, Nicolas Negre, Qunhua Li, Joanna O Mieczkowska, Matthew Slattery, Tao Liu, Yong Zhang, Tae-Kyung Kim, Housheng Hansen He, Jennifer Zieba, Yijun Ruan, Peter J Bickel, Richard M Myers, Barbara J Wold, Kevin P White, Jason D


[78] Thomas D Schneider. Information content of individual genetic sequences. Journal

362


coid receptor agonist, with full anti-inflammatory properties in human ocular cells. 


[104] Donna Karolchik, Angela S Hinrichs, Terrence S Furey, Krishna M Roskin, Charles W Sugnet, David Haussler, and W James Kent. The ucs table browser data retrieval


Publishing Agreement

It is the policy of the University to encourage the distribution of all theses, dissertations, and manuscripts. Copies of all UCSF theses, dissertations, and manuscripts will be routed to the library via the Graduate Division. The library will make all theses, dissertations, and manuscripts accessible to the public and will preserve these to the best of their abilities, in perpetuity.

I hereby grant permission to the Graduate Division of the University of California, San Francisco to release copies of my thesis, dissertation, or manuscript to the Campus Library to provide access and preservation, in whole or in part, in perpetuity.

[Signature]
Author Signature

[Signature]
6/14/2013
Date