

Oligo – Anleitung

by Aaron Sievers

Inhaltsverzeichnis

Inhalt

Inhaltsverzeichnis	2
Installation Oligo Software-Package	5
Download der Software	5
Installation Python	6
Python Bibliotheken	7
Test der Installation.....	8
Download von Genomen (Sequenzdaten + Gen-Annotationen)	9
Accession ID.....	9
Download	10
Sequence (*.seqs) Dateien	11
Sequenzsuche.....	12
Suche mit Sequenzdateien	13
Suche in Genomen	13
Lesen der Ergebnisse.....	14
Suche nach Logos	15
Suche nach Tandem-Repeats (TR) / Kettensuche	16
Kettensuche auf Sequenzdateien / in Genomen	16
<i>k</i> -mer Suche.....	18
<i>k</i> -mer Suche auf Sequenzdateien / in Genomen	18
Einlesen von <i>k</i> -mer Spektren.....	19
Darstellung von <i>k</i> -mer Spektren.....	19
Genom-Regionen (Gene, Introns, Intergenic Region)	20
Gene	20
Intergenische Regionen.....	21
Exons	21
Introns	21
Codierende Sequenzen (CDS).....	22
Protein-Codierende (PC) Gene	22
Nicht-Protein-Codierende (nPC) Gene	22
Pseudogene	23
Non-Coding-RNA Gene	23
Non-Coding RNAs (ncRNA)	23
Long Non-Coding RNAs (lncRNA)	24

Micro RNA (miRNA	24
Sequencing Gaps	24
Weitere.....	24
Repeat-Masker	25
Locus-Operationen und Filter.....	27
Simple Filter.....	27
Speichern von Loci.....	27
Shuffle Loci	27
Sort Loci	28
Cluster	28
Parameterbestimmung	28
Statistische Auswertung.....	30
Speichern & Lesen von Clustern.....	31
Cluster in Loci Convertieren	32
Vorberechnete Distanzen / Distanzmatrix	32
Clustering in Correlationsdaten.....	33
<i>k</i> -mer Analyse.....	36
Zusammenfassen von Spektren	36
Korrelation von <i>k</i> -mer-Spektren.....	36
Korrelation vieler Spektren	37
Darstellung als Heatmap	37
Mean Correlation	38
Mean Correlation VS. <i>k</i>	39
Correlation Contribution	41
Most Contributing Words.....	42
Different Complexities / Repeats	44
Längenverteilungen.....	46
Loci in other Loci	48
Next-Neighbour-Distanzen (1D).....	49
Motivation	49
Berechnung der Distanzen	49
Distanzverteilungen.....	50
Berechnung und Speicherung von Distanzverteilungen	50
Modellrechnung	51
Darstellung von Distanzverteilungen	51
Maps.....	52

Erstellen aus Loci	52
Visualisierung	53
Kurven	53
Chromosomen-Bilder	53
Map-Korrelation	55
Boot-Correlation.....	57
Shuffle.....	58
Deutung der Ergebnisse:	58
Masken	59
Masken Erstellen	59
Masken Kombinieren	59
Masken speichern & laden	59
Masken laden/verwenden	59
Masken für Mensch und Maus.....	60
Erstellen von Standard-Masken für Genome	60
Density Profile	60
Visualisieren	61
Kombinieren von Density Profiles	62
3D Density Profiles	63
Einlesen von Hi-C-Daten	63
Distanz-Heatmaps	64
Berechnung 3D Density Profiles	65
Tandem-Repeat-Modelle	65
Theoretischer Hintergrund.....	65
Anwendung in Oligo	67
Anhang:	69
Windows Kommandozeile.....	69

Installation Oligo Software-Package

Im Folgenden wird beschrieben, wie das Oligo-Software-Paket erhalten und installiert werden kann. Es handelt sich um die Inhouse-Software, entwickelt von Aaron Sievers.

Bei Verwendung in Publikationen folgende Referenz angeben:

<https://www.ncbi.nlm.nih.gov/pubmed/28422050>

K-mer Content, Correlation, and Position Analysis of Genome DNA Sequences for the Identification of Function and Evolutionary Features. Sievers et al. 2017

Download der Software

Das Software-Paket kann, in gezippter (komprimierter) Form heruntergeladen werden von:

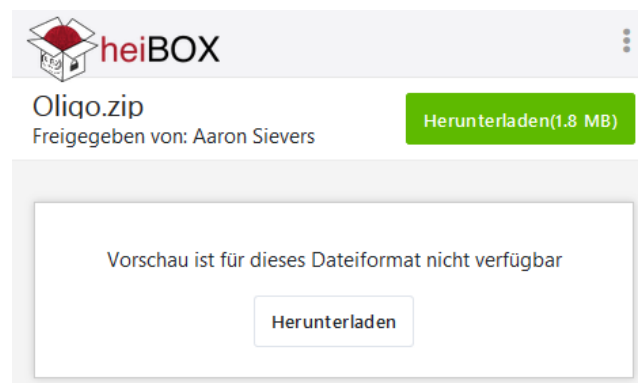
HeiBox:

<https://heibox.uni-heidelberg.de/d/cc9c4ad1e47841efa247/>

Das dort verlangte Passwort lautet:

KIP-Genomik

Es sollte ein Fenster dieser Art erscheinen:



Im komprimierten Verzeichnis befindet sich ein Ordner mit der Bezeichnung „Oligo“. Dieser muss entpackt und kann beliebig im Dateisystem abgelegt werden. Es empfiehlt sich den entsprechenden Pfad (Ort) zu notieren oder direkt in diesem Verzeichnis zu arbeiten.

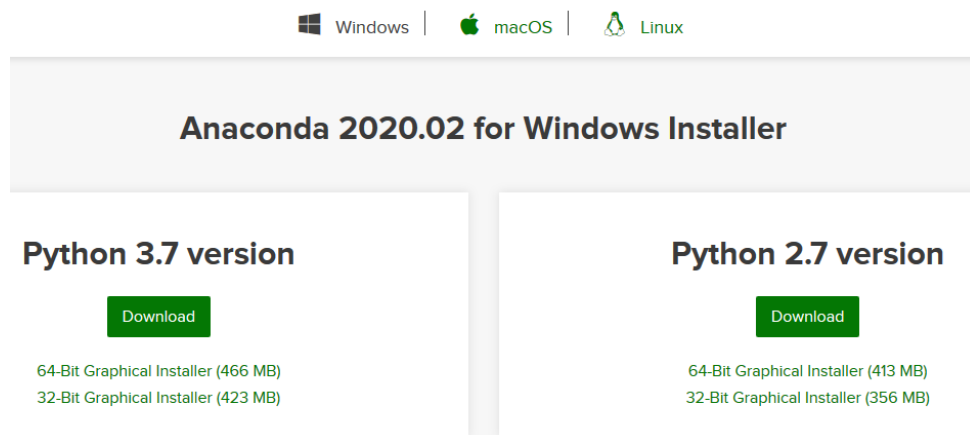
z.B.:

Name	Änderungsdatum	Typ	Größe
data	20.04.2020 09:57	Dateiordner	
Oligo	20.04.2020 09:57	Dateiordner	
my oligo script.py	20.04.2020 09:57	PY-Datei	0 KB
my second oligo script.py	20.04.2020 09:57	PY-Datei	0 KB

Installation Python

Oligo sollte auf dem derzeitigen stand mit Python 2.7 und auch Python 3.7 kompatibel sein. Grundsätzlich ist die aktuelle Version >3.6 empfehlenswert. Während es generell möglich ist die notwendigen Pakete einzeln jeder Python-Distribution hinzuzufügen, empfehle ich wegen der Einfachheit *Anaconda*:

<https://www.anaconda.com/distribution/#download-section>



Anaconda kommt mit vielen nützlichen Paketen (z.B. numpy, matplotlib, pandas) vorinstalliert und bietet die Möglichkeit weitere benötigte Pakete einfach zu installieren.

Nach dem Download kann Anaconda wie andere Applikationen mit einem grafischen Interface leicht installiert werden.

Ich empfehle bei der entsprechenden Frage Anaconda zur Standard-Python-Umgebung zu machen und einen Eintrag unter PATH erstellen zu lassen (diese Option ist standardmäßig nicht ausgewählt).

Packages required for Oligo:

(only relevant if you install the packages manually)

```
biopython
pandas
scipy
sklearn (conda install -c anaconda scikit-learn)
matplotlib
seaborn
```

Python Bibliotheken

Lediglich eine (nicht-standard) Bibliothek wird zusätzlich zu den durch Anaconda vorinstallierten Bibliotheken benötigt. Es handelt sich um **Biopython**, ein Software-Paket zum Bearbeiten von Standard-Formaten der Bioinformatik, darunter besonders wichtig, Sequenzdateien.

Bei erfolgreicher Anaconda-Installation sollte folgender Befehl eingegeben in Die Kommandozeile (siehe unten):

```
conda install -c anaconda biopython
```

Sollte es dabei zu einem Fehler kommen wäre folgender Befehl einen Versuch wert:

```
pip install biopython
```

Sollte auch dies scheitern, bleibt die Möglichkeit einer manuellen Installation. Dazu das Packet hier downloaden:

<https://pypi.org/project/biopython/1.76/>

Direktdownload:

https://files.pythonhosted.org/packages/d1/c4/9a1a1d79e228cd7626b2fe5c5386cdb67a63249e03737ad7dd8a5cb4d36e/biopython-1.76-cp27-cp27m-macosx_10_6_intel.whl

Dann folgenden Befehl ausführen:

```
pip install C:/Users/Sievers/Downloads/biopython-1.76-cp27-cp27m-macosx_10_6_intel.whl
```

(logischerweise mit dem im Einzelfall passenden Dateipfad)

Test der Installation

Zum Testen der Installation kann einfach ein Python-Script erstellt werden, welches das Oligo-Packet importiert. Dieses kann wie folgt aussehen:

Name	Änderungsdatum	Typ	Größe
 my_oligo_script.py	18.03.2021 12:32	PY-Datei	1 KB

Inhalt "my oligo script.py":

```
import sys
sys.path.insert(1, 'N:/Software/KIP/')
import Oligo
```

Notiz: Selbstverständlich muss beim Pfad der passende Pfad zu deiner Oligo-Installation hinterlegt werden.

Dieses Script kann dann über die Kommandozeile* ausgeführt werden über:

* Ein Kommandozeilenfenster kann man in Windows aufrufen über folgende (vorinstallierte Anwendung):

C:/Windows/System32/cmd.exe

Download von Genomen (Sequenzdaten + Gen-Annotationen)

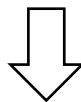
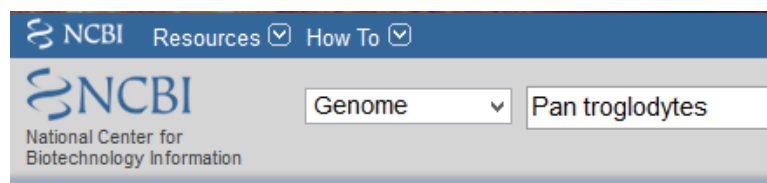
Oligo besitzt ein Modul zum automatischen Download von Genomen, inklusive Sequenzen und Genom-Annotationen (d.h. Positionen und Aufbau aller Gene).

Accession ID

Benötigt wird nur eine sogenannte „Accession ID“. Diese findet man auf der Website des NCBI:

<https://www.ncbi.nlm.nih.gov/>

Sucht man dort beispielsweise (in der Kategorie „Genome“) nach *Pan troglodytes* (lat. Schimpanse), erhält man folgendes Bild:



Organism Overview : [Genome Assembly and Annotation report \[6\]](#) : [Organelle Annotation Report \[1\]](#)

ID: 202



Pan troglodytes (chimpanzee)

The chimpanzee genome sequence is an important tool in understanding primate evolution

Lineage: Eukaryota[823]; Metazoa[445]; Chordata[272]; Craniata[266]; Vertebrata[266]; Euteleostomi[261]; Mammalia[122]; Eutheria[115]; Euarchontoglires[40]; Primates[22]; Haplorrhini[17]; Catarrhini[14]; Hominidae[6]; Pan[2]; Pan troglodytes[1]

Pan troglodytes, or chimpanzee, is a primate very closely related to humans. The chimpanzee and other apes are most closely related to humans, followed by Old World monkeys; including the rhesus macaque and baboon. The chimpanzee is an important model to study biology, disease, and evolution. Research with *Pan troglodytes* has provided [More...](#)

Summary

Sequence data: genome assemblies: 6; sequence reads: 3 (See [Genome Assembly and Annotation report](#))
Statistics: median total length (Mb): 3050.4
median protein count: 40465
median GC%: 40.8085
NCBI Annotation Release: 105

Publications (limited to 20 most recent records)

1. A mitogenomic phylogeny of living primates. Finstermeier K, et al. PLoS One 2013
2. Major chimpanzee-specific structural changes in sperm development-associated genes. Kim RN, et al. Funct Integr Genomics 2011 Sep
3. Chimpanzee and human Y chromosomes are remarkably divergent in structure and gene content. Hughes JF, et al. Nature 2010 Jan 28

[More...](#)

Representative (genome information for reference and representative genomes)

Reference genome:

◦ [Pan troglodytes Clint_PTRv2](#)

Submitter: The International Chimpanzee Chromosome 22 Consortium

Loc	Type	Name	RefSeq	INSDC	Size (Mb)	GC%	Protein	rRNA	tRNA	Other RNA	Gene	Pseudogene
Chr	1	NC_036879.1	CM009238.2	224.24	41.9	7,926	1	46	2,012	3,826	673	
Chr	2A	NC_036880.1	CM009239.2	108.02	41.1	2,720	1	4	896	1,354	225	
Chr	2B	NC_036881.1	CM009240.2	128.76	39.6	3,140	1	4	810	1,386	262	
Chr	3	NC_036882.1	CM009241.2	196.56	39.7	4,888	1	4	1,339	2,248	436	
Chr	4	NC_036883.1	CM009242.2	189.15	38.3	3,223	-	1	1,175	1,816	369	
Chr	5	NC_036884.1	CM009243.2	159.32	39.7	2,971	-	18	1,032	1,768	334	
Chr	6	NC_036885.1	CM009244.2	168.37	39.7	3,999	1	133	1,249	2,305	435	
Chr	7	NC_036886.1	CM009245.2	156.05	40.9	3,783	-	13	1,120	2,035	394	
Chr	8	NC_036887.1	CM009246.2	143.34	40.2	2,771	-	6	900	1,515	286	
Chr	9	NC_036888.1	CM009247.2	110.51	41.5	3,241	-	3	795	1,496	278	
Chr	10	NC_036889.1	CM009248.2	129.81	41.8	3,702	-	3	796	1,539	279	
Chr	11	NC_036890.1	CM009249.2	130.78	41.6	4,957	-	14	939	2,276	384	
Chr	12	NC_036891.1	CM009250.2	131	40.9	4,362	-	10	1,061	1,985	346	

Interessant ist insbesondere die Tabelle mit der Überschrift „Reference Genome“ und dort die Spalte „RefSeq“ (kurz für Reference Sequence). Die in dieser Spalte befindlichen IDs (z.B. NC_036879.1) sind die besagten „Accession IDs“.

Diese können einfach kopiert oder abgeschrieben werden, letztlich sollten sie in einer Python-Liste stehen:

z.B.:

```
ids = [  
    'NC_036879.1',  
    'NC_036880.1',  
    'NC_036881.1',  
    'NC_036882.1',  
    'NC_036883.1',  
    'NC_036884.1',  
    'NC_036885.1',  
    'NC_036886.1',  
    'NC_036887.1',  
    'NC_036888.1',  
    'NC_036889.1',  
    'NC_036890.1',  
    'NC_036891.1',  
    'NC_036892.1',  
    'NC_036893.1',  
    'NC_036894.1',  
    'NC_036895.1',  
    'NC_036896.1',  
    'NC_036897.1',  
    'NC_036898.1',  
    'NC_036899.1',  
    'NC_036900.1',  
    'NC_036901.1',  
    'NC_036902.1',  
    'NC_006492.4',  
]
```

Download

Die Sequenzen können anschließend durch folgenden Befehl runtergeladen werden:

```
Oligo.File.download_genbanks(<ids>, labels=<labels>)
```

Der erste Parameter ist einfach die besagte IDs-Liste, der zweite Parameter ist eine zweite Liste von beliebigen zusätzlichen Labels, die man den Sequenzen geben möchte.

Wenn es sich um Teile eines Genoms handelt, wie im Beispiel, dann sollte das Label „Genome“ nicht vergessen werden, um später leicht das gesamte Genom aufrufen zu können.

Bsp.:

```
import Oligo  
  
ids = [  
    'NC_036879.1',  
    'NC_036880.1',  
]  
  
Oligo.File.download_genbanks(ids, labels=['Genome'])
```

Sequence (*.seqs) Dateien

Eine Reihe von Metadaten wird, beim Download, automatisch in einer Datei mit der Bezeichnung default.seqs gespeichert. Die Datei ist menschenlesbar. Es handelt sich um eine durch Tab („\t“) getrennte Tabelle mit den Spalten:

id: Accession ID der Sequenz (als Referenz für spätere Publikationen extrem wichtig)

name: Name der Sequenz, z.B. c1 für Chromosom 1

organism: lateinischer Name des Organismus, z.B. *Homo sapiens*

topology: Topologie des DNA/RNA Moleküls (in der Regel linear oder circular)

length: Die Länge der Sequenz in Basenpaaren (bp)

file: Der Dateipfad der Sequenzdatei

labels: hinzugefügte Labels (durch Komma getrennt), z.B. Genome

Eine valide Sequence-Datei könnte so aussehen:

id	name	organism	topology	length	file	labels
NC_000001.11	c1	Homo sapiens	linear	248956422	Homo_Sapiens_c1.gb	Genome, DNA
NC_000002.12	c2	Homo sapiens	linear	242193529	Homo_Sapiens_c2.gb	Genome, DNA
NC_000003.12	c3	Homo sapiens	linear	198295559	Homo_Sapiens_c3.gb	Genome, DNA
NC_000004.12	c4	Homo sapiens	linear	190214555	Homo_Sapiens_c4.gb	Genome, DNA
NC_000005.10	c5	Homo sapiens	linear	181538259	Homo_Sapiens_c5.gb	Genome, DNA
NC_000006.12	c6	Homo sapiens	linear	170805979	Homo_Sapiens_c6.gb	Genome, DNA
NC_000007.14	c7	Homo sapiens	linear	159345973	Homo_Sapiens_c7.gb	Genome, DNA
NC_000008.11	c8	Homo sapiens	linear	145138636	Homo_Sapiens_c8.gb	Genome, DNA
NC_000009.12	c9	Homo sapiens	linear	138394717	Homo_Sapiens_c9.gb	Genome, DNA
NC_000010.11	c10	Homo sapiens	linear	133797422	Homo_Sapiens_c10.gb	Genome, DNA
NC_000011.10	c11	Homo sapiens	linear	135086622	Homo_Sapiens_c11.gb	Genome, DNA
NC_000012.12	c12	Homo sapiens	linear	133275309	Homo_Sapiens_c12.gb	Genome, DNA
NC_000013.11	c13	Homo sapiens	linear	114364328	Homo_Sapiens_c13.gb	Genome, DNA
NC_000014.9	c14	Homo sapiens	linear	107043718	Homo_Sapiens_c14.gb	Genome, DNA
NC_000015.10	c15	Homo sapiens	linear	101991189	Homo_Sapiens_c15.gb	Genome, DNA
NC_000016.10	c16	Homo sapiens	linear	90338345	Homo_Sapiens_c16.gb	Genome, DNA
NC_000017.11	c17	Homo sapiens	linear	83257441	Homo_Sapiens_c17.gb	Genome, DNA
NC_000018.10	c18	Homo sapiens	linear	80373285	Homo_Sapiens_c18.gb	Genome, DNA
NC_000019.10	c19	Homo sapiens	linear	58617616	Homo_Sapiens_c19.gb	Genome, DNA
NC_000020.11	c20	Homo sapiens	linear	64444167	Homo_Sapiens_c20.gb	Genome, DNA
NC_000021.9	c21	Homo sapiens	linear	46709983	Homo_Sapiens_c21.gb	Genome, DNA
NC_000022.11	c22	Homo sapiens	linear	50818468	Homo_Sapiens_c22.gb	Genome, DNA
NC_000023.11	cX	Homo sapiens	linear	156040895	Homo_Sapiens_cX.gb	Genome, DNA
NC_000024.10	cY	Homo sapiens	linear	57227415	Homo_Sapiens_cY.gb	Genome, DNA

Generell ist es natürlich auch möglich die sogenannten Genbank (*.gb) Dateien manuell runterzuladen und ohne Metadaten zu verwenden oder manuell Daten in die *.seqs Datei einzutragen, die automatische Verwendung ist allerdings wesentlich komfortabler und weniger fehleranfällig.

Sequenzsuche

Eine Hauptfunktion des Oligo-Software-Pakets ist die Suche. Die einfachste Möglichkeit einer solchen Suche ist die Suche nach einer definierten Sequenz (query sequence) innerhalb einer anderen Sequenz (data sequence).

Der hierfür verwendete Befehl ist:

Oligo.Search.search_seq()

Parameter:

```
data_seq (str) :      sequence to search other sequence (query) in.
query_seq (str) :      sequence to search for
output_filename (str) : name of the file were results are saved.
data_seq_name (str) :  name of the data sequence
query_seq_name (str) :  name of the query sequence
allowed_mismatches (int) : number of allowed mismatches in the search hits.
search_type (int) :    type of base identification algorithm (affects
                        performance)
                        0 : only exact matches between query and data bases are considered
                        1 : R, D, W, H, V, M, Y, P, N, - allowed in query sequence.
                        2 : Everything allowed in query and data sequence.

clear_file (bool) :    if the output file is cleared before adding new
                        results.
verbose (int) :        if docstrings are printed.
```

Bsp.:

```
data_seq = 'TACGATCGATCGACTAGCTAGCTACGATCGACTAGCTG'
query_seq = 'GAT'

Oligo.Search.search_seq(data_seq, query_seq, output_filename='search_test.loci',
allowed_mismatches=0, search_type=0)
```

Erzeugt eine Ergebnisdatei (Loci-Datei) mit folgendem Inhalt:

\$head	position	strand	number of mismatches
\$data name:	TACGATCGATCGACTAGCTA		
\$name:	GAT		
\$length:	3		
3	0	0	
4	1	0	
7	0	0	
8	1	0	
25	0	0	
26	1	0	

Die erste Zeile gibt die Spaltendefinitionen also „position“ (die Position des hits in der data sequence), der Strang (0 für Strangrichtung, 1 für Gegenrichtung), dann folgen die Namen der beiden Sequenzen und die Länge des Queries.

Danach folgt die Tabelle der eigentlichen Ergebnissen. Gefunden wurde die Sequenz GAT also an den Positionen 3, 4, 7, 8, 25, 26 Prüfung:

```

      TACGATCGATCGACTAGCTAGCTACGATCGACTAGCTG
0      TAC
1      ACG
2      CGA
3      GAT
4      ATC
7      GAT
8      ATC
25      GAT
26      ATC

```

Notiz: ATC ist das reverse Komplement von GAT also ein Treffer auf dem Gegenstrang.

Suche mit Sequenzdateien

Eher als die Definition von Sequenzen im Code, werden Sequenzen aus Sequenzdateien ausgelesen. Dies kann ebenfalls in Oligo erfolgen. Der Befehl lautet:

```
Oligo.File.read_sequence_file(<query_seq_filename>)
```

Bsp.:

```

import Oligo

data_seq_id, data_seq = Oligo.File.read_sequence_file('data/Escherichia coli str. K-12 substr. MG1655.gb')
query_seq_id, query_seq = Oligo.File.read_sequence_file('data/alu_winner.fasta')

Oligo.Search.search_seq(data_seq, query_seq, output_filename='Alu_in_EColi.loci', allowed_mismatches=3,
search_type=0)

```

Notiz: Die verwendete **alu_winner.fasta** Datei kann ebenfalls von HeiBox heruntergeladen werden.

Suche in Genomen

Noch komfortabler gestaltet sich die Sache mit einer validen Sequence-Datei (*.seqs). In diesem Fall kann man das gesamte Genom eines Organismus laden über:

```
genome = Oligo.File.read_genome(<organism>, seqs_filename=<seqs filename>)
```

Der erste Parameter ist die Bezeichnung des Organismus, wie sie auch im *.seqs-File verwendet wird (z.B. Homo sapiens). Der Parameter „seqs_filename“ ist optional und gibt die Möglichkeit neben der default.seqs weitere Sequence Dateien zu verwenden (für den Anfang ist diese Option nicht unbedingt notwendig).

Bsp.:

```

import Oligo

genome = Oligo.File.read_genome('Homo sapiens', seqs_filename=Oligo.File.search('Homo sapiens.seqs'))
query_seq_id, query_seq = Oligo.File.read_sequence_file('data/alu_winner.fasta')

```

```
for chromo in genome:
    chromo.load_seq()
    Oligo.Search.search_seq(data_seq=chromo.get_seq(), query_seq=query_seq, data_seq_name=str(chromo),
output_filename='%s_Alu_winner.loci' % str(chromo), allowed_mismatches=0, search_type=0)
    chromo.unload_seq()
```

Notiz: Erst die Methode „load_seq()“ lädt tatsächlich die Sequenzdatei, d.h. erst hier wird Arbeitsspeicher verbraucht. Da es unnötig ist alle Sequenzen im RAM zu belassen, wird nach jeder Suche mittels „unload_seq()“ die Sequenz wieder verworfen. Dies verhindert Probleme mit dem Arbeitsspeicher, ist aber (besonders bei kleinen Genomen) nicht notwendig.

Lesen der Ergebnisse

Loci-Dateien können als Loci-Objekte wieder geladen werden über den Befehl:

```
Oligo.Locus.read(<input_filename>)
```

Bsp.:

```
Import Oligo
```

```
loci = Oligo.Locus.read('Homo sapiens c1_Alu_winner.loci')
Oligo.Loci.basic_stats(loci)
```

Hier werden auch eine Reihe von statistischen Basisinformationen ausgegeben, was zur Verifikation einer erfolgreichen Suche sehr hilfreich sein kann.

```
Basic Stats for 34305 loci:
n = 34305
sum = 583185bp
coverage sum = 583185bp
mean length = 17.0bp
std length = 0.0bp
range = 26815-248943391
n density = 0.0001378172581
cov. density = 0.0023428933877
```

Weitere Analysemöglichkeiten für Loci-Objekte werden in den folgenden Kapiteln besprochen.

Suche nach Logos

Analog zur Suche nach spezifischen Sequenzen, kann auch nach Sequenzlogos, repräsentiert durch Wahrscheinlichkeitsmatrizen gesucht werden.

Hierzu dient der Befehl:

`Oligo.Search.search_logo()`

Beispiel:

```
import Oligo

query_matrix = Oligo.Matrix.read_jaspar('E:/Daten/CTCF Binding
Sites/MA0139.1.jaspar')

data_seq =
'TGGCCACCAGGGGCGCTATGGCCACCAGGGGCGCTATGGCCACCAGGGGCGCTATGGCCACCAGGGGCGCTA'

Oligo.Search.search_logo(data_seq, query_matrix, output_filename='Logo_test2.loci',
t_min=1e6)
```

Das Ergebnis ist folgende Datei:

```
#date: 2020-04-30 13:04:01.858000
start  length strand
0      19      ?
19     19      ?
38     19      ?
```

Notiz: Drei positive Ergebnisse sind gut, da ich als data_seq dreimal die CTCF-Consensus-Sequenz verwendet hatte.

Notiz: Der Wert für t_min ist momentan etwas Tricky. Eine Heuristik, wie man auf einen sinnvollen Wert kommt erläutere ich noch in einer kommenden Version. Wichtig ist, dass jeder Wert über 1 bedeutet, dass der Wert über dem Erwartungswert liegt.

Suche nach Tandem-Repeats (TR) / Kettensuche

Ähnlich wie die Suche nach einzelnen Sequenzen, funktioniert auch die Suche nach Tandem-Wiederholungen (engl. Tandem Repeats, kurz: TR). Dabei handelt es sich von Head-to-Tail Wiederholungen bzw. Ketten einer vorgegebenen Wiederholungs-Einheit (engl. Repeat Unit), im Algorithmus auch (aus technischen Gründen) als „seed“ bezeichnet.

Der Befehl funktioniert analog zur Sequenzsuche:

Oligo.Search.search_chains()

Parameter:

data_seq (str) :	sequence to search TRs in.
data_seq_name (str) :	name of the data sequence.
chain_seed_seq (str) :	sequence of the repeat unit (seed).
query_seq_name (str) :	name of the repeat unit.
output_filename (str) :	name of the file where results are saved.
search_type (int) :	type of base identification algorithm (affects performance) 0 : only exact matches between query and data bases are considered 1 : R, D, W, H, V, M, Y, P, N, - allowed in query sequence. 2 : Everything allowed in query and data sequence.
min_length (int) :	minimum length of a chain (in bp) to be counted. Be careful! Small chains are very frequent.
min_purity (float) :	minimum purity for a chain to be counted (1.0 = only exact matches)
allow_shifts (bool) :	if shifts of chains are counted (e.g. if CACACA counts as chain of the unit AC)

Bsp.:

```
Oligo.Search.search_chains(data_seq='AAAAAAGCTGCTCGTCAAATCGTCG',
chain_seed_seq='A', output_filename='chain_search_test.loci',
search_type=0, min_length=3)
```

```
#date:      2020-04-21 11:23:38.773000
#RESULTS: generated by search_chains.cpp
$data sequence name:
$chain seed sequence name: (A)n
$chain seed length: 1
$allow shifts: 1
$minimum length:3
$minimum purity:0.000000
$maximum gap size: 1
location    length    mismatches
0           6         0
16          3         0
```

Kettensuche auf Sequenzdateien / in Genomen

Die Suche mit Sequenzdateien und Genomen funktioniert analog zur direkten Sequenzsuche aus dem Kapitel oben:

Bsp.:

```
import Oligo
```



```

genome = Oligo.File.read_genome('Homo sapiens', seqs_filename=Oligo.File.search('Homo
sapiens.seqs'))
chain_seed = 'A'

for chromo in genome:
    chromo.load_seq()
    Oligo.Search.search_chains(data_seq=chromo.get_seq(), chain_seed_seq=chain_seed,
data_seq_name=str(chromo), chain_seed_seq_name='(A)n', output_filename='%s_(A)n.loci'
% str(chromo), min_length=20)
    chromo.unload_seq()

```

Die Ausgabedateien sind erneut im Loci-Format und können darum auf die gleiche Weise geöffnet werden:

Bsp.:

```

Import Oligo

loci = Oligo.Locus.read('Homo sapiens c1_(A)n.loci')
Oligo.Loci.basic_stats(loci)

```

Hier werden auch eine Reihe von statistischen Basisinformationen ausgegeben, was zur Verifikation einer erfolgreichen Suche sehr hilfreich sein kann.

```

Basic Stats for 75312 loci:
n = 75312
sum = 2089008bp
coverage sum = 2089008bp
mean length = 27.738049713193117bp
std length = 12.100181241774154bp
range = 35483-248943665
n density = 0.000302569402881
cov. density = 0.00839268513881

```

Weitere Analysemöglichkeiten für Loci-Objekte werden in den folgenden Kapiteln besprochen.

k-mer Suche

Die *k*-mer Suche extrahiert jedes (DNA-)Wort einer spezifizierten Länge *k* aus einer gegebenen Data-Sequenz. Im Gegensatz zur bisherigen Suche wird also nicht nach einer definierten Sequenz gesucht. Der Befehl funktioniert dennoch analog:

```
Oligo.Search.search_kmer()
```

Parameter:

<code>data_seq :</code>	Sequenz in welcher die Worte gesucht werden
<code>k :</code>	Länge der gesuchten Worte in bp
<code>output_filename :</code>	Name für die Ausgabedatei
<code>data_seq_name</code>	Name der Data-Sequenz
<code>loci</code>	Liste von Loci-Objekten. Wenn angegeben wird lediglich innerhalb dieser Objekte gesucht (z.B. für Suchen nur in Genen)

Bsp.:

```
Oligo.Search.search_kmer('ACGTACGCA', k=2, output_filename='test.kmer')
```

Ergebnisdatei:

```
$Sequence Name: ACGTACGCA
Row Name   Frequency
AC    2
GC    1
CA    1
GT    1
TA    1
CG    2
```

Da keine Positionen aufgenommen werden, handelt es sich nicht um Loci-Dateien.

k-mer Suche auf Sequenzdateien / in Genomen

Diese Suche funktioniert analog zu den anderen Suchen.

Bsp.:

```
genome = Oligo.File.read_genome('Homo sapiens',
seqs_filename=Oligo.File.search('Homo sapiens.seqs'))
k = 3

for chromo in genome:
    chromo.load_seq()
    Oligo.Search.search_kmer(data_seq=chromo.get_seq(), k=k,
data_seq_name=str(chromo), output_filename='%s_k=3.kmer' % str(chromo))
    chromo.unload_seq()
```

Einlesen von k-mer Spektren

Die entstehenden *.kmer Dateien können als kmer-Spektren eingelesen werden:

```
spectrum = Oligo.Kmer.KmerSpectrum.read(<input_filename>)
```

Bsp.:

```
spectrum = Oligo.Kmer.KmerSpectrum.read('Homo sapiens_c1_k=3.kmer')  
spectrum.basic_stats()
```

Wie für Loci können einige statistische Basisinformationen ausgegeben werden:

```
n = 64  
f = 1.0
```

Notiz:

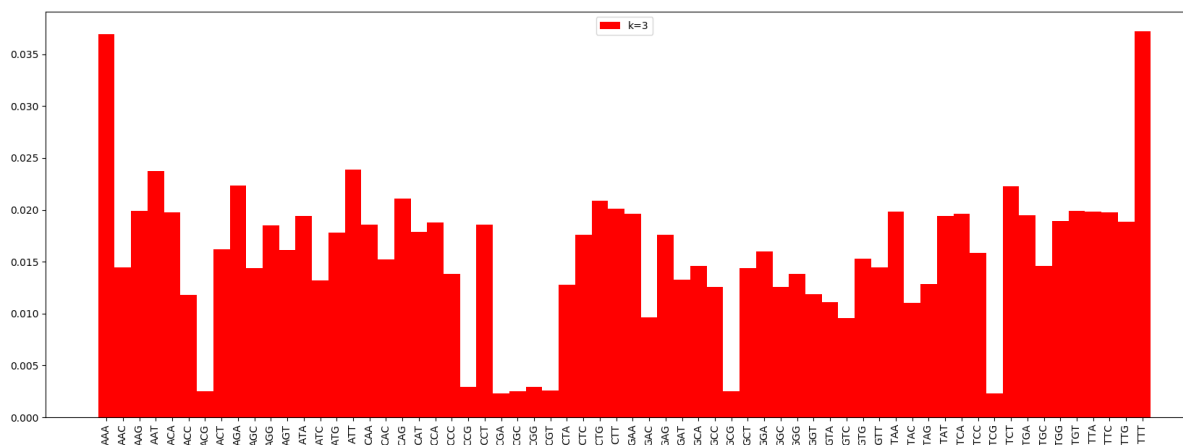
Standardmäßig werden alle Worte, welche nicht ausschließlich aus A, C, G, T oder U bestehen gefiltert.

Darstellung von k-mer Spektren

Oligo beinhaltet auch die Möglichkeit zum Erstellen von Grafiken.

Ein k-mer Spektrum kann beispielsweise als Histogramm dargestellt werden mit:

```
import Oligo  
  
spectrum = Oligo.Kmer.KmerSpectrum.read('Homo sapiens c1_k=3.kmer')  
spectrum.basic_stats()  
  
kmers = sorted(spectrum.get_kmers())  
drawer = Oligo.Plot.HistoDrawer(bins=spectrum.get_bins(), label='k=3')  
drawer.plot('Homo sapiens_k=3.kmer.png', xticklabels=kmers,  
xticks=range(len(kmers)), figsize=(20,7), xticklabels_rotation=90)
```



Notiz: Ab etwa k=4 werde diese Bilder aber schon relativ schwierig darzustellen.

Genom-Regionen (Gene, Introns, Intergenic Region)

Oligo hat die Möglichkeit verschiedene Klassen von Genen, Introns und intergenische Bereiche als Loci-Objekte aus Genbank-Dateien zu abzuleiten. Anschließend kann mit diesen Bereichen, wie mit nativen Loci-Objekten verfahren werden.

Gene

Um alle Gene aus einer Genbank-Datei zu lesen wird folgender Befehl verwendet:

```
Oligo.File.read_genes(<input_filename>)
```

Bsp.:

```
loci = Oligo.File.read_genes('data/Homo_Sapiens_c1.gb')
Oligo.Loci.basic_stats(loci)
```

```
Basic Stats for 5078 loci:
n = 5078
sum = 148250656bp
coverage sum = 138188555bp
mean length = 29194.693974005513bp
std length = 74459.28621625825bp
range = 11873-248937069
n density = 2.03997027284e-05
cov. density = 0.555140890599
```

Der read_genes Befehl kann alternativ auch mit einem chromosom-Objekt, Chromosom-Namen oder einem Organismus-Name, als Parameter ausgeführt werden, was das Einlesen aller Gene eines Organismus stark erleichtert:

Bsp.:

```
genome = Oligo.File.read_genome('Homo sapiens',
seqs_filename=Oligo.File.search('Homo sapiens.seqs'))
chromo = genome[0]
loci = Oligo.File.read_genes(chromosome=chromo)
Oligo.Loci.basic_stats(loci)
```

```
loci = Oligo.File.read_genes(organism='Homo sapiens',
seqs_filename=Oligo.File.search('Homo sapiens.seqs'))
Oligo.Loci.basic_stats(loci[0])
```

```
loci = Oligo.File.read_genes(chromo_name='Homo sapiens_c1',
seqs_filename=Oligo.File.search('Homo sapiens.seqs'))
Oligo.Loci.basic_stats(loci)
```

Die Befehle sind alle gleichermaßen funktionsfähig und liefern die gleichen Ergebnisse. Je nach Anwendung, können einige davon komfortabler sein als andere.

Alle in diesem Kapitel folgenden Befehle, bieten die gleiche Funktionalität.

Intergenische Regionen

Die intergenischen Regionen bilden die Bereiche der Genome, die nicht mit Genen bedeckt sind, also, neben den Introns in der klassischen Genetik als funktionslos gelten.

Der Befehl zum Auslesen funktioniert völlig analog zum `read_genes`:

`Oligo.File.read_intergenics(<input_filename>)`

```
loci = Oligo.File.read_intergenics(chromo_name='Homo sapiens c1')
Oligo.Loci.basic_stats(loci)
```

```
Basic Stats for 3322 loci:
n = 3322
sum = 110767867bp
coverage sum = 110767867bp
mean length = 33343.72877784467bp
std length = 380311.57759883517bp
range = 0-248956422
n density = 1.33437007702e-05
cov. density = 0.444928739376
```

Notiz: Wie erwartet ergibt die Summe aus der coverage density von Genen und intergenischen Bereichen 1.0 (beides gemeinsam ist das gesamte Genom).

Exons

Exons sind die Teile der Gene, die von der RNA-Polymerase in mRNA bzw. funktionale RNA übersetzt werden, sie beinhalten also die Informationen für Proteine und gelten klassisch als funktional.

```
loci = Oligo.File.read_exons(chromo_name='Homo sapiens c1')
Oligo.Loci.basic_stats(loci)
```

```
Basic Stats for 1034 loci:
n = 1034
sum = 783871bp
coverage sum = 537208bp
mean length = 758.0957446808511bp
std length = 680.0530610778192bp
range = 126645-248912793
n density = 4.15617994938e-06
cov. density = 0.00215931636194
```

Es sollte auffallen, dass Exons nur einen sehr geringen Anteil des Genoms ausmachen.

Notiz: Nur als „Exon“ annotierte Exons werden zur Berechnung verwendet. Dies schließt unter Umständen RNAs aus.

Introns

Introns sind die Teile der Gene, die nicht zu mRNA übersetzt werden und darum klassisch als nicht funktional (weil nicht Teil des Proteins) gelten.

```
loci = Oligo.File.read_introns(chromo_name='Homo sapiens cl')
Oligo.Loci.basic_stats(loci)
```

```
Basic Stats for 23739 loci:
n = 23739
sum = 113440886bp
coverage sum = 108265635bp
mean length = 4778.671637389949bp
std length = 13750.259854785107bp
range = 89005-248916601
n density = 9.54034053361e-05
cov. density = 0.435103006019
```

Notiz: Anders als bei Exons werden zur Berechnung auch mRNAs und CDS beachtet. Gene komplett ohne Exons, mRNA und CDS werden nicht berücksichtigt (schlechte Datenlage erscheint wahrscheinlicher als ein Gen ohne Exons).

Codierende Sequenzen (CDS)

Die CDS bildet den Teil der Exons, der am Ende tatsächlich in Aminosäuren übersetzt wird. Dieser Teil des Genoms gilt also höchst funktional.

```
Loci = Oligo.File.read_cds(chromo_name='Homo sapiens cl')
Oligo.Loci.basic_stats(loci)
```

```
Basic Stats for 149954 loci:
n = 149954
sum = 23332742bp
coverage sum = 3637417bp
mean length = 155.59933046134148bp
std length = 206.01539254310026bp
range = 69090-248918363
n density = 0.000602589664789
cov. density = 0.014616948469
```

Protein-Codierende (PC) Gene

Alle Gene, die tatsächlich für Proteine kodieren (viele Gene kodieren für funktionale RNA oder Transkripte ohne bekannte Funktion oder werden gar nicht abgelesen).

```
loci = Oligo.File.read_pc_genes(chromo_name='Homo sapiens cl')
Oligo.Loci.basic_stats(loci)
```

```
Basic Stats for 2257 loci:
n = 2257
sum = 119234764bp
coverage sum = 114526520bp
mean length = 52828.87195392113bp
std length = 101657.49029305812bp
range = 69090-248919146
n density = 9.06971867428e-06
cov. density = 0.460223002723
```

Nicht-Protein-Codierende (nPC) Gene

Gene, welche (wie oben beschrieben) nicht für Proteine kodieren.

```
loci = Oligo.File.read_npc_genes(chromo_name='Homo sapiens cl')
Oligo.Loci.basic_stats(loci)
```

```
Basic Stats for 2821 loci:
n = 2821
sum = 29015892bp
coverage sum = 27493721bp
mean length = 10285.676001417936bp
std length = 30122.211468197318bp
range = 11873-248937069
n density = 1.1332721819e-05
cov. density = 0.110449731252
```

Pseudogene

Funktionslose Gene (werden eventuell Transkribiert).

```
loci = Oligo.File.read_pseudo_genes(chromo_name='Homo sapiens cl')
Oligo.Loci.basic_stats(loci)
```

```
Basic Stats for 1372 loci:
n = 1372
sum = 3645427bp
coverage sum = 3637031bp
mean length = 2657.0167638483963bp
std length = 10905.277058538215bp
range = 11873-248937069
n density = 5.51169597151e-06
cov. density = 0.0146109395852
```

Non-Coding-RNA Gene

```
loci = Oligo.File.read_rna_genes(chromo_name='Homo sapiens cl')
Oligo.Loci.basic_stats(loci)
```

```
Basic Stats for 2023 loci:
n = 2023
sum = 81398491bp
coverage sum = 74435109bp
mean length = 40236.52545724172bp
std length = 102474.90168133528bp
range = 14361-248936684
n density = 8.12703326732e-06
cov. density = 0.299029464706
```

Non-Coding RNAs (ncRNA)

```
Basic Stats for 56290 loci:
n = 56290
sum = 26946580bp
coverage sum = 2630434bp
mean length = 478.70989518564573bp
std length = 987.4183543346586bp
range = 17368-248826439
n density = 0.000226237732305
cov. density = 0.0105720984747
```

Long Non-Coding RNAs (lncRNA)

```
loci = Oligo.File.read_lncRNAs(chromo_name='Homo sapiens cl')
Oligo.Loci.basic_stats(loci)
```

```
Basic Stats for 55628 loci:
n = 55628
sum = 26878761bp
coverage sum = 2610764bp
mean length = 483.18762134177035bp
std length = 991.5526978355502bp
range = 29925-248743944
n density = 0.000223662502917
cov. density = 0.0104970520379
```

Micro RNA (miRNA)

```
loci = Oligo.File.read_miRNAs(chromo_name='Homo sapiens cl')
Oligo.Loci.basic_stats(loci)
```

```
Basic Stats for 469 loci:
n = 469
sum = 10162bp
coverage sum = 5024bp
mean length = 21.66737739872068bp
std length = 1.3303981434305872bp
range = 17368-248826439
n density = 1.88497950704e-06
cov. density = 2.01921898579e-05
```

Sequencing Gaps

```
gaps = Oligo.File.read_sequence_gaps(chromo_name="Homo sapiens cl")
Oligo.Loci.basic_stats(gaps)
```

```
n = 165
sum = 18465408bp
coverage sum = 18465285bp
mean length = 111911.56363636363bp
std length = 1396869.901813433bp
range = 10000-228658364
n density = 7.216321040460189e-07
cov. density = 0.08075843919005692
```

Weitere

Analoge Funktionen existieren für:

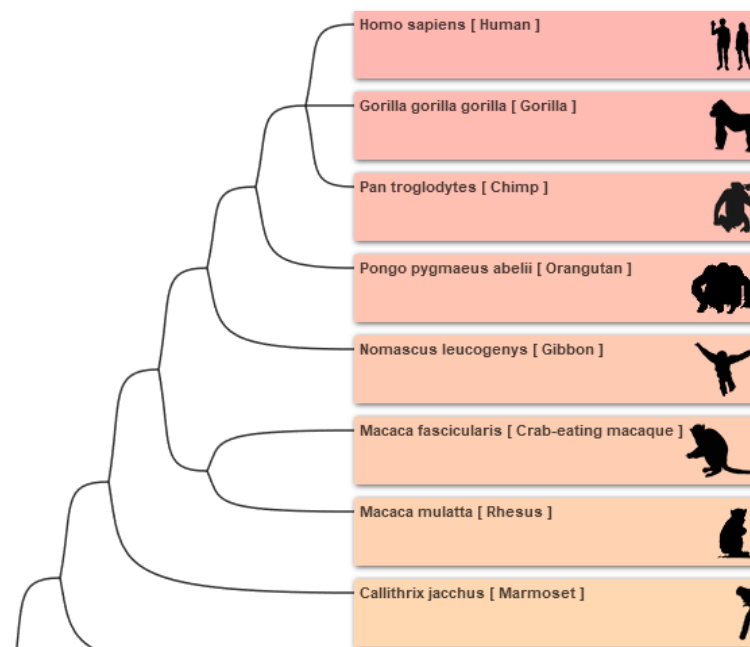
```
read_genes
read_exons
read_introns
read_CDS
```


read_mRNAs
read_intergenics
read_protein_coding_genes
read_pseudo_genes
read_non_protein_coding_genes
read_ncRNAs
read_lncRNAs
read_miRNAs
read_snoRNAs
read_snRNAs
read_tRNAs
read_rRNAs
read_precursor_RNAs
read_gaps
read_sequence_gaps

Repeat-Masker

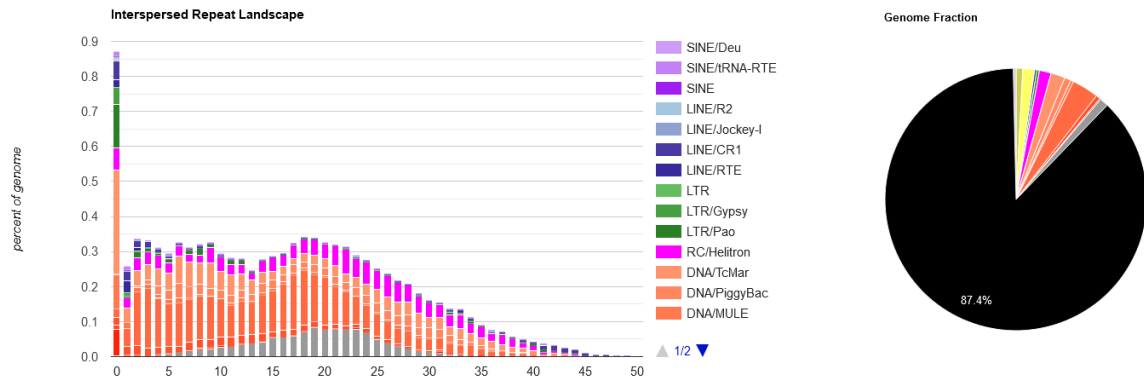
Repeatmasker bietet auf seiner Website (<http://www.repeatmasker.org/>) eine relativ umfängliche Sammlung von annotierten Repeats, darunter z.B. die Alu- und L1- Positionen mit angegebenen Subfamilien (z.B. AluSx, AluY).

Man navigiert hierzu über den Link „Genome Analysis and Downloads“ (<http://www.repeatmasker.org/genomicDatasets/RMGenomicDatasets.html>) zu folgender Ansicht:



Klickt hier auf den gewünschten Organismus:

ce10



Und wählt auf dieser Seite unter „Downloads“ die Datei mit der Endung „*.fa.out.gz“ zum Download. Die Datei muss noch entpackt werden (z.B. mit 7zip) und enthält eine Tabelle der Form:

SW	perc	perc	perc	query	position	in query	matching	repeat	position	in repeat	repeat	ID
score	div.	del.	ins.	sequence	begin	end	(left)	repeat	begin	end	(left)	
508	0.0	0.0	0.0	chrI	1	432	(15071991)	+	(GCCTAA)n	Simple_repeat	1 432 (0)	1
1226	10.0	0.0	0.0	chrI	566	595	(15071828)	+	(GCCTAA)n	Simple_repeat	1 41 (240)	2
344	22.2	0.0	0.0	chrI	596	676	(15071747)	C	RCS5	Satellite	(41) 1387 1307	3
1226	10.0	0.0	0.0	chrI	677	846	(15071577)	+	(GCCTAA)n	Simple_repeat	42 281 (0)	2
432	21.9	2.4	0.0	chrI	1622	1744	(15070679)	+	LONGPAL1	DNA/MULE-MuDR	136 261 (2330)	4
8509	0.6	0.0	0.1	chrI	2052	3026	(15069397)	+	PALTTTAAA3	DNA	1 974 (529)	5
4521	1.1	0.2	0.2	chrI	3124	3652	(15068771)	+	PALTTTAAA3	DNA	974 1502 (1)	6
1259	20.4	2.5	4.6	chrI	4423	4750	(15067673)	+	CELE2	DNA	1 321 (4)	7

Die Spalten sind hier durch unterschiedliche Anzahl an Leerzeichen getrennt, was ein automatisches Auslesen erschwert. Eine entsprechende Funktion wurde in Oligo integriert:

```
loci = Oligo.Locus.read_repeatmasker('ce10.fa.out')
```

Das Beispiel liest alle Einträge in der Datei als Loci. Um nach Chromosomen zu trennen ist auch die Verwendung eines Filters möglich:

```
loci = Oligo.Locus.read_repeatmasker('ce10.fa.out', filter_func=lambda
row,c: row[4] == c, filter_args=('chrII',))
```

Der Beispielcode filtert alle Einträge, die auf Chromosom II von C. Elegans liegen. Anschließend kann wie mit allen Loci-Objekten verfahren werden.

Locus-Operationen und Filter

Grundsätzlich können Loci auf vielfältige Weise gefiltert und kombiniert werden. In diesem Abschnitt werden eine Reihe grundlegender Operationen und in Oligo implementierter Operationen diskutiert.

Simple Filter

Da Loci in der Regel in gewöhnlichen Listen abgelegt sind, können diese auch wie gewöhnliche Python-Listen gefiltert werden:

```
filtered_loci = [locus for locus in loci if len(locus) > 10]
```

Das Codebeispiel filtert alle loci mit einer Länge von 10 oder weniger, es verbleiben also all Loci mit einer Mindestlänge von 11 bp.

Speichern von Loci

Nach der Anwendung mehrerer Filter oder anderen komplexer Operationen kann es sinnvoll sein das Ergebnis in einer Datei zu speichern. Hierfür existiert ein einfacher Befehl:

```
Oligo.Locus.save(loci, output_filename)
```

Hierbei ist loci eine Liste von Loci, die gespeichert werden sollen und output_filename der Name bzw. Pfad der Datei.

```
loci = Oligo.File.read_lncRNAs(chromo_name='Homo sapiens c1')
filtered_loci = [locus for locus in loci if len(locus) > 10]
Oligo.Locus.save(filtered_loci, output_filename='Homo sapiens c1
lncRNA larger 10bp.loci')
filtered_loci = Oligo.Locus.read('Homo sapiens c1 lncRNA larger
10bp.loci')
```

Der Beispielcode liest alle Long-Noncoding-RNAs des menschlichen Chromosom 1, filtert diese anhand ihrer Länge und speichert das Ergebnis in eine Datei. Anschließend wird die Datei wieder als loci gelesen.

Shuffle Loci

Für einige statistische Auswertungen ist ein zufälliges verteilen (Mischen) von gegebenen Loci (z.B. auf einem Chromosom) sehr nützlich. Der Vorgang ist weniger trivial als man annehmen könnte, da sich Loci für die passende Null-Hypothese oftmals nicht überlappen sollen und auch nicht innerhalb von Sequencing-Gaps platziert werden dürfen.

Der zugehörige Befehl lautet:

```
Oligo.Loci.shuffle(loci, boundaries=(0, None), allow_overlap=True, prevent_loci=None)
```

loci: Liste von Loci, die gemischt werden sollen

boundaries: Grenzen, in denen die Loci verteilt werden sollen. Als Untergrenze ist oftmals 0 die gewünschte Wahl, als Obergrenze die Länge des Chromosoms.

allow_overlap: Wahrheitswert, ob sich die neu platzierten Loci überschneiden dürfen (True: die Loci dürfen sich überschneiden)

prevent_loci: Eine zweite Liste von Loci, mit denen keine Überschneidung stattfinden darf. Je nach Anwendungsfall, sollten hier gaps oder bestimmte Loci eingetragen werden

Hinweis: Das Shuffling geschieht „in place“ d.h. die Loci in der übergebenen Liste loci werden beim Aufruf verändert.

Hinweis: Auch wenn die übergebene Liste von Loci sortiert war, ist die Liste nach dem Mischen nicht länger sortiert!

Sort Loci

Für einige Anwendungen ist es sinnvoll Loci zu sortieren. Hierfür existiert in Oligo keine explizite Funktion. Mit reinem Python ist dies allerdings leicht möglich:

```
sorted_loci = sorted(loci, key=lambda l: l.start)
```

Cluster

Es ist möglich nach Clustern innerhalb von Loci-Dateien zu suchen. Hierzu existiert eine DBSCAN-Implementierung innerhalb von Oligo. Der zugehörige Befehl lautet:

```
clustering = Oligo.Loci.cluster.db_scan(eps=eps, min_points=2, loci=loci)
```

Die Parameter sind die üblichen DBSCAN-Parameter (<https://de.wikipedia.org/wiki/DBSCAN>)

eps: Die Nachbarschaftslänge – die Distanz, die zwei Punkte haben, damit sie noch zu einem Cluster gehören.

minPts: Die minimale Anzahl an Punkten, die ein Cluster haben muss, um als Cluster zu gelten.

Parameterbestimmung

Eine sinnvolle Abschätzung der Parameter ist nicht ganz trivial. Eine Möglichkeit wäre die typischen Abstände zweier Punkte bei einer zufälligen Verteilung der Loci zu berechnen.

Beispiel:

```
import Oligo
import numpy as np

loci = Oligo.Locus.read('Homo sapiens cl_genes.loci')
```

```
Oligo.Loci.shuffle(loci, boundaries=(0, 248956422),
allow_overlap=False, prevent_loci=None)
loci = sorted(loci, key=lambda l: l.start)

dists = [abs(locus.start - loci[i+1].start) for i, locus in
enumerate(loci[:-1])]

print(np.mean(dists), np.std(dists))
```

Im Script oben wird eine Loci-Datei gelesen. Dann werden die Loci zufällig gemischt und nach Startposition sortiert. Anschließend werden die Distanzen zum jeweils folgenden Locus berechnet und die entsprechenden Mittelwerte mit Standardabweichung berechnet. Das Ergebnis lautet:

$$m = 48807 \quad \sigma = 38829$$

Im vorliegenden Fall könnte man nun $m - 1\sigma = 9978$ als Wert für eps verwenden. Die Distanzverteilung ist aber nicht unbedingt symmetrische, sodass diese Berechnung eventuell kein sinnvolles Ergebnis liefert. Eventuell besser funktioniert ein Perzentil:

```
print(np.percentile(dists, 33))
```

In diesem Beispiel würde der Wert ermittelt, bei dem 33% der Distanzen kleiner sind (was bei einer symmetrischen Verteilung etwa dem 1σ -Abstand entsprechen würde). Das Ergebnis im Beispiel wäre:

$$P_{33\%} = 26013$$

Das ist offensichtlich deutlich höher als der zunächst berechnete Wert. Selbstverständlich kann es legitim sein hier auf 10% oder noch weniger zu gehen abhängig davon, wie sicher man sein will, dass das Ergebnis eine hohe Signifikanz hat. Im Beispiel ist:

$$\begin{aligned} P_{10\%} &= 9588 \\ P_{5\%} &= 5814 \\ P_{1\%} &= 1785 \end{aligned}$$

Bei $\text{eps}=1785$ ist also die Wahrscheinlichkeit, dass sich, bei zufälliger Verteilung ein weiteres Element im Umkreis befindet, lediglich 1%. Setzt man nun also minPts auf das Minimum von 2 sollten nur etwa 1% aller Loci in Clustern sein.

Beispiel:

```
import Oligo
import numpy as np

loci = Oligo.Locus.read('Homo sapiens cl_genes.loci')

Oligo.Loci.shuffle(loci, boundaries=(0, 248956422),
allow_overlap=False, prevent_loci=None)
```

```

clustering = Oligo.Loci.cluster.db_scan(eps=1785, min_points=2,
loci=loci)

print('Found %s Cluster.' % (len(clustering)))
print('%s / %s Loci in Clusters' % (sum([len(cluster.members) for
cluster in clustering]), len(loci)))

```

In diesem Beispiel werden Loci gelesen, dann zufällig gemischt und schließlich wird ein Clustering mit $\text{eps} = 1785$ und $\text{min_pts}=2$ durchgeführt. Das Ergebnis sind 57 gefundene Cluster mit insgesamt 114 Elementen (das sind etwa 2.2% aller Gene auf c1).

Anmerkung: Das ist deutlich größer als die erwarteten 1%. Hier spielt die Tatsache eine Rolle, dass links und rechts von jedem Gen die Chance besteht, dass der Abstand unterschritten wird, sowie die Möglichkeit, dass mehrere Gene in diesem Abstand liegen. Die korrekte Formel wäre eher:

$$E[\# \text{ cluster members}] = 1 - (1 - P_{1\%})^2 \approx 1.99\%$$

Angewendet auf die echten Gene auf c1 (ohne shuffle) ergeben sich übrigens 234 Cluster mit 553 Genen (>10%).

Statistische Auswertung

Wollte man dieses Ergebnis nun auf Signifikanz prüfen, könnte man das Shuffling 10 oder 100-mal wiederholen und so eine Verteilung von Clusteranzahl und Clustergröße erhalten, mit der man den empirischen Wert vergleichen kann. Beispiel:

```

import Oligo
import numpy as np

loci = Oligo.Locus.read('Homo sapiens c1_genes.loci')

clustering = Oligo.Loci.cluster.db_scan(eps=1785, min_points=2, loci=loci)
n_cluster = len(clustering)
n_genes_in_clusters = sum([len(cluster.members) for cluster in clustering])
mean_cluster_members = np.mean([len(cluster.members) for cluster in clustering])
std_cluster_members = np.std([len(cluster.members) for cluster in clustering])
mean_cluster_size = np.mean([abs(min([locus.start for locus in cluster.members])-
max([locus.get_end() for locus in cluster.members])) for cluster in clustering])
std_cluster_size = np.std([abs(min([locus.start for locus in cluster.members])-
max([locus.get_end() for locus in cluster.members])) for cluster in clustering])

print('# Clusters: %s' % n_cluster)
print('# Loci in Clusters: %s' % n_genes_in_clusters)
print('Loci per Cluster: %s +/- %s' % (mean_cluster_members, std_cluster_members))
print('Size of Clusters: %s +/- %s' % (mean_cluster_size, std_cluster_size))

n_clusters_list = []
n_genes_in_clusters_list = []
mean_cluster_members_list = []
mean_cluster_size_list = []

for i in range(10):
    Oligo.Loci.shuffle(loci, boundaries=(0, 248956422), allow_overlap=False,
prevent_loci=None)
    loci = sorted(loci, key=lambda l: l.start)

    clustering = Oligo.Loci.cluster.db_scan(eps=1785, min_points=2, loci=loci)
    n_cluster = len(clustering)

```

```

    n_genes_in_clusters = sum([len(cluster.members) for cluster in clustering])
    mean_cluster_members = np.mean([len(cluster.members) for cluster in
clustering])
    mean_cluster_size = np.mean([abs(min([locus.start for locus in cluster.members])-
max([locus.get_end() for locus in cluster.members])) for cluster in clustering])

    n_clusters_list.append(n_cluster)
    n_genes_in_clusters_list.append(n_genes_in_clusters)
    mean_cluster_members_list.append(mean_cluster_members)
    mean_cluster_size_list.append(mean_cluster_size)

print('Expected # Clusters: %s +/- %s' % (np.mean(n_clusters_list),
np.std(n_clusters_list)))
print('Expected # Loci in Clusters: %s +/- %s' % (np.mean(n_genes_in_clusters_list),
np.std(n_genes_in_clusters_list)))
print('Expected Loci per Clusters: %s +/- %s' % (np.mean(mean_cluster_members_list),
np.std(mean_cluster_members_list)))
print('Expected Size of Clusters: %s +/- %s' % (np.mean(mean_cluster_size_list),
np.std(mean_cluster_size_list)))

```

Das Skript wiederholt das Clustering 10-mal und gibt allerlei statistische Informationen aus:

Expected # Clusters: 52.3 +/- 7.6
Expected # Loci in Clusters: 105.4 +/- 15.1
Expected Loci per Clusters: 2.016 +/- 0.016
Expected Size of Clusters: 32705 bp +/- 9708 bp

Empirische Ergebnisse:

Clusters: 234
Loci in Clusters: 553
Loci per Cluster: 2.4 +/- 1.1
Size of Clusters: 36118 bp +/- 87435 bp

Es gibt also signifikant mehr Cluster als erwartet und entsprechend mehr Gene in Clustern. Die Loci pro Cluster sind nur insignifikant größer, während die Größe der Cluster stark schwankt. Es wäre nun in einem weiteren Schritt möglich die Cluster rauszufiltern, die größer sind als erwartet oder die mehr Gene enthalten.

Speichern & Lesen von Clustern

Man könnte direkt die Cluster filtern und speichern, damit man sie später weiter auswerten kann:

```

import Oligo
import numpy as np

loci = Oligo.Locus.read('Homo sapiens c1_genes.loci')

clustering = Oligo.Loci.cluster.db_scan(eps=1785, min_points=2,
loci=loci)

filtered_clusters = [cluster for cluster in clustering if
len(cluster.members) > 2.032]

```

```
Oligo.Loci.cluster.Cluster.save(filtered_clusters,
'filtered_clusters.cluster')

filtered_clusters =
Oligo.Loci.cluster.Cluster.read('filtered_clusters.cluster')
```

Cluster in Loci Convertieren

Es ist auch möglich Cluster direkt in Loci umzuwandeln, sodass man diese beispielsweise auf Maps darstellen kann:

```
import Oligo

loci = Oligo.Locus.read('Homo sapiens c1_genes.loci')

clustering = Oligo.Loci.cluster.db_scan(eps=1785, min_points=2,
loci=loci)

cluster_loci = [cluster.to_locus() for cluster in clustering]
Oligo.Locus.save(cluster_loci, 'cluster_loci.loci')
```

Vorberechnete Distanzen / Distanzmatrix

Anstelle von Loci, akzeptiert der Befehl *Oligo.Loci.cluster.db_scan* auch eine quadratische Distanzmatrix als Parameter. In dieser Matrix entspricht dann jede Zeile und jede Spalte einer Element und der Eintrag an der Stelle *i,j* gibt die Distanz zwischen Element *i* und *j* an. Die Berechnung über eine solche Distanzmatrix kann (insbesondere bei mehrdimensionalen Datensätzen) schneller sein, braucht aber gegebenenfalls viel Arbeitsspeicher.

IM folgenden Beispiel wird eine Distanzmatrix für die Gene auf Chromosom 1 berechnet und anschließend ein Clustering durchgeführt.

```
import Oligo
import numpy as np

loci = Oligo.Locus.read('Homo sapiens c1_genes.loci')

distances = []
for locus1 in loci:
    distances.append([])
    for locus2 in loci:
        distances[-1].append(abs(locus1.start-locus2.start))
loci_names = [str(locus) for locus in loci]
matrix = Oligo.Matrix.Matrix(data=distances, row_names=loci_names
,col_names=loci_names)
matrix.save('Homo sapiens c1_gene_distances.matrix')

matrix = Oligo.Matrix.Matrix.read('Homo sapiens c1_gene_distances.matrix')

clustering = Oligo.Loci.cluster.db_scan(eps=1785, min_points=2,
distances=matrix.matrix)

n_cluster = len(clustering)
n_genes_in_clusters = sum([len(cluster.members) for cluster in clustering])
mean_cluster_members = np.mean([len(cluster.members) for cluster in clustering])
std_cluster_members = np.std([len(cluster.members) for cluster in clustering])
mean_cluster_size = np.mean([abs(min([loci[i].start for i in cluster.members])-
max([loci[i].get_end() for i in cluster.members])) for cluster in clustering])
std_cluster_size = np.std([abs(min([loci[i].start for i in cluster.members])-
max([loci[i].get_end() for i in cluster.members])) for cluster in clustering])
```



```
print('# Clusters: %s' % n_cluster)
print('# Loci in Clusters: %s' % n_genes_in_clusters)
print('Loci per Cluster: %s +/- %s' % (mean_cluster_members, std_cluster_members))
print('Size of Clusters: %s +/- %s' % (mean_cluster_size, std_cluster_size))
```

Ausgabe:

```
# Clusters: 234
# Loci in Clusters: 553
Loci per Cluster: 2.4 +/- 1.1
Size of Clusters: 36118 +/- 87435
```

Das Ergebnis ist also identisch mit der Berechnung direkt über die Loci.

Clustering in Correlationsdaten

Anstelle einer Distanzmatrix, könnte als Eingabe für das Clustering auch eine Korrelationsmatrix verwendet werden, wie sie beispielsweise durch die Befehle *Oligo.Maps.correlation_matrix* oder *Oligo.Kmer.KmerSpectrum.correlate_spectra* generiert werden.

Hierbei gibt es zunächst zwei technische Hürden. Einmal bedeutet eine höhere Korrelation größere Ähnlichkeit – sie ist also eine Art inverse Distanz. Ein möglicher naiver Ansatz wäre also ein einfacher Vorzeichenwechsel. Unglücklicherweise akzeptiert DBSCAN aber keine negativen Distanzen, darum muss eine komplexere Umwandlung geschehen. Um negative Werte zu vermeiden wäre folgende Formel sinnvoll:

$$dist_j = \max(\{r_i\}) - r_j$$

Also der maximal vorkommende Korrelationswert minus den Wert. Entsprechend müsste man den Wert für eps anpassen:

$$eps' = \max(\{r_i\}) - eps$$

Angenommen man will also Korrelationen ab 0.75 als ausreichend ähnlich definieren und die maximal vorkommende Korrelation wäre 0.99, dann müsste als $eps = 0.99 - 0.75 = 0.24$ verwendet werden. Folgendes Skript verwendet diese Variante:

```
import Oligo
import numpy as np

matrix = Oligo.Matrix.Matrix.read('Homo sapiens_k=3_correlation.matrix')

max_value = max(matrix.get_all_values())
matrix.modify(func=lambda matrix,i,j,x: max_value-x)

clustering = Oligo.Loci.cluster.db_scan(eps=max_value-0.98, min_points=2,
distances=matrix.matrix)

n_cluster = len(clustering)
n_chromos_in_clusters = sum([len(cluster.members) for cluster in
clustering])
mean_cluster_members = np.mean([[len(cluster.members) for cluster in
clustering]])
```

```
std_cluster_members = np.std([[len(cluster.members) for cluster in
clustering]])

print('# Clusters: %s' % n_cluster)
print('# Chromosomes in Clusters: %s' % n_chromos_in_clusters)
print('Chromosomes per Cluster: %s +/- %s' % (mean_cluster_members,
std_cluster_members))
```

Eine korrekte Abschätzung von ϵ ist analog zum Problem der Definition einer signifikanten Korrelation. Eine zuverlässige Variante wäre die genes von Referenzdaten (z.B. shuffling von Maps vor der Korrelation) und daraus die Berechnung des maximal möglichen Wertes bei zufälligen Maps. Bei Korrelationen mit Fehlerabschätzung (z.B: Boot-Correlate) wäre es auch denkbar, anstelle der Korrelationswerte, Signifikanzen zu verwenden. Dann wäre $\epsilon = 1\sigma$ oder $\epsilon = 2\sigma$, je nach gewünschter Signifikanz der Ergebnisse, möglich.

Custom Labels/Attributes

Es ist gelegentlich sinnvoll Loci mit zusätzlichen Attributen bzw. Labels zu versehen. Hierfür existieren entsprechende Funktionen, die in diesem Abschnitt beschrieben werden.

Hinzufügen & Auslesen von Labels

Labels können mit den Funktionen *set_value* und *get_value* an einzelne Loci annotiert werden. Beide Funktionen haben zwei Parameter: Eine Bezeichnung/Name des Labels und der Wert (value) des Labels.

Beispiel:

```
import Oligo

loci = Oligo.File.read_lncRNAs(chromo_name='Homo sapiens c11') # Read some Loci

for locus in loci: # Annotate a very meaningful label to all loci
    if len(locus) > 1000:
        locus.set_value('very long', 1)
    else:
        locus.set_value('very long', 0)

# Print Labels for all Loci:
for locus in loci:
    print(locus.get_value('very long'))
```

Speichern & Lesen von Labels in Dateien

Zum Speichern der Labels wird der gewöhnliche Speicher-Befehl für Loci verwendet. Es muss hier lediglich eingegeben werden, dass das entsprechende Label gespeichert werden soll. Dies geschieht über den Parameter *saved_features*. Hierbei handelt es sich um eine Liste, die bestimmt, welche Werte in die Loci-Datei gespeichert werden. Standardmäßig ist der Wert [start, length, strand]. Diese kann entsprechend um die Namen der Labels ergänzt werden.

Beispiel:

```
Oligo.Locus.save(loci, 'test.loci', saved_features=['start', 'length', 'strand', 'very long'])
```

Ausgabedatei:

```
#date: 2024-03-14 09:25:53.160379
start  length strand  very long
138955 162    ?      0
138046 65     ?      0
```

134889	58	?	0
134563	233	?	0
133587	191	?	0
131466	58	?	0
127133	4240	?	1
138955	162	?	0
138046	65	?	0
134889	58	?	0
134563	233	?	0
133587	191	?	0
131466	58	?	0

Anmerkung: Eine Reihe von Namen sind bereits für bestimmte Features der Loci reserviert und sollten daurm nicht für Labels verwendet werden. Folgende Namen sind reserviert: *start, end, length, strand, name, Loc, labels, id, sequence, chromosome, genes, gene_name*

Das Auslesen der Loci funktioniert analog mit der gewöhnlichen Lese-Funktion für Loci. Hierbei wird der Parameter *custom_labels* zum Auslesen zusätzlicher Labels verwendet.

Beispiel:

```
loci = Oligo.Locus.read('test.loci', custom_labels=['very long']) # Read loci with custom label

for locus in loci: # Print Label values
    print(locus.get_value('very long'))
```

Anmerkung: Die Custom-Labels warden stets als String (Text) eingelesen. Es ist darum eventuell notwendig die Werte wieder in Zahlen (Integer oder Float) zu konvertieren. Beispielsweise mittels: `locus.set_value(<name>, int(locus.get_value(<name>)))`

Beispiel: Gaps zwischen Loci

Eine mögliche Anwendung für die beschriebenen Labels ist das Speichern der Gaps zwischen je zwei Loci. Folgender Code bestimmt diese Abstände und speichert das Ergebnis als Labels:

```
import Oligo

loci = Oligo.File.read_lncRNAs(chromo_name='Homo sapiens c11') # Read Loci
loci = sorted(loci, key=lambda locus: locus.start) # sort loci by starting position

for i,locus in enumerate(loci):
    if i < len(loci)-1:
        next_locus = loci[i+1]
        dist = next_locus.start - (locus.start + len(locus))
        locus.set_value('gap', dist)
        locus.set_value('partner strand', next_locus.strand)

Oligo.Locus.save(loci[:-1], 'Homo Sapiens c11_lncRNA_gaps.loci',
saved_features=['start','length','strand','gap', 'partner strand'])
```

Anmerkung: Die -1 für die Rand-Loci kommen daher, dass der letzte Locus im Datensatz keinen nächsten Locus besitzt und darum in der Analyse kein Label erhalten kann.

k-mer Analyse

Im Folgenden wird die Durchführung einer *k*-mer-Analyse analog zu unserem Paper (Sievers et al. 2018) beschrieben. Es wird davon ausgegangen, dass *k*-mer-Dateien (*.kmer) für verschiedene *k* bereits gesucht wurden (siehe oben für Details).

Zusammenfassen von Spektren

Eine *k*-mer-Suche ergibt stets das Ergebnis für eine Sequenz (bzw. ein Chromosom). Oft sind wir aber an den Spektren für ganze Organismen interessiert. Um also mehrere Spektren zu einem Gesamtspektrum zu verbinden, existiert ein einfacher Befehl:

```
Oligo.Kmer.combine(<spectra>)
```

Diese kombinierten Spektren können dann auch leicht als eigene Spektren gespeichert werden:

```
KmerSpectrum.save(<output filename>)
```

Bsp.:

```
# Read All Spectra for Homo sapiens k=3
genome = Oligo.File.read_genome('Homo sapiens',
seqs_filename=Oligo.File.search('Homo sapiens.seqs'))
spectra = [Oligo.Kmer.KmerSpectrum.read('%s_k=3.kmer' % chromo) for chromo in
genome]

# Combine Spectra
combined_spectrum = Oligo.Kmer.combine(spectra)

# Save Spectrum
combined_spectrum.save('Homo sapiens_k=3.kmer')
```

Korrelation von *k*-mer-Spektren

Ein Korrelationskoeffizient ist ein Maß für Ähnlichkeit zwischen zwei Vektoren. Ein *k*-mer Spektrum kann als Vektor mit 4^k -Komponenten aufgefasst werden und entsprechend bietet eine Korrelation ein Maß für die Ähnlichkeit von *k*-mer Spektren.

Wir verwenden die Pearson-Korrelation, bei der ein Ergebnis von 0 keine Korrelation (keine Ähnlichkeit), 1 perfekte Übereinstimmung und -1 Antikorrelation (ein hoher Wert in einem Vektor impliziert einen niedrigen Wert beim anderen) bedeutet.

Zwei Spektren können korreliert werden über den Befehl:

```
KmerSpectrum.correlate()
```

Bsp.:

```
# Read All Spectra for Homo sapiens k=3
genome = Oligo.File.read_genome('Homo sapiens',
seqs_filename=Oligo.File.search('Homo sapiens.seqs'))
```

```
spectra = [Oligo.Kmer.KmerSpectrum.read('%s_k=3.kmer' % chromo) for chromo in
genome]

# Correlate Spectra of c1 and c2
print(spectra[0].correlate(spectra[1]))
```

Das Ergebnis war bei mir etwa 0.994, was einen sehr hohen Korrelationswert darstellt. Die beiden ersten menschlichen Chromosomen sind also sehr ähnlich in ihrer 3-mer-Zusammensetzung.

Korrelation vieler Spektren

Es gibt eine Funktion um viele *k*-mer-Spektren paarweise zu Korrelieren und die Ergebnisse direkt als Matrix zu erhalten (dies erleichtert auch die Darstellung als Heatmap):

```
Oligo.Kmer.KmerSpectrum.correlate_spectra(<spectra1>, <spectra2>)
```

spectra1 und spectra2 sind unabhängige Listen von Spektren, die miteinander paarweise korreliert werden sollen. Wird spectra2 nicht angegeben, so werden die Spektren in der ersten Liste miteinander korreliert.

Bsp.:

```
# Read All Spectra for Homo sapiens k=3
genome = Oligo.File.read_genome('Homo sapiens')
spectra = [Oligo.Kmer.KmerSpectrum.read('%s_k=3.kmer' % chromo, name=str(chromo))
for chromo in genome]

# Correlate
matrix = Oligo.Kmer.KmerSpectrum.correlate_spectra(spectra)

# Display Matrix
matrix.show()
```

Darstellung als Heatmap

Oligo hat eine einfache Möglichkeit um Matrizen als Heatmaps darzustellen:

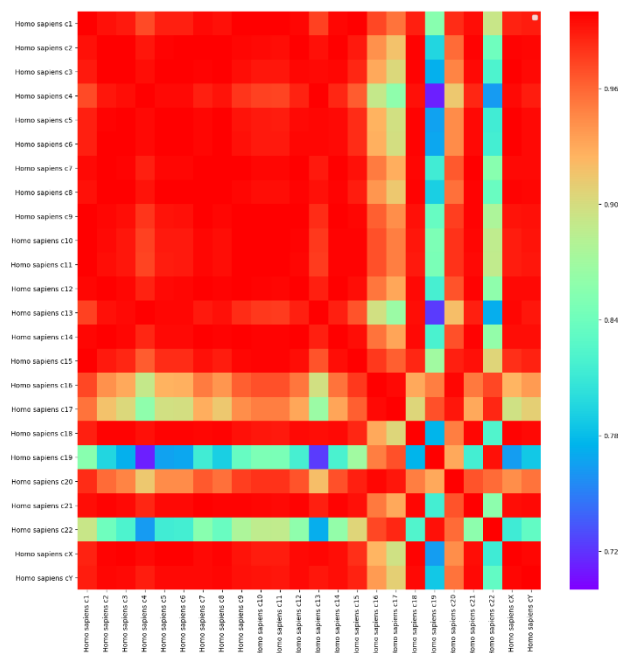
```
# Read All Spectra for Homo sapiens k=3
genome = Oligo.File.read_genome('Homo sapiens',
seqs_filename=Oligo.File.search('Homo sapiens.seqs'))
spectra = [Oligo.Kmer.KmerSpectrum.read('%s_k=3.kmer' % chromo, name=str(chromo))
for chromo in genome]

# Correlate
matrix = Oligo.Kmer.KmerSpectrum.correlate_spectra(spectra)

# Save Matrix
matrix.save('Homo sapiens_k=3_correlation.matrix')

# Display Matrix
drawer = Oligo.Plot.HeatmapDrawer(data=matrix.matrix, xticklabels=matrix.col_names,
yticklabels=matrix.row_names, vmin=0.7, vmax=1., cmap='rainbow')
drawer.plot('Homo sapiens_k=3.kmer.hm.png', figsize=(15,15))
```

Ergebnis:



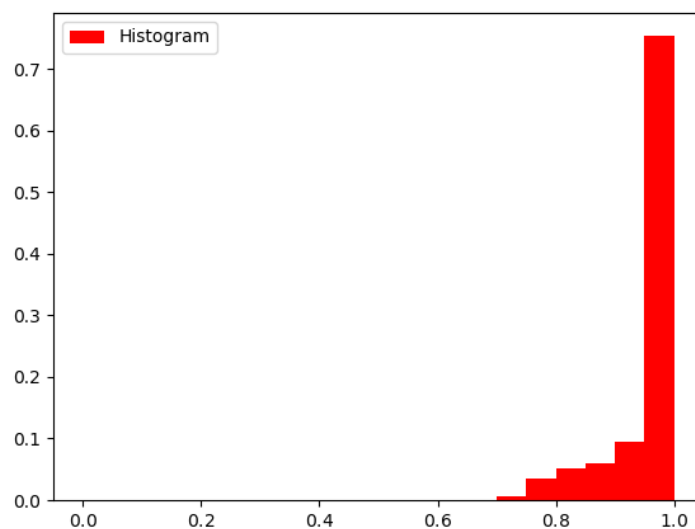
Mean Correlation

Heatmaps sind eine schöne Sache für die Übersicht aber es gilt als nächsten Schritt die Konservierung innerhalb einer solchen Heatmap zu quantifizieren und zu vergleichen. Dazu können die Verteilungen innerhalb der Heatmaps berechnet werden:

```
# Read Data
matrix = Oligo.Matrix.Matrix.read('Homo sapiens_k=3_correlation.matrix')

# Display Distribution
values = matrix.get_all_values()
bins = Oligo.Loci.bins.create_bins(values, n_bins=20)

drawer = Oligo.Plot.HistoDrawer(bins=bins)
drawer.plot('Homo sapiens_k=3_correlation_distribution.png')
```



Mean Correlation VS. k

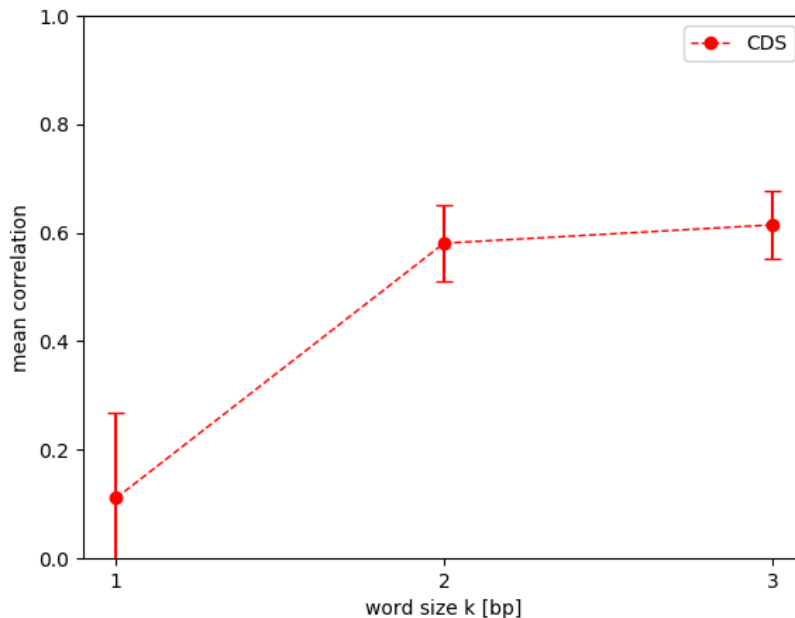
Es kann sinnvoll sein verschiedene Datensätze zu vergleichen. Hierzu können mit Numpy einfach statistische Größen aus den Matrix-Daten ermittelt werden:

```
matrix = Oligo.Matrix.Matrix.read('Homo sapiens_k=3_correlation.matrix')  
  
m = np.mean(matrix.matrix)  
e = np.std(matrix.matrix)/np.sqrt(len(matrix))
```

Die Ergebnisse können für verschiedene Datensätze (z.B. verschiedene Wortlängen k) mit Oligo dargestellt werden:

```
ks = [1,2,3]  
mean_values = []  
err_values = []  
  
for k in ks:  
    matrix = Oligo.Matrix.Matrix.read('Homo sapiens_k=%s_CDS_correlation.matrix' %  
k)  
    m = np.mean(matrix.matrix)  
    e = np.std(matrix.matrix)/np.sqrt(len(matrix))  
    mean_values.append(m)  
    err_values.append(e)  
  
drawer = Oligo.Plot.CurveDrawer(x=ks, y=mean_values, y_err=err_values, label='CDS',  
linestyle='o--', color='red', err_color='red')  
drawer.plot('heatmap_summaries.png', xticks=ks, xlabel='word size k [bp]',  
ylabel='mean correlation', ylim=(0,1))
```

Ergebnis:



Es ist ebenso möglich mehrere solche Kurven in ein Diagramm zu zeichnen:

Code:

```
# CDS
```

```

ks = [1,2,3]
mean_values = []
err_values = []
for k in ks:
    matrix = Oligo.Matrix.Matrix.read('Homo sapiens_k=%s_CDS_correlation.matrix' % k)
    m = np.mean(matrix.matrix)
    e = np.std(matrix.matrix)/np.sqrt(len(matrix))
    mean_values.append(m)
    err_values.append(e)

drawer1 = Oligo.Plot.CurveDrawer(x=ks, y=mean_values, y_err=err_values, label='CDS',
linestyle='o--', color='red', err_color='red')

# Introns
ks = [1,2,3]
mean_values = []
err_values = []
for k in ks:
    matrix = Oligo.Matrix.Matrix.read('Homo sapiens_k=%s_introns_correlation.matrix' % k)
    m = np.mean(matrix.matrix)
    e = np.std(matrix.matrix)/np.sqrt(len(matrix))
    mean_values.append(m)
    err_values.append(e)

drawer2 = Oligo.Plot.CurveDrawer(x=ks, y=mean_values, y_err=err_values, label='Introns',
linestyle='o--', color='blue', err_color='blue')

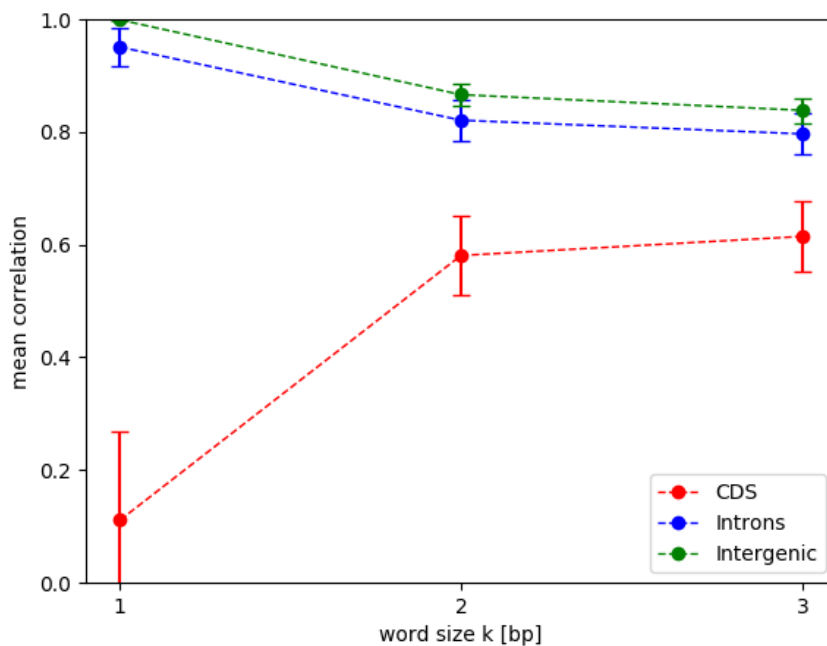
# Intergenic
ks = [1,2,3]
mean_values = []
err_values = []
for k in ks:
    matrix = Oligo.Matrix.Matrix.read('Homo sapiens_k=%s_intrgenic_correlation.matrix' % k)
    m = np.mean(matrix.matrix)
    e = np.std(matrix.matrix)/np.sqrt(len(matrix))
    mean_values.append(m)
    err_values.append(e)

drawer3 = Oligo.Plot.CurveDrawer(x=ks, y=mean_values, y_err=err_values, label='Intergenic',
linestyle='o--', color='green', err_color='green')

drawer = Oligo.Plot.MultiDrawer([drawer1, drawer2, drawer3])
drawer.plot('heatmap_summaries.png', xticks=ks, xlabel='word size k [bp]', ylabel='mean
correlation', ylim=(0,1))

```

Ergebnis:



Notiz: Der Code oben ist selbstverständlich nicht sehr elegant. Es wäre besser eine Funktion zu definieren.

Correlation Contribution

Zu einer k -mer-Analyse zählt auch die Bestimmung der Korrelationsbeiträge für verschiedene Wortgruppen. Dabei handelt es sich um den Wert, den eine zuvor definierte Gruppe an Worten (eine Gruppe kann auch nur ein Wort sein), zum mittleren Korrelationswert beiträgt (siehe Sievers et al. 2018 für Details).

Hierzu wird zunächst eine Datei erstellt, welche die Beiträge der einzelnen Worte speichert (diese können dann einfach durch Addition zu Beiträgen von Gruppen verrechnet werden).

Der Befehl lautet:

```
Oligo.Kmer.correlation_contributions()
```

Parameter:

spectra : Liste von k -mer Spektren

k : Wortlänge k (wird automatisch ermittelt, wenn nicht angegeben)

symmetric: Angabe ob die Korrelationsmatrix Symmetrisch ist (dieses Angabe spart gegebenenfalls 50% der Rechenzeit). Standardwert ist True.

sort : Gibt an ob die Worte sortiert werden sollen (sollte momentan immer True sein)

output_filename : Name der Ausgabedatei

allow_delete : Angabe ob Spektren durch die Funktion aus dem RAM gelöscht werden dürfen. Sollte auf True stehen, wenn es Probleme mit dem RAM gibt und die Spektren nicht nach der Ausführung noch im Skript benötigt werden.

Beispiel:

```
# Read All Spectra for Homo sapiens k=3
genome = Oligo.File.read_genome('Homo sapiens',
seqs_filename=Oligo.File.search('Homo sapiens.seqs'))
spectra = [Oligo.Kmer.KmerSpectrum.read('%s_k=3.kmer' % chromo,
name=str(chromo)) for chromo in genome]

# Calculate Correlation Contributions
Oligo.Kmer.correlation_contributions(spectra, output_filename='Homo
sapiens_k=3_correlation_contributions.matrix')
```

Das Ergebnis ist folgende Datei:

```
#Correlation Contribution Matrix
#Generated by Oligo.Kmer.Spectra.save_correlation_contributions()
$n = 276
$total sum = 0.9580732277179503
Row Name      Mean Correlation Contribution      Std      Correlation      Contribution
      Normalized Correlation Contribution      Std      Normalized      Correlation
Contribution
TTT      0.15455561204306265 0.014631138444325415      0.16131920564276814
      0.015271419784034205
```

AAA	0.14661132534367757	0.015400505841975005	0.15302726462036037
	0.01607445589379187		
CGA	0.05519005307944738	0.003770306965319691	0.057605255509441004
	0.00393530145320963		
TCG	0.054451541001694165	0.0035406029594702132	0.05683442499629506
	0.0036955452433459924		
CGC	0.05283929083401051	0.0025173660709803857	0.05515162025753423
	0.002627529919583015		
GCG	0.05268954108920713	0.0025708882220735027	0.0549953172313448
	0.0026833942831250403		
ACG	0.052030027160166255	0.0035608269747678975	0.05430694194857881
	0.003716654292959931		
CGT	0.05184835464836894	0.0035101689918252883	0.05411731916553743
	0.0036637794380145815		
CCG	0.0497880480498299	0.002250297216518444	0.051966850350699006
	0.0023487737173059956		
CGG	0.04911863702925867	0.00223478068564526	0.051268144864307634
	0.0023325781589454485		
ATT	0.02219945991335663	0.008967362056164687	0.02317094275375367
	0.009359787745582023		

Es handelt sich um eine Liste der k -mer Worte mit den Korrelationsbeiträgen, sowie deren Standardabweichungen (Absolutwerte und dann normiert auf 1 – also 4 Zahlenwerte pro Zeile).

Most Contributing Words

Da die Daten sortiert sind, sind diese einfach aus der Tabelle auslesbar:

```
# Einlesen der Daten
matrix = Oligo.Matrix.Matrix.read('Homo
sapiens_k=3_correlation_contributions.matrix')

# Ausgabe der Top 10 Worte mit normalisiertem Content
top10 = matrix.col('Normalized Correlation Contribution')[0:10]
words = matrix.row_names[0:10]

for word, cont in zip(words, top10):
    print(word, cont)
```

Ergebnis:

```
TTT 0.161319205643
AAA 0.15302726462
CGA 0.0576052555094
TCG 0.0568344249963
CGC 0.0551516202575
GCG 0.0549953172313
ACG 0.0543069419486
CGT 0.0541173191655
CCG 0.0519668503507
CGG 0.0512681448643
```

Different G/C-Contents

Um nun den Beitrag aus einer definierten Gruppe zu bestimmen, müssen Ergebnisse dieser Art nur gefiltert werden. Nach einem GC Gehalt der Worte, beispielsweise über:

```
# Einlesen der Daten
matrix = Oligo.Matrix.Matrix.read('Animalia_k=7_correlation_contributions.matrix')
```

```
# Ausgabe der Top 10 Worte mit normalisiertem Content
top10 = matrix.col('Normalized Correlation Contribution')
words = matrix.row_names

x = 0.0
for word, cont in zip(words, top10):
    if Oligo.Kmer.gc_content(word) > 0.5:
        x += cont
print(x)
```

Das Ergebnis hier war 0.22, folglich werden 88% durch Worte mit weniger als 50% GC-Gehalt erzeugt und folglich sind diese Worte dann für den größten Teile der Korrelation und damit der Ähnlichkeit der Sequenzen verantwortlich.

Ergebnisse für verschiedene Gruppen können dann beispielsweise als Histogramm dargestellt werden:

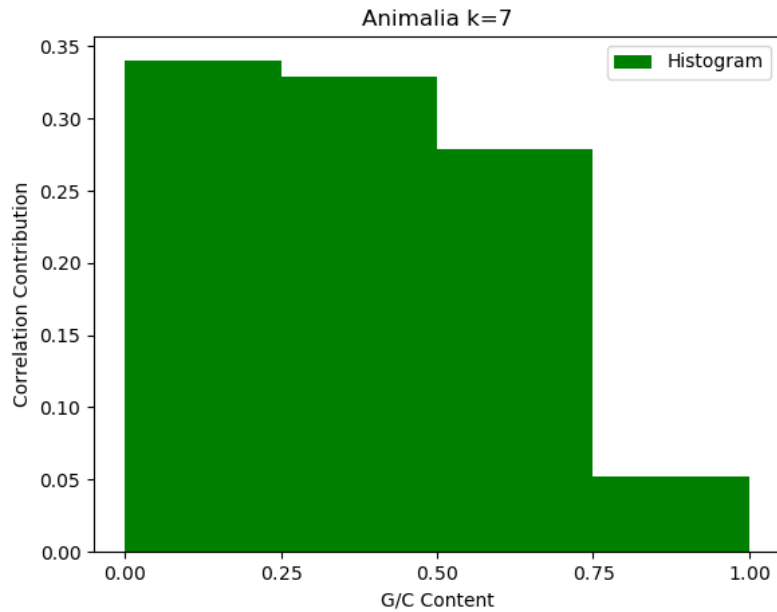
Code:

```
# Einlesen der Daten
matrix = Oligo.Matrix.Matrix.read('Animalia_k=7_correlation_contributions.matrix')
conts = matrix.col('Normalized Correlation Contribution')
words = matrix.row_names

# Analyse der Daten
gcs = [.0, .25, .50, .75, 1.]
data = []
for i in range(1, len(gcs)):
    x = 0.0
    for word, cont in zip(words, conts):
        gc_content = Oligo.Kmer.gc_content(word)
        #print word, gc_content, gcs[i-1], gcs[i], gc_content > gcs[i-1] and
gc_content <= gcs[i]
        if gc_content > gcs[i-1] and gc_content <= gcs[i]:
            x += cont
    data.append(x)

print(data)
# Darstellung der Daten
bins = [Oligo.Bin(data[i-1], gcs[i-1], gcs[i]) for i in range(1, len(gcs))]
drawer = Oligo.Plot.HistoDrawer(bins=bins, color='green')
drawer.plot('GC_data.png', title='Animalia k=7', xticks=gcs, xlabel='G/C Content',
ylabel='Correlation Contribution')
```

Ergebnis:



Different Complexities / Repeats

Eine weitere Gruppe, die bisher relevant war, sind Repeats. Die entsprechenden Filter sind andere aber die Funktionsweise identisch.

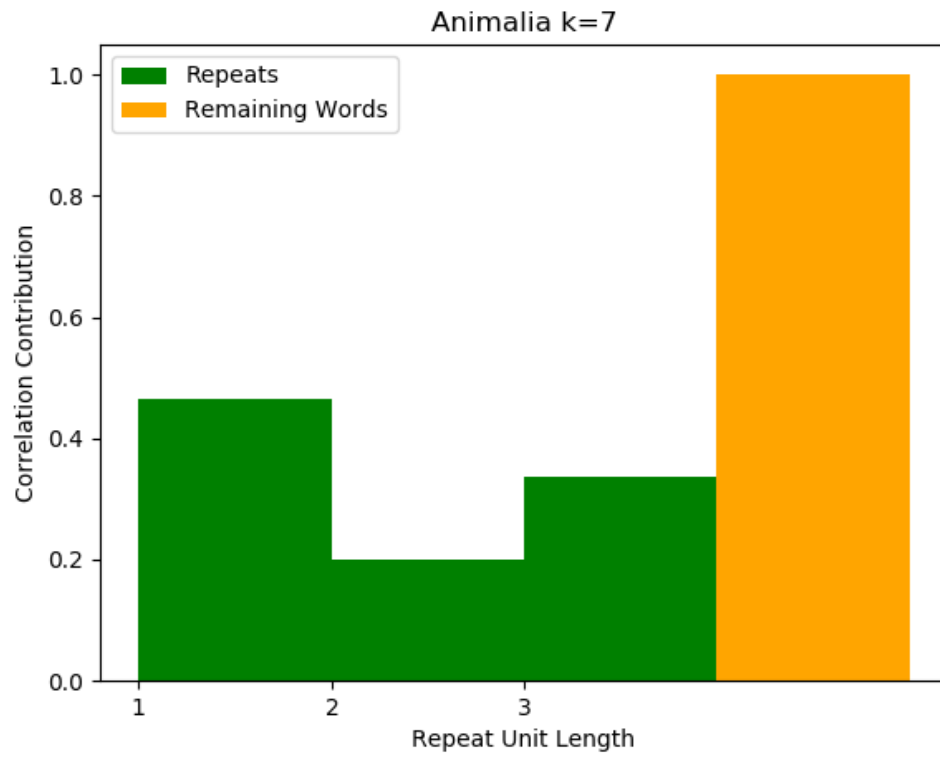
Code:

```
# Einlesen der Daten
matrix = Oligo.Matrix.Matrix.read('Animalia_k=7_correlation_contributions.matrix')
conts = matrix.col('Normalized Correlation Contribution')
words = matrix.row_names

# Analyse der Daten
us = [1,2,3]
data = []
for u in us:
    x = 0.0
    for word, cont in zip(words, conts):
        if Oligo.Kmer.is_repeat(word, u):
            x += cont
    data.append(x)
rest = 1.-sum(data)

# Darstellung der Daten
bins = [Oligo.Bin(data[i-1], u, u+1) for i,u in enumerate(us)]
drawer1 = Oligo.Plot.HistoDrawer(bins=bins, color='green', label='Repeats')
bins = [Oligo.Bin(rest, us[-1]+1, us[-1]+2)]
drawer2 = Oligo.Plot.HistoDrawer(bins=bins, color='orange', label='Remaining Words')
drawer = Oligo.Plot.MultiDrawer([drawer1, drawer2])
drawer.plot('complexity_data.png', title='Animalia k=7', xticks=us, xlabel='Repeat Unit Length', ylabel='Correlation Contribution')
```

Ergebnis:



Längenverteilungen

Es kann, beispielsweise bei TR oder externen Quellen von Interesse sein die Längenverteilungen von Elementen (Loci) zu betrachten. Diese Möglichkeit besteht in Oligo.

Folgender Code berechnet die Längenverteilung und speichert sie ab:

```
import Oligo

loci = Oligo.Locus.read('E:/Daten/Loci/Homo sapiens c1_Al_u_repeatmasker.loci')
lengths = [len(locus) for locus in loci] # Collect Lengths
bins = Oligo.Loci.bins.create_bins(lengths, bin_width=10) # Create Bins
Oligo.Loci.bins.save_bins(bins, 'Homo sapiens c1_Al_u_repeatmasker.lengths.bins')
```

Folgender Code trainiert und speichert ein Markov-Modell als Referenz für die Längen:

Notiz: Das Modell ist nur für Repeats valide! Bei den meisten anderen Modellen würde man wohl eher eine Normalverteilung erwarten.

```
import Oligo
import numpy as np

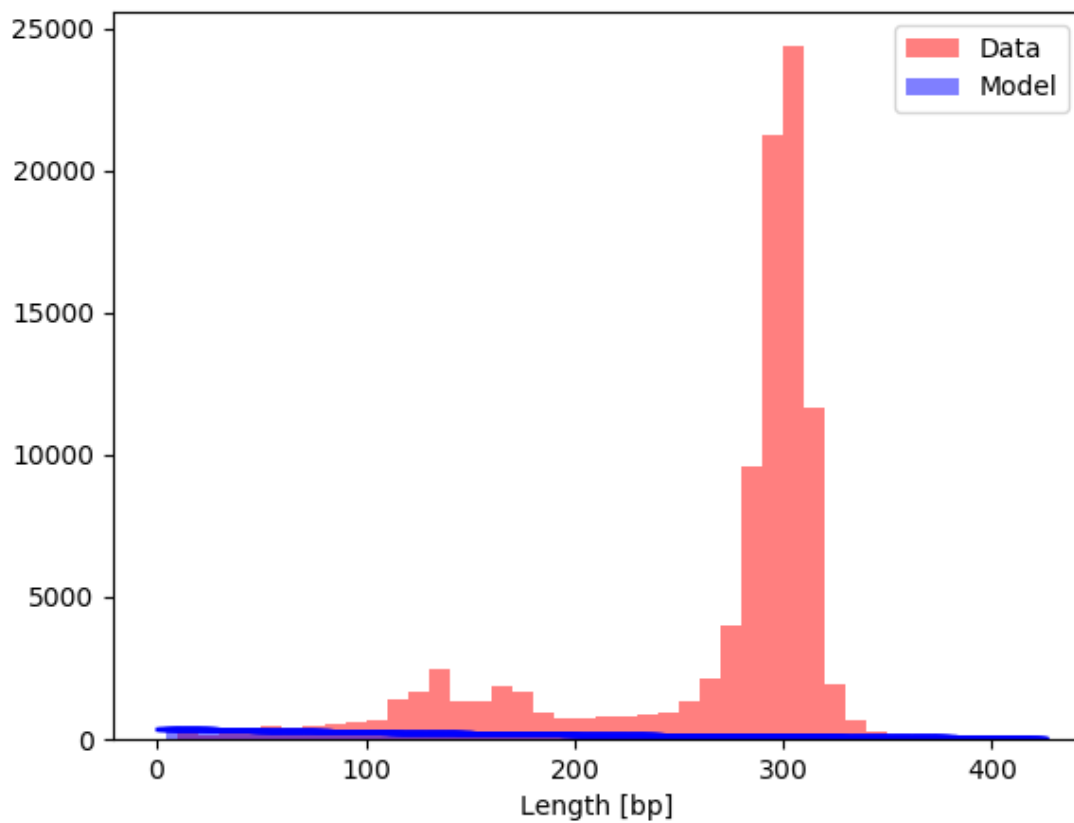
model = Oligo.Loci.create_length_model(lengths=lengths, min_length=4)
max_length = max(lengths)
model_bins = []
t1, t2 = model.get_transition('occupied->occupied').copy(),
model.get_transition('occupied->free').copy()
for i in range(1,max_length):
    trans = [t1]*(i-1) + [t2]
    p = model.get_chain_p(trans)
    n = len(lengths)
    model_bins.append(Oligo.Loci.bins.Bin(count=n*p, start=i+3, end=i+1+3,
std=np.sqrt(n*p*(1.-p))))
Oligo.Loci.bins.save_bins(model_bins, 'Homo sapiens
c1_Al_u_repeatmasker.lengths.model.bins')
```

Folgender Code list die oben generierten Dateien und erstellt ein Histogramm:

```
Import Oligo

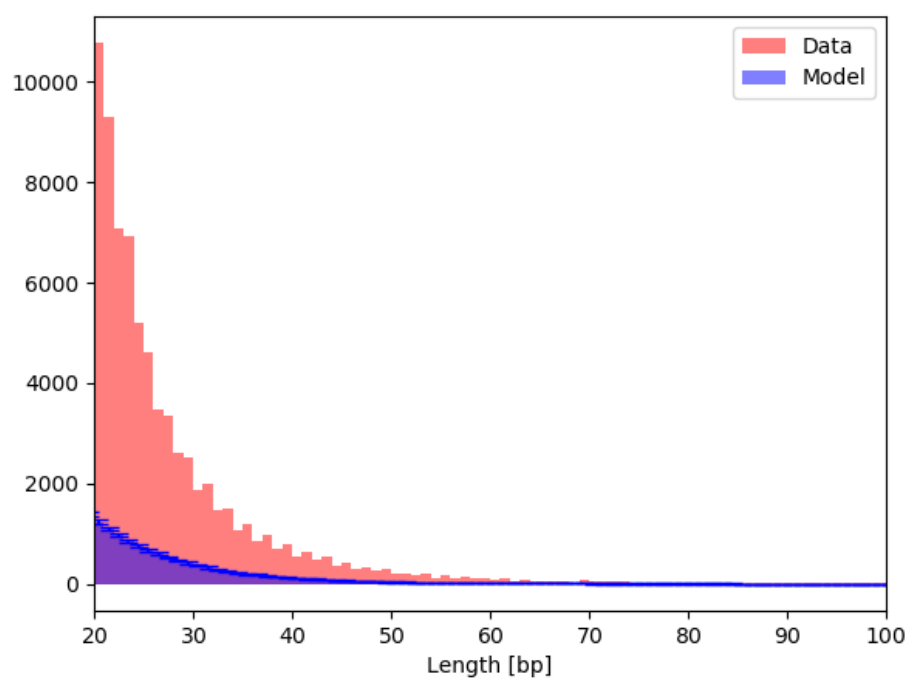
bins = Oligo.Loci.bins.read_bins('Homo sapiens c1_Al_u_repeatmasker.lengths.bins')
model_bins = Oligo.Loci.bins.read_bins('Homo sapiens
c1_Al_u_repeatmasker.lengths.model.bins')

drawer1 = Oligo.Plot.HistoDrawer(bins=bins, color='red', label='Data',
relative=False, alpha=0.5)
drawer2 = Oligo.Plot.HistoDrawer(bins=model_bins, color='blue', label='Model',
relative=False, alpha=0.5)
drawer = Oligo.Plot.MultiDrawer([drawer1,drawer2])
drawer.plot(output_filename='Homo sapiens c1_Al_u_repeatmasker.lengths.png',
xlabel='Length [bp]')
```



Notiz: Es sind deutlich zwei Populationen sichtbar. Die Alu-Dimere mit einer Länge um etwa 300bp und die (wenigen) Alu Monomere mit der Hälfte der Länge.

Beispiel mit Repeats:



Loci in other Loci

Es ist oft von Interesse, wie viele Loci aus einer Gruppe sich innerhalb einer Gruppe anderer Loci befinden, sagen wir wie viele Alus oder TRs liegen innerhalb von Genen.

Dazu existiert ein einfacher Befehl:

```
Oligo.Loci.loci_in_loci(loci1, loci2, target_length=248956422)
```

Beispiel:

```
import Oligo

loci1 = Oligo.Locus.read('E:/Daten/loci/Homo sapiens c1_Alu_repeatmasker.loci')
#loci1 = Oligo.Locus.read('E:/Daten/loci/Homo sapiens c1_(A)n.loci')

loci2 = Oligo.File.read_genes('E:/Daten/genbank/Homo sapiens c1.gb')

Oligo.Loci.loci_in_loci(loci1, loci2, target_length=248956422)
```

Das Ergebnis ist folgende Ausgabe:

```
Loci n = 98824
Loci with overlap k = 61160 (62.0%)
Target Length L = 248956422
Size of Loci2 S = 138243136 (56.0%)
p [S/L] = 0.555290499797
p-value (x<k) = 0.9999999999999999
p-value (x>k) = 1.1102230246251565e-16
model mean = 54876.02835191775
model std = 156.21744826208962
```

Es liegen also 62% aller Alus auf Chromosom 1 innerhalb von Genen (wahrscheinlich in Introns). Nach der angewendeten Binomialstatistik hat dieser Wert eine sehr kleinen p-Value, Alus treten demnach also gehäuft innerhalb von Genen auf.

Next-Neighbour-Distanzen (1D)

Motivation

Nachdem bestimmte Objekte als interessant identifiziert sind, ist es von Interesse ob diese (in der linearen Sequenz = 1D) nahe beieinander liegen. Dies ist deswegen interessant, weil ein geringer Abstand impliziert, dass Sequenzen wahrscheinlich gemeinsam vererbt werden was, bei einem funktionalen Zusammenhang, gut wäre, noch wichtiger aber ist, dass ein geringer Abstand für Biomoleküle rein statistisch vorteilhaft ist um physikalischen Kontakt herzustellen, welcher mangels langreichweitiger Wechselwirkungen (Wasser und Ionen schirmen elektromagnetische Wechselwirkungen ab) in Zellen notwendig ist um zu interagieren.

Kurz: Sollten zwei Elemente näher beieinander liegen als statistisch zu erwarten, ist dies ein Indiz für einen funktionalen Zusammenhang.

Ein signifikantes Ergebnis bei der Distanz zum nächsten Element (Nachbarn), könnte bedeuten, dass eine 1-zu-1 Beziehung, wie Beispielsweise zwischen Promotor und Gen besteht (Ein Gen braucht nur einen nahen Promoter, die Promoter-Dichte spielt keine Rolle).

Berechnung der Distanzen

Liegen zwei Loci-Dateien vor, so kann die Next-Neighbour-Distanz mit folgendem Befehl berechnet werden:

```
Oligo.Loci.distances.nearest_neighbor_distances()
```

Parameter:

loci1 : Liste von loci zur Berechnung von Distanzen (für jeden Locus dieser Liste wird genau eine Distanz berechnet)

loci2 : List zum Suchen des nächsten Nachbarn

dist_func : Funktion zur Distanzberechnung. Im Normalfall ist hier

Oligo.Loci.distances.interspace_distance die richtige Wahl

output_filename : Name der Ausgabedatei

probe_func : Funktion die nach der Distanzberechnung ausgeführt wird. Im Normalfalls:

Oligo.Loci.distances.add_lengths_probe

Beispiel:

```
import Oligo

loci1 = Oligo.File.read_miRNAs(chromo_name='Homo sapiens c1',
seqs_filename=Oligo.File.search('Homo sapiens.seqs'))
loci2 = Oligo.Locus.read('Homo sapiens c1_(A)n.loci')

Oligo.Loci.distances.nearest_neighbor_distances(loci1, loci2,
dist_func=Oligo.Loci.distances.interspace_distance, output_filename='Homo sapiens
c1_miRNA_(A)n_NN.distances', probe_func=Oligo.Loci.distances.add_lengths_probe)
```

Das Ergebnis ist eine Distanz-Datei (*.distances) mit der Form:

```
#date: 2020-04-24 13:19:34.865000
#RESULTS:      created by Oligo.distances.nearest_neighbor_distances
#distance function: <function interspace_distance at 0x0000000005FE73C8>()
#valid pair filter: None()
Distance      Labels
6113  Loc2:35483..35512;length2:29;length1:15009;Loc1:653635;overlapping:0
18047 Loc2:35483..35512;length2:29;length1:68;Loc1:102466751;overlapping:0
4188  Loc2:35483..35512;length2:29;length1:1370;Loc1:107985730;overlapping:0
4980  Loc2:35483..35512;length2:29;length1:138;Loc1:100302278;overlapping:0
0      overlap_end:92688;length2:25;length1:176186;overlap_start:92663;overlapping:1
;Loc2:92663..92688;Loc1:100996442
6139  Loc2:205999..206028;length2:29;length1:14983;Loc1:102723897;overlapping:0
9552  Loc2:178313..178338;length2:25;length1:68;Loc1:102465909;overlapping:0
5400  Loc2:638016..638040;length2:24;length1:1543;Loc1:107075141;overlapping:0
5603  Loc2:638016..638040;length2:24;length1:89;Loc1:102465432;overlapping:0
5561  Loc2:1172759..1172833;length2:74;length1:95;Loc1:406984;overlapping:0
4807  Loc2:1172759..1172833;length2:74;length1:90;Loc1:406983;overlapping:0
3672  Loc2:1172759..1172833;length2:74;length1:83;Loc1:554210;overlapping:0
8664  Loc2:1283698..1283719;length2:21;length1:15542;Loc1:116983;overlapping:0
12390 Loc2:1283698..1283719;length2:21;length1:61;Loc1:102465434;overlapping:0
```

Diese können eingelesen werden über:

`Oligo.Loci.distances.Distance.read(<path>)`

Distanzverteilungen

Berechnung und Speicherung von Distanzverteilungen

Um nun aus diesen Distanzen eine Distanzverteilung zu berechnen, wird eigentlich nur ein Bining-Algorithmus verwendet.

Code:

```
dists = Oligo.Loci.distances.Distance.read('Homo sapiens
c1_miRNA_(A)n_NN.distances')
dists = [d for d in dists if not int(d.labels['overlapping'])]
dist_bins = Oligo.Loci.bins.create_bins(dists, bin_width=100)
Oligo.Loci.bins.save_bins(dist_bins, Oligo.File.translate_output_filename('Homo
sapiens c1_miRNA_(A)n_NN.distances.bins'))
```

Das Ergebnis ist ein Bins-Datei der Form:

```
#date: 2020-04-24 13:28:14.093000
Start End    Count Std
0      100    112   None
100    200     8    None
200    300     8    None
300    400    10    None
400    500     3    None
500    600     5    None
600    700     4    None
```

Modellrechnung

Zur Testung der Signifikanz können analoge Bins mit der Nullhypothese einer komplett zufälligen Verteilung der Loci berechnet werden:

```
dists = Oligo.Loci.distances.Distance.read('Homo sapiens
c1_miRNA_(A)n_NN.distances')
dists = [d for d in dists if not int(d.labels['overlapping'])]

dist_bins = Oligo.Loci.bins.create_bins(dists, bin_width=100)
Oligo.Loci.bins.save_bins(dist_bins, 'Homo sapiens
c1_miRNA_(A)n_NN.distances.bins')

model = Oligo.Loci.distances.DistanceModel.from_distances(dists)
model_bins = model.to_bins(n=len(dists), bin_bounds=[b.get_bounds() for b in
dist_bins], bin_width=len(dist_bins[0]), bin_start=0, bin_end=dist_bins[-1].end)
Oligo.Loci.bins.save_bins(model_bins, 'Homo sapiens
c1_miRNA_(A)n_NN_model.distances.bins')
```

Darstellung von Distanzverteilungen

Sinnvollerweise können Distanzverteilungen (oder allgemeiner Bin-Files) als Histogramme dargestellt werden.

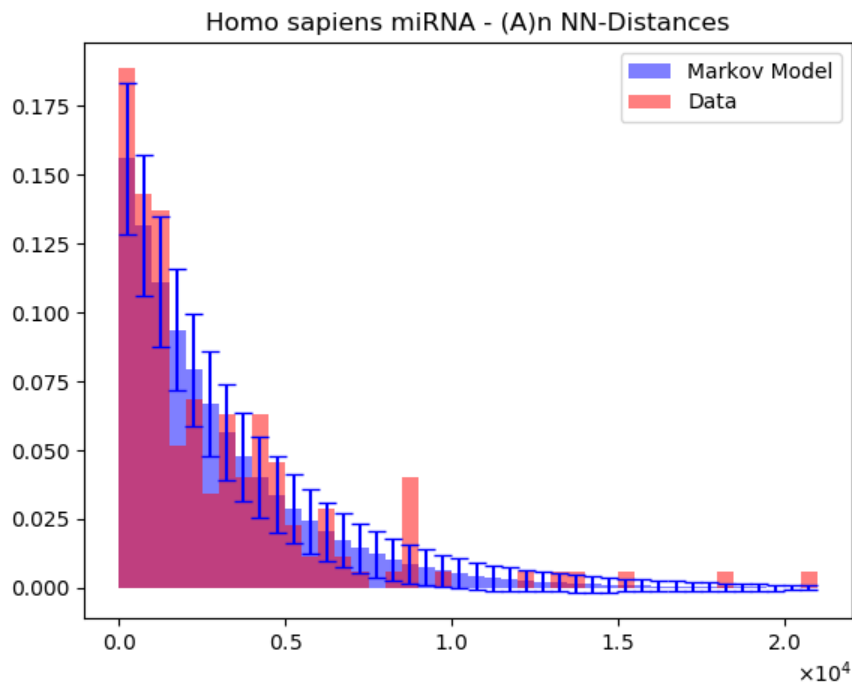
Code:

```
model_bins = Oligo.Loci.bins.read_bins('Homo sapiens
c1_miRNA_(A)n_NN_model.distances.bins')
dist_bins = Oligo.Loci.bins.read_bins('Homo sapiens
c1_miRNA_(A)n_NN.distances.bins')

drawer1 = Oligo.Plot.HistoDrawer(bins=model_bins, color='blue', alpha=0.5,
label='Markov Model')
drawer2 = Oligo.Plot.HistoDrawer(bins=dist_bins, color='red', alpha=0.5,
label='Data')

drawer = Oligo.Plot.MultiDrawer([drawer1, drawer2])
drawer.plot(title='Homo sapiens miRNA - (A)n NN-Distances', output_filename='Homo
sapiens c1_miRNA_(A)n_NN.distances.png')
```

Ergebnis:



Notiz: Das Ergebnis oben ist bis auf wenige Bins insignifikant.

Maps

Maps sind genomische Daten mit räumlicher Auflösung und definierter Auflösung. Das heißt wir können die lokale Dichte bzw. deren (lineare) Verteilung von beliebigen Elementen auf Chromosomen berechnen, darstellen und analysieren.

Erstellen aus Loci

Maps können einfach mit beliebiger Auflösung aus Loci erstellt werden.

`Oligo.Maps.DensityMap()`

Beispiel:

```
import Oligo

loci = Oligo.Locus.read('E:/Daten/loci/Homo sapiens c1_Al_u_repeatmasker.loci',
target=Oligo.Orga.Chromosome('c1','Homo sapiens',length=248956422))

map = Oligo.Maps.DensityMap(loci=loci, resolution=100000)

map.save('Homo sapiens c1_Al_u_repeatmasker.map')
```

Notiz: Natürlich muss kein Chromosome-Objekt erstellt werden, dieses kann auch über `read_genome` direkt ohne Angabe von Länge geladen werden.

Notiz: Die Auflösung ist in Basenpaaren (bp)

Das Ergebnis ist folgende Datei:

```
#date: 2020-05-06 11:45:24.606000
```

```

$Name: None
$Organism: Homo sapiens
$Resolution: 100000
$Target Lengths: c1:248956422
Target Start Value
c1 0 16.0
c1 100000 67.0
c1 200000 17.0
c1 300000 20.0
c1 400000 30.0
c1 500000 56.0
c1 600000 23.0
c1 700000 89.0

```

Neben einigen Metadaten, wie der Auflösung, sind die einzelnen Bins, mit dem Namen des Chromosoms, der Position und der dortigen Dichte (Anzahldichte in [Anzahl/bp]) als Tabelle angegeben.

Visualisierung

Kurven

[coming soon]

Chromosomen-Bilder

Maps (*.map) können über einen einfachen Befehl eingelesen und visualisiert werden.

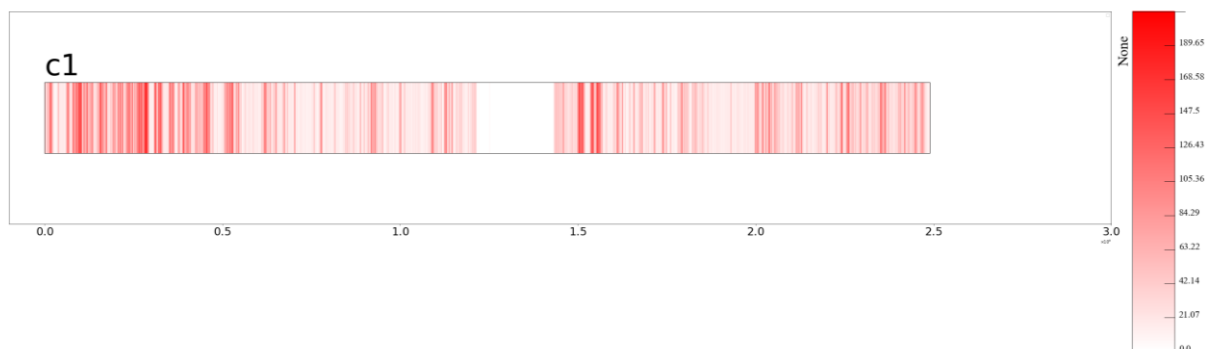
Beispiel:

```

map = Oligo.Maps.DensityMap.read('Homo sapiens c1_Alucrepeatmasker.map')
drawer = Oligo.Plot.MapDrawer(maps=[map])
drawer.plot('Homo sapiens c1_Alucrepeatmasker.map.png', xlim=(-1e7,3e8), ylim=(-1,2),
figsize=(60,15), dpi=60, adjust_right=.75, adjust_bottom=0.4, yticks=[],
xtick_size=28)

```

Ergebnis:



Natürlich können auch mehrere Chromosomen und mehrere Maps angezeigt werden:

Zum Erstellen einer Map mit mehr als einem Chromosom muss lediglich eine Liste von Loci mit mehreren (validen) targets übergeben werden:

```

genome = Oligo.File.read_genome('Homo sapiens')
loci = []
for chromo in genome:

```

```

    loci += Oligo.Locus.read('Alu_repeatmasker_%s.loci' % (chromo.name),
target=chromo)
map = Oligo.Maps.DensityMap(loci=loci, resolution=1e6)
drawer = Oligo.Plot.MapDrawer(maps=[map])
drawer.plot('Homo sapiens_Alu.map.png', xlim=(-1e7,3e8), ylim=(-1,30),
figsize=(60,60), dpi=60, adjust_right=.75, adjust_bottom=0.4, yticks=[],
xtick_size=28)

```

Ergebnis:

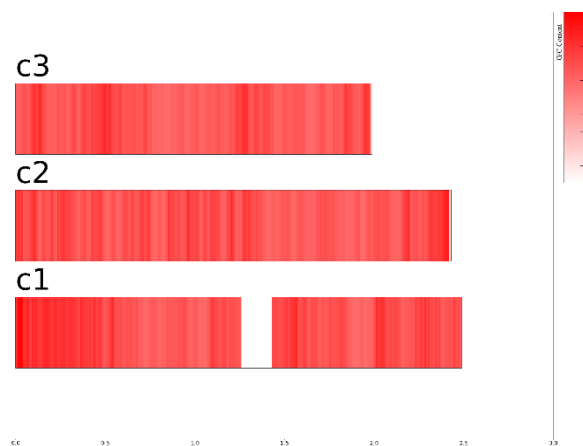


Es kommt zuweilen vor, dass nur eine Auswahl oder eine bestimmte Reihenfolge von Chromosomen bevorzugt wird (nicht zu Zufallswahl im Beispiel oben) dafür kann über einen Parameter eine Reihenfolge und Auswahl bestimmt werden:

```

map = Oligo.Maps.DensityMap.read('Homo sapiens_G+C_Content_1000kb.map')
drawer = Oligo.Plot.MapDrawer(maps=[map], selected_targets = ['c1', 'c2', 'c3'])
drawer.plot('Homo sapiens_G+C.map.png', xlim=(-1e7,3e8), ylim=(-1,5),
figsize=(60,60), dpi=60, adjust_right=.75, adjust_bottom=0.4, yticks=[],
xtick_size=28)

```



Zur Darstellung mehrerer Maps werden diese als Liste an den Drawer übergeben:

```

map1 = Oligo.Maps.DensityMap.read('Homo sapiens_G+C_Content_1000kb.map')
map2 = Oligo.Maps.DensityMap.read('Homo sapiens_(A)n_1000kb.map')

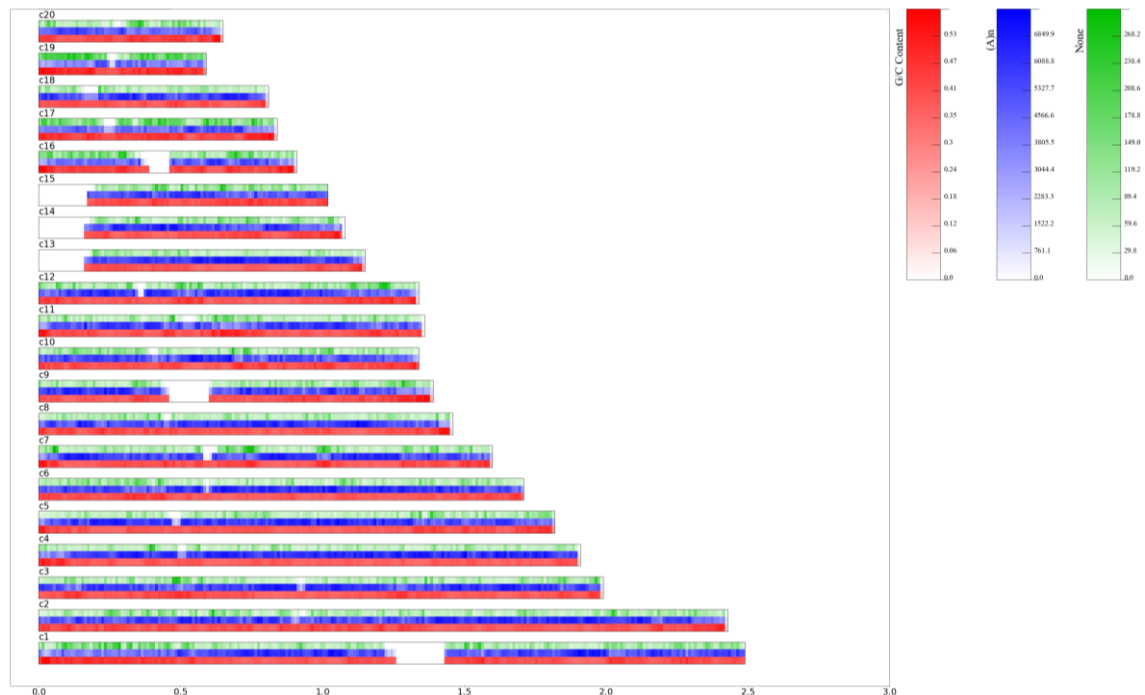
```

```
map3 = Oligo.Maps.DensityMap.read('Homo sapiens_(A)n_larger12bp_1000kb.map')

drawer = Oligo.Plot.MapDrawer(maps=[map1, map2, map3])

drawer.plot('Homo sapiens_test.map.png', xlim=(-1e7,3e8), ylim=(-1,30),
figsize=(60,60), dpi=60, adjust_right=.75, adjust_bottom=0.4, yticks=[],
xtick_size=28)
```

Ergebnis:



Notiz: Maps zum Ausprobieren befinden sich [comming soon] im HeiBox-Ordner.

Map-Korrelation

Um die Ähnlichkeit der Verteilungen auf den Maps zu quantifizieren, können die Maps (vielmehr die zugrundeliegenden Verteilungen) korreliert werden. Hierzu existiert ein einfacher Befehl:

Oligo.Maps.DensityMap.correlate()

Bsp.:

```
import Oligo

chromo = Oligo.Orga.Chromosome('c1','Homo sapiens',length=248956422)
loci1 = Oligo.Locus.read('E:/Daten/loci/Homo sapiens c1_Alucrepeatmasker.loci',
target=chromo)
loci2 = Oligo.File.read_lncRNAs('E:/Daten/genbank/Homo sapiens c1.gb')
Oligo.Loci.add_target(loci2, chromo)

map1 = Oligo.Maps.DensityMap(loci=loci1, resolution=100000)
map2 = Oligo.Maps.DensityMap(loci=loci2, resolution=100000)

map1.save('Homo sapiens c1_Alucrepeatmasker.map')
map2.save('Homo sapiens c1_lncRNA.map')
```

```
map1 = Oligo.Maps.DensityMap.read('Homo sapiens c1_Alucrepeatmasker.map')
map2 = Oligo.Maps.DensityMap.read('Homo sapiens c1_lncRNA.map')

print(Oligo.Maps.correlate(map1, map2))
```

Das Ergebnis ist:

0.042

Es gibt folglich kaum nachweisbare Korrelation zwischen Alu und lncRNAs auf Chromosom 1.

Es können auch mehrere Korrelationen Paarweise durchgeführt werden:

Oligo.Maps.correlation_matrix()

Bsp.:

```
map1 = Oligo.Maps.DensityMap.read('Homo sapiens c1_Alucrepeatmasker.map')
map2 = Oligo.Maps.DensityMap.read('Homo sapiens c1_lncRNA.map')
map3 = Oligo.Maps.DensityMap.read('Homo sapiens c1_genes.map')

matrix = Oligo.Maps.correlation_matrix([map1, map2, map3], names=['Alu', 'lncRNA', 'Genes'])
matrix.show()
```

Das Ergebnis ist:

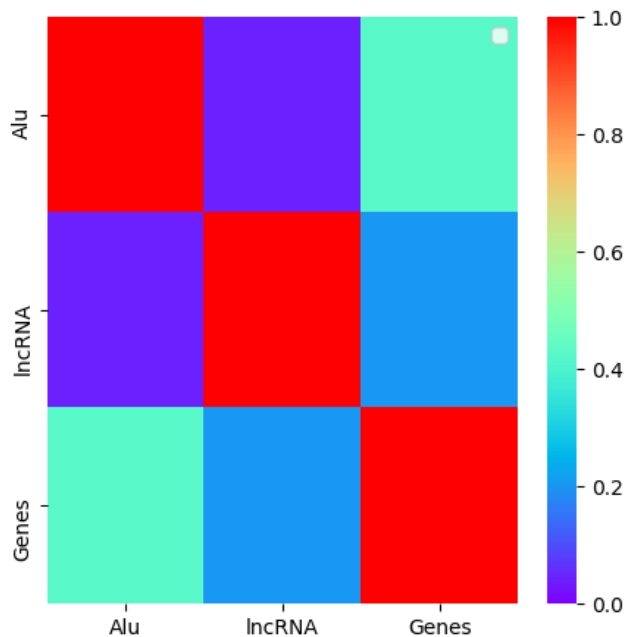
	Alu	lncRNA	Genes
Alu	1.0	0.0416319618585	0.425679998231
lncRNA	0.0416319618585	1.0	0.204878677664
Genes	0.425679998231	0.204878677664	1.0

Es gibt also eine Korrelation von 42.6% zwischen Alu und Genen eine schwächere zwischen lncRNA und Genen und kaum Korrelation zwischen Alu und lncRNA.

Das Ergebnis ist ein Matrix-Objekt und kann wie bei der Korrelation im Kapitel oben als Heatmap dargestellt werden:

```
drawer = Oligo.Plot.HeatmapDrawer(data=matrix.matrix, xticklabels=matrix.col_names,
yticklabels=matrix.row_names, vmin=0.7, vmax=1., cmap='rainbow')
drawer.plot('Homo sapiens_c1-Map_correlations.png', figsize=(15,15))
```

Ergebnis:



Boot-Correlation

Hierbei handelt es sich, um eine fortgeschrittener Variante der Korrelation, welche einen Boot-Strapping-Algorithmus verwendet, um die Korrelation zu beschleunigen und zeitgleich eine Fehlerabschätzung durchzuführen.

Beim Boot-Strapping wird eine großer Datensatz (hier eine Map) in kleinere Datensätze geteilt, die dann wie unabhängige Messreihen behandelt werden, um eine statistische Fehlerabschätzung durchzuführen. Hierbei ist es notwendig, dass die kleinen Datensätze (samples) ausreichend groß sind, damit weiterhin solide statistische Aussagen getroffen werden können. Für unsre Maps haben sich Größen von 50-200 Bins (bei 30-100 samples) als ausreichend erwiesen, damit typische Korrelationen erkannt werden. Eine Garantie für jeden Datensatz gibt es jedoch nicht. Bei Unsicherheiten empfiehlt sich ein Vergleich mit dem klassischen Korrelationswert. Stimmen die Ergebnisse überein, ist die Größe der Samples ausreichend. Grundsätzlich bedeuten größere Samples zuverlässigere Ergebnisse und höhere Rechenzeiten.

`Oligo.Maps.DensityMap.boot_correlate()`

Bsp.:

```
import Oligo

chromo = Oligo.Orga.Chromosome('c1','Homo sapiens',length=248956422)
loci1 = Oligo.Locus.read('E:/Daten/loci/Homo sapiens c1_Alucrepeatmasker.loci',
target=chromo)
loci2 = Oligo.File.read_lncRNAs('E:/Daten/genbank/Homo sapiens c1.gb')
Oligo.Loci.add_target(loci2, chromo)

map1 = Oligo.Maps.DensityMap(loci=loci1, resolution=100000)
map2 = Oligo.Maps.DensityMap(loci=loci2, resolution=100000)

map1.save('Homo sapiens c1_Alucrepeatmasker.map')
map2.save('Homo sapiens c1_lncRNA.map')

map1 = Oligo.Maps.DensityMap.read('Homo sapiens c1_Alucrepeatmasker.map')
map2 = Oligo.Maps.DensityMap.read('Homo sapiens c1_lncRNA.map')

print(Oligo.Maps.boot_correlate(map1, map2, sample_size=10, repeats=25))
print(Oligo.Maps.boot_correlate(map1, map2, sample_size=25, repeats=25))
```

```
print(Oligo.Maps.boot_correlate(map1, map2, sample_size=50, repeats=25))
print(Oligo.Maps.boot_correlate(map1, map2, sample_size=100, repeats=25))
print(Oligo.Maps.boot_correlate(map1, map2, sample_size=200, repeats=25))
print(Oligo.Maps.boot_correlate(map1, map2, sample_size=500, repeats=25))
```

Ergebnis:

```
(0.16552071253524528, 0.31884264737516704)
(0.0716380215362161, 0.13805174674507517)
(0.07378542935516397, 0.16426254811660027)
(0.045948527271276296, 0.08472931849028073)
(0.03668360455291122, 0.06901927363685964)
(0.037279016421044156, 0.02812413745590036)
```

Die Pearson-Korrelation war bei: 0.042

Gut erkennbar ist, dass die Ergebnisse keine Signifikanz dieses Resultats anzeigen. Die Werte sind allerdings konsistent über 0. Diese schwache Korrelation wird erst bei sample sizes über 500 angezeigt. Eine derart hohe Sensitivität wird bei typischen Analysen aber nicht notwendig.

Shuffle

Der Boot-Correlate-Befehl besitzt auch einen shuffle-Parameter, über den man die Signifikanz der Ergebnisse weiter verifizieren kann. Setzt man diesen, werden zufällig gemischte Listen korreliert. Ein Vergleich zwischen diesem Ergebnis und dem Ergebnis ohne shuffle erlaubt eine sehr genaue Einschätzung, ob der Korrelationswert zufällig oder tatsächlich signifikant ist. Beispiel:

```
print(Oligo.Maps.boot_correlate(map1, map2, sample_size=10, repeats=25, shuffle=True))
print(Oligo.Maps.boot_correlate(map1, map2, sample_size=25, repeats=25, shuffle=True))
print(Oligo.Maps.boot_correlate(map1, map2, sample_size=50, repeats=25, shuffle=True))
print(Oligo.Maps.boot_correlate(map1, map2, sample_size=100, repeats=25, shuffle=True))
print(Oligo.Maps.boot_correlate(map1, map2, sample_size=200, repeats=25, shuffle=True))
print(Oligo.Maps.boot_correlate(map1, map2, sample_size=500, repeats=25, shuffle=True))
```

Ergebnis:

```
(0.02305364439905868, 0.35208444370496905)
(-0.0376234473248843, 0.14971022689795935)
(-0.05163560487222577, 0.09799623749194503)
(0.006523059881173074, 0.07364072301058278)
(0.0035513661884717575, 0.08060867640324842)
(-0.005488489899782721, 0.0374854524166825)
```

Hier streuen die Werte offensichtlich um 0. Beachtet man die Fehlergrenzen, wird klar, dass der Wert selbst bei 500 nur eine extrem geringe Signifikanz aufweist.

Deutung der Ergebnisse:

Obwohl der Korrelationswert also konsistent und signifikant über 0 liegt, befindet er sich innerhalb der Range, die auch bei einer zufälligen Verteilung auftritt. Die Angezeigte Korrelation ist damit mit einiger Wahrscheinlichkeit das Ergebnis einer real vorhandenen aber zufälligen positiven Korrelation.

Masken

Im Allgemeinen existieren auf genomischen Sequenzen Lücken (Gaps) oder andere Strukturen (z.B. Telomere, Zentromere), die zu Artefakten in den Analysen, insbesondere in den Korrelationen führen können (z.B. werden bei Gaps alle Elemente eine reduzierte Anzahl zeigen, was zu einer Korrelation führt).

Um diese oder beliebige andere Bereiche bei der Korrelation von Maps zu entfernen, existieren Masken. Diese können gespeichert werden, sodass derselbe Filter dann für mehrere Analysen (auf dem demselben Genom, bei gleicher Auflösung) verwendet werden kann.

Masken sind im wesentlichen langen Listen in denen für jedes Chromosom und jeden Map-Bin eine 1 oder eine 0 steht, wobei 1 bedeutet der Bin soll in der Korrelation verwendet werden, während 0 bedeutet, der Bin wird ignoriert.

Masken Erstellen

Im üblichen Anwendungsfall werden Masken aus bestehenden Maps (z.B. für Gaps) erstellt. Folgender Code erstellt eine Maske für Zentromere aus einer entsprechenden Map:

```
map = Oligo.Maps.DensityMap.read('HS_c1_centromeres_10kbp.map')
mask = Oligo.Maps.MapMask.mask_from_map(map, mask_function=lambda li: 1 if li == 0 else 0)
```

Die gezeigte Mask-Funktion ist immer dann 1, wenn in der Map eine 0 steht also wenn sich dort gerade kein Zentromer befindet. Der Befehl basierend auf Gaps usw. wäre analog.

Masken Kombinieren

Es ist meistens sinnvoll, Gaps und Zentromere oder andere Kombinationen aus Masken zu verwenden. Entsprechend existiert eine Möglichkeit zum Kombinieren von Masken:

```
mask = Oligo.Maps.MapMask.combine_masks([mask1,mask2])
```

Masken speichern & laden

Um einmal erstellte Masken erneut zu verwenden, können diese als Dateien gespeichert werden. Hierfür existiert folgender Befehl:

```
mask.save(output_filename)
```

Entsprechend ist es auch möglich Masken aus Dateien auszulesen:

```
mask = Oligo.Maps.MapMask.read(input_filename)
```

Masken laden/verwenden

Beim klassischen `correlate` und bei `boot_correlate` existiert die Möglichkeit zur Verwendung einer Maske durch den `map_mask`-Parameter.

Beispiele:

```
Oligo.Maps.correlate(map1, map2, map_mask=mask)
```

```
Oligo.Maps.boot_correlate(map1, map2, sample_size=50, repeats=30,
map_mask=mask)
```

Masken für Mensch und Maus

Relevante Masken für die Genome von Mensch (hg38) und Maus, sind in der Heibox abgelegt unter:

Test Data \ Masken

Erstellen von Standard-Masken für Genome

Ein häufiger Anwendungsfall ist das Erstellen von kombinierten Gap- und Centromere-Masken für ganze Genome, z.B. um einen Bias bei einer Map-Correlation zu vermeiden. Das folgende Skript kann hierfür als Prototyp betrachtet werden.

```
import Oligo

genome = Oligo.File.read_genome('Homo sapiens')

# Generate Gap Loci
chromo_gaps = {}
for chromo in genome:
    chromo_gaps[str(chromo)] = Oligo.File.read_sequence_gaps(chromosome=chromo)
    Oligo.Locus.save(chromo_gaps[str(chromo)], '%s_gaps.loci' % str(chromo))

# Create Coverage Map from Gaps Loci
map = Oligo.Maps.DensityMap(target_loci=chromo_gaps, name='Homo sapiens Gaps',
target_lengths={str(chromo):len(chromo) for chromo in genome}, resolution=1e6,
mode=2)
map.save('Homo sapiens_Gaps_1Mbp.map')

# Create Mask from Gap Map
gap_mask = Oligo.Maps.MapMask.mask_from_map(map, mask_function=lambda li: 1 if li <
0.1 else 0)
gap_mask.save('Homo sapiens_Gaps_1Mbp.mask')

# Generate Centromere Loci
chromo_centros = {}
for chromo in genome:
    chromo_centros[str(chromo)] = Oligo.File.read_centromere(chromo.filename)
    Oligo.Locus.save(chromo_centros[str(chromo)], '%s_centromeres.loci' %
str(chromo))

# Create Coverage Map from Centromere Loci
map = Oligo.Maps.DensityMap(target_loci=chromo_centros, name='Homo sapiens
Centromeres', target_lengths={str(chromo):len(chromo) for chromo in genome},
resolution=1e6, mode=2)
map.save('Homo sapiens_Centromeres_1Mbp.map')

# Create Mask from Centromere Map
centro_mask = Oligo.Maps.MapMask.mask_from_map(map, mask_function=lambda li: 1 if li
== 0 else 0)
centro_mask.save('Homo sapiens_Centromere_1Mbp.mask')

# Combine Masks
mask = Oligo.Maps.MapMask.combine_masks([gap_mask,centro_mask])
mask.save('Homo sapiens_Gaps+Centromere_1Mbp.mask')
```

Density Profile

Eine weitere Information über die Umgebung von Elementen liefern „Density Profiles“.

Diese zeigen die Dichteverteilung um die Elemente an.

Grundlage zur Berechnung der Profile sind Loci und Map-Daten in der gewünschten Auflösung des Profils.

Folgendes Beispiel erzeugt das Profil von Poly-A TR um Alu Loci:

```

Import Oligo

chromo = Oligo.Orga.Chromosome('c1','Homo sapiens',length=248956422)
loci1 = Oligo.Locus.read('E:/Daten/loci/Homo sapiens c1_Alu_repeatmasker.loci',
target=chromo)
loci2 = Oligo.Locus.read('E:/Daten/loci/Homo sapiens c1_(A)n.loci', target=chromo)

poly_A_map = Oligo.Maps.DensityMap(loci=loci2, resolution=1000000)

profile = Oligo.Maps.density_profile(loci1, poly_A_map, consider_strain=True,
limit=1e7)

Oligo.Maps.save_density_profile('Homo sapiens c1 (A)n around
Alu.density_profile.dat', profile)

```

Ergebnis:

distance density n	mean density	map stderr	model err	map density	std density	model
-3000000	324.35033780030756	70.79185623824279		0.20703795656791485	0.0	
	327.437394164	64.49356967449452	327.437394164	97036		
-2000000	320.90020044555183	70.79185623824279		0.22581534195530578	0.0	
	327.437394164	70.68056275847351	327.437394164	97970		
-1000000	322.63199832923647	70.79185623824279		0.22258484445666749	0.0	
	327.437394164	69.90794324360145	327.437394164	98642		
0	344.3268560263585	70.79185623824279		0.2110796858706078	0.0	327.437394164
	66.35561113718764	327.437394164	98824			
1000000	335.17939991456444	70.79185623824279		0.21147880985145132	0.0	327.437394164
	66.24689752442258	327.437394164	98129			
2000000	327.98662802059147	70.79185623824279		0.20393684142567112	0.0	327.437394164
	63.43911024074877	327.437394164	96766			
3000000	327.1701451860798	70.79185623824279		0.22111173713307283	0.0	327.437394164
	68.49576252438878	327.437394164	95963			

Es handelt sich um eine Tabelle mit verschiedenen Parametern, welche später zur Visualisierung verwendet werden. Die erste Spalte „distance“ und zweite Spalte „mean density“ sind die wesentlichen Werte.

Notiz:

Die Rechenzeit ist linear zur Anzahl an Elementen in loci1 und zum Distanz-Limit (limit). Es ist darum empfehlenswert bei einer großen Anzahl an Elementen (mehr als ein paar 1000) kein unnötig großes Limit zu wählen.

Visualisieren

Selbstverständlich können die Density Profiles auch visualisiert werden:

```

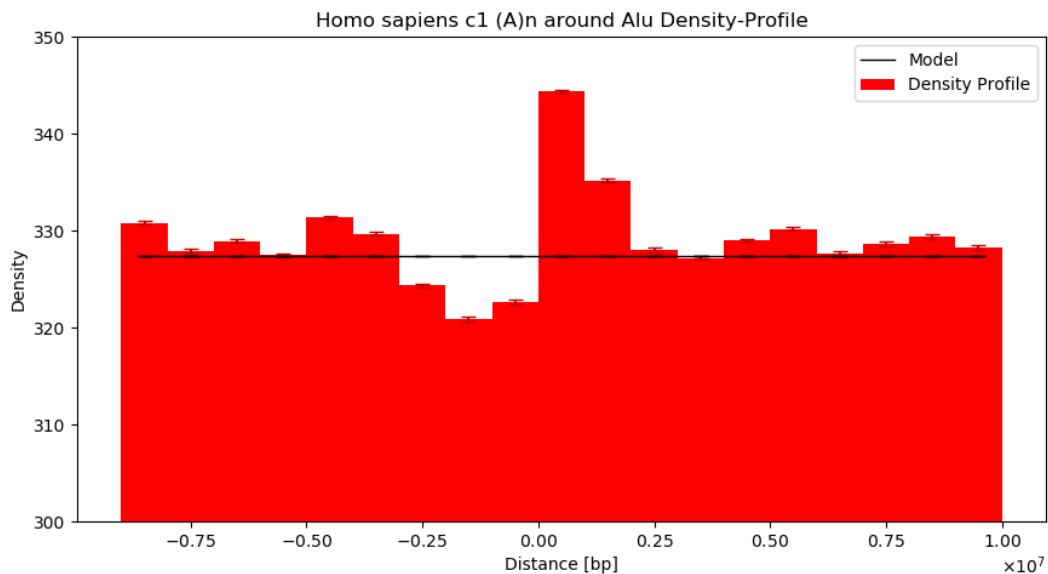
profile = Oligo.Maps.read_density_profile('Homo sapiens c1 (A)n around Alu.density_profile.dat')

dists = sorted(profile['distance'])
res = dists[1]-dists[0]
bins = [Oligo.Loci.bins.Bin(float(profile['mean density'])[i]), float(profile['distance'])[i]),
float(profile['distance'])[i]+res, std=float(profile['err'])[i])] for i in range(len(profile['distance']))]
drawer1 = Oligo.Plot.HistoDrawer(bins=bins, color='red', label='Density Profile', relative=False, err_color=(.75,0,0))
drawer2 = Oligo.Plot.CurveDrawer(x=[bin.start+len(bin)/2. for bin in bins], y=[float(profile['model density'])[i] for
i in range(len(profile['distance']))], y_err=[float(profile['model err'])[i] for i in range(len(profile['distance']))],
label='Model', err_color='black', color='black')
drawer = Oligo.Plot.MultiDrawer([drawer1,drawer2])

```

```
drawer.plot(title='Homo sapiens c1 (A)n around Alu Density-Profile', output_filename='Homo sapiens c1 (A)n around Alu.density_profile.png',figsize=[10,5], xlabel='Distance [bp]', ylabel='Density')
```

Ergebnis:



Das Bild oben zeigt eine offenkundige Häufung von Poly-A direkt downstream (also hinter) Alu-Elementen, was der Tatsache entspricht, dass ein Poly-A-Tail zur Alu-Sequenz gehört. Der Mangel an Poly-A direkt vor Alu-Elementen ist hingegen eine andere Sache.

Kombinieren von Density Profiles

[coming soon]

3D Density Profiles

Auf Grundlage von Hi-C-Daten, können die 3D-Distanzen zwischen beliebigen Regionen des Genoms abgeschätzt werden. Da es sich hierbei um eine eher grobe Übersetzung handelt, welche hinzu auf einigen Annahmen beruht, muss hier mit viel Rauschen gerechnet werden.

Einlesen von Hi-C-Daten

Die Daten können runtergeladen werden unter:

[coming soon]

Ein Testset für Chromosom 1 befindet sich im HeiBox-Verzeichnis:

HiCtool_chr1_40kb_observed.txt

Die Daten können eingelesen werden über:

Oligo.Matrix.HiCMatrix.read()

Beispiel:

```
import Oligo

matrix = Oligo.Matrix.HiCMatrix.read('HiCtool_chr1_40kb_observed.txt',
resolution=40*1000)
```

Diese Matrizen können direkt als Heatmap dargestellt werden. Wegen ihrer Größe ist es allerdings empfehlenswert nur eine Sub-Matrix zu betrachten. Diese erhält man über:

d.sub_matrix()

Beispiel:

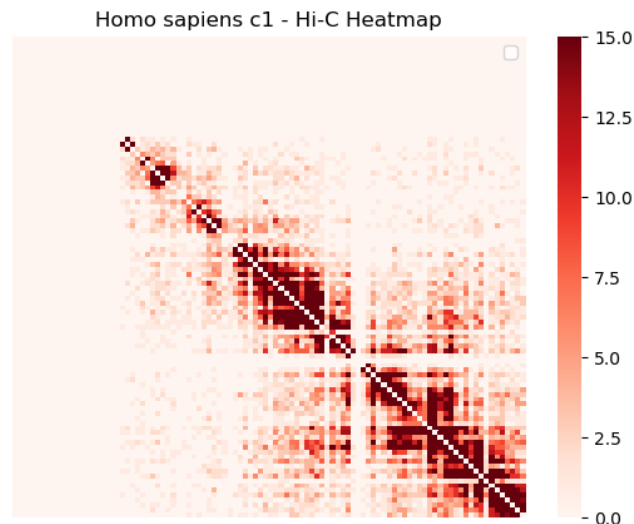
```
import Oligo

matrix = Oligo.Matrix.HiCMatrix.read('HiCtool_chr1_40kb_observed.txt',
resolution=40*1000)

sub_matrix = matrix.sub_matrix(0, 100)

drawer = Oligo.Plot.HeatmapDrawer(sub_matrix.complete_matrix(none_filler=1e6),
cmap='Reds', vmin=0, vmax=15)
drawer.plot('Hi-C-Heatmap.png', title='Homo sapiens c1 - Hi-C Heatmap')
```

Ergebnis:



Notiz: Bereiche, die lokal stark mit anderen Bereichen interagieren (Dreiecke um die Diagonale) nennt man TADs.

Distanz-Heatmaps

Die Daten können in Distanzen übersetzt werden über:

`Oligo.Matrix.HiCMatrix.distances()`

Beispiel:

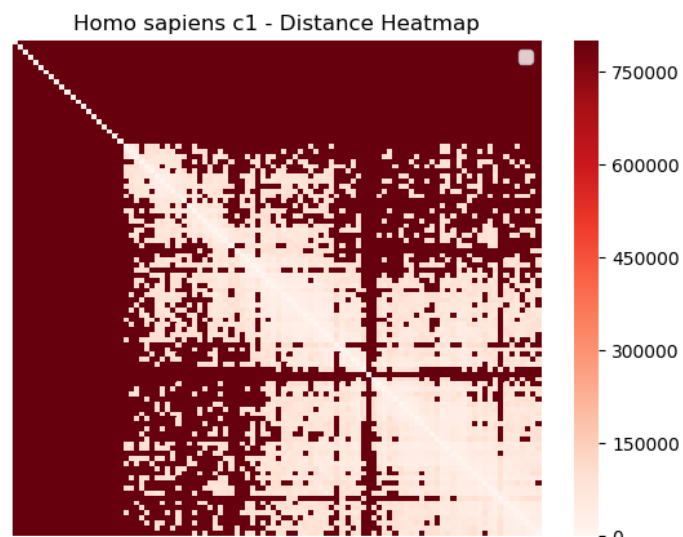
```
import Oligo

matrix = Oligo.Matrix.HiCMatrix.read('E:/Daten\Hi-C/GSM1267196_contact_matrices/GSM1267196_observed/HiCtool_chr1_40kb_observed.txt',
resolution=40*1000)
distance_matrix = matrix.distances()

sub_matrix = distance_matrix.sub_matrix(0, 100)

drawer = Oligo.Plot.HeatmapDrawer(sub_matrix.complete_matrix(none_filler=1e6),
cmap='Reds', vmin=0, vmax=40000*100)
drawer.plot('Distance-Heatmap.png', title='Homo sapiens c1 - Distance Heatmap')
```

Ergebnis:



Berechnung 3D Density Profiles

Mithilfe dieser 3D-Distanzen und Maps mit gleicher Auflösung, wie die Hi-C-Daten (im Beispiel 40kb), können 3D-Density-Profiles analog zu den 1D-Density-Profiles erstellt werden.

Beispiel:

Tandem-Repeat-Modelle

Zur Berechnung von Referenzdaten existiert in Oligo die Möglichkeit Erwartungswerte sowie Schwankungsbreiten für die Anzahl und Abdeckung von Short Tandem Repeats (STRs), basierend auf Nucleotid oder Dinucleotid-Profilen zu erstellen.

Theoretischer Hintergrund

In beiden Fällen werden Markov-Modelle verwendet, um Wahrscheinlichkeiten für eine anschließende Binomialstatistik zu erstellen.

Die Wahrscheinlichkeit für das Auftreten einer Sequenz W , im Nukleotidmodell (z.B. ATATAT) ist allgemein gegeben durch:

$$p_W = \prod_{i=1}^{l_W} p_{W_i}$$

Hierbei ist l_W die Länge von W und p_{W_i} die Wahrscheinlichkeit des Auftretens des Nucleotides $W_i \in \{A, C, G, T\}$.

z.B.:

$$p_{ATATAT} = p_A p_T p_A p_T p_A p_T = p_A^3 p_T^3$$

Diese Wahrscheinlichkeit ist korrekt für k-mer-Worte, schließt allerdings nicht aus, dass diese Sequenz Teil einer längeren Kette ist (z.B. ATATATATAT). Aus diesem Grund muss ausgeschlossen werden, dass sich vor sowie nach der Sequenz eine weitere Wiederholung befindet. Dies beeinflusst die Wahrscheinlichkeit:

$$p'_W = p_{start} p_W p_{end}$$

$$p_{start} = (1 - p_{W_1})$$

$$p_{end} = \begin{cases} (1 - p_{W_1}) & \text{wenn } l_W \text{ gerade} \\ (1 - p_{W_2}) & \text{wenn } l_W \text{ ungerade} \end{cases}$$

p_{start} ist gerade die Wahrscheinlichkeit, dass sich vor der Sequenz kein weiteres passendes Nukleotid befindet. p_{end} ist komplexer, da das Ende der Sequenz abhängig, ob der Repeat innerhalb einer Repeatunit endet oder nicht unterschiedlich ausfällt.

Anmerkung: Diese Berechnung setzt voraus, dass z.B. TATATAT nicht als ATATAT gewertet werden soll – was innerhalb von Oligo so gezählt wird. TATATAT würde als dort als Wiederholung von TA und nicht von AT gewertet.

Mit der Wahrscheinlichkeit p_W (auf den Strich wir nun verzichtet), kann nun, ausgehend von einer Binomialverteilung, die erwartete Anzahl von STRs sowie deren Varianz berechnet werden:

$$N_W = L p_W$$

$$\Delta N_W = \sqrt{L p_W (1 - p_W)}$$

Hierbei ist L die Länge der Gesamtsequenz (z.B. ein Chromosom). Diese Binomialstatistik ist valide, solange $p_W \ll 1$. Denn nur dann, können die Wahrscheinlichkeiten des Auftretens eines STRs an jeder beliebigen Position innerhalb der Gesamtsequenz als unabhängige Zufallsereignisse gewertet werden – andernfalls würde z.B. das Auftreten eines STRs verhindern, dass weitere STRs innerhalb des STRs beginnen und entsprechend wäre p_W an diesen Stellen 0.

Diese Bedingung ist aber in der Regel gegeben und zudem würde die Anzahl von STRs andernfalls systematisch überschätzt. Sind demnach die empirischen Daten höher als die modellierten Erwartungswerte ist dies immer vertrauenswürdig.

Aus der Anzahl N_W lässt sich dann leicht die Abdeckung (Coverage) berechnen:

$$C_W = l_W N_W = L l_W p_W$$

$$\Delta C_W = l_W \Delta N_W = \sqrt{L l_W^2 p_W (1 - p_W)}$$

Das Dinukleotid-Modell verwendet dieselben Grundannahmen. Hierbei wird lediglich die Berechnung von p_W verändert, indem anstelle der Wahrscheinlichkeiten von Nukleotiden die von Dinukleotiden treten.

$$p_W = p_{start} \left(\prod_{i=1}^{l_W} p_{W_i W_{i+1}} \right) p_{end}$$

Hier ist nun p_{XY} die Wahrscheinlichkeit des Auftretens des Dinukleotides XY ($X, Y \in \{A, C, G, T\}$). Also die bedingte Wahrscheinlichkeit $p(Y|X)$.

$$p_{start} = 1 - p_{W_2 W_1}$$

$$p_{end} = \begin{cases} (1 - p_{W_2 W_1}) & \text{wenn } l_W \text{ gerade} \\ (1 - p_{W_1 W_2}) & \text{wenn } l_W \text{ ungerade} \end{cases}$$

Anwendung in Oligo

In Oligo existieren zwei Klassen als Repräsentation der beiden oben beschriebenen Modelle:

```
Oligo.Repeats.TRMonomerModel  
Oligo.Repeats.TRDimerModel
```

Beschreibung:

Oligo.Repeats.TRMonomerModel()

Parameter:

nuc_data (dict) : Dictionary mit den Nucleotiden als Keys und deren Häufigkeiten als, z.B.: {'A':0.2, 'T':0.2, 'C':0.3, 'G':0.3}

data_normalized (bool) : Wahrheitswert, ob die Angaben in nuc_data bereits auf 1 normiert sind. Falls nicht, wird die Normalisierung durchgeführt. Defaultwert: False

consider_inverse (bool) : Wahrheitswert, ob inverse STRs als derselbe STR gewertet werden. z.B., ob TATA auch zu (AT)_n zählt. Defaultwert: True

Oligo.Repeats.TRDimerModel

Parameter:

dinuc_data (dict) : Dictionary mit den Dinucleotiden als Keys und deren Häufigkeiten als, z.B.: {'AA':0.02, 'AC':0.02, ...}

data_normalized (bool) : Wahrheitswert, ob die Angaben in dinuc_data bereits auf 1 normiert sind. Falls nicht, wird die Normalisierung durchgeführt. Defaultwert: False

Beide besitzen eine Methode get_expectation-Methode, mit identischen Parametern:

Oligo.Repeats.TRModel.get_expectation()

Parameter:

tr (str) : Repeat-Unit des STRs, für welchen Erwartungswerte generiert werden sollen, z.B.: 'AG'.

sequence_length (int) : Länge der Sequenz (in bp), für die Erwartungswerte generiert werden sollen.

l_min (int) : Minimallänge der STRs, die gezählt werden sollen. Default: 4

l_max (int) : Maximallänge der STRs, die gezählt werden sollen. Default: 50
Es zeigt sich, dass Längen über 50 praktisch nie zu den Ergebnissen beitragen.

consider_inverse (bool) : Wahrheitswert, ob inverse STRs als derselbe STR gewertet werden. z.B., ob TATA auch zu (AT)_n zählt. Defaultwert: True

Returns:

Tupel (n,c,en,ec) mit folgenden Werten:

n : Expectation value für die Anzahl an STRs
c : Expectation value für coverage der STRs
en : Fehler von n
ec : Fehler von c

Beispiel:

Berechnung von Erwartungswerten für eine Sequenz der Länge 10000 bp für (AT)_n mit n = 4 und n = 5:

```
import Oligo

model = Oligo.Repeats.TRMonomerModel(nuc_data={'A':0.3,'T':0.3,'C':0.2, 'G':0.2})

(n,c,en,ec) = model.get_expectation(tr='AT', sequence_length=10000, l_min=4, l_max=4)
print('Expected Number of (AT)n with n=4 %s +/- %s' % (n,en))

(n,c,en,ec) = model.get_expectation(tr='AT', sequence_length=10000, l_min=5, l_max=5)
print('Expected Number of (AT)n with n=5 %s +/- %s' % (n,en))
```

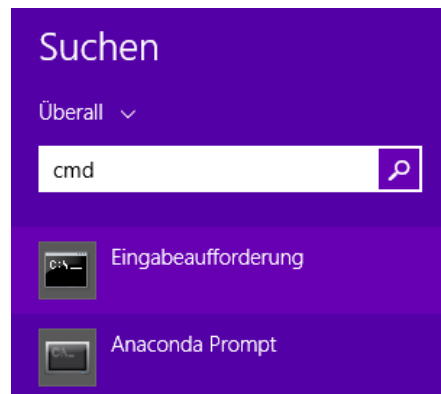
Ausgabe:

```
Expected Number of (AT)n with n=4 79.38 +/- 8.891846871151122
Expected Number of (AT)n with n=5 23.813999999999997 +/- 4.877052867275482
```

Anhang:







Windows Kommandozeile

Das Tool kann über die Windows-Suche leicht gefunden werden, indem man den Suchbegriff „cmd“ eingibt:



Es kann auch direkt navigiert werden zu:

C:\Users\<Username>\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\System Tools

Name	Änderungsdatum	Typ	Größe
 Ausführen	22.08.2013 08:54	Verknüpfung	1 KB
 Dieser PC	22.08.2013 08:54	Verknüpfung	1 KB
 Eingabeaufforderung	22.08.2013 08:52	Verknüpfung	2 KB
 Explorer	22.08.2013 08:54	Verknüpfung	1 KB
 Hilfe und Support	22.08.2013 08:54	Verknüpfung	1 KB
 Systemsteuerung	22.08.2013 08:54	Verknüpfung	1 KB
 Windows Defender	22.08.2013 08:56	Verknüpfung	2 KB

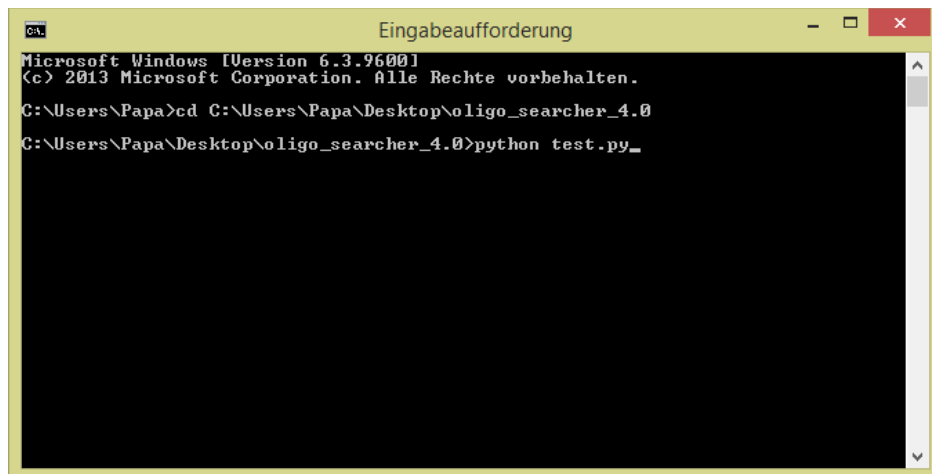
Dort befindet sich eine Verknüpfung zu „Eingabeaufforderung“ (hier als Kommandozeile bezeichnet):



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. Alle Rechte vorbehalten.
C:\Users\Papa>
```

Um innerhalb dieser Umgebung das Verzeichnis (Arbeitsverzeichnis) zu wechseln wird der „change directory“ (cd) Befehl verwendet:

z.B.



```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. Alle Rechte vorbehalten.
C:\Users\Papa>cd C:\Users\Papa\Desktop\oligo_searcher_4.0
C:\Users\Papa\Desktop\oligo_searcher_4.0>python test.py_
```