Contents

はじめに

現代では脳は電気で動いている、と信じられています。
しかし、どのような挙動なのかはまだまだ分かっていません。
だから、貴方は研究をしたくなります。(それは火を見るより明らかです)
しかし、脳の解析は難しく、技術的な入門書、特に和書に乏しい現状があります。
だから同人誌を書くことにしました。
本書では脳磁図、脳波、MRI 解析を「体で覚える」べく実践していきます。
さぁ、MNE/python、freesurfer の世界で良い生活を送りましょう！

...というか、周りに MNEpython 使いほとんど居ない...一人じゃ辛い。

本書の目的・限界・対象者・使い方

MNE/python や freesurfer を用いて脳内の電源推定...特にソースベース解析を行うための
解析環境の構築と解析の基礎を概説します。可能な限り効率的な解析環境を構築し、楽をします。
僕が個人的に考えている事もちょくちょく書きます。
僕は elekta 社の MEG を使っているので elekta 前提で書きます。

本書の限界は僕のスキル不足 (2 年と少しかやっていない) と、これが同人誌であることです。
不確実なものとして、疑って読んでいただければ幸いです。
この同人誌は不完全なため、日々更新しています。

本書の対象者は以下のとおりです

- 脳波/脳磁図計を使って研究をしたい初心者
- 脳磁図計を使って研究しているけれど、コーディングが苦手な中級者
- 頭部 MRI 研究で freesurfer を使いたい初心者
  また、前提条件としてターミナルやプログラミングを怖がらないことがあります。
  (プログラミング未経験者の質問にも出来るだけ答えたいと思います)

脳研究の経験者は MNE/python とはから読んでいけばいいです。
MNE/freesurfer 経験者なら OS の準備から読めばいいです。
コンピュータは自転車みたいなもので、基本は体で覚えていくしかないと思っています。
分からなければググることが大事です。qiita[1]等で検索するのも良いでしょう。

[1]日本のプログラマ用の SNS の一つです。

センサーベース/ソースベース解析とはなんぞや

脳の中の電気信号を調べる方法としては脳波や脳磁図[2]が有名です。
脳波や脳磁図のセンサーで捉えた信号を直接解析する方法を
センサーレベルの解析と言います。これは伝統的なやり方であり、
今でも多くの論文がこの方法で出ている確実な方法です。

しかし、脳波や脳磁図は頭蓋骨を外して直接電極をつけないと
発生源 (僕達はソースと呼びます) での電気活動はわかりません[3]。
普段計測している脳波・脳磁図は所詮は「漏れでた信号」に過ぎないのです。
では、一体どうすれば脳内の電気信号を非侵襲的に観察できるのでしょうか？
方法は残念ながらない[4]のですが、推定する方法ならあります。
その中の一つの方法として、脳磁図と MRI を組み合わせ、
MNE という python パッケージを使って自ら解析用スクリプトを実装する方法があります。
ソースベース解析というのはあくまで推定であり、
先進的である一方でまだまだ確実性には劣るやり方との指摘もあります。

ちなみに、脳波のソースベース解析もあるにはあるのですが、
脳波は電流であるため磁力と違って拡散しやすい性質があります。
実際、脳波でのソースベース解析とセンサーベース解析の結果が
不一致であったという研究が発表されています。[5]

スラングや用語の説明と **unix** 系のお約束

本書では下記の言葉を使っています。伝統的なスラングを含みます。
適宜読み替えていってください。それ以外にも色々スラングあるかもです…。

- hoge:貴方の環境に応じて読み替えてください、という意味のスラング
  fuga,piyo も同じ意味です。ちなみにこれは日本語です。
  英語が好きな方は foo とか bar とかになりますね。
- 叩く:(コマンドをターミナルから) 実行するという意味の他動詞
- 回す、走らせる:重い処理を実行するという意味の他動詞
- ターミナル:いわゆる「黒い画面」のこと。Mac ならユーティリティフォルダにある。
- .bash_profile:ホームディレクトリにある隠し設定ファイルです。
  環境によって.bashrc だったりしますし、両方あることもあります。

[2]脳波は電気信号を捉えますが、脳磁図は磁場を捉えます。電気と違って骨を貫通しやすく拡散しにくいので空間分解能に優れますが、ノイズに弱いです。値段も高いです。http://www.elekta.co.jp/products/functionalmapping.html
[3]動物実験では脳に電極刺す実験はされていますが、人に刺すと警察に捕まります。
[4]他に脳の活動を調べる方法として磁力を照射する fMRI や赤外線を照射する NIRS などがあります。fMRI は電気信号ってわけでも無さそうです。NIRS は赤外線で脳血流を捉えるのですが、頭皮の血流をいっぱい拾ってしまうので大変です。
[5]http://biorxiv.org/content/early/2017/03/29/121764

貴方の環境でどちらが動いているか (両方のこともある) 確認して設定してください。
- 実装:プログラミングのことです。プログラムを書くことです。

本書で「インストールにはこうします」とか言ってコマンドを示した場合は
文脈上特に何もない場合、ターミナルでそれを叩いてくださいという意味です。
python の文脈になったら python です。この辺りは見慣れれば判別できます。

それぞれの研究に必要な環境・特色のまとめ

本書ではまず環境を構築しますので、色々インストールが必要です。
必要物品についてまとめると下記です。

- 脳波センサーレベル研究
  python(本書では anaconda 使用)、MNE/python
  安価で普及していますが、まだ多くの謎が眠っている分野です。
  脳の深部の信号に強いですが、脳脊髄液や頭蓋骨を伝わって行くうちに
  信号が拡散してしまうため、空間分解能が低いです。
- MEG センサーレベル研究
  python(本書では anaconda 使用)、MNE/python
  ノイズに弱く、脳の深部に弱く、莫大な資金が必要な希少な機器です。
  それさえクリアできれば処理の重い脳波みたいなものです。
  ※ノートでは厳しいです。
- MRI 研究
  freesurfer、mricrogl(mricron)
  莫大な資金が必要ですが、それなりに普及しています。
  ネタが尽きようとも、新たな理論を持ち出してくる根性の分野です。
  ※ノートでは無理です。
  ※グラフ理論で解析する場合は python 必要
- MEG/EEG+MRI ソースレベル研究
  MEG と MRI を組み合わせた応用編となるため、紹介したものの全てが必要です。
  本書の本題です。MRI の空間分解能と脳磁図の時間分解能を備えた
  まさに †最強の解析†...のはずなんですが、どうなんでしょうね？
  ※膨大な計算量が必要なため、でかいコンピュータが必要です。

MNE/python とは

脳磁図を解析するための python[6]用 numpy,scipy ベースのパッケージです。
自由度が非常に高いです。(引き換えに難易度が高いです。)
wavelet 変換、コネクティビティ、その他あらゆる事が出来ます。
出来るのですが…使いこなすためには生理学、数学、工学の知識が必要です。
ちなみに元来脳磁図用なのですが、脳波を解析することも出来ます。
C 言語で実装された MNE/C というのもありますが、古いバージョンと考えていいです。
最近は MNEpython に機能を移しています、移行がまだ完全ではないところがあるなら
必要かもしれません。両方共フリーウェアですが、MNE-C は登録が必要です。
開発は活発で、最近新バージョンは MNE/python 0.16.1 です。
freesurfer は 6.0 が、python は python3.7 が最新です。
導入と紹介を書いていこうと思います。
最近 MNE がアップデートされて、python3 シリーズが使えるようになりました！
python は python3.6 を使っていくことになります。

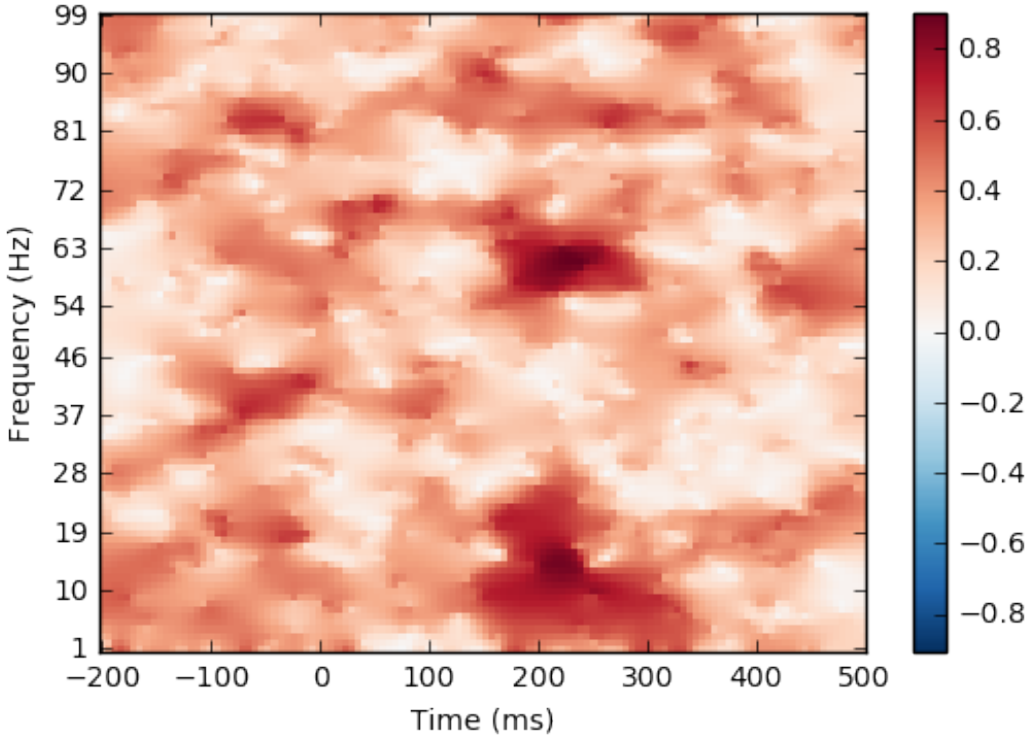


Figure 1: wavelet 変換の出力例

[6]コンピュータ言語の一つ。速度を犠牲にして、読み書きやすさを追求した言語。科学計算の世界では現時点では広く普及しています。MATLAB と似ていますが、python は無料でオブジェクト指向の汎用言語なので、応用範囲が Web サーバーとか機械の制御にまで及び、習得して損をすることはまずないでしょう。

freesurfer とは

頭部 MRI を解析する為のソフトです。自動で皮質の厚さやボリュームを測れるだけでなく最近は
fMRI でコネクティビティの算出が出来るようになるなど、かなり賢いです。特に厚さに強いです。
反面、激重な上にサイズが大きくターミナル使う必要があります。
Unix 系 OS じゃないと動きません。
その上、違う CPU 使ったら結果が変わる仕様があり、正しく扱わないとジャジャ馬と化します。
最近頭部 MRI 研究で勢力を伸ばしつつあり、最早スタンダードの一つだそうです。フリーウェアです。

それぞれのソフトの関係性～それ **office** に例えるとどうなの？ ～

いきなり MNEpython と言われても初心者にはよくわからないと言われます。
unix 系もコンピュータ言語も触ったことない人には例え話のほうが良いかもしれないので、
初心者のために、登場するソフトの名前を例え話で話してみます。
凄く乱暴な例えではあります。

MNE	役割	オフィスに例えると？
anaconda,pip,homebrew	ソフトをインストールするソフト	app store,google play, 人事部
spyder,jupyter	実際に色々書いたりするソフト	word,excel, 筆記用具
python	言語	日本語、命令書の書式
MNE	言語で動く命令セット	excel の関数、社内文書に従って動く部下
mricron	変換・ 表示用ソフト	画像変換ソフト, 通訳
freefurfer/freeview	MRI 画像処理ソフト	何でも一人でこなそうとする部下

MRI と脳磁図計と **ELEKTA** 製ソフトの準備

必要ですが　†億単位の金†　が必要なので本書では割愛します。
読者の中で個人的に買える人が居るなら買うと良いんじゃないかな。

脳波計

脳波研究は気軽で良いですね。これは脳磁図研究ほどお金はかかりません。
臨床応用されている脳波計は †千万単位の金†くらいしか要りません。
もしお金がなくても、病院にはそれなりにある機械です。

あと、最近は 10 万円くらいで買える脳波計もありますね。openBCI[7]とか。

コンピュータの準備

必要な性能はどこまでやるかにもよります。脳波解析なら普通の市販のノートでも十分です。
MRI やソースベース解析やるなら高性能なのがいいです。
また、高性能でも 24 時間計算し続けるような場合ノートではダメです。
その場合は...小さくてもデスクトップ機を使って下さい。
ノートは性能に限界があるだけでなく、排熱機構が弱いので
数日計算し続けると火災が発生する可能性があります。[8]
メモリいっぱい、CPU は多コアがいいです。ソースベース解析するなら nvidia の GPU とか
載ってるやつも良いかもしれないです[9]。どの程度のものが必要かは実験系によります。
メモリが大量に必要で、GPU より CPU 使う場面が多いです。
freesurfer は OS や CPU が変わったら結果が変わるという仕様がありますから
「このコンピュータを使う」と固定する必要があります。

OS の準備

OS は linux か MAC が普通と思います、windows ではどうなのでしょう？[10]
よく分かりませんが、MNEpython は動きます。
Unix 系コマンドラインツールは動きません。freesurfer は辛いです。
僕は新しめの debian 系 linux である UBUNTU[11]または MAC を使います。

linux でも新しめのメジャーな linux ディストリビューションを勧める理由は
CUDA 等の技術に対応していたり、ユーザーが難しいことを考えなくて良いことが多いからです。
debian 系を使う理由はパッケージ管理ソフトの apt が優秀でユーザーが多いことです。
MAC の場合は apt の代わりに homebrew(https://brew.sh/index_ja.html) を用いることになります。
以下、UBUNTU16.04LTS か macos10.12 を想定して書いていきます。
UBUNTU16.04LTS は下記サイトから無料でダウンロードできます。

https://www.ubuntulinux.jp/ubuntu

[7]ハードウェア、ソフトウェア共にオープンソースという夢広がりんぐな脳波計なのですが、使ったこと無いのでどの程度の性能なのかよく知りません。レビュー求む！
[8]あくまで本番環境ではの話です。例えばノートを通してサーバーやワークステーションを動かすとか、スクリプトの雛形を作るという用途であればソースベース解析でもノートは実用性に優れています。
[9]ブランドにこだわらずに探せば 20 コアとかのマシンが 40 万円も出せば買えます。グラボは nVidia 製にして下さい。AMD も頑張ってますが、まだまだ科学計算に弱いです。
[10]anaconda は os の垣根を越えているので、大丈夫なのでしょうけれど僕は試していません。
[11]UBUNTU は Canonical 社によって開発されているオープンソースの linux ディストリビューションであり、人気があります。debian というディストリビューションをベースに作られています。

僕自身は少しでも速く処理して欲しいので、誤差範囲かもですが linux では軽量デスクトップ環境に
変えています...ここは任意です。MAC を使う場合は homebrew というパッケージマネージャを
インストールすると色々楽になることがあります。
https://brew.sh/index_ja.html

開発環境の準備（脳波、脳磁図の場合。出来るだけリッチに。）

- freesurfer だけ使う人は開発環境は要りません。読み飛ばして下さい。
- 試すだけだとか、質素な開発環境でいい人も読み飛ばして下さい。

開発環境は MNE 使うなら必要です。詳しい人からは「docker[12] じゃダメなん？」という質問が
来そうですが、セットアップは自分でできなければ困ることもありましょう。
僕は anaconda[13] を使います。何故ならインストールが楽だからです。

https://www.continuum.io/downloads

このサイトからインストールプログラムをダウンロードします。
anaconda は 2 と 3 があり、それぞれ python2 と 3 に対応しています。
anaconda3 を入れるのがいいと思います。
anaconda に jupyter という repl[14] と spyder という IDE が付いてきます。
これらを使うのもまたいいと思います。

それぞれの使用感

開発環境は色々あるので軽く紹介します。

spyder

とても素直な挙動の IDE で ipython の補完機能も手伝って使いやすいです。
ただし、動作が重めなのと、企業のバックアップがなくなって今後が辛いです。

jupyter, ipython

repl というか、shell と言うかちょっと珍しい開発環境です。
これだけで完結することも出来なくはないレベルの開発環境です。

[12] 最近流行りの仮想化環境です。性能が高いのが特徴ですが、反面使いこなすのには力が必要です。
[13] 昔は source activate コマンドでしたが、このコマンドは anaconda 以外の仮想環境ツールと衝突してクラッシュするという不具合がありました。今後は conda activate コマンドを使うのがいいでしょう。
[14] 特定言語用の対話型インターフェイスのこと。

強みとしては
- web ベースなので遠隔操作可能
- cython や R といった他言語との連携が容易
- 対話的インターフェイスがメイン
僕は jupyter と他の IDE やテキストエディタを組み合わせるのがいいと思います。

atom, visual studio code

atom も vscode も現代的なテキストエディタです。
python 用ではありませんが、プラグインを入れて python の IDE として使うことが出来ます。
企業のバックアップがしっかりしているので、安心です。

vim

漢のエディタ

僕の今のおすすめは**?**

- jupyter
- anaconda
- visual studio code
  という組み合わせです。

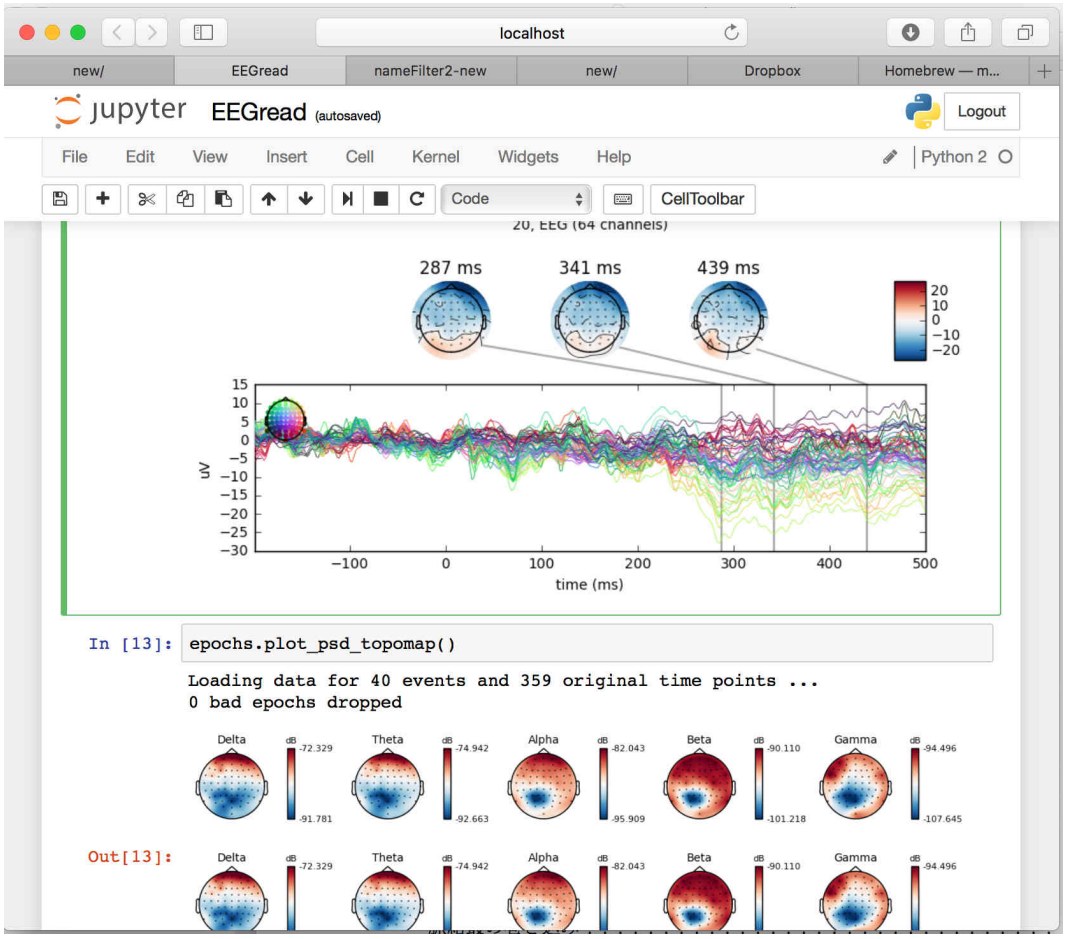


Figure 2: jupyter の画面。web ベースでインタラクティブにコーディング・共有できる。まぁ、触ってみればわかります。git 併用するのが良いかと思います。

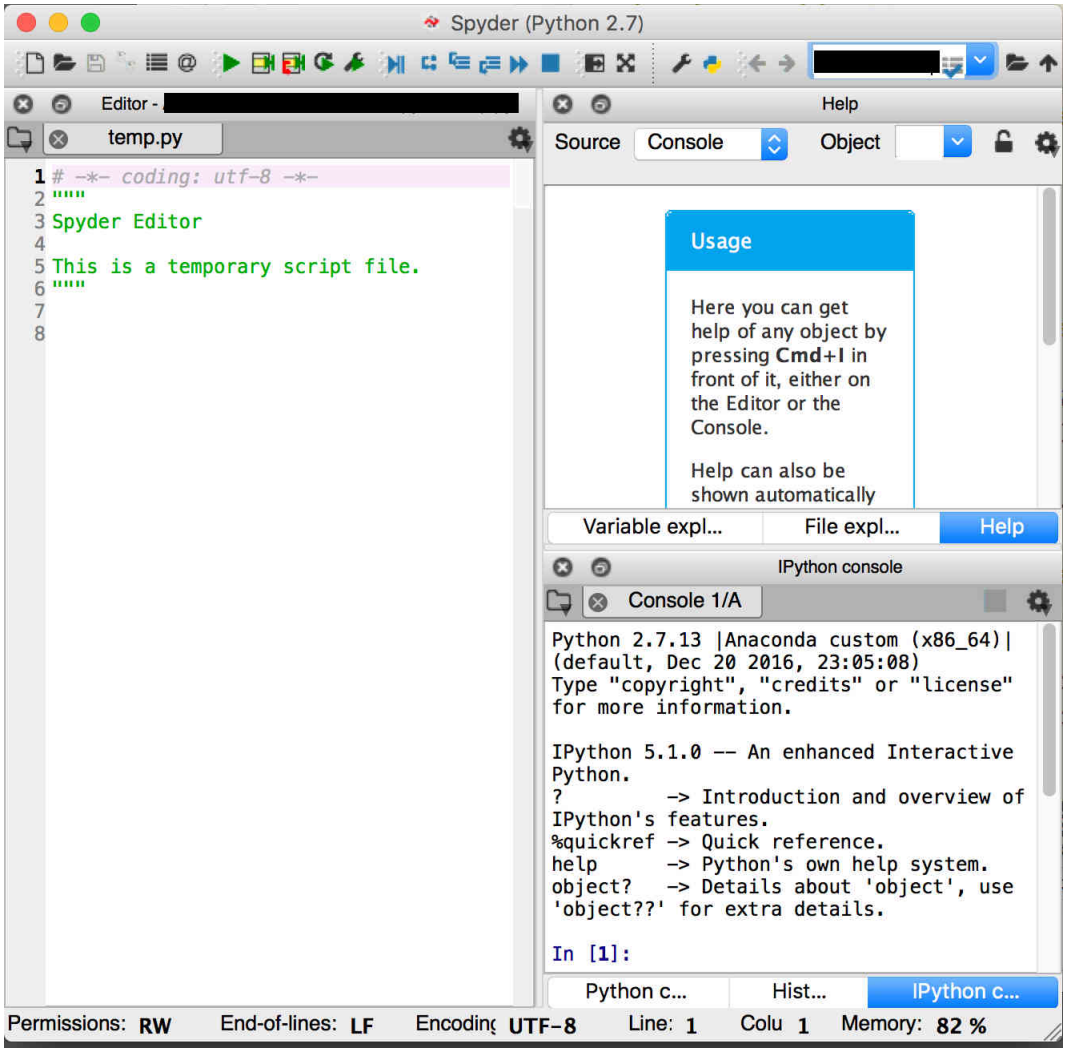


Figure 3: spyder の画面。ごく普通の素直な挙動の IDE。

MAC は anaconda のインストーラーをダウンロードしてクリックしていけばどうにかなります。
linux では anaconda はダウンロード後、ターミナルで以下のようにコマンドを叩いて
インストールします。bash です。ただの sh じゃインストールできません。

```
    bash Anaconda3-hoge-Linux-x86_64.sh
```

インストール先はホームフォルダでいいかとか、色々質問が出てきますが、
そのままホームフォルダにインストールするのが~~気持ち悪くても~~スムーズに行くかと思います。

jupyter の設定 **(**やり得**)**

素の jupyter でも強力ですが、折角なので拡張しておきましょう。ターミナルで下記を叩いてください。

```
    conda install -c conda-forge jupyter_contrib_nbextensions
    jupyter contrib nbextension install --user
    ipcluster nbextension enable --user
```

これで extension が使えるようになります。jupyter は機能が拡張できるので便利です。

jupyter での **plot**

jupyter は plot の方法を指定できます。
表示したい場合は、予め下記コードを jupyter 上に書いておいてください。

jupyter 上に直接出力したい時

```
    %matplotlib inline
```

python2 環境下で別ウィンドウで表示したい時
これはスクロールが必要な時に便利です。

```
    %matplotlib qt
```

python3 環境下で別ウィンドウで表示したい時
python3 と python2 は使う qt のバージョンが違うので
qt5 が必要になります。

```
    %matplotlib qt5
```

三次元画像をグリグリ動かしながら見たい時
(mayavi 使用)

```
    %gui qt
```

これについては後でまた詳しく記載します。

anaconda 仮想環境

mne は python3 に移行したのですが、freesurfer はまだ python2 です。
だから、その辺りを二刀流する必要があります。

anaconda は python の仮想環境[15]を作ることが出来ますのでそれを利用するのが楽です。

では、ipython からやっていきましょう。
ここでは、hoge という名前の python3.6 環境を jupyter 上に作ってみましょう。

```
    ipython kernel install --user
    conda create -n hoge python=3.6 anaconda
    conda activate hoge
    ipython kernel install --user
    conda info -e
```

1 行目から順に何をやっているか述べます。

1. 今の python の環境を jupyter に載せておく
2. conda で別バージョンの python 環境を作る
3. 切り替える
4. jupyter に組み込む
5. 確認

conda activate コマンドで python の環境を切り替えられます。
これで jupyter で色んな環境を切り替えられると思います。
ちなみに間違って環境を作った場合は以下のコマンドで消せます。

```
    conda remove -n python3 --all
```

R を jupyter で動かすために (非常に便利)

anaconda を使っているなら下記で R がインストールできます。

```
    conda install libiconv
    conda install -c r r-essentials
    conda install -c r rpy2
```

これにより R が動くようになり、貴方は少しだけ楽になります。
何故なら、実験結果を同じ環境で動く R に吸い込ませられるので、
「実験結果を入力するだけでワンクリックで統計解析結果まで出る」[16]ような
スクリプトが実現できるからです。具体的には jupyter 上で

```
    %load_ext rpy2.ipython
```

[15]仮想環境にも色々あります。例えば、pipenv などです。anaconda も同じ様な感じで使えます。
[16]同様に、matlab や C 等と連携をすることが簡単なのが jupyter の強みの一つと思います。

とした後

```
    %%R -i input -o output
    hogehoge
```

という風に記述すれば hogehoge が R として動きます。plot も出来るし、引数、返り値も
上述のとおり直感的です。さて、この-i ですが、通常の数字や一次元配列は普通に入りますが、
R ならデータフレームからやりたいものです。その場合は pandas というモジュールを使って
受け渡しをします。例えばこのような感じです。

```
    import pandas as pd
    data=pd.Dataframe([二次元配列])
```

```
    %%R -i data
    print(summary(data))
```

python と R をシームレスに使いこなすことがこれで出来るようになります。

CUDA

CUDA をご存知でしょうか？
GPU を科学計算に用いる方法の 1 つで、Nvidia 社が開発しているものです。
つまり、GPGPU です。
これは MNEpython でも使うことが出来るので、やってみましょう。
これのインストールも詰まるとそれなりに面倒です。
まずは、Nvidia のサイトからインストーラーをダウンロードします。

https://developer.nvidia.com/cuda-downloads

このサイトには色々な OS に対応した CUDA が置いてあります。
僕は ubuntu なら deb(network) をお勧めします。面倒臭さが低いです。
インストーラーをダウンロードしてダブルクリックするだけではダメで、
ダウンロードのリンクの下にある説明文を刮目して読みましょう。
こんな感じに書いてあります (バージョンによって違います)

```
    sudo dpkg -i cuda-repo-ubuntu1604_9.1.85-1_amd64.deb`
    sudo apt-key adv --fetch-keys http://hogehoge.pub
    sudo apt-get update`
    sudo apt-get install cuda`
```

こんな感じのがあるはずなので、実行して下さい。
そして、これが大事なのですが、bashrc にパスを通す必要があります。

これは CUDA のインストールガイドに書いてあります。
インストールガイドへのリンクは先程の説明の下に小さく書いてあります。
具体的には下記のような感じです。

```
    export PATH=/usr/local/cuda-9.1/bin${PATH:+:${PATH}}
    export LD_LIBRARY_PATH=/usr/local/cuda-9.1/lib64\
          ${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
```

これで CUDA へのリンクが貼れたはずです。
bash を再起動しましょう。
MNEpython の CUDA インストールのページに従ってコマンドを叩きます。
http://martinos.org/mne/stable/advanced_setup.html#advanced-setup

```
    sudo apt-get install nvidia-cuda-dev nvidia-modprobe
    git clone http://git.tiker.net/trees/pycuda.git
    cd pycuda
    ./configure.py --cuda-enable-gl
    git submodule update --init
    make -j 4
    python setup.py install
    cd ..
    git clone https://github.com/lebedov/scikit-cuda.git
    cd scikit-cuda
    python setup.py install
```

これでインストールできてたら成功です。
python で

```
    import mne
    mne.cuda.init_cuda()
```

としたら Enabling CUDA with 1.55 GB available memory…
的なメッセージが出たりします。
そして、一番確実なのは MNEpython に付属した
テストツールを回してみることです。

```
    pytest test_filter.py
```

このテストツールは MNEpython の中にあります。
場所的には anaconda の中の lib/python3/site-package/mne/tests
的な場所にあると思うのですが、環境によって違うかもです。

このテストがエラーを吐かなければ...おめでとうございます！
貴方は MNEpython を CUDA で回すことが出来ます！

バージョン管理 **git**

バージョン管理を知っているでしょうか？
貴方はスクリプトを書くことになるのですが、ちょっとしたミスでスクリプトは動かなくなります。
そんなリスクを軽減するために、貴方はスクリプトのコピーを取ります。
コピーを取り続けるうちに、貴方のコンピュータはスクリプトで埋め尽くされ、収集つかなくなります。
さらに、他の人がスクリプトを手直しする時、引き継ぎとかも大変です。
だから、貴方は git を使ってください。
git を知らない人は、とりあえず github desktop とか source tree をダウンロードして
体でそれを知ってください。詳しくは git でググってください。
こことか参考になります。
http://www.backlog.jp/git-guide/

git サーバー

git 単体でもいけるのですが、折角だから git のサーバーを導入してみましょう。
一番いいのは github のプライベートリポジトリを使うことなんですが有料です。
他にも bitbucket だとか gitlab とか色々あるのですが...僕自身は研究室のローカルなサーバーで
実現したかったんです~~趣味もある~~
なので、gitbucket を採用しました。gitbucket は github のクローンを目指して開発されたものです。

gitbucket 導入

gitbucket をググって gitbucket.war をダウンロードしてください。
で、java というか、jdk をインストールします。めんどいんで詳しくはググってください。
http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

このまま

```
    java -jar gitbucket.war
```

でも動くのですが、安定性に欠けるらしいので僕は PostgreSql を導入します。

とりあえず、いくつか unix ユーザーを作りましょう...。
そんで、データベース上に自分と gitbucket を登録します。

```
useradd postgres
passwd postgres
su postgres
createuser -d hoge
createdb hoge
createdb gitbucket
exit
```

上記で一応なんとかなると思うのですが、念のため確認を...
postgresql にログインして下記を叩けばちゃんとデータベースが出来たかを確認できます。

```
\du
```

.gitbucket/database.conf を下記のように書き直します。

```
db {
  url = "jdbc:postgresql://localhost/gitbucket"
  user = "test"
  password = "test"
}
```

で、

```
java -jar gitbucket.war
```

これで ip+:8080 にアクセスすれば gitbucket 動いてます。
(もちろん、これだけではセキュリティ面等、不十分です。
セキュリティ詰める自信がないならローカルだけで使いましょう。)

jupyter で作ったスクリプトのバージョン管理 **(**小技**)**

jupyter を僕は使いますが、jupyter のファイルは git しにくいです。
でも、何とかなります。

方法は 2 つありますが、僕は方法 2 が楽でいいと思っています。

方法 **1**

```
jupyter notebook --generate-config
```

このコマンドで jupyter のコンフィグファイルが作成されます。場所は/home/hoge/.jupyter です。
その上で、下記 URL に記載されている通りに書き加えます。
http://jupyter-notebook.readthedocs.io/en/latest/extending/savehooks.html
すると、jupyter で編集したファイルが python のスクリプトとしても保存されます。
あとは git[17]などで管理すればいいです。ただし、この方法は計算結果がファイル内に残りません。
しかも散らかります。
どちらかというと素直に py ファイルにしてダウンロードして git を使うほうが良いかもしれません。

方法 **2**

git を使いますが、git 側の設定だけでもどうにかなります。
まず、jq をインストールします。
.gitattibute に書きを書き加えます。
無ければ作ってください。

```
    *.ipynb diff=ipynb
```

そして、下記を.git/config に

```
    [diff "ipynb"]
    textconv=jq -r .cells[] |{source,cell_type}
    prompt = false
```

下記を.gitignore に

```
    .ipynb_checkpoints/
```

これで jupyter notebook のファイルを git で管理しやすくなります。

厳密な実験

研究するならもちろん厳密な実験が良いに決まっています。
しかし、厳密な実験というものは苦しみに満ちています。

厳密な視覚実験

まず厳密な視覚実験の苦しみを述べます。

[17] プログラミング用バージョン管理ソフト。敷居は高いが多機能。

苦しみ **1**　本当に見てるの？

視覚実験の場合、貴方は被検者さんに画像とかを見せることになります。
しかし、貴方の提示した画面を被験者さんは本当に見ているでしょうか？
見ているとして、本当に真正面から見ているんでしょうか？
視野が少しでもずれたら大きく結果が変わるんじゃないかと厳しい人は言います。[18]
本来は、瞼なんか全部切り取って、片目を潰して、
眼球運動の筋肉を全て切除して、視神経の一部を切除してやりたいくらいです。[19]

それを解決する方法としてこの 2 つが有力です。

- fixation
- eye tracker

fixation は「実験中はここを注目しておいてね！」という
小さな印です。問題点としては、その印自体が脳に影響するかも
ということと、本当にそれを見ているかも保証できない事ですが...
まぁ、流石に良いんじゃないでしょうか？

もう一つはカメラで眼球を監視して、動いていないときのデータだけ
使うというやり方です。これは値段が超高いです。
機械に対する需要が少なく、量産効果が無いからです。

苦しみ **2**　**MEG** におけるノイズ問題

MEG はノイズに弱いです。当然のことながら、MEG のシールドルームに
視覚刺激提示用の画面を置くと物凄いノイズが乗って酷くなります。
ノイズの出にくい画面というのがあるのですが、これがまたお高いです。

厳密な聴覚実験

聴覚実験の苦しみを述べます。

必要物品

- インサートイヤホン
- 音圧計

[18] 厳しすぎない？
[19] 動物実験でしてる人は居るらしいですが、人間にやると警察に捕まります。

理由

ヘッドホンではダメです。

聴覚実験の場合、音の大きさが重要になるからです。
音の大きさが不揃いだと実験になりませんが、
音は反射したり、干渉したりする性質があることはご存知でしょう。
そして、耳の形は人によって全然違います！
ヘッドホンを付けたときのズレも毎回変わります！

そういう意味でズレたりしないインサートイヤホンが選ばれるのですが、
イヤホンから出た音の大きさを、耳の中と同じ条件で
測ることって出来るでしょうか？

実は無理じゃないらしいんですが、かなり値の張る音圧測定器が
必要になってきます。

resting state 実験

ならば、何も刺激をせずにすればいいじゃないかとなりそうですが、
これもこれで色々考える所があります。

- 目を閉じているか: 脳波や脳磁図は目を開けてるかどうかで変わります
- MRI 由来の音: MRI はガンガン音がする上に狭い場所に押し込められます

そのへんは考えておいたほうが良いかなと思います。

~~苦しすぎて死にたくなった~~**~~?~~**

~~厳格すぎる実験で結果が出ても~~
~~そんな厳格な実験でしか出ない結果なんて臨床応用できなくね？~~

私からは以上です。

実験のディレイの測定

被験者に何かを見せたり聞かせたりしてその反応を脳波や脳磁図で拾ってくる実験をしたいとします。
脳はかなりの性能なので、30ms 後には反応が始まります。
さて...ここで困ったことが起こります。実はコンピュータから画面やスピーカへ信号を送る時、
一瞬で届いてくれないことがあります。理由は、現代的なコンピュータは様々なタスクを

同時進行でやっていたり、性能を確保するために様々な工夫をしますが、それが仇になるのです。
実験中に他の処理が割って入ってきたり、刺激に沢山の演算が必要だったりして
時間を取られたりすると信号が一瞬で届きません。
なので、どのくらいの時間で出来るかを測定しておく必要があります。

ここでは僕がどうやったかを書いておきます。
僕がやったのは被験者の目の前の画面に図が表示されるまでと、
「画面提示」の信号が脳磁図計に届くまでの差を調べることです。

必要物品

- オシロスコープ本体
  picoscope2000 というのを使いました。windows で動きます。かなり可愛いオシロスコープですが、
  入力 2 チャンネル、トリガー出力 1 チャンネル、μ s 単位の反応速度を持っています。
  25000 円くらいですが、家電量販店には売ってないです。
- センサー
  これは明るくなると抵抗が減るダイオードです。
  光センサが RPM22PB で 200 円くらいです。amazon で買ったあと品切れになりました。
  他の通販サイトにはあるようです。スペック上μ s 単位の反応速度のようです。
- 乾電池
  上記センサは 5V くらいの電圧では全然問題ないため、
  乾電池二本直列程度なら回路の途中に抵抗は要らないようでした。
  あまり逆の電流は流さない方がいいのかもしれません。
- ヒートシュリンクチューブ
  ドライヤーを当てると縮むチューブです。
  光センサーが壊れたら嫌なのでヒートシュリンクチューブで守りました。

あとはジャンパー線、鰐口クリップ付 BNC 同軸ケーブル、
半田と半田ごて、電池入れが必要でした。BNC 同軸ケーブル周りはやや入手難度が高かった印象です。

どのようにしたかというと、光センサーを電池につなげてオシロスコープに繋ぎます。
オシロスコープには 2 つの入力チャンネルがあるので、もう一方を刺激提示用コンピュータに繋ぎます。
さらに、ディレイ測定用コンピュータにも繋ぎます。
それで、刺激提示させた刺激を光センサーで捉え、差分を測定用コンピュータで受け取ります。

図にするとこうでしょうか...

もちろん、聴覚実験のときはまた違うと思います。
先輩と協力して聴覚実験のディレイを測ったときは、

直接イヤホンジャックからの信号をオシロスコープにブチ込んで測りました。
音の種類にもよりますが、聴覚実験の方が少し簡単かもです？

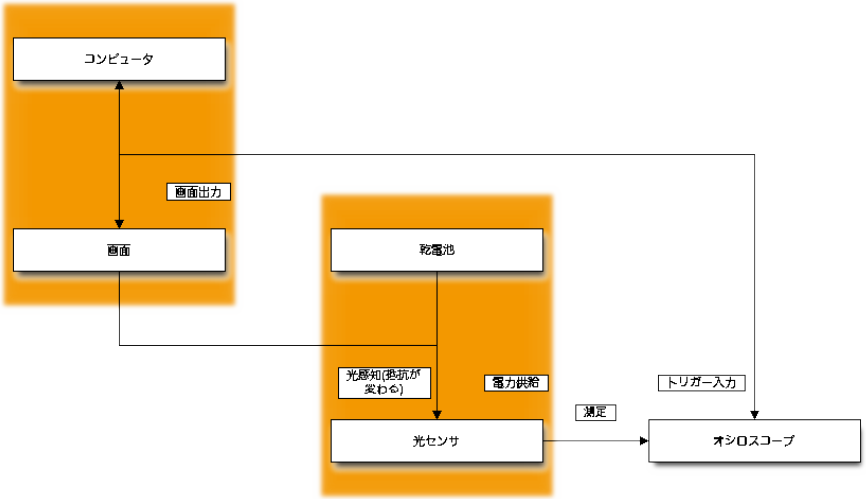


Figure 4: オシロスコープ図

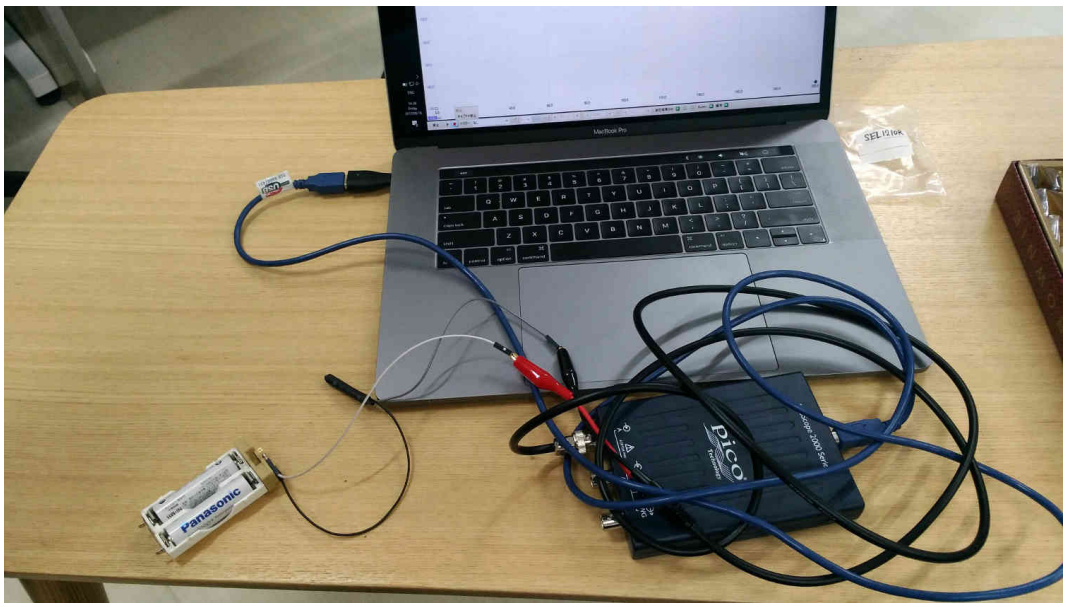


Figure 5: 光センサー＋オシロスコープセット。総額 3 万円くらいしました。半田ごてを使わないといけないので、やけどに注意する必要があります。手前の小さな箱がオシロスコープ、左にあるのが電池、オシロスコープと電池の間にあるのが光センサーを回路にはんだ付けしたものをヒートシュリンクチューブで保護したもの、奥にあるのはコンピュータです。

コンピュータクラスタ (計算量が膨大な場合)

これは無いならないでもいいですが、解析したい人数や観察したい場所が
多くなるなら役に立つかもしれないので記しておきます。
科学計算はときに膨大な演算が必要になります。
ただし、パソコン初心者とかにとっては敷居高いかもしれません。
以下、UBUNTU16.04 を想定して書きます。

やり方としてはまず nfs を使ってディスクを共有します
僕はホームディレクトリをそのまま共有しました。共有元ではこうです。

```
    apt install nfs-kernel-server
```

次に、/etc/exports を書き換えます。

```
    /home/hoge/ fuga(rw)
```

ディレクトリ、ip、オプションの順です。詳しくはググって下さい。

次に共有先のコンピュータをいじります。
まず、他のコンピュータのディスクを読むためのソフトをインストールします。

```
apt install nfs-common
```

そして、起動時にそのホームディレクトリを読みに行くように設定します。
具体的には/etc/fstab をいじります。home ディレクトリを他のところから読む設定です。
fuga は他のコンピュータの ip、hoge は

```
fuga:/home/hoge /home/piyo ext4 rw 0 0
```

あとは再起動すれば、毎回ホームディレクトリが共有されます。
必ず共有元を先に起動するようにしてください。（実はクラスタやめる時に一寸面倒くさいです）
通信方式は ssh を使うので ssh-agent を使えるようにします。まず、秘密鍵と公開鍵を作ります。[20]

```
ssh-keygen -t rsa
```

ssh-add コマンドで鍵を登録することが出来ます。

```
ssh-add hoge/fuga
```

下記のコマンドでちゃんと登録できたか確認できます。

```
ssh-add -l
```

下記のコマンドで SSH-agent を起動出来ます。

```
eval `ssh-agent`
```

また、ssh-agent は-A オプションつけてやるのがお勧めですが、これはまた別の話。
ssh はネットワークの基礎的な技術なので、本書ではここまでにしておきます。
詳しい事はインターネット上に分かりやすい記事がたくさんあると思いますので、検索してみて下さい。

あとは ipyparallel の機能を用いて並列計算します。ipyparallel をインストールしましょう。

```
conda install ipyparallel
```

pip でも良いみたいです。ずっと先でこれの使い方を解説します。

下記の公式サイトの説明の通りにしていけば良いです。
https://ipyparallel.readthedocs.io/en/latest/

とはいえ、英語の公式サイトを見ながらゴリゴリ設定する...というのもきついものがあります。
ネット上ではこの記事などが参考になります。
http://qiita.com/chokkan/items/750cc12fb19314636eb7

[20] いわゆる鍵認証方式というやり方です。セキュリティと利便性を両立できる良いやり方であり、
クラスタ作る時に限らず日常生活の中でもお勧めです。パスワード入れなくてもいいので楽ですし。

何故か僕がやると正攻法で動いてくれなかったので、下記のやり方をとりました。
http://qiita.com/uesseu/items/def93d1a0e829aec8e86

maxfilter のインストール (elekta のやつ)

maxfilter というフィルタが MEG 研究ではほぼ必須です。
これは外から飛んでくるノイズを数学的に除去するフィルタです。

これについては MNEpython にもあるのですが、elekta 社の maxfilter もあります。
一長一短ですが、何も考えずに使うなら elekta 社でしょうか…。
僕は以前は elekta のを使っていましたが、最近 MNE に移行しました。
MNE のは後で解説します。

以前、コンテナを使って導入したことがあるので導入方法を説明しておきます。
DANA というソフトと maxfilter というソフトを ELEKTA 社から貰う必要があります。
また、環境は Redhat5 または CentOs5 の 64bit 版を使うことになっています。
僕は docker[21] で centos5 のコンテナをダウンロードしてインストールを試みました。

```
    docker run -it --name centos5 -v ~:/home/hoge centos:5
```

これで centos5 がダウンロードされ、centos5 の端末に入ります。
ELEKTA 社製のソフトは 32bit,64bit のソフトが混在しています。
依存しているものとしては 32bit と 64bit の fortran、which コマンドです。
また、neuromag というユーザーを neuro というグループに入れる必要があります。

```
    yum install compat-libf2c-34.i386
    yum install compat-libf2c-34.x86_64
    yum install which
    useradd neuromag
    groupadd neuro
    usermod -a neuromag neuro
```

その上で、DANA と maxfilter のインストールスクリプトをそれぞれ動かします。

```
    sh install
```

僕は難しいこと考えるのが嫌だったので、インストールファイルを HDD にコピーして
スクリプトを動かしました。インストールできたら

```
    /neuro/bin/admin/license_info
```

[21] 最近流行りの仮想化環境です。性能が高いのが特徴ですが、反面使いこなすのには力が必要です。

として出力結果を ELEKTA に送り、ライセンスを取得します。
最後に脳磁図計のキャリブレーションファイルを入れる必要があります。
つまり「人が入っていない時の状態」を入れることになります。
/neuro/databases/sss/sss_cal.dat
/neuro/databases/ctc/ct_sparse.fif
この 2 つが必要です。ライセンスなどは日本法人の人に聞いたほうが良いです。
細則があります。以上で elekta の maxfilter のインストールは終わりです。

freesurfer のインストール

freesurfer をインストールしましょう。
下記の url からダウンロードできます。windows 版？ そんなものはない。

```
    https://surfer.nmr.mgh.harvard.edu/fswiki/DownloadAndInstall
```

で、ダウンロードしたファイルを

```
    tar -C /usr/local -xzvf hoge.tar.gz
```

Mac ならインストーラーもあります！
ね、簡単でしょう？ でも、まだ終わっていません。このままでは動きません。
設定をしないといけないのです。設定ファイルはホームディレクトリに
ある隠しファイルです。

テキストエディタは何でも良いですが、とにかく編集しましょう。
「隠しファイルなにそれ」な人は、unix 系の勉強をしましょう！
僕はとても優しいので教えますが、「.」で始まるファイル名は
隠しファイルになります。

freesurfer のダウンロードページに、Setup & Configuration という所があります。
四角で囲んである部分をコピーして、隠しファイルの.bash_profile に追記しましょう。

貴方が使っているシェルに応じてどれをコピペするかが決まるのですが、
大抵は bash と思います。

で、コピペし終わったら、保存して閉じるんですが、MRI の解析結果の
保存先 (subject_dir) を決めてあげたい場合は下記のようにします。

```
    export SUBJECTS_DIR=hoge
```

これは決めてあげたほうが良いです。何故なら、標準の subject_dir は
読み書きに管理者権限が必要だったりするからです。

最後にライセンスキーを入れましょう。
freesurfer の公式サイトに登録して、ライセンスキーをメールでもらい、
freesurfer のディレクトリに突っ込みます。

面倒いので、あとは freesurfer のサイトを読んで下さい。

MNE/python のインストール (脳波、脳磁図をする場合)

こちらは anaconda の存在下ではかなり簡単です。
mne 0.16 からは少しインストールの仕方が変わりました。
仮想環境で構築することになります。
このやり方のメリットは、いつでも同じ環境を整える事ができるので、
ソフトのバージョンが変わっても対応しやすいということです。
反面、毎回仮想環境に入らないといけないという小さなデメリットがあります。

公式サイトをみながら頑張りましょう。
http://martinos.org/mne/stable/install_mne_python.html

anaconda のバージョンは新しくしておきましょう。
新しくすればこのように確認できます。

```
    $ conda --version && python --version
    conda 4.4.10
    Python 3.6.4 :: Continuum Analytics, Inc.
```

要約すれば...
- curl[22] で environment.yml をダウンロードする
- conda env create -f environment.yml

これで mne の仮想環境が整いました。
下記のコマンドで mne の環境に入れます。

```
    conda activate mne
```

今後は mne を使うときは必ず上記のコマンドを打って下さい。[23]

これで完結...と言いたいところなのですが、残念ながら
mnepython と freesurfer のコマンドラインツール群にはまだ
python2 依存の部分があります。

[22] unix 界隈では大人気のダウンローダー
[23] 昔は source activate コマンドでしたが、このコマンドは anaconda 以外の仮想環境ツールと衝突してクラッシュするという不具合がありました。今後は conda activate コマンドを使うのがいいでしょう。

なので、python2 の環境も作りましょう。
ここ、公式に書いてない落とし穴です。

```
    conda create -n python2 python=2.7 anaconda
```

mne の環境に入るには

```
    conda activate mne
```

です。
さっきの python2 に入るのはもちろん

```
    conda activate python2
```

ちなみに、出るのは

```
    conda deactivate
```

mac なら下記も必要です。

```
    pip install --upgrade pyqt5>=5.10
```

MNE 環境を複数作りたい！

MNE の環境が複数欲しくなることもあると思います。
僕は欲しくなりましたし、今後 MNE がバージョンアップしていくたびに
古いのを残しながら音故知新する必要が出てくるはずです。
さっき色々やったなかで curl で environment.yml をダウンロードしたはずです。
この enviroment.yml は普通にテキストエディタで開けます。
内容はインストールすべきパッケージの列挙です。
一番上の所に

```
    name: mne
```

とあると思うので、単純にこいつを別の名前に変えてから
続きのコマンドを叩いていけばいいだけです。

jupyter kernel

jupyter を使うのであれば、上記の環境を jupyter に登録する必要があります。
まずは、仮想環境に入って下さい。

```
conda activate mne
```

では、登録しましょう。下記は「今いる環境を jupyter に登録する」やつです。

```
ipython kernel install --user --name hoge
```

もし、要らなくなったら

```
ipython kernelsupec uninstall hoge
```

ですね。

CUDA

CUDA[24](GPGPU) についてもそのサイトに記載があります。
CUDA は nvidia の GPU しか動きません。インストールについては
nvidia のサイトも参照して下さい。
ソースベースの解析をする場合はスピードが 6 倍くらいになります。

僕の環境では下記二行のコマンドを予め入れていないと動かないです。
.bash_profile や.bashrc に書き加えておけばいいでしょう。

```
export LD_PRELOAD='/usr/$LIB/libstdc++.so.6'
export DISPLAY=:0
```

さらに、jupyter 内で下記を実行しないといけません。

```
%gui qt
```

MNE/C のインストール (古い方法)

これは不要かもしれません。
下記サイトにメールアドレスを登録し、ダウンロードさせていただきましょう。
http://www.nmr.mgh.harvard.edu/martinos/userInfo/data/MNE_register/index.php
ダウンロードしたものについてはこのサイトの通りにすればインストールできます。
http://martinos.org/mne/stable/install_mne_c.html
僕はホームディレクトリに入れました。

[24] nVidia の GPU を使った高速な計算ができる開発環境

```
tar zxvf MNE-hogehoge
mv MNE-hogehoge MNE-C
cd MNE-C
export MNE_ROOT=/home/fuga/MNE-C
. $MNE_ROOT/bin/mne_setup_sh
```

これで MNE-C も動くようになるはずです。

コラム *1-SNS* の活用

```
皆さんはSNSはしていますか？SNSには様々な効能と副作用があります。
時に炎上する人だって居ます。廃人になる人も居ます。
しかし、最先端の科学にとって、SNSは大変有用なのです。
twitterでMEGやMRIの研究者をフォローしてみてください。
いい情報、最新の情報がピックアップされ、エキサイティングです。
僕は新着情報はtwitterで研究者、開発者、有名科学雑誌のアカウントを
フォローしてアンテナはってたこともありました。
(脳の疾患が増悪して今はしてない)
ちなみに、若いエンジニアはよくするらしいです。
```

freesurfer を使う (MRI)

ここからターミナルを使っていくことになります。下記は必要最低限の bash のコマンドです。
- cd :閲覧するフォルダへ移動する
- ls :今開いているフォルダの内容を確認する

まず、ターミナルを開き MRI の画像データがある場所まで移動します。
例えばフォルダの名前が DATA なら下記のようにします。

```
cd DATA
```

辿っていって、目的のファイルを見つけたならば、freesurfer で解析します。
例えばファイルの名前が hoge.nii なら下記です。

```
recon-all -i ./hoge.nii -subject (患者番号) -all
```

このコマンドを走らせると、完遂するのにおよそ丸 1 日かかります。
かかりすぎですね？ 下記で 4 コア並列できます。

```
recon-all -i ./hoge.nii -subject (患者番号) -all -parallel
```

やっている事は、頭蓋骨を取り除き、皮質の厚さやボリュームの測定、標準脳への置き換え、
皮質の機能別の色分け等、色々な事をしてます。詳しくは freesurfer のサイトを見て下さい。

recon-all 同時掛け (freesurfer)

recon-all はマルチスレッド処理をすることができます。しかし、効率はあまり良くないです。[25]
つまり、マルチコア機なら一例ずつマルチスレッドでかけるより、
同時多数症例をシングルスレッドで掛かける方が速く済みます。
ターミナルを沢山開いて処理させたりすると速いですが煩雑です。
なので、スクリプトを書いて自動化することをおすすめします。
MNEpython を使う人はプログラミングの習得は必須なので良いとして、
freesurfer しか使わない人でもスクリプトは書けるようになる方が便利です。
僕のおすすめは python、sh のいずれかです。

freesurfer の解析結果の表示

freeview というコマンドで解析済みの画像を表示できます。
上から解剖的に分けたデータを乗せることで部位別の表示ができます。

コマンドラインでは以下のようにすればいいですが、freeview と叩いてから
画面上からやっていってもいいと思います。
(多くの人は普通の画面上からしたほうが分かりやすいでしょう)

```
freeview -v <subj>/mri/orig.mgz \
hoge/mri/aparc+aseg.mgz:colormap=lut:opacity=0.4 \
```

orig.mgz というのはオリジナル画像。グレイスケールで読みこみましょう。
aparc+aseg.mgz は部位別データ。部位別データには色を付けて読み込みましょう。

画面左側に表示されているのは読み込んだ画像一覧です。
上に半透明の画像を重ねあわせていって上から見ています。
色々できますので、遊んで体で覚えるのが良いと思います。

[25] 理由は openMP というライブラリを使った並列化だからです。openMP はマルチスレッドを簡単に実装する優れたライブラリなのですが、メモリの位置が近い場合にスレッド同士がメモリ領域の取り合いをしてしまうため速度が頭落ちになるのです。

解析結果のまとめ

recon-all が終わった時点で、下記コマンドを入力しましょう。

```
    asegstats2table --subjects hoge1 hoge2 hoge3 ...\
      --segno hoge1 hoge2 hoge3 ... --tablefile hoge.csv
```

subject には subject(つまり解析済みデータの通し番号) を入れます。
segno には見たい位置を入力します。その位置というのは
$FREESURFER_HOME/FreeSurferColorLUT.txt に書かれていますので参照しましょう。

ちなみに、freesurfer6.0 の時点でこのコマンドは python2 に依存しています。
python3 を使っている人は python2 を何らかの形で併用しましょう。

これで hoge.csv というファイルが出力されます。
このファイルの中には既に脳の各部位のボリュームや皮質の厚さ等、
知りたい情報が詰まっています。しかし、このまま使うのは危険です。
freesurfer は時にエラーを起こしますので、クオリティチェックと修正が必要です。

画像解析の修正

個別な修正は freeview を用いてすることになります。
下記を参照して下さい。
http://freesurfer.net/fswiki/Tutorials

この freesurfer のサイトには、説明用のスライドと動画があり、とてもいいです。
以下、要約です。

- 脈絡叢や各種膜を灰白質と間違える
  - freeview で修正して recon-all(オプション付き)
- 白質の中で低吸収域を「脳の外側」と間違える
  - freeview で修正して recon-all(オプション付き)
- 白質の中で薄い部分を灰白質と間違える (controlpoint より小さい部分)
  - freeview で修正して recon-all(オプション付き)
- 頭蓋骨をくりぬく時に間違って小脳などを外してしまう
  - recon-all(オプション付き)
- 白質を freesurfer が少なく見すぎてしまう
  - freeview で controlpoints を付け加えて recon-all(オプション付き)

これは、問題にぶつかった時に上記サイトのスライドでも見ながら頑張るのが良いと思います。
皮髄境界などは freesurfer は苦手としているそうです。

SkullStrip のエラー

Freesurfer は脳だけを解析するために Skull Strip という作業をします。要するに、頭蓋骨を
外してしまうわけです。この時に watershedmethod[26] という方法を使うのですが、頭蓋骨を
切り取ろうとして脳まで取ったり逆に眼球や脈絡叢まで脳と間違うことがあるので修正が必要です。

脈絡叢の巻き込み

脈絡叢を巻き込んでいる場合は brainmask.mgz を編集します。
Brush value を 255、Eraser value を 1 にして Recon editing
shift キーを押しながらマウスをクリックして脈絡叢を消していきます。編集がおわったら

```
    recon-all  -s <subject>   -autorecon-pial
```

とします。

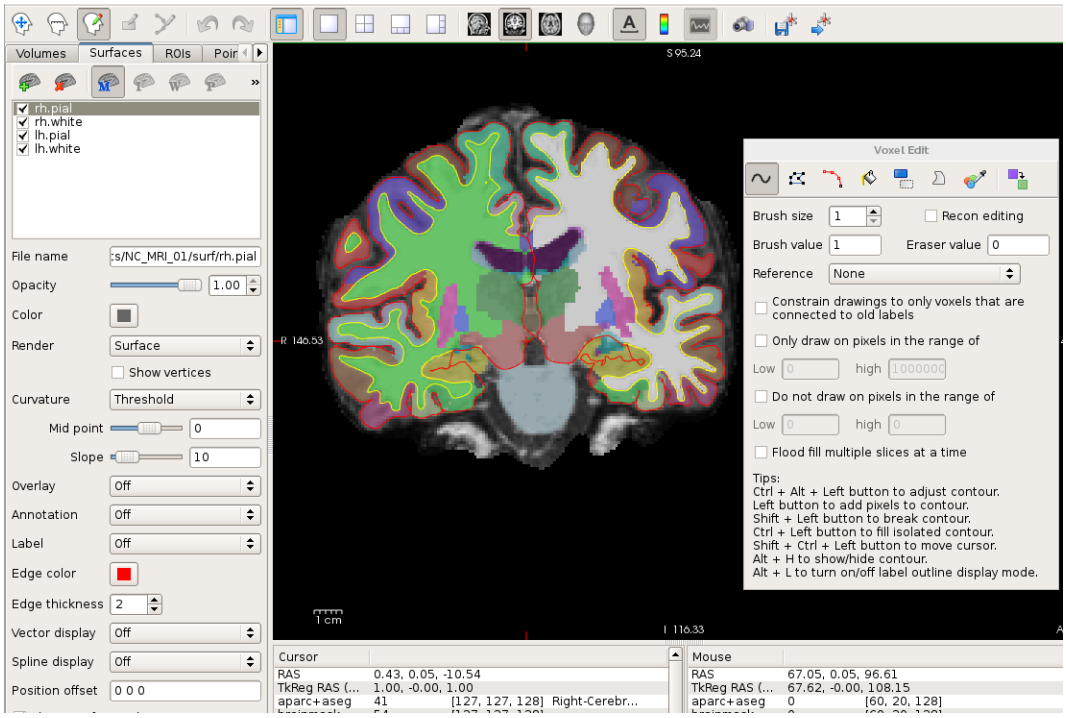


Figure 6: freeview による編集

[26]脳に水を流し込むシミレーションをすることで切っていいところと悪い所を分ける処理

眼球が白質と間違われた時

上記と同様にして、編集がおわったら

```
    recon-all   -s <subject>  -autorecon2-wm   -autorecon3
```

頭蓋骨と間違って脳をえぐっているとき

頭蓋骨と間違って脳実質まで取られた画像が得られた場合は

```
    recon-all  -skullstrip  -wsthresh 35  -clean-bm  -no-wsgcaatlas  -s <subj>
```

で調整します。この-wsthresh が watershedmethod の閾値です。
標準は 25 なのですが、ここではあまり削り過ぎないように 35 にしてます。

白質の内部に灰白質があると判定されるとき

時々、白質の中の低吸収域を灰白質とか脳溝と間違えることがあります。これも freeview で編集します。
wm.mgz を開いて色を付け、半透明にし、T1 強調画像に重ねます。
Brush value を 255、Eraser value を 1 にして
Recon editing をチェックして編集します。

```
    recon-all -autorecon2-wm -autorecon3 -subjid <hoge>
```

白質が厳しく判定されているとき

実は、freesurfer は brainmask.mgz で白質を全部 110 という色の濃さに統一します。
しかし、時々これに合わない脳があります。
そんな時は brainmask.mgz にコントロールポイントをつけて recon-all をします。

File -> New Point Set を選びます。
Control points を選んで OK して、選ばれるべきだった白質を
クリックしていきます。そして下記でいいそうです。

```
    recon-all   -s <subject>   -autorecon2-cp   -autorecon3
```

mricron/crogl(MRI を使う場合)

mricron が必要になることもあるので、入れましょう。UBUNTU なら

```
    sudo apt install mricron
```

MAC なら http://www.mccauslandcenter.sc.edu/crnl/mricron/から
インストーラーをダウンロードします。この mricron ファミリーの中にある dcm2nii というソフトが MRI の形式の変換に大変有用です。

さて、今はより新しいやつがあります。
mricrogl というやつです。(まだ詳しくない)
これは mricron では変換できないものを変換することが出来ます。
ここからダウンロード出来ます。
http://www.mccauslandcenter.sc.edu/mricrogl/

以上で freesurfer/MNE/python のインストールは終了しました。
これで jupyter 経由でゴリゴリ計算していくことができます。

mricron による MRI のファイルの変換

mricron は mri の画像の閲覧が出来るソフトですが、
この中に dcm2niigui というソフトがあるはずなので、そのソフトを起動します。

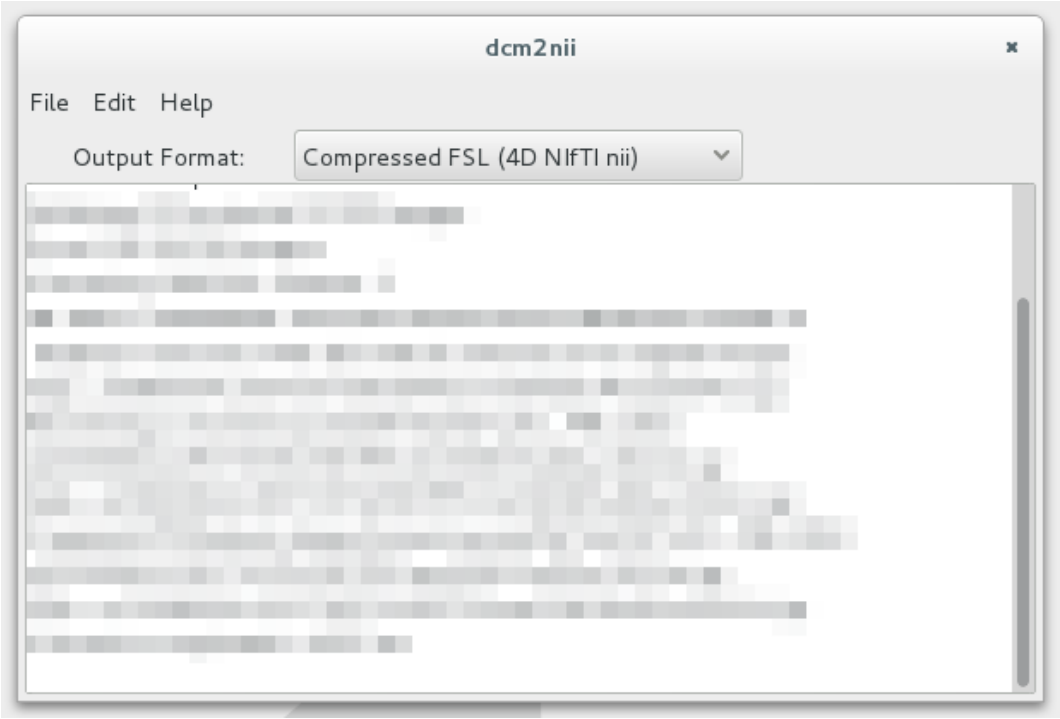


Figure 7: dcm2nii の画面

ちなみに、mricron 自体は mri 閲覧ソフトで、これもこれで有用です。

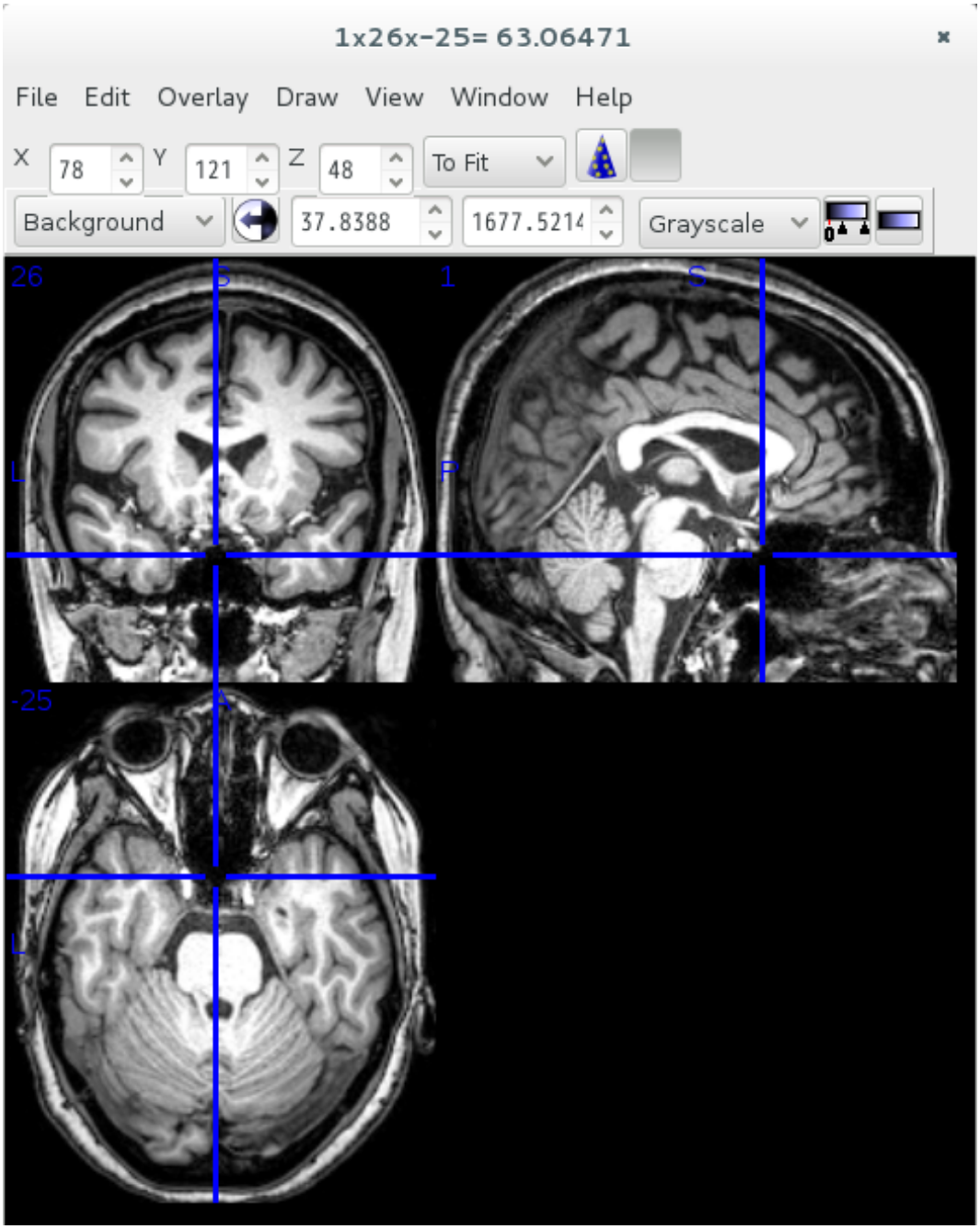


Figure 8: mricron による 3DMRI 画像の閲覧

例えば手元にある MRI の形式が dicom ならば、方言を吸収するために NIFTI 形式に直した方が僕の環境では安定していました。dcm2niigui の画面に dicom のフォルダをドラッグしてください。ファイルが出力されるはずです。

さて、出力されたファイルですが、3 つあるはずです

- hogehoge:単純に nifti に変換された画像
- ohogehoge:水平断で切り直された画像
- cohogehoge:脳以外の不要な場所を切り取った画像

となります。どれを使っても構わないと思います。

MNE を使う

いよいよ解析の準備に入ります。以下、MNE の公式サイトのチュートリアルのスクリプトなのですが...
かなり流暢な pythonista が書いていると思われます。
そのため、初心者が見るには敷居が高目です。
一回はそれをなぞろうと思いますが、その後は噛み砕いてシンプルに紹介します。
http://martinos.org/mne/stable/tutorials.html
http://martinos.org/mne/stable/auto_examples/index.html
http://martinos.org/mne/stable/python_reference.html

エディタの起動

jupyter を使うならターミナルで下記を叩いてください

```
    jupyter notebook
```

すると、ブラウザが起動し、画面が表示されるはずです。
起動しなければ、下記 URL にアクセスしてください。
http://localhost:8888
jupyter はブラウザで動かすものですが、別にネットに繋がるものじゃないです。
ちなみに、下記の様にして起動すると、lan 内で別の jupyter に接続できます。

```
    jupyter notebook --ip hoge
```

jupyter はターミナルで ctr-c を二回叩けば終了できます。
では、左上の new ボタンから python2 を起動しましょう。

spyder ならターミナルで

```
    spyder
```

です。

atom でも visual studio code でもいいと思います！
もちろん、vimmer[27]なら vim を使ってもいいと思います！
notepad.exe はお勧めしません！

MNEpython を使う前に学んでおくべきパッケージ

とりあえず、python と numpy[28]の基礎を学ばねばなりません。
これは最低限のことです。これが書けないのであれば mne/python は無理です。

他に学んでおくべきパッケージは

- matplotlib(作図用。seaborn で代用可能なこともある。)
- pandas(python 版 excel。必須ではないが、やりやすくなる。)

の 2 つでしょう。

pandas や matplotlib もググってください。qiita も結構良いです。
毎日何らかの課題に向けて python スクリプトを書きましょう。指が覚えます。
適当にググって良いサイトを見つければいいでしょう。
Python 入門から応用までの学習サイト
http://www.python-izm.com/

numpy で遊ぼう

詳しくはググって下さい。numpy は本書では語りつくせるわけがありません。以上です。

...ではあんまりなので、ほんのさわりだけ紹介しておきます。
下記サイトが参考になります。

Python の数値計算ライブラリ NumPy 入門
http://rest-term.com/archives/2999/

```
    import numpy as np
    a=np.array([5,6])
    b=np.array([7,8])
```

解説します。
1 行目は numpy を使うけれども長いから np と略して使うよ、という意味です。
二行目と三行目で、a と b に 5,6 と 7,8 を代入しました。ここから下記を入力します。

```
    print(a+b)
```

[27]vim 使いの事。僕は毎日 vim を起動しています。
[28]python 用行列計算ライブラリ。科学計算に広く用いられています。

結果

```
    [12,14]
```

このように計算できます。
ちなみに、numpy の配列と素の python の配列は違うものであり、素の python ならこうなります。

```
    a=[5,6]
    b=[7,8]
    print(a+b)
```

結果

```
    [5,6,7,8]
```

numpy と普通の list は list 関数や numpy.array 関数で相互に変換できます。
他に numpy.arange 等非常に有用です。

```
    import numpy as np
    np.arange(5,13,2)
```

結果

```
    array([5,7,9,11])
```

これは 5〜13 までの間、2 刻みの数列を作るという意味です。
そのほか、多くの機能があり MNEpython のベースとなっています。
出力結果が numpy 配列で出てくるので、MNE があるとはいえ使い方は覚える必要があります。

maxfilter をかけると数学的に脳の外のノイズを取ることができます。
これがないとかなりノイズだらけのデータになりますので、必須です。

解析を始める前の **warning!**

ここまでは単なる unix 系の知識だけで済んでいましたが、この辺りからは数学の知識、
python を流暢に書く技術、脳波脳磁図のデータ解析の常識等、色々必要です。
python を書くのは本気でやればすぐ出来ますが、
微分方程式だとか行列計算を全部理解して応用するのはかなり面倒くさいです。
同人誌で完璧に説明するのは無理なので、一寸だけしかしません。
また、データ解析の常識は進化が速いうえにその手の論文を
読めていないと正確なところは書けません。

僕はあまり強くないのでここからは不正確な部分が交じるでしょう。
本書は純粋な技術書であることに留意し、最新の知識を入れ続けましょう。

作図用おまじないセット

下記は jupyter のコマンド

```
    %matplotlib inline
    %gui qt
```

%matplotlib inline については、この設定なら jupyter 上に表示されます。
もし、別窓[29]を作りたいなら、inline を変えてください。
python3 の場合

```
    %matplotlib qt5
```

python2 の場合

```
    %matplotlib qt
```

となります。
下の%gui qt は mayavi による 3D 表示のためのものです。
mayavi が python3 で動くかどうかは僕はまだ確認してないです。

他に、こういうのもあります。

```
    import seaborn as sns
```

matplotlib の図を自動で可愛くしてくれるゆるふわパッケージです。

データの読み込みとフィルタリング・リサンプル (公式サイト版)

ついに　MNE を使い始めます。
まずは下記リンクを開けてください。
http://martinos.org/mne/stable/auto_tutorials/plot_artifacts_correction_filtering.html
ちょっと小難しい文法を使っているように見えます。
小難しい部分は初心者は混乱するだけなので無視してください。
難しいなら読み飛ばして、次に移ってください。簡単にまとめています。
公式サイトでは脳磁図前提としていますが、ここではついでに脳波の読み込みの解説もやります。

[29] 生の波形を見たいときなどにはそのほうが向いてる

是非脳波、脳磁図のファイルを手元において、読み込んだり
フィルタを掛けてみてください。(でないと、覚えられません)

ここでは
- データの読み込み
- パワースペクトル密度のプロット (以下 psd)[30]
- notch filter と low pass filter を使って要らない波を除去する
- サンプリングレートを下げて処理を軽くする
をしています。

はじめの cell(パラグラフの事)[31]で大事な関数は以下です。

- mne.io.read_raw_fif:脳磁図のデータを読み込みます。
  ここではつけていませんが、通常 preload=True をつけた方がいいです。
  preload をつけると、メモリ上に脳磁図データを読み込み、
  色々と処理ができるようになります。(付けないと処理できないです…)
  脳波を解析するなら下記公式サイトの **Reading raw data** セクションに
  各種形式に対応した読み込み関数が書いてありますから、読み替えてください。
  http://martinos.org/mne/stable/python_reference.html

読み込みの詳細は後で書きます。
- mne.read_selection:脳磁図の一部を取り出しています。
- mne.pick_types:データの中から欲しいデータだけ取り出します。
- plot.psd:psd プロットを行います。

基本は体で慣れるしかありませんが、jupyter や spyder や ipython の
コンソール上で tab キーを押せば中の関数が見えるので、入力自体は楽です。
例えば「raw.」と書いて tab を押せば、plot 関数だけでも色々出てきます。
だから、色々プロットして遊んでみてください。

次の cell では notch filter をかけています。

- notch_filter
  これは特定の周波数を削除するフィルタです。
  何故それをするかというと、送電線の周波数が影響するからです。
  西日本では 60Hz、東日本では 50Hz です。それを除去できます。[32]
  関数内に np.arange と書いてあるのは numpy の関数。

[30]各周波数ごとの波の強さをあらわしたもの。フーリエ変換の結果算出されるものの 1 つ。
[31]jupyter 独自の単位です。通常のプログラミングでは行ごと、関数ごとですが、jupyter では数行をひと塊りにしてプログラムを書きます。
スッキリ見やすい解説用コードを書けるのが jupyter の強みです。MNE 公式サイトでは jupyter を採用しているので、今後 jupyter を前提に話していきます。
[32]敢えて除去しない研究者もいます。notchfilter によって、除去する周波数の周辺の信頼性が失われるとのことです。

60 から 241 までの間で 60 ごとの等差数列を返すものです。
つまり、ここでは 60Hz を除去しています。

次に low pass filter をかけます。

- filter
  これは分かりやすいでしょう。ある周波数以上、以下の波を除去します。
  バンドパスフィルタと言います。
  ERP をする時は遅い周波数成分を除去するときは注意が必要です。
  その場合は 0.1Hz 未満でするのがいいのかもしれません。
  また、ICA でノイズ除去する時は 1Hz くらいでかけるといいです。

最後にサンプリングレートを変えています。
理由は今後処理がかなりのものになるので負担を軽くしたいからです。

- resample
  ここでは 100Hz まで下げていますが、最低見たい周波数の 2〜3 倍以上の周波数が必要です。
  また、周波数は元の周波数の約数である必要があります。

以上...MNE の公式サイトは一寸詳しいです。僕にはちょっとつらかったですね...。

データの読み込みと **filter,resample(**僕の解説**)**

公式サイトは python をバリバリ書ける上に生理学をきちんと理解できている人向けに感じます。
本書はあくまで初心者 (具体的には僕) 向けです。
先ずは大雑把に理解して体を動かすべきと思うので、以下は極めて乱暴な僕なりのまとめです。
大まかに理解した上で公式サイトに取り組めば良いのではないでしょうか？

極めて乱暴にまとめると、ノイズ取りの第一段階はこうです。

```
raw=mne.io.Raw('hoge',preload=True)     #読み込み
raw.filter(1,100)     #0.1~100だけ残すバンドパスフィルタ
raw.notch_filter([60,100])     #この場合、60と100Hzを消してる
raw.resample(sfreq=100)     #100Hzにリサンプルする
raw.save('fuga')
```

ちなみに、第 0 段階があります。
それは badchannnel の指定、interpolation、maxfilter 等ですが、
とりあえず読めなきゃ話にならないので。

- 1 行目で読み込みます。脳波と脳磁図では読み込み方が違うので、次セクションを参照。
  preload を True にしてください。そうしないとメモリ上に読み込んでくれません。[33]

[33] preload しないと各種処理が出来ないので、ほぼ必須です。何故 preload が標準で False なのかはよくわかりませんが、False も使いみちがあります。例えば生波形を素早く表示するだけならば preload は False が軽いです。

- 2 行目でバンドパスフィルタかけてます。1Hz 未満の波と、100Hz 以上の波を消しています。[34]
- 3 行目で送電線のハムノイズを取っています。[35]
- 4 行目でデータを間引いて処理を軽くしています。必ず元データの約数に設定し、
  wavelet 変換するならば wavelet 変換の最高の周波数の 2〜3 倍以上の周波数にしてください。
- 5 行目で掃除した結果を'fuga' という名前で保存しています。

あとは、plot を色々してみてください。

以下、本書ではこのような乱暴な解説をしてとりあえず手で覚えた後、
理屈を覚えていくスタイルにしていきます。

脳波読み込みの問題

脳波はすんなり読み込めたでしょうか？ そうでもないかもしれないですね。
なにしろ、脳磁図と違って脳波は沢山の形式があるのです。
例えば、ヘッダーファイルを要求する形式があったりもしますし、
モンタージュや眼球運動チャンネルの設定を追加せねばならぬ場合もあります。
このセクションは試行錯誤が要求されます。

さて...脳波は色々な企業が参入していますが、
脳波のファイルには以下の情報が入ってたり入ってなかったりです。

- 波形データ
- チャンネル名と空間データ
- 測定条件

このあたりは脳波計のユーザーが設定できる所もあったりするので、
脳波計の管理者に聞いたりするのが早いかもしれません。
また、モンタージュ (センサーの空間情報) を指定せねばならぬ事もあります。
その時は、下記のように書きます。

```
    raw=mne.io.read_raw_edf(filename,preload=True,
           montage='biosemi64',
           eog=['eye-l','eye-r'],exclude=['X1','X2','X3','X4'])
```

解説します...。

- filename
  これは問題ないですね

[34]バンドパスフィルタについては賛否両論と思います。何故なら、時間周波数解析をすると要らない周波数は消えちゃうので、意味が無いという考えがあるからです。詳しくは参考文献の analyzing neural ...を読んで下さい。個人的にはソースレベル解析の場合はした方がいいと思います。
[35]notch フィルタもバンドパスフィルタ同様賛否両論です。

- preload
  これも問題ないです。前のセクションを御覧ください。
- montage
  これは場合によっては問題ありですのであとで解説します。
- eog
  これは、眼球運動は何番目のチャンネルだよ、というやつですね。単純。
- exclude
  これは、このチャンネルはいらないよ、というやつです。
  余りチャンネルが有ることは日常茶飯事です。数字で指定もできます。

event 情報が読み込めない場合

EDF 形式は event 情報が文字列として入ってたりします。
そんな時は MNEpython では読めません。なので、別のソフトを使います。
使うソフトは pyedflib です。インストールしましょう。

```
pip install pyedflib
```

そして、コードを書くのですが、たいへん面倒いです。

```
import pyedflib
edf = pyedflib.EdfReader('hoge.edf')
annot = edf.read_annotation()
annot
```

これで annot(list 形式) にイベント情報が入ります。
しかし、annot の中を覗くと分かると思いますが、たまーにこの annot の中に
2 行で 1 つのイベントとかが入ってたりして、そいつを 1 つのイベントとして
書き直すスクリプトを書かないといけなかったりするので面倒臭いです。
頑張って書いて下さい。

そもそも少しも読み込めない場合

EDF 形式はメジャーなのですが、そんな中にも色々な形式があります。
EDF+C だとか EDF+D だとか。EDF+D は凄く読み込みにくいです。
pyedflib は EDF+D を読めません。しかし、万事休すではありません。
open source のいいソフトがあります。edfbrowser というソフトです。

https://www.teuniz.net/edfbrowser/

このサイトには windows 版が公開されていますね。
このソフトは tools メニューから EDF+D を EDF+C に変換する事ができます。

脳波のセンサーの位置が変則的な場合

さて…montage の話をします。montage は要するにセンサーの空間情報です。
この世には色々な脳波の取り付け方があります。「は？ 10-20 法しかねえよ！」
と言われそうですが、あるものは仕方ないのです。センサーの数の違いもありますし。

MNEpython では出来合いのモンタージュセットがあります。
10-20 法ならだいたいセンサーはこの辺だよ、というやつですね。
それは上記のように文字列で指定できます。大抵はこれで事足ります。
しかし、時々凄くニッチなセンサー配置の脳波計があったりします。
そういうのは MNEpython で対応できないこともしばしばです。
そんな時には文字列じゃなくてモンタージュ情報を別途読み込んで、
montage 変数に入れなきゃなりません。めんどいです。
詳しくは mne.io の解説記事をみて下さい。形式ごとの解説記事があります。

ちなみに下記のように raw を読み込んだ後で指定する事も可能です。

```
    raw.set_montage(mont)
```

脳波のセンサーからソースベース出来んの？

これについては大きな声では言いたくないのですが、出来ます。
ちょっと捻ったやり方が必要です。
ただし、脳波のセンサーを位置情報としたソースベース解析が
どの程度の精度を持っているかは…お察しください。

まず、上記の raw.set_montage(mont) に一つオプションを入れます。

```
    raw.set_montage(mont, set_dig=True)
```

これをすると、raw.info の中に raw.info[“dig”] という項目が入ります。
この dig の中に位置情報が入りますから、これを使って位置合わせが出来ます。
ただし、set_montage 関数から入れてきた montage 情報は、
どういうわけか meg の montage 情報に比べて縮尺がやたら大きかったりします。
僕がやったときはなんと千倍のサイズでした ()
この大きさになると無理感が出てくるので、ちっちゃくしちゃいました。

```
    for n in raw.info['dig']:
        n['r']=n['r']/1000
```

無理矢理感あふれるやり方ですね…。
こうすることにより、MEG と同じ様に mne coreg が出来るようになります。
mne coreg については後述します。ソースベース解析の所を御覧ください。

脳波のセンサーの名前が変則的な場合

もう一つかなり面倒くさい問題があります。
MNEpython はチャンネルの位置情報を自動で設定する時に
ファイルの中に記述されているチャンネルの名前を参照して
位置情報を当てはめていきます。これの何が困るのでしょうか？

脳波計が montage の’Fp1’ という風な普通の名前で出力してたら良いのですが、
例えば’EEG-Fp1’ という風な名前だったら名前を変えてあげないと読めないのです。
名前は大事なのです。

変える方法としては、raw.rename_channels 関数を使う方法があります。
mne.channels.read_montage の解説記事を開いてみて下さい。

まず、チャンネルの名前を表示しましょう。
いっぱいモンタージュ情報が書いてありますが、ここでは 10-20 法を見てみます。

```
    mont=mne.channels.read_montage('standard_1020')
    print(mont.ch_names)
    mont.plot()
```

凄くたくさんのチャンネル名と図が出てきましたね？
次に、読み込んだ脳波のチャンネルリストを見てみましょう。

```
    print(raw.ch_names)
```

チャンネル名が同じ名前になっているでしょうか？
なっていなかったら書き換えていかねばなりません。
書き換えるには、python の辞書形式を利用します。
2 つのチャンネル名をよーく見比べて変えていって下さい。
下記のような辞書を作っていきます。

```
    channel_list = {
        ”EEG Fp1-Ref”: ”Fp1”, ”EEG Fp2-Ref”: ”Fp2”,
```

```
        "EEG F3-Ref": "F3", "EEG F4-Ref": "F4",
        "EEG C3-Ref": "C3", "EEG C4-Ref": "C4",
        "EEG P3-Ref": "P3", "EEG P4-Ref": "P4",
        "EEG O1-Ref": "O1", "EEG O2-Ref": "O2",
        "EEG F7-Ref": "F7", "EEG F8-Ref": "F8",
        "EEG T3-Ref": "T3", "EEG T4-Ref": "T4",
        "EEG T5-Ref": "T5", "EEG T6-Ref": "T6",
        "EEG Fz-Ref": "Fz", "EEG Cz-Ref": "Cz",
        "EEG Pz-Ref": "Pz","EEG A1-Ref":"A1",
        "EEG A2-Ref":"A2"}
```

脳波の基準電極や眼球運動や心電図もこんな風に辞書にして下さい。
では、この辞書を使ってチャンネル名を変えましょう。

```
    raw.rename_channels(channel_list)
```

これで、脳波のチャンネルの名前を変え終わりました。
最後に、用意した montage と脳波をくっつけます。

```
    mont=mne.channels.read_montage('standard_1020')
    raw.set_montage(mont)
```

これで上手くいけば普通に MNEpython で解析できます。

基準電極

基準電極の設定は下記のような感じでできます。

```
    raw = mne.set_eeg_reference(
      raw, ref_channels=['LMASTOID'])[0]
```

が、普通脳は研究では全体の平均で設定することが多いようですから、下記のようなのが普通でしょうか。

```
    raw2=mne.set_eeg_reference(raw)[0]
```

ちなみに、初期設定では全体の平均を基準電極としていますから、この設定は実は不要です。

末尾の [0] はこの関数が list 形式で結果を出してくるから必要です。
詳細は
http://martinos.org/mne/stable/python_reference.html
を見て、各自読み替えてください。

このようなスクリプトははじめは面倒ですが、
一度書いてしまえば後は使いまわしたり自動化出来ます。

トリガーチャンネル

もう一つの鬼門がトリガーチャンネルです。つまり、刺激提示の時刻を記録したものです。
これは通常下記で表示できます。

```
    mne.find_events(raw)
```

raw の中の刺激提示チャンネルが読めない場合はどうにかしてテキスト形式とかで書き出してください。
そこからは…貴方はもちろん pythonista なので書けるはずです。
例えば、pandas を使って

```
    import pandas as pd
    shigeki=pd.read_csv('hoge.csv')
```

後はゴリゴリスクリプト書いてください。

僕もこのようなトリガーチャンネルについて苦労しました。
僕の場合はトリガーが脳波と同じように波形として記録されていたのです。脳波の波形は

```
    raw.get_data()
```

で出力することが出来ます。内容はチャンネルごとの numpy 形式の数列です。
サンプリング周波数を鑑みてがんばってください。
トリガーチャンネルは信号が入ったら波形が跳ね上がっていたので、
僕はその跳ね上がりを検知するようなスクリプトを書くことで解決しました。

bad channel の設定

苦行その 1 です。次にダメなチャンネルの設定や眼球運動の除去を行います。
http://martinos.org/mne/stable/auto_tutorials/plot_artifacts_correction_rejection.html
これには 2 つのやり方があります。

やり方 **1**

jupyter で%matplotlib qt としたあとで raw.plot() でデータを見ながらひたすら下記のように
badchannel を設定していってください。それだけです。

```
    raw.info['bads'] = ['MEG 2443']
```

badchannel は、例えば明らかに一個だけ滅茶苦茶な波形...
振幅が大きくて他のとぜんぜん違う動きしているとか、
物凄い周波数になっているとか、毛虫っぽいとか、そういうやつを選んでください。

やり方 **2**

raw.plot()
をした上で、画面上でポチポチクリックしていけば、raw に bad が
入っていくように出来ています。便利ですね！
もちろん、あとで保存しないとちゃんと残りません。

```
    raw.save('hoge.fif')
```

interpolation

選び終わったら、badchannel を補正します。
隣接するチャンネルを平均したようなやつで置き換えることになります。
それには下記を走らせるだけでいいです。

```
    raw.interpolate_bads()
```

後で badchannel を無視した ICA を掛けるとか、色々出来るわけです。

maxfilter

MNEpython に maxfilter があります。
MEG 使いの人はこれを使うのも一つの手です。
https://mne-tools.github.io/stable/generated/mne.preprocessing.maxwell_filter.html

さて、maxfilter には 2 つファイルが必要です。
この 2 つのファイルは、それぞれの施設によって違うものです。
一つは calibration 用の dat ファイル、一つは crosstalk 用の fif ファイルです。
これについては elekta の機械ならあるはずなので、そこから抜き出すといいでしょう。
ここについては僕は詳しくないので、周囲の賢者に聞いて下さい。

もう一つ、MNE の maxfilter には特徴があって、
badchannel を設定してあげないとうまく動きません。
因みに、elekta のは自動で badchannel を設定しちゃうそうです。

```
    from mne.preprocessing import maxwell_filter
    cal = 'hoge.dat'
    cross = 'fuga.fif'
    raw = maxwell_filter(raw,calibration=cal,
                         cross_talk=cross, st_duration=10)
```

この maxwell_filter 関数で行います。
calibration と cross_talk は見てのとおりと思いますが、
st_duration も大事なやつです。
MNEpython の標準の設定では st_duration は None なのですが、
実際は数値を設定しないと酷いことになります。
公式サイトには「俺たちの MEG はキレイだから None で良いんだ」と
ドヤ顔していましたが、町中の MEG だと地下鉄通るだけで酷いことになるので、
大草原の小さなラボとかでないなら設定してあげましょう。
元祖 elekta maxfilter ではここが 10 になっています。

この st_duration の数字は実は highpass filter の役割も果たします。
だから、注意が必要です。
1/st_duration 以下の周波数がカットされるので、
遅い周波数を見たい人は気をつけて下さい。
その他、いろいろな理由で st_duration は出来れば大きな値が良いそうですが、
計算コストが上がるという欠点がありますので、程々に。

ICA をかけよう

苦行その 2 です。ICA は日本語で言うと独立成分分析と言い、
何をするかというとノイズ取りです。前回やったノイズとは違うノイズを取ります。
例えば眼球運動や心電図が脳波、脳磁図に混じることがあるので、これを除去するのです。
これは ICA という方法 (独立成分分析) で波を幾つかの波に分け、
その上で眼球運動や心電図っぽい波を除去するフィルタを作ります。

順を追って内容を説明します。

```
    from mne.preprocessing import ICA
    from mne.preprocessing import create_eog_epochs, create_ecg_epochs
```

まずは、ICA のモジュールをインポートします。

```
    picks_meg = mne.pick_types(raw.info, meg=True, eeg=False, eog=False,
                               stim=False, exclude='bads')
```

次に、どのような波に ICA をかけるか選びます。基本、解析したい脳磁図 (脳波) に
ICA をかけるので、それを True にします。badchannel も弾きます。

```
    n_components = 25
    method = 'fastica'
    decim = 4
    random_state = 9
```

n_components は ICA が作る波の数です。
ICA で作る波の数は何個が良いのか僕にはよく分かりません。
多分現時点で決まりはないと思うので、ここではひとまず適当に 25 個にしています。

method は ica の方法です。
方法は三種類選べます。API 解説ページをご参照ください。

decim はどの程度詳しく ICA をかけるかの値です。
数字が大きくなるほど沢山かけますが、数字を入力しなければ最大限にかけます。

random_state は乱数発生器の番号指定です。
python では乱数テーブルを指定することが出来ます。
そうすると、再現可能な乱数 (厳密には乱数ではない) が生成できるようになります。
実は ICA は乱数を使うので、結果に再現性がないのですが、
この擬似乱数テーブルを用いることにより再現性を確保しつつ乱数っぽく出来るのです。
便利ですね！

```
    ica = ICA(n_components=n_components,
            method=method, random_state=random_state)
    ica.fit(raw, picks=picks_meg, decim=decim,reject = dict( grad=4000e-13))
```

ica のセットを作り、データに適用しています。
この時点ではまだ何も起こっていません。
先に%matplotlib qt と入力した上で下記を実行してください。

```
    ica.plot_sources(raw)
```

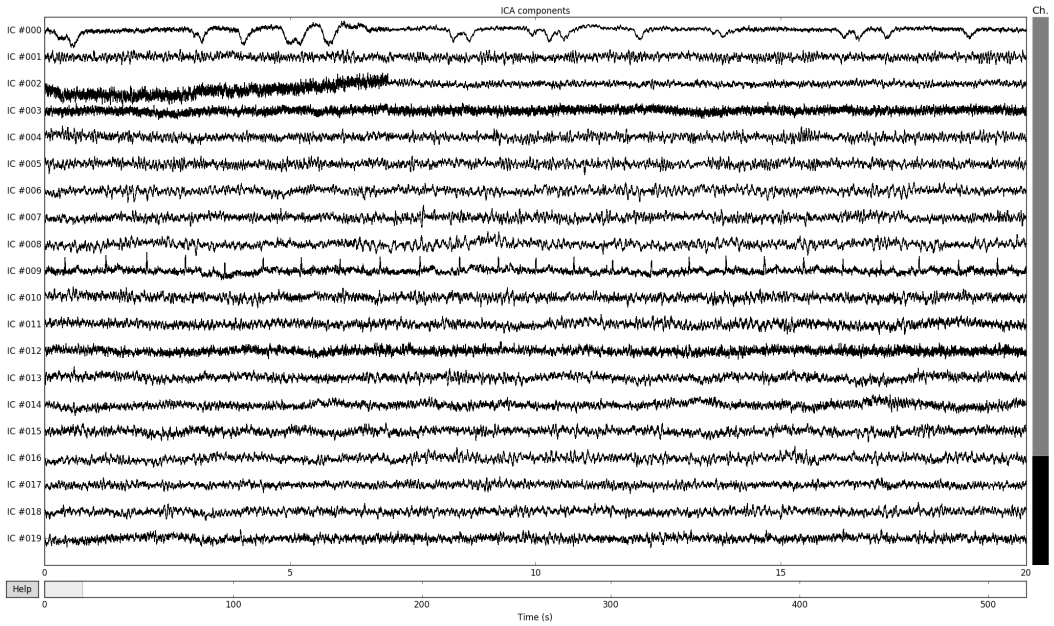


Figure 9: ica で分離した波。明らかに眼球運動や心電図が分離された図が出てくると思います。

個人的には生波形を見るのが明快で好きです。
ちなみに、これを凄く詳しく見るには下記のようになります。重いですが、これも結構良いです。

```
    ica.plot_properties(raw, picks=0)
```

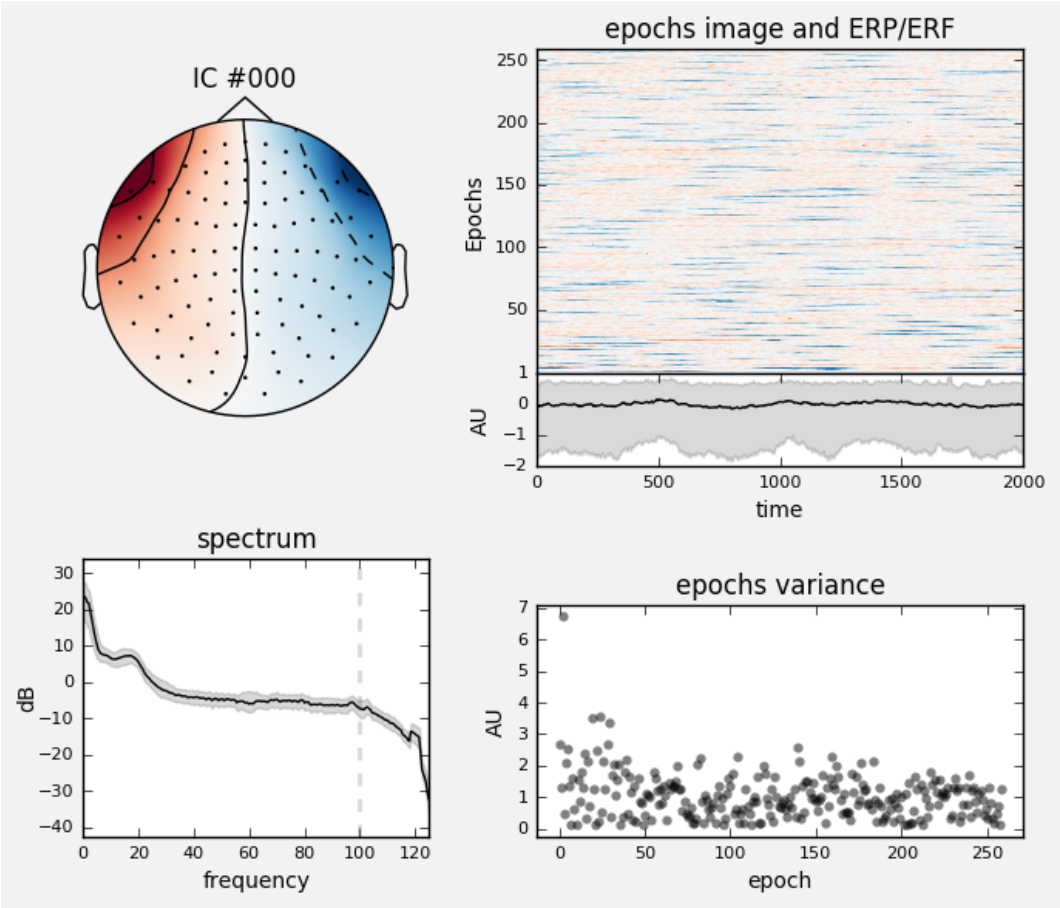


Figure 10: ica property の図。左上を御覧ください。これこそが典型的な眼球運動の topomap です。

最後に、0 番目と 10 番目の波を raw データから取り除きます。

```
    filtered_raw=ica.apply(raw,exclude=[0,10])
```

これで ica はかけ終わりです。
上記の出力結果や取り除いたチャンネル、random_state は保存しておきましょう。

Epoch と Evoked

なんのことやら分かりにくい単語ですが、波形解析には重要なものです。
epoch は元データをぶつ切りにしたものです。
元データ (raw) に「ここで刺激したよ！」という印を付けておいて、
後からその印が入っているところだけ切り出してきます。
evoked は切り出したものを加算平均したものです。

例えば元データ (raw) に刺激提示したタイミングを記録しているならば、
下記のコードでその一覧を取得できます。

```
    events=mne.find_events(raw)
```

この events 情報からほしいものを抜き出してきて、
epoch や evoked を作ります。
上記 events の内容は例えばこうなります。

```
    221 events found
    Events id: [1 2 4 7 8]
    Out[205]:
    array([[ 15628,      0,      2],
           [ 18053,      0,      2],
           [ 20666,      0,      4],
           [ 23131,      0,      1],
           [ 25597,      0,      8],
```

この場合刺激チャンネルには 1,2,4,8 という刺激が入っています。
このうち、刺激情報 1 を使って切り出したいときは下記です。

```
    epochs=mne.Epochs(raw,event_id=[1],events=events)
```

先程の events を使っています。
event_id は配列にしてください。ここは [1,2] とかも出来るのでしょう。
evoked を作るのはとても簡単で、下記のとおりです。

```
    evoked=epochs.average()
```

データの **plot**、主に **jupyter** 周り、そして **PySurfer**

是非自ら plot してみてください。
何をやっているのか理解が早まると思います。

```
    epochs.plot()
    evoked.plot()
```

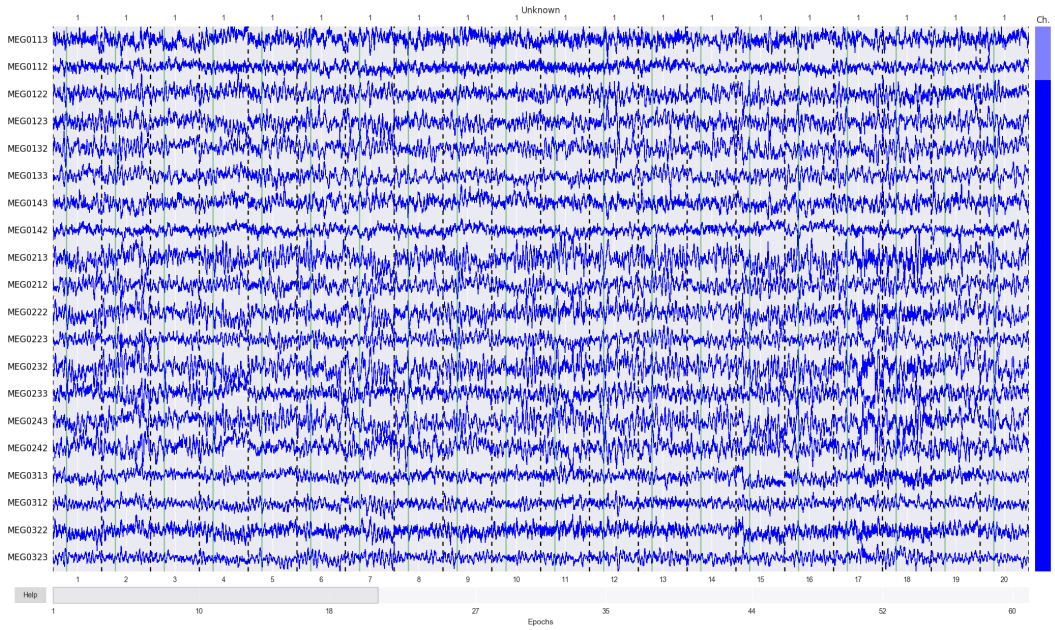


Figure 11: epochs の例

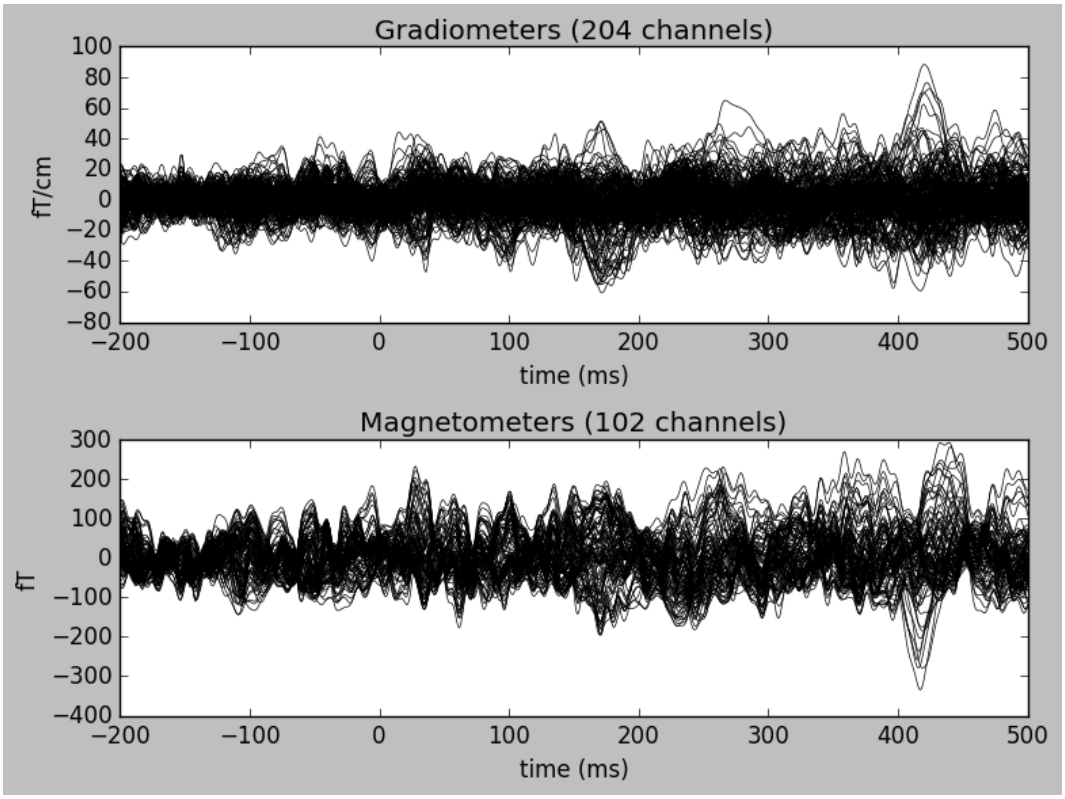


Figure 12: evoked の例

epochs や raw をプロットしたとき、どうなったでしょうか？
jupyter ではどのように表示するかを選ぶことが出来ます。

jupyter にそのまま表示したい場合は下記を先に jupyter 上で実行してください。

```
%matolotlib inline
```

別の window に表示したいときは下記のようにしてください。

```
%matplotlib qt
```

また、3D 画像を表示したい場合は

```
%gui qt
```

jupyter に表示するメリットは jupyter 自体を実験ノート風に使えること、
別ウィンドウに表示するメリットは raw や epoch 等大きなデータを表示する時に
スクロールさせることが出来ることです。

実は jupyter 上でスクロール出来る表示もあるのですが、重くてあまり良くないです。
詳しくは qiita で検索してください。親切な記事がいくらでもあります。

また、PySurfer については例えば下記のような感じです。
これは mac の場合ですが、ubuntu も同じ感じです。
subject や subjects_dir は freesurfer の設定で読み替えてください。
jupyter で下記の呪文を唱えましょう。

```
import surfer
%gui qt
```

そしてこうです。この場合ブロードマン 1 を赤く塗っています。

```
brain = surfer.Brain(subject, ”lh”, ”inflated”,
subjects_dir=subjects_dir)
brain.add_label(”BA1.thresh”, color=”red”)
```

注意すべき点として、拡張子や左右半球にかんしては add_label 関数では
省略して入力する必要があります。

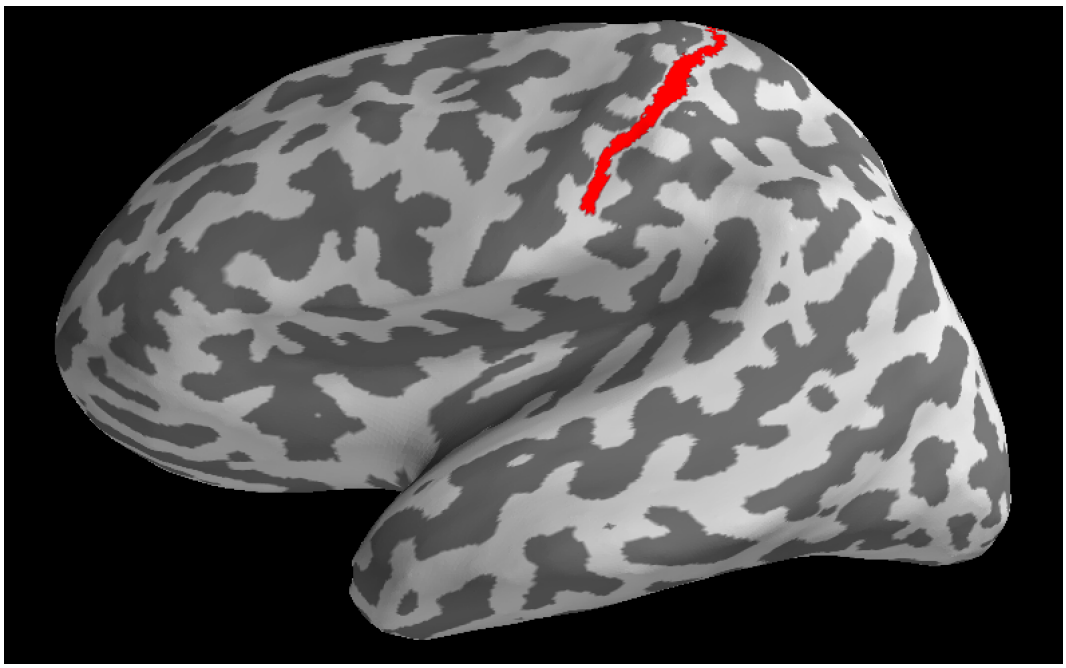


Figure 13: pysurfer で表示した freesurfer のラベルファイル

ちなみに、label ファイルはそれぞれの subject の中の label フォルダの中にあります。
この label についてはブロードマンの脳磁図ベースの古典的なものが多いですね。
新しい系は annot ファイルの中に多いです。

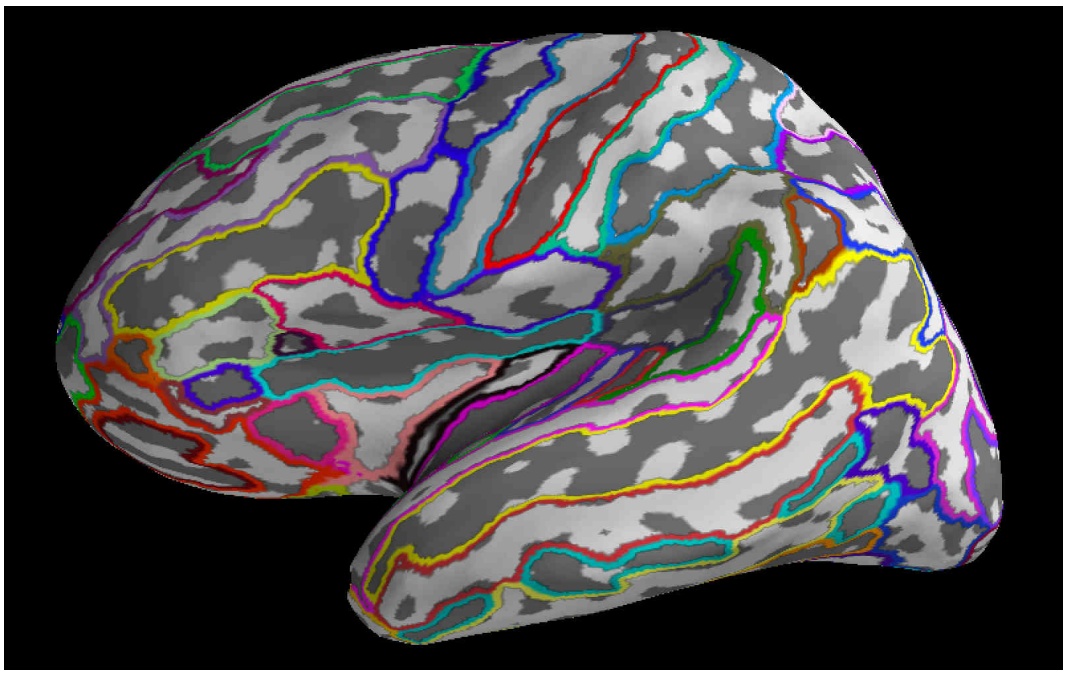


Figure 14: pysurfer で表示した freesurfer の annotation ファイル

```
    brain = surfer.Brain(subject, ”lh”, ”inflated”,
    subjects_dir=subjects_dir)
    brain.add_annotation('aparc.a2009s')
```

沢山表示されていますね。僕はちょっと気持ち悪いなぁと思いました。

多チャンネル抜き出し

もし、多チャンネルの evoked を平均したものを割り出したいなら
貴方は numpy を使うことになります。
ここでは脳波の evoked を例にしておきます。他のデータでも応用きます。
下記のチャンネルを選択したいとします。

```
    channel=['Fz','FCz','FC1','FC2','Cz','C1','C2','F1','F2']
```

python の配列では、中の項目を逆引きで探し出す.index() 関数があります。
加工した波形データは.data 変数の中に格納されています。その一番初めの情報が
チャンネル別なので、1 チャンネル...例えば’Fz’ なら下記のようにすれば割り出せます。

```
    evoked.data[evoked.info['ch_names'].index('Fz')]
```

この’Fz’ を for 文で書きかえていけば良いのです。

```
    data=[]
    for ch in channel:
      data.append(evoked.data[evoked.info['ch_names'].index(ch)])
```

ちなみに、下記のように書くのがより pythonic と思われます。
これはリスト内包表記と言って、pythonista が好んで使う方法です。

```
    data=[evoked.data[evoked.info['ch_names'].index(ch)] for ch in channel]
```

センサーレベル **wavelet** 変換

これは解析のゴールの一つと言えましょう。

そもそも **wavelet** 変換とは何なのか

特定の周波数の波の強さや位相を定量化するための計算方法です。
僕は数学が苦手なので、適当な説明です。フーリエ変換という言葉をご存知でしょうか？
これは波を sin 波の複合体として解釈することで波を一つの式として表す方法です。
ほぼ全ての波はフーリエ変換によって近似的に変換できるのです。
凄いですね！ しかし、これには欠点があります。不規則な波の変化に対応できないのです。
何故なら、sin 波は未来永劫減衰しない波だからです。
フーリエ変換において、波は未来永劫つづくのが前提なのです。
(擬似的に切り取ることは出来る)
そこで、減衰する wavelet という波を使って波を表す方法を使います。
そのため、減衰する波を単純な数式で表現する必要があります。
これを理解するためには高校数学を理解する必要があります。
詳しくは後半の「初心者のための波形解析」を御覧ください。

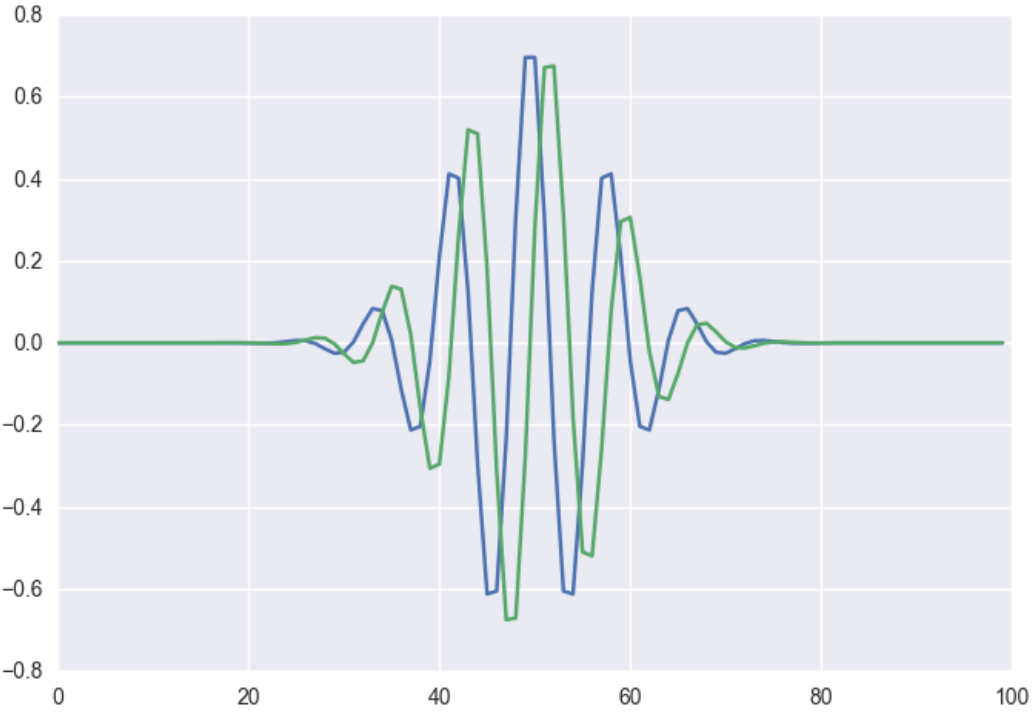


Figure 15: wavelet の例。これは morlet wavelet という種類。morlet はモルレと読む。青は実数部分、緑は虚数部分。

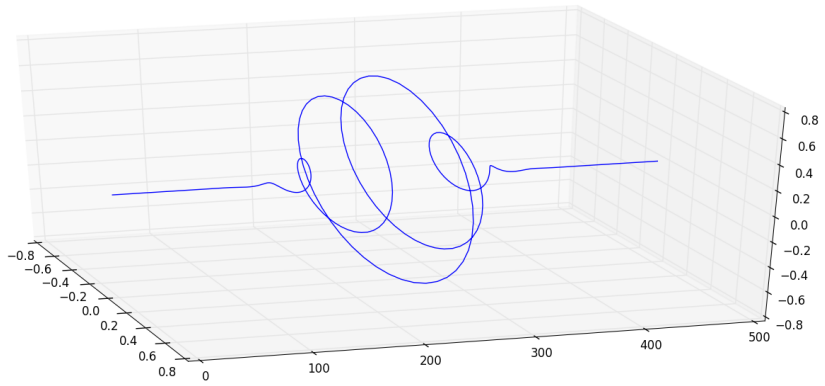


Figure 16: morlet wavelet の実数軸、虚数軸、角度軸による 3d plot。

wavelet 変換にまつわる臨床的な単語

wavelet 変換に登場する単語としては以下のものが挙げられます。

単語	型	内容	特徴
evoked power	波を加算平均した後に wavelet 変換、波の強さ	ノイズにやや強い	
induced power	wavelet した後に結果加算平均、波の強さ	ノイズに弱いが後期成分に強い	
phase locking factor	同一部位での位相同期性	ノイズにやや強い	

このなかで、phase locking factor は別名 inter-trial coherence(itc) といいます。
MNEpython では itc という言い方しています。[36]
それぞれ生理学的には違うものを見ているらしいです。
MNEpython では induced power と itc の計算方法が実装されています。[37]
これらの特徴の違いが何故生まれるかについても後半の
「初心者のための波形解析」を御覧ください。

では evoked power,induced power,phase locking factor について
解析を行いましょう。

wavelet 変換の実際

morlet のやり方は臨床研究的にメジャーなやり方と僕は思っています。
下記のスクリプトで実行できます。

```
    freqs=np.arange(30,100,1)
    n_cycles = 6
    evoked_power=mne.time_frequency.tfr_morlet(evoked,n_jobs=4,
      freqs=freqs,n_cycles=n_cycles, use_fft=True,
      return_itc=False, decim=1)
```

- freqs : どの周波数帯域について調べるか。
  上の例では 30Hz から 100Hz まで 1Hz 刻みに計算しています。
- n_cycles : 一つの wavelet に含まれる波のサイクル数。
  5~7 という値で固定する方法がよく用いられます。
  MNE ではこのサイクル数を可変にすることも出来ます。
- n_jobs : CPU のコアをいくつ使うか。重い処理なのです。
  ちなみに、n_jobs を大きくするよりも、n_jobs を 1 にして
  同時にたくさん走らせたほうが速いです…が、メモリは食います。

[36] ちなみに phase locking value という全然別のものがあります。これはコネクティビティ用語ですので分野が違います。あとで書きます。
[37] これは実質 itc と似たようなもの…という考え方もあります。

- use_fft : FFT による高速 wavelet 変換を行うかどうか。
  数学の話になるので、詳しい所は本書では扱いません。
  要するに速く計算するかどうかです、True でいいかと。
- decim : この値を大きくすると処理が軽くなりますが、
  出力結果がちょっと荒くなります。
- return_itc : これを True にすると phaselocking factor も
  算出してくれます。

この関数は evoked も epochs も引数として取ることが出来ます。
return_itc が True か False かでも大きく挙動が違います。
挙動の組み合わせについてですが、下記のとおりです。

return_itc	引数	返り値 1 つ目	返り値 2 つ目
False	evoked	evoked_power	なし
False	epochs	induced_power	なし
True	epochs	induced_power	phaselocking_factor

itc を計算したい時は返り値が 2 つになりますから、下記のです。

```
freqs=np.arange(30,100,1)
n_cycles = 6
induced_power,plf=mne.time_frequency.tfr_morlet(epochs,n_jobs=4,
  freqs=freqs,n_cycles=n_cycles, use_fft=True,
  return_itc=False, decim=1)
```

ここで一つ注意点があります。
wavelet 変換は基準になる波を実際の波に掛け算して行うのですが、
波の始まりと終わりのところだけは切れちゃうはずです。
そこは十分注意して下さい。
どの程度の wavelet の波の長さなのかについては、
頑張って計算して下さい。(余裕があれば書くかも)

データの集計について

データの集計についてですが...実は結構面倒くさいです。
MNE は個人個人のデータを解析するモジュールだからです。
貴方は個人個人のデータを MNE で解析した後、
そのデータを自分で集計する必要があります。numpy を使う必要性はここで出てきます。

MNE のオブジェクト (itc,power,evoked,epochs,raw 等) は
ユーザーがいじることが出来るようになっています。

中の実データはそれぞれのオブジェクトの中の data という変数か、
または get_data 関数で抽出してくることになります。
power なら power.data に、raw なら raw.get_data() に入っています。
こうして出してきた配列は numpy 形式の配列です。

ピックアップした情報は多次元配列ですから、内容は膨大です。直接見ても整理つきません。
そこで便利な変数が numpy にはあります。例えば evoked のデータを作ったならば

```
    evoked.data.shape
```

とすればデータの構造が確認できます。

データの構造としてはこんな感じのようです。括弧がついているのはオブジェクト内の関数です

形式	データ	1 次元目	2 次元目	3 次元目
raw	raw.get_data()	チャンネル	波形	
epochs	epochs.get_data()	チャンネル	波形	
evoked	evoked.data	チャンネル	波形	
itc	itc.data	チャンネル	周波数	波形
power	power.data	チャンネル	周波数	波形

揃っていませんね…。
(どうせ使うのは evoked 以下くらいなので大して困りません。)

それぞれのオブジェクトは
object.save(filename)
とすれば保存できます。
読み込みは多くの形式に対応する必要があってか一寸複雑です。

形式	読み込み関数	備考
raw	mne.io.Raw()	脳磁図の場合。脳波とかは公式サイト API 参照
epochs	mne.read_epochs()	
evoked	mne.read_evoked()	条件によって配列で返されることあり
itc	mne.time_frequency.read_tfrs()	条件によって配列で返されることあり

形式	読み込み関数	備考
power	mne.time_frequency.read_tfrs()	条件によって配列で返されることあり

例えば

```
    itc=mne.time_frequency.read_tfrs('/home/hoge/piyo')[0]
```

という感じで読み込みます。行の最後についている [0] は上記のごとく
条件によって配列で返されることがある関数だからです。この場合は行列として返されます。
そうじゃない関数の場合は [0] は不要です。実際に手を動かして練習すればわかると思います。

さて、実データのみではサンプリング周波数やチャンネルの名前が分からず
困ったことになりますが、mne/python ではこれらは
それぞれの object の中の info という python 辞書形式変数に入っています。
例えば print(itc.info) とか print(itc.info[“ch_names”]) とかで
読めたりしますから確認してみてください。僕はこの info を使ってチャンネルを抽出したりします。

ここまでの知識で、自分で numpy 形式で脳波脳磁図を扱えるようになります。

あとは下記のようにすれば良いと思います。

1. power なり itc なり波形なり、個人レベルで計算する
2. numpy 形式で 1 チャンネル抜き出したり数チャンネルの平均取ったりする
3. 個人個人で数字が出てくるので、それを保存する
4. R でその数字を統計解析する

例えば hoge チャンネルの fugaHz から piyoHz、
foo 番目から bar 番目 (秒 × サンプリング周波数) の反応までの
実データを抽出したいなら、

```
    itc.data[hoge,huga:piyo,foo:bar]
```

です。ちなみに、wavelet 変換時に decim の値を設定している場合は
(秒 × サンプリング周波数/wavelet 変換の decim の値) となります。
API ページで time_frequency.tfr_morlet() 関数をご参照ください。

2 は numpy の mean 等で実現します。
import numpy as np の後

```
    np.mean(itc.data[hoge,huga:piyo,foo:bar])
```

などとすれば良いと思います。

3 は python の基本構文通りなので解説しません。

3 と 4 は jupyter ならシームレスに扱うことが出来ます。これは超楽なので僕のオススメのやり方です。

R と padas の連携、特に ANOVA について

「R を jupyter で動かすために」である程度書きましたが、再掲します。
jupyter 上で

```
    %load_ext rpy2.ipython
```

とした後

```
    %%R -i input -o output
    hogehoge
```

という風に記述すれば hogehoge が R として動きます。
データの受け渡しには pandas を使うのが良いです。

```
    import pandas as pd
    data=pd.Dataframe([二次元配列])
```

```
    %%R -i data
    print(summary(data))
```

さて…これを応用します。
前述の np.mean() 関数で特定の時間、周波数など切り出した数値 (配列ではない) があります。
この数値を仮に num という変数に入れるとします。これに背景情報を付けます。データの背景情報に「疾患群、健常者群」「右脳、左脳」「刺激提示、プラセボ」という分類を作ったとしましょう。
…日本語は色々と面倒なので、下記のような分類に変えます。
[“disease”,“normal”],[“right”,“left”],[“stimuli”,“placebo”]

そして、上記で出したデータが
[“disease”,“left”,“stimuli”] という背景情報に合致するのであれば、
次のような配列を作ります。

```
    ['disease','left','stimuli',num]
```

この配列をさらに大きな配列に入れていきます。

```
    data=[['disease','left','stimuli',num]]
```

仮に、次のデータが
[“disease”,“left”,“placebo”,num2] なら、

```
    data.append(['disease','left','placebo',num2])
```

とすれば追加されます。[38]
さて...これで被験者の背景情報まで含まれた 2 次元配列が出来ました。
これを pandas を使って R の DataFrame とほぼ同等のものにします。

```
    import pandas as pd
    df=pd.DataFrame(data,
            columns=('group','hemisphere','test','value'))
```

これで、横軸に colums のラベルの付いたデータフレームが出来ます。
jupyter の R ではこれを読み込めます。具体的には下記のようにします。

```
    %%R -i df
    print(summary(aov(df$value~df$group*df$hemisphere*df$test,data=df)))
```

ここで python からいきなり R を書き始めます。
python の scipy での統計もいいのですが「なんで統計ソフト使わないん？ 舐めてるん？」
と reject を食らう可能性もありますから辞めましょう。
今回は多重比較です。多重 ANOVA を用います。aov が R の ANOVA 関数です。
これを summary 関数に読ませることで結果を簡単にまとめます。
さらに、print 文を使うことで画面上に表示します。
中の式は、データフレーム内の掛け算になっています。
ANOVA 詳しい人は知っていると思いますが、これは相互作用を算出するものです。
相互作用を計算しない場合は’+’ 演算子を使ってください。結果が算出されると思います。
あとは ANOVA の本でも読んで下さい。本書では割愛します。

Connectivity

Connectivity を脳波でやってみましょう。Connectivity は要するに、
脳のあちこちの繋がり具合を調べる指標です。MRI とかでよくされている手法ですね。

MNEpython では脳波と脳磁図でこれを計算することが出来ます。
実装されている計算方法を列挙してみます。
まずは、何はなくても計算しやすくする変換をせねば始まりません。
変換方法は下記の 3 つが提供されています。

[38]pythonista はリスト内包表記とか使うんでしょうが、ここは簡単のために append 使っています。というか、この程度の処理なら append で困りません。

- multitaper
- fourier
- morlet wavelet

このうち、multitaper と fourier は離散の計算方法、
morlet wavelet は連続 wavelet 変換です。(今までやってたのと同じ)

フーリエ変換や wavelet 変換をした上で、それぞれの値を比較するのです。
比較の方法は下記のとおりです。

- Coherence: Coherency の絶対値
- Coherency: 純粋に計算で出されたやつ
- ImaginaryCoherence: 同一ソースの影響を除いたもの
- Phase-Locking Value: 純粋に計算で出されたやつ
- Phase Lag Index: 同一ソースの影響を除いたもの
- Weighted Phase Lag Index: PhaseLagIndex に重みを付けたもの

…多すぎですね (´・ω・`)
どれが良いとかは…よくわかりません。色々やってみたり先行研究を見るのが良いかも？
一応 ImaginaryCoherence と PhaseLagIndex 系は同一ソースの影響が少ないので
性能がちょっといいかもしれません？
とりあえず、計算方法を書いておきます。まずは、epoch を作ります。作り方は前述のとおりです。
眼球運動や心電図のデータは要らない[39]ので、
pick_channel や drop_channel で要らないのを外していきます。

```
    epochs.pick_channels(['hoge'])
    epochs.drop_channels(['fuga'])
```

では、始めましょう。

```
    from mne.connectivity import spectral_connectivity as sc
    cons = sc(epochs, method='coh', indices=None,
              sfreq=500, mode='multitaper', fmin=35, fmax=45, fskip=0,
              faverage=False, tmin=0, tmax=0.5, mt_bandwidth=None,
              mt_adaptive=False, mt_low_bias=True,
              cwt_frequencies=None, cwt_n_cycles=7,
              block_size=1000, n_jobs=1)
```

…基本、我流の僕はソースコードが汚いんですが、今回はあまりにも
一行あたりが長すぎる気がします。オブジェクト指向的にはちょっと…。
自前でオブジェクト作ったほうが楽かもです。
やむを得ず圧縮のために sc と短縮しました…。では、解説いきます。

[39]大抵は、の話です。心臓の鼓動と脳波のコネクティビティの研究も一応あることは知ってます！

- method: そのまま method ですね。上記の通り。
- indices: どことどこの connectivity を見たいかです。
- sfreq: サンプリング周波数です。
- fmin,fmax: 見たい周波数帯域です
- fskip: どのくらい飛び飛びで解析するかです。
- faverage: 最終的に幾つかの周波数を平均した値を出すかどうかです。
- tmin,tmax: どこからどこまでの時間見るか
- cwt_frequencies: morlet wavelet の時の周波数 (numpy 形式の数列)
- cwt_n_cycles: morlet wavelet の波の数
- block_size,n_jobs: 一度にどのくらい計算するか

この関数は、中々詰~~め込み~~多機能な関数です。
なんと、上記の沢山の method を全部できます。出来るがゆえの大変さもあります。
この関数には 5 つの返り値と、実質 4 つのモードがあると考えるとやりやすいと思います。
それぞれについて解説します。

5 つの返り値

さっきの関数には 5 つの返り値があります。順に

- con: connectivity numpy 形式
  幾つかパターンあるのであとで書きます
- freqs: 周波数
  何故ここで周波数が出てきたか一瞬怪訝に思いそうですが、
  これは fourier と multitaper モードの時に力を発揮します。
  というのは、離散的な演算をするので、周波数は関数が勝手に設定するからです。
  morlet wavelet の場合は cwt_frewuencies で設定した物が出てきます。
- times: 解析に使った時刻のリスト
- n_epochs: 解析に使った epoch の数
- n_tapers: multitaper の時だけ null として出力
  DPSS という値が格納されます。

上記のコードでは cons という変数にタプルを入れているので、
cons[0] が con、cons[2] が times です。

この中で大事なのは con です。何故なら、これが結果だからです。
con の中で一番大事なのは中に入っている三角行列です。
三角行列というのは、行列の対角より上か下が全部 0 で出来ている行列です。

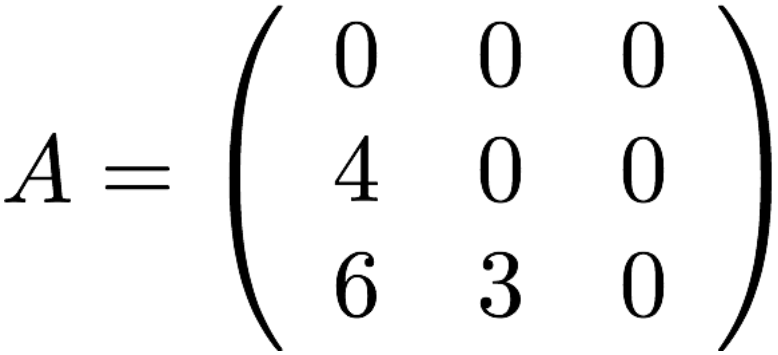


Figure 17: 三角行列の例

fourier/multitaper モード

fourier や multitaper は時間軸がないです。その辺が morlet wavelet と違うところです。
con の内容は [チャンネル数 X チャンネル数 X 周波数] という三次元配列になります。
この内、チャンネル数 X チャンネル数 の部分が三角行列になります。
周波数は、関数が勝手に「これがいいよ」と言って抜き出してきた
離散的な周波数になります。

ここで、幾つかの周波数について個別にやりたいなら話は違うのですが、
加算平均したいなら下記のコードで十分です。

```
    conmat=np.mean(con,axis=(2))
```

これで、conmat に三角行列が入りました。

wavelet モード

morlet wavelet は乱暴に言うと fourier に時間軸を与える拡張版です。
[チャンネル数 X チャンネル数 X 周波数 X 時間] という 4 次元になります。
この場合は下記のコードで三角行列を作りましょう。

```
    conmat=np.mean(con,axis=(2,3))
```

三角行列が出来ました。

plot

さっきの 2 つは三角行列を作るモードでした。
三角行列がある場合は綺麗な plot が出来ます。下記のとおりです。

```
    mne.viz.plot_connectivity_circle(conmat, epochs.ch_names)
```

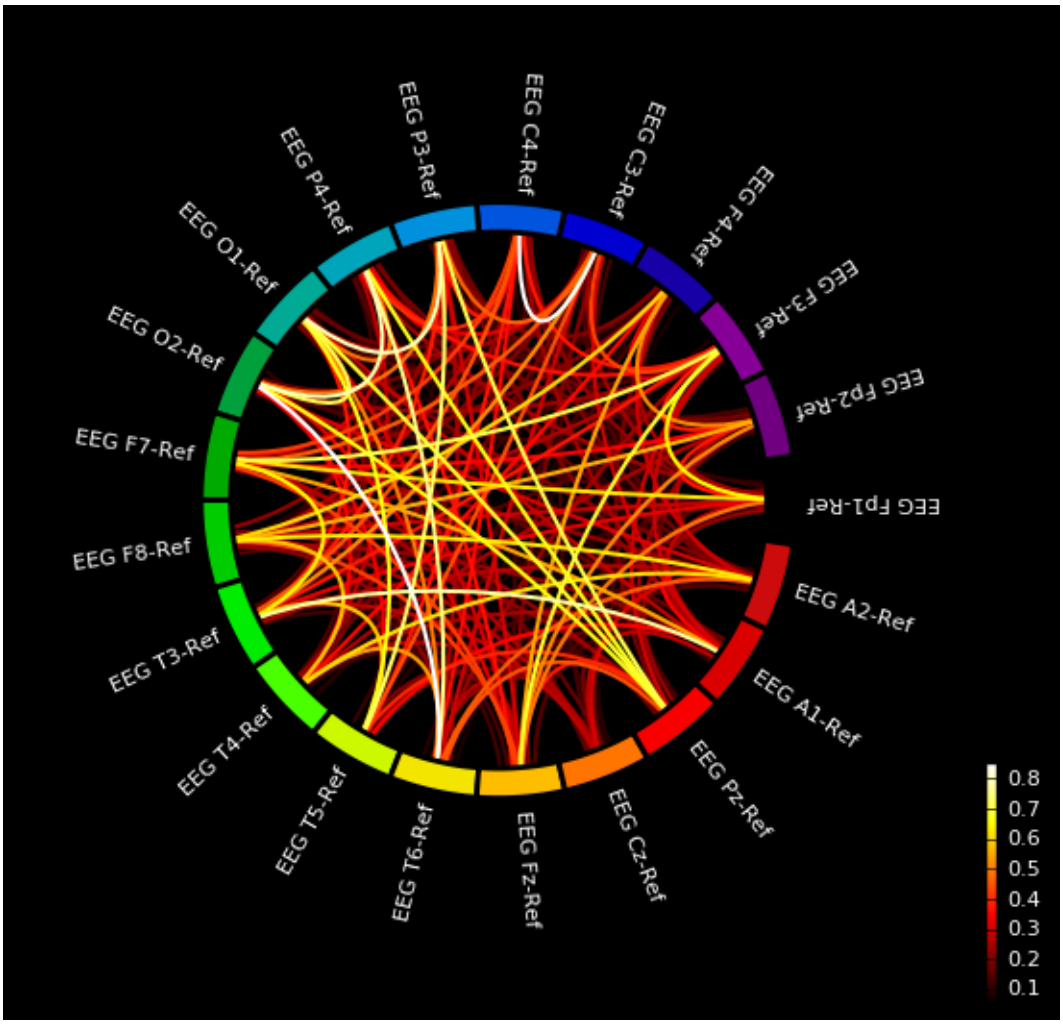


Figure 18: 僕の脳波のコネクティビティの図。花火みたいで綺麗なので好きです。

indices モード

三角行列関連は、要するに全部入りな感じの計算でした。
indices というところに引数を入れると、特定の connectivity だけ計算してくれます。

```
    indices = (np.array([0, 0, 0]),
               np.array([2, 3, 4]))
```

このように numpy 配列を作ります。1 列目は何番目のチャンネルとそれぞれを見比べたいか。
2 列目はそれぞれのチャンネルです。ここでは
0 → 2 , 0 → 3 , 0 → 4 番目のチャンネルを比べています。

で、この indices を引数として入れるとどうなるかというと、
fourier/multitaper モードなら [見比べたチャンネルの数 X 周波数] となります。
morlet wavelet モードなら [見比べたチャンネルの数 X 周波数 X 時間] となります。

ソースレベル **MEG** 解析

ついにソースレベルの解析を行います。これが MNE/python の真髄です。
難しいのです。頑張りましょう。

ソースレベル解析については冒頭の記述を見ていただくとして、
MRI と MEG をくっつけていきます。(MRI がない場合は標準脳を使える)
目標は「脳内の信号を算出するための式を作る」事です。
式さえできればなんとか計算できるわけです。
必要物品は以下の通り

- 脳の中の見たい場所リスト
- センサーの位置情報
- 脳波か脳磁図
- 皮膚や頭蓋骨の抵抗値や、その分布

これで、脳の中の活動量を X、センサーで捉える活動量を Y とすると
$AX = Y$という形式に落とし込めるはずです。
ここの A を計算するために、抵抗とか距離とかが必要なんですね！
このことを ForwardSolution という感じに言います。
ここから $X = A^{-1}Y$という風に変えれば X を計算できます。
これを InverseSolution と言い、A^{-1} のことを
InverseOperator と言います。

手順としては以下のとおりです。

1. 「推定するべき脳の部位」と EEG/MEG のセンサーの位置をすり合わせる。
   この作業は手動で行われる。(やればわかる)
   この重ね合わせ情報は trans というファイル形式で保存される。

2. MRI から脳の形を取ってきて、骸骨の抵抗とかも加味して計算できる形にする。
   これを BEM という。
3. 脳の形から「推定するべき脳の位置」を特定する。
   この脳内の位置情報をソーススペース (source space) という。
4. 脳の部位情報と頭の形情報とセンサーの位置から、
   脳活動によってどのようにセンサーに信号が届くかを計算する。
   これを脳磁図における順問題 (forward solution) という。
5. ノイズについて考慮する。この時、covariance matrix と言うものが必要になる。
6. 上記の脳部位とセンサーの関係性から、特定の脳部位での電源活動の波形を推定する。
   これを脳磁図における逆問題 (inverse solution) という。
   逆問題を解くために数式を作る。その数式を inverse operator という。
   逆問題には決まった解答はない。「最も良い解を得る方法」が幾つか提案されている。
7. 脳全体で推定した波形のうち、欲しいものをとってくる。

その後は色々なストーリーがあるでしょう。

- 推定された波形を wavelet 変換する。
- PSD や ERP をしてみる。
- 脳の各部位のコネクティビティを算出する。
- 何か僕達が思いつかなかった凄いアイデアを実行する。

などなど。
でははじめましょう。

手順 **1**、**trans**

GUI での操作となります。
下記のコードを実行すると画面が立ち上がります。

```
    mne.gui.coregistration()
```

subject や meg への path を指定しない場合は、GUI 上で指定することになります。
もし 0 から立ち上げた場合、山のようにある MRI の subject から該当の
subject を探さねばならなくなります。

python の関数に色々入れてから起動すれば、
既にデータが読み込まれているので、楽です。

```
    mne.gui.coregistration(subject = subject,subjects_dir = subjects_dir,
                           inst = file_path)
```

inst は meg データ…raw でも epoch でも良いらしいですが、どれかを指定して下さい。

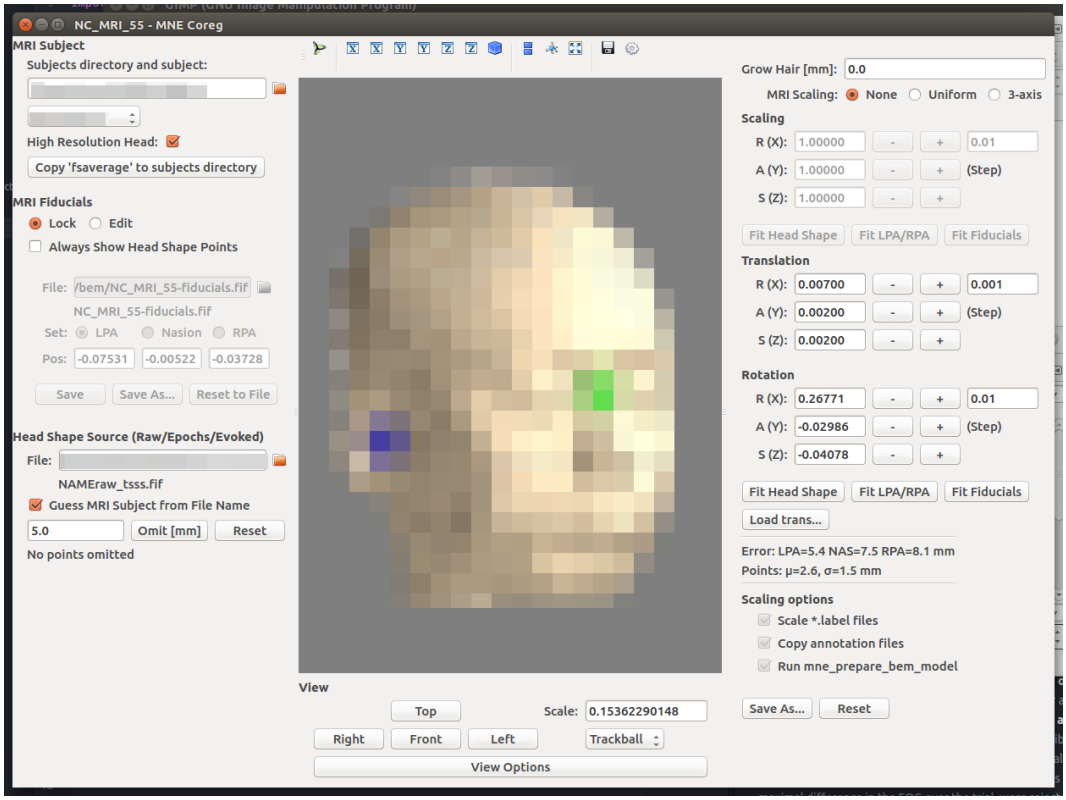


Figure 19: mne coregistration の画面。大して苦行ではない。

手順はこうです。

1. 必要ならば、MRI の subject を読み込む
2. 必要ならば、fif ファイルを読み込む
3. 左側、set のところで耳と眉間の位置を入力 (MEG のスタイラスでポチるところです)
4. それの一寸上の所、lock をポチる。
5. 右側、Fit LPA/RPA ボタンを押す。
6. 中の人の顔データをマウスでグリグリしながら、右上の ± ボタンを押して調整。
7. ちゃんと fit したら右下の save as ボタンを押して保存。

あとで、保存した trans を

```
    trans = mne.read_trans('/Users/hoge/fuga/trans.fif')
```

みたいな感じで読み込んで使います。
注意点として、脳波とかの場合は表示が projection モードになっているかもしれません。
そうなっていたらうまく重ね合わせる厳しくなるので、

手順 2、 BEM 作成

上記の通り、MRI から抽出してくる形データとして、BEM と言うものを使います。
BEM は脳の全体を包み込むサランラップみたいなデータです。
頭蓋骨とか皮とか、そういう絶縁体を考慮するために、BEM は三枚一組で
出力されます。実装上は 3 枚あるということを意識しなくても大丈夫です。

作るためには freesurfer による解析データが必要となります。
freesurfer を既に使っているなら Subject 関連は既に馴染んだ言葉でしょうか？
もちろん SUBJECT や SUBJECTS_DIR は読み替えてください。

```
    mne watershed_bem -s subject -d subjects_dir
```

これにより、BEM が作成されました。
再び python に戻り、下記を入力してみてください。

```
    mne.viz.plot_bem(subject = subject, subjects_dir = subjects_dir,
                     brain_surfaces = 'white', orientation = 'coronal')
```

これで BEM が表示されるはずです。

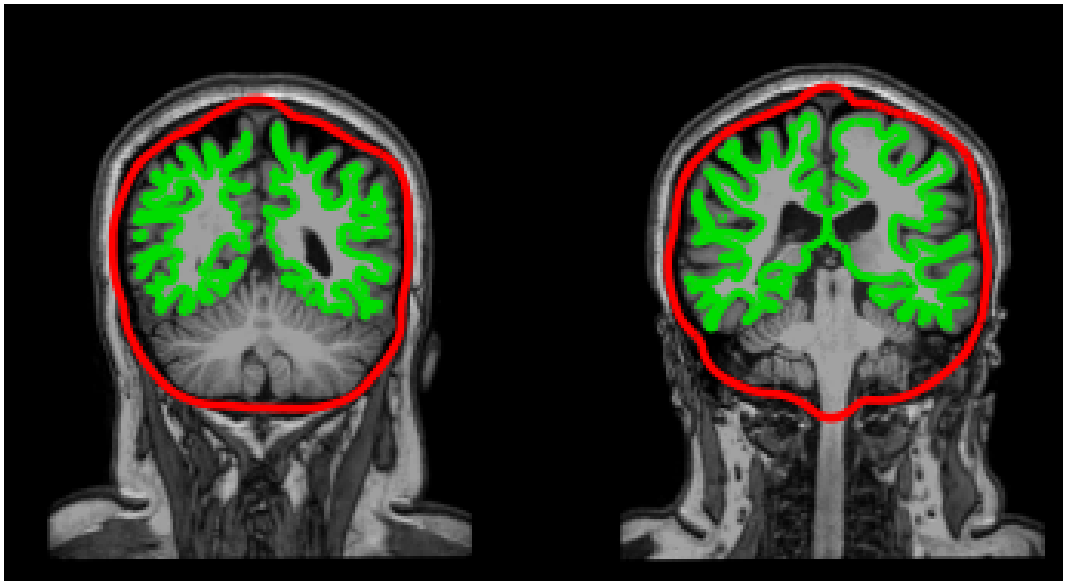


Figure 20: BEM の図示。

もし、標準脳を使うなら、以下のコマンドをターミナルから叩いて下さい。

```
    mne coreg
```

gui の画面が現れると思います。
'fsaverage → SUBJECTS_DIR' というボタンを押して下さい。
freesurfer の標準脳である fsaverage が現れます。
以降、subject には fsaverage を入れると標準脳を使うことになります。

手順 3、ソーススペース作成

脳磁図で見れる空間のうち、どの部分の電源を推定するかを
設定する必要があります。その設定がソーススペースです。
subjects_dir は環境変数に設定していれば要らないです。

```
    src = mne.setup_source_space(subject = subject, spacing = 'oct6',
          subjects_dir = subjects_dir)
```

もちろん、標準脳が欲しい場合は黙って fsaverage。
これで、src という変数にソーススペースが入りました。

見慣れぬ単語が出てきました。oct6 とは何でしょうか？
それはここに書いてあります。
http://martinos.org/mne/stable/manual/cookbook.html#setting-up-source-space

ソーススペースを作るためには計算上正十二面体や正八面体で
区画分けするので、その設定ですね。
やり方によってソーススペースの数も変わるみたいです。
臨床的に意味があるかはわかりません。

標準脳を使う場合は'fsaverage' を subject に指定して下さい。
ない場合は手順 2 の mne.gui.coregistration() でボタンを押して下さい。

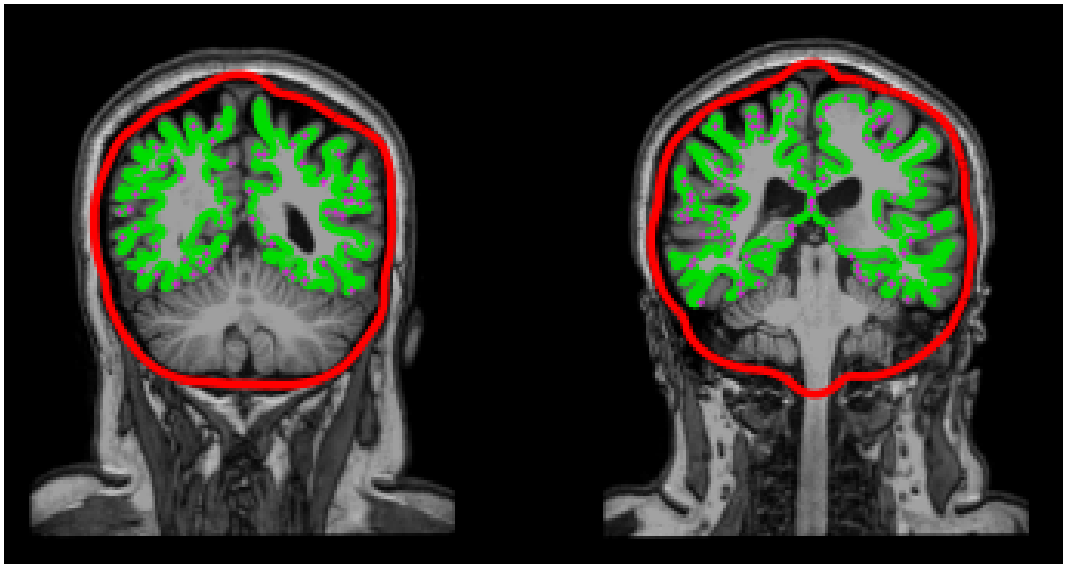


Figure 21: ソーススペースの図示。小さい点々がソーススペース。

手順 **4**、順問題

先程作った BEM は 3 枚あります。
EEG の場合は 3 枚必要です。何故なら、磁力と違って電力は
脳脊髄液と頭蓋骨と頭皮を素通りしにくいからです。
だから、BEM を三枚仮定するのです。
MEG の場合は一枚だけで十分だそうです。

では、BEM で順問題を解く準備をしましょう。

```
    conductivity = (0.3,)
    model = mne.make_bem_model(subject = 'sample', ico = 4,
                               conductivity = conductivity,
                               subjects_dir = subjects_dir)
    bem = mne.make_bem_solution(model)
```

これにより、BEM を読み込み、順問題解きモードに入りました。
ico はどの程度細かく順問題を解くかの数値です。ico の数字が高いほうが詳しいです。
conductivity は電気や磁力の伝導性のパラメータです。
EEG の場合はこれが (0.3, 0.006, 0.3) とかになったりします。

では、先程作った色々なものと組み合わせて順問題を解きます。

```
    trans = mne.read_trans('/hoge/fuga')
    mindist = 5
```

```
    fwd = mne.make_forward_solution(raw.info,
                                    trans = trans,
                                    src = src, bem = bem,meg = True,
                                    eeg = False,
                                    mindist = mindist, n_jobs = 4)
```

ここまでやった方にとって、上記のパラメータはだいたい分かるでしょう。
mindist は頭蓋骨から脳までの距離です。単位は mm。
ここで使うのは raw.info です。
mindist は頭蓋骨からみて、一番浅い部分にあるソーススペースの距離です。

手順 **5**、コヴァリアンスマトリックス関連

次に、MAP 推定という方法を用いて脳活動を推定します。
この推定には covariance matrix というものを使ってソースベースのデータの
ノイズ周囲の事を計算していかねばなりません。
これには MEG を空撮りした空データを使います。下記でからの部屋データを読み込みます。

```
    cov = mne.compute_raw_covariance(raw_empty_room,
                                     tmin = 0, tmax = None)
```

これでコヴァリアンスを作ることになりますが…MNE には更に追加の方法があります。
上記の空室の方法は広く行われている方法ですが、
誘発電位を見たい場合は restingstate(脳が何もしていない時の活動電位)
がノイズ (本来ノイズではないが、ここではノイズ) として乗る可能性があります。
それを含めるならば、下記のようにすることも出来ます。

```
    cov = mne.compute_covariance(epochs, tmax = 0., method = 'auto')
```

ちなみに、この method = auto というのは MNE に実装された新しいやり方だそうです。
tmax = 0 にしているので、刺激が入る前までの波を取り除きます。
つまりベースラインコレクションみたいな感じになるのです。
ちなみに、epochs で covariance…特に auto ですると結構重いです。

他に、raw データから covariance matrix を作る方法もあります。

```
    compute_raw_covariance(raw, tmin = 0, tmax = 20)
```

これは resting state とかに良さそうですね？
…最早当然ですが、tmin,tmax は時間です。単位は秒です。

手順 **6**、逆問題

最終段階です。順問題とコヴァリアンスを組み合わせて逆問題を解きましょう。
下記のとおりです。

```
    inverse_operator = make_inverse_operator(epochs.info, fwd,
                                        cov,loose = 0.2, depth = 0.8)
```

inverse_operator と言うのは何かというと、逆問題を算出するための式です。
この inverse_operator を作るために頑張ってきたと言っても過言なしです。

ここで、第一引数に epochs.info を入れていますが、info なら
raw でも evoked でも良いはずです。

さて、ここで loose と depth という耳慣れぬ物が出てきました。
一寸大事なパラメータです。
脳内の電流源推定と言っても、電流の向きを考慮しなくてはならないわけです。
loose はその向きがどのくらいゆるゆるかの指標です。
脳磁図はコイルで磁場を測る関係上、脳の表面と水平な方向の成分を
捉えやすいように出来ています。
でも、脳波複雑だから完全な水平ってないよね？ どのくらいのを想定する？
という風なパラメータです。
loose は 0〜1 の値をとりますが、loose が 1 というのは超ユルユル、
どの方向でも良いですよということです。
ちなみに、loose が 0 の時は一緒に fixed を True にする必要があります。
fixed が True の時は、MNEpython が脳の形に沿って自動調整してくれます。

depth は何かというと、どのくらい深い部分を見たいか、です。
MNE という計算手法は脳の表面の情報を拾いやすい偏った計算方法です。
故に、深い部分に対して有利になるようにする計算方法があります。
depth を設定すると、脳の深い所を探れるわけです。
depth を None に設定すると、ほぼ脳の表面だけ見ることになります。

他に limit_depth_chs というパラメータもあります。
これを True にすると、完全に脳の表面だけ見ます。
即ち、マグネトメータをやめて、グラディオメータだけで見るのです。

ここまで長かったので保存しておきましょう！

```
    write_inverse_operator('/home/hoge/fuga',
                           inverse_operator)
```

この inverse_operator が作れたら、あとは色々出来ます。

手順 **7** ソース推定

まずは、ソース推定をやってみましょう。

```
    from mne.minimum_norm import apply_inverse
    source = apply_inverse(evoked, inverse_operator)
```

これで出てきたものの中に data という変数があります。
まさに膨大な数です。脳内の膨大な場所について電流源推定したのです。
これは、一つ一つが脳内で起こった電流と考えて良いと思います。
...とは言え、推定ではあるので本物かどうかはわかりませんが。
細かい所は公式サイト見てください。

さて...こんな膨大な数列があっても困りますよね？
脳のどこの部位なのかわかりませんし。
そこで、freesurfer のラベルデータを使います。
それによって、脳のどの部分なのか印をつけてやるのです。

手順 **8**、前半ラベル付け

freesurfer にはいくつかのアトラスがあります。
詳しくはここをみて下さい。
https://surfer.nmr.mgh.harvard.edu/fswiki/CorticalParcellation
desikan atlas とか Destrieux Atlas とか色々ありますよね。
こういうのを読み込まねばなりません。
ターミナルでこのように打ってみて下さい。

```
    ls $SUBJECT_DIR
```

freesurfer のサブジェクトが沢山出てくるはずです。
サブジェクトの中身には label というディレクトリがあります。
この中にいっぱいそういう freesurfer のアトラスが入っています。

ファイルの形式には二種類あり、annot 形式と label 形式があります。
annot 形式は新しく開発されたアトラスが入っていて、
label 形式はブロードマンと思います。
annot 形式の内容はこのように読みます。

```
    mne.read_labels_from_annot(subject,annot_fname = 'hoge')
```

詳しくは公式サイト (ry
他にも読み方があります。
こうして読んだら、label のリストが出てきます。
単体の label は下記で。

```
    mne.read_label(filename, subject = None)
```

これで label を読み込めたら、次はそれを当てはめることになります。

手順 **8** 後半、**label** 当てはめ

では、label をベースにデータを抜き出しましょう。

```
    source_label = mne.extract_label_time_course(stcs,
                        labels, src, mode = 'mean_flip')
```

ここでは stc がソースのデータ、src が左右半球のソーススペースのリストです。
mode はいくつかあります。
mean: それぞれのラベルの平均です
mean_flip: すみません、わかりません＞＜
pca_flip: すみません、わかりません＞＜
max: ラベルの中で最大の信号が出てきます

殆どわからなくてごめんなさい。
ただ、これで脳内の波形が取り出せたわけです。
これで、色々出来ます。なにしろ、今まで wavelet 等していたわけですから。

その後のお楽しみ **1**、ソースベース **wavelet**

ソースベースで wavelet やりたいなら、特別に楽ちんな関数が
実装されています。

induced_power と phaselocking_factor を算出する関数は下記です。
※ label を選ばなければ激重注意！[40]

```
    induced_power, itc = source_induced_power(
            epochs, inverse_operator, frequencies, label,
            baseline = (-0.1, 0),
```

[40] label を選ばない場合これは激重です。何故なら 306 チャンネルの MEG からソースに落とし込むと計算方法によっては 10000 チャンネルくらいになります。ROI を絞ったとしても「人数 × タスク ×ROI の数 ×EPOCH の数」回 wavelet 変換して power と itc に落とし込むのですから…途方もない計算量です。label を選びましょう。

```
            baseline_mode = 'zscore',
            n_cycles = n_cycles, n_jobs = 4)
```

基本は以前 wavelet 変換で行った事に、いくつか追記するだけです。
まず、ベースラインコレクションはここでは zscore でしています。
やり方は色々あります。label は freesurfer のラベルデータです。
baseline 補正の時間についてはデータの端っこすぎると値がブレるので、
そこのところはデータ開始時点～刺激提示の瞬間の間で適切な値にしておいてください。
これで算出された wavelet 変換の結果の取扱は、前に書いた wavelet 変換の結果と同じです。

その後のお楽しみ **2**、ソースベース **connectivity**

ソースベースでコネクティビティ出来ます。

```
    from mne.connectivity import spectral_connectivity
    con, freqs, times, n_epochs, n_tapers = spectral_connectivity(
                source_label, method = 'coh', mode = 'multitaper',
                sfreq = 500, fmin = 30,
                fmax = 50, faverage = True, mt_adaptive = True)
```

使い方はセンサーベースコネクティビティと同じです。
この場合、さっき計算して出したラベルごとのデータと、
ラベルリストを放り込めば、先述の 5 つの変数が出てくるので楽ちんです。

コラム ***3-markdown*** で同人誌を書こう！

```
    皆さんもこのような科学系同人誌書きたいですよね？
    書いてコミケにサークル参加したいですよね？
    **難しいLaTexなんて覚えなくても大丈夫。そう、markdownならね！**
    LaTeXは添えるだけ。手順は下記。

    macならmactexとpandocをインストールします。
    ubuntuやwindowsならTeXliveをインストールします。
    mactexはググれば出てきます。pandocは
    brew install pandoc

    ubuntuなら
    sudo apt install texlive—lang—japanese
```

```
sudo apt install texlive-xetex
sudo apt install pandoc

これでpandocでmarkdownからpdfに変換できるようになります。
例えばDoujinshi.mdというマークダウンファイルを作って

pandoc Doujinshi.md -o out.pdf \
-V documentclass=ltjarticle --toc --latex-engine=lualatex\
-V geometry:margin=1in -f markdown+hard_line_breaks --listings

四角で囲われているところはコードの引用の書式に従って書いた後、
コードの上の```の末尾に{frame=single}と書き加えてください。

これで同人誌に出来るPDFになります。詳しくはググってください。
良い同人ライフを！
```

初心者のための波形解析

ここまで実践を行ってきましたが、ブラックボックスでした。
でも、理解する努力はすべきでしょう。
ここから理論編に入ります。

内容は高校数学 (地獄級) でギリギリやれますし工学部生は大抵理解しています (?) が、
初心者には結構難しいです。また、本書はあんま頭のいい人が書いてるわけじゃないです。
だから、ここでは物凄くざっくり (やや不正確な) 解説をします。
...というか、算数が苦手なので不正確な話しかできません。

波形解析で得たいものと、必要な変換

脳波や脳磁図を解析して貴方は何を得たいでしょうか？
脳波や脳磁図の特定の波長...例えばα波、β波、γ波等の強さを求めたいですね！ それもミリ秒単位で！
また、どのくらい位相が揃っているかとかも見たいです。
波と波の関係性とかもみたいですね。
なので、必要なのは下記を兼ね備えたデータです。

- 波の強さ

- 波の位相
- 波の位置 (ミリ秒単位)
- 注目した周波数以外は無視できる
- 波と波の関係性

波というものは位相があります。出っ張りもあれば凹みもあって、しかもいろいろな波が
重なっていたりして定量化しにくいです。特定の周波数の波の出っ張りと凹みを
両方同じように評価するにはどうすればいいでしょう？

これらを実現するのが時間周波数解析です。
本当は色々な種類の時間周波数解析があるのですが、
ここでは以下の 3 つの解析を解説しようと思います。

- フーリエ変換
- 連続 morlet wavelet 変換
- ヒルベルト変換

フーリエ変換とは

調べたい波を、全て sin と cos だけで表してしまうやりかたです。
周波数ごとに長い sin,cos 波を大きくしたり小さくしたりしながら当てはめます。
要するに、周波数ごとに
$Acosx + Bsinx$
という形に直していきますが、実はもう一寸スマートなやり方があって
$Acosx + Bisinx$
という複素数の形式 (!?) に落とし込んでいくのを目指します。
理由は波の強さ、位相を観察するためには複素数のほうが都合がいいからです。

連続 **morlet wavelet** 変換とは

しかし、sin も cos も未来永劫絶対に減衰しない波であるため、
不規則な波のフーリエ変換はきついです。実は完全にダメではなく、
「一部を無理やり切り取ってきて、切り取った波が永遠に続く波と
想定した上で変換すること」は一応可能です。[41]
そこで、フーリエ変換にほんの少しの細工を施して時間軸を加味したのが
連続 morlet wavelet 変換です。(以下、wavelet 変換)

理解するためにはフーリエ級数を理解する必要があります。

[41] この、永遠に続く波じゃないと無理っぽい感じになることをフーリエ変換の不確定性と言います。連続する波をぶつ切りにすると、端っこが不揃いになったりして、そこの処理もしないといけないです。

ヒルベルト変換

上記とはちょっと違った風な時間周波数解析ですが、
その性質や使い方は wavelet 変換によくにています。
wavelet 変換は一つの波から「実数部分と虚数部分を抽出してくる変換」
と言えますが、ヒルベルト変換は
「元の波から架空の虚数軸部分を作っちゃう変換」みたいなイメージです。
こちらは bandpass-filter に応用されたりします。

フーリエ級数

早速ですが、フーリエ変換の時、元の波は下記のように表します。

$$f(x) = a_0 + \sum_{k=1}^{\infty}(a_n \cos\frac{2\pi nt}{T} + b_n \sin\frac{2\pi nt}{T})$$

これは何かというと、波を変換している式です。解説します。それぞれ...

- $f(x)$:元の波を表す式
- a_0:波のベースの高さ
- t:時間
- n:解析したい周波数に対応した変数
- T:周期 (任意の定数)
- 右辺:特定周波数の波を表す式

かなり複雑な式っぽいですが、$\sum$ の中を御覧ください。
解析したい周波数が変数として与えられてます。これは、特定の周波数だけにしか対応しないのです。

大抵の波はこの単純な波の足し算で説明することが出来るのです。

実はこの方程式を積分したりして解けば、各 n に対する a も b も算出することが出来ます。
右辺全体をフーリエ級数、a と b をフーリエ係数と言います。
この式によって規則的なほとんど全ての波を別形式に書き換えられます。
凄いですね！ ですが、それだけでは面白くありません。
波をこのように別の式に書き換えた所で僕達がほしい
「波の強さ」「波の位相同期性」「波の位置」の情報がないからです。
なので、上の式を数学的に変換して公式を求めます。

複素フーリエ級数

高校数学の複素数平面を覚えているでしょうか？ 全ての二次元の座標は
複素数平面上で下記の式によって表現できます。

$$r(cos\theta + isin\theta)$$

この式です。最終的にはこの様な形に成れば、波の強さも位相も分かるはずです。

さて、貴方はオイラーの公式をご存知でしょうか？
この世で最も美しく偉大な公式の一つです。高校数学 (地獄級) でギリギリ出てきます。下記です。

$$e^{i\theta} = cos\theta + isin\theta$$

何度見ても美しい公式ですね。[42]証明はググってください。
これは美しいだけでなく役に立ちます。まずはこのオイラーの公式による変換を考えます。

端折りますが、オイラーの公式を変形すると sin と cos を e と i で表現することが可能ですので、
フーリエ級数の式から sin と cos を排除できます。高校数学 (地獄級) で弄くり倒すと、

$$f(x) = \sum_{n=-\infty}^{\infty} C_n e^{i\pi t/T}$$

と変形できます。Cn は変換 a,b を複素数でまとめたものです。
この式の右辺を複素フーリエ級数といいます。
フーリエ変換とはフーリエ係数を求める計算のことです。

$f(x)$ は元の波、$e^{i\pi t/T}$ はオイラーの公式を見ると...極座標で言うと半径 1 の円ですね。

ここで知りたいのは Cn です。
Cn を求めるにはやはり高校数学 (地獄級) によって下記のように書き換えられます。

$$C_n = \frac{1}{T}\int_{-T/2}^{T/2} x(t)e^{-2\pi nt/T}$$

$f(x)$ は脳波や脳磁図の結果ですので、フーリエ変換は
脳波や脳磁図に絶対値 1 の複素数を掛け算して積分する操作といえそうです。

i 乗とか実際に計算するのはアレなので、実際の計算をする時も
オイラーの公式で展開...と言いたいところなのですが、
python3 なんかは複素数の演算ができるので、e^i を計算できたりします。凄いですね！！！

[42]ちなみに、θ が 1 の場合はオイラーの等式と言います。

というわけで、上の式をそのまま python3 流に書けばフーリエ変換は
自分でコーディングすることが出来るわけです。でも、僕は自分ではしないです。
既にそれをするスクリプトが開発されているからです。[43]

ところで、フーリエ変換には FFT という超速いアルゴリズムがあります。
これは畳み込みの定理を組み合わせることで wavelet 変換にも応用できます。
結果は変わらないので MNE とかでは使うといいでしょう。

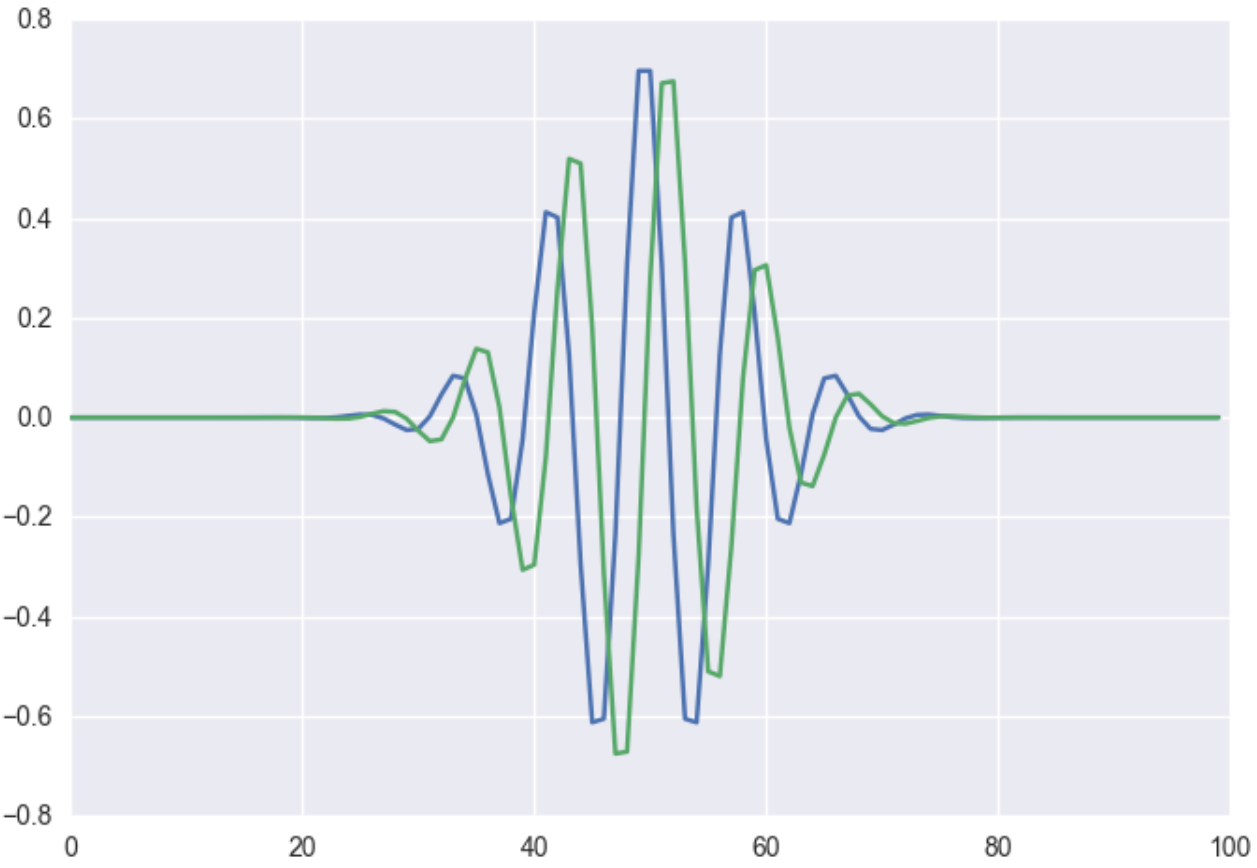


Figure 22: 再掲。morlet wavelet の 2D プロット

[43]既に作られているものをもう一回作る無駄のことを、業界では「車輪の再発明」と言います。

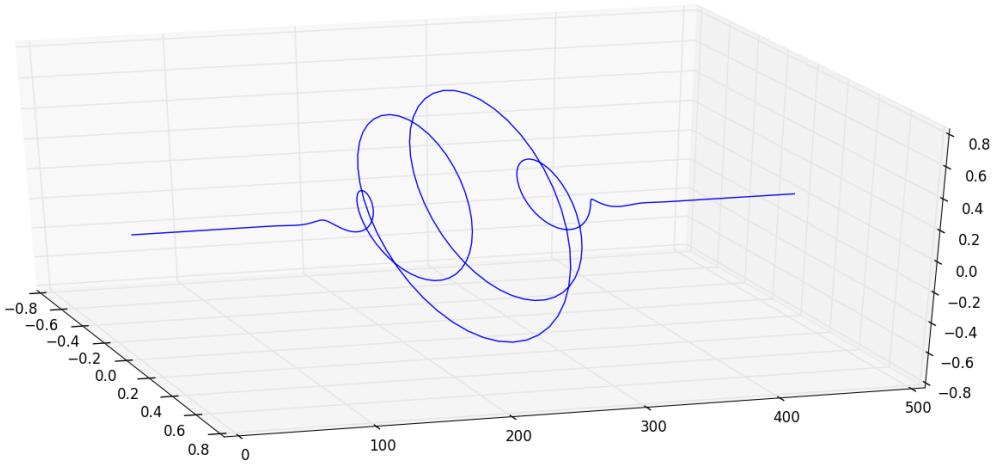


Figure 23: 再掲。3D プロット

結局何をしているのか

要するに基準となる複素数の波 (エネルギー 1) を物差しにして、

波 $=$ 特定の周波数の波1 $+$ 特定の周波数の波2 $+$ 特定の周波数の波3...

特定の周波数の波 $=$ 定数 $*$ 基準の波

という感じに変えていく計算です。
沢山ある定数 (数列ですね) さえわかれば波が表せます。
で、この定数を複素数にしたものが複素数版フーリエ級数。
複素数だから「定数」の絶対値や位相が計算しやすい、ということです。

さて、今あなたは周波数ごとに
$A + Bi$ または (r, θ) という形の複素数を得ることが出来ています。
これをどう料理していくかが次の問題です。
ここまでくれば高校の複素数です。

スペクトル解析

得たかったものは何だったか思い出しましょう。
複素数から「絶対値、位相」が分かるというのは
高校時代に数学をしたことのある人は明らかでしょう。
では、脳波の強さとは何でしょうか？ それはエネルギーです。

即ち、絶対値の二乗に他なりません！
脳波の位相はもちろん、上記の複素数の位相成分ですね。

フーリエ変換で出て来る数値 (フーリエなら係数に相当するやつ) は
スペクトルと業界 (脳波以外でも波を使う業界なら全て) で呼ばれています。
特に、自分自身に自分自身を掛け算したものの積分は「パワースペクトル」といいます。
$f(\omega)$ の絶対値の二乗が力 (パワー)、そして位相がそのまま位相です。

この、パワースペクトルの時間単位の平均値が
パワースペクトル密度 (PSD) と言われ、脳波解析の結果の一つです。

では、位相についてはどうするでしょうか？
一つの脳波の位相を取り出したところで意味はありません。
位相の良いところは他の波とリズムが揃っているか調べられる所にあります。
「毎回同じ位相を取り続けるかどうか」であったり、
「他の場所の位相とどのくらい差があるか」であったり、
色々わかります。

毎回同じかどうかは PhaseLockingFactor とか InterTrialCoherence と呼ばれ、
それぞれ PLF、ITC と略されます。
他の場所の位相とどのくらい差があるかは PhaseLockingValue と呼ばれ、
PLV と略されます。

さて、実際の計算は大したことはありません。
複素数 $A + iB$ について考えてみましょう。
これの絶対値は $ (A + iB)(A - iB) $であることは自明です。
すなわち、複素共役同士を掛け算したものです。
これ、自分自身をかけ合わせると位相が 0 になりますが、
他の波の複素共役と掛け合わせると位相の引き算になるのです。
コネクティビティはこういうのを使っていきます。

さて、これまで「複素共役を掛け算して積分したもの」がいっぱい出てきました。
長いので、よく下記のように略されます。
Sxx (x と x、つまり自分と自分の関係、パワースペクトル)
Sxy (x と y、つまり自分と他人の関係、クロススペクトル)
以下、このように書いていきます。

そして **wavelet** 変換へ

一旦話を戻し、まずは Power を考えます。
フーリエ変換の時点では周波数ごとの複素数が一つづつありました。

これでは時間の次元が失われていますね。いつその Power だったんだよと思います。
この理由は、フーリエ変換に使う sin と cos が永遠に続く波であるからです。
永遠に続く波を使えば、そりゃ時間軸は表現できるわけ無いです。

それを解決するのが wavelet 変換です。
時間軸を表現するためにものさしに使うものは減衰する波の必要があります。
減衰させるには色々あるのですが、morlet wavelet では以下のようにします。

$$e^{-iwx}e^{-x^2}$$

この、左側がフーリエ変換のためのやつ、右に付け加えたのが減衰するやつ。

こんな感じの式に色々と定数を付けたのが morlet wavelet です。
何故定数をつけるかというと、掛け算するときに掛け合わせるものの
絶対値が 1 じゃないとエネルギーを計算するのが超面倒だからです。
くるくる回りながら減衰する波です。

3D プロットのとおりですね。ちなみに、これは他にも色々あるやり方の一つです。

morlet wavelet も計算結果が複素数平面として出てきますから、
さっきと同じように Power と位相が分かるはずです。
しかも、時間別に！

初めてフーリエ級数とか聞いた人はここまで読んで意味がわからなかったと思います。
そんな人には高校数学の美しい物語の複素フーリエ級数関連の記事をお勧めします。

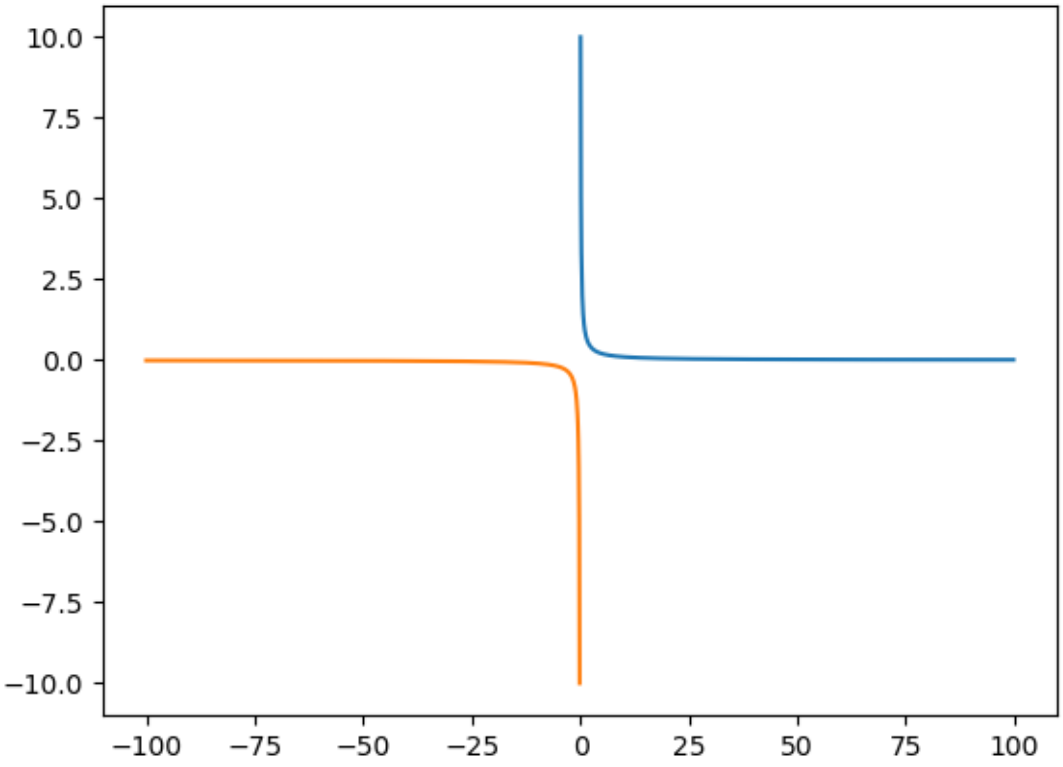
wavelet 逆変換と **bandpass filter**

上記のように wavelet 変換は数値を出すのに応用できるのですが、
これは実は bandpass filter にも応用することが出来ます。
何故なら、周波数を分けちゃう事ができるのと、wavelet 変換は逆変換が出来るからです。
wavelet 逆変換が出来るのは、上記を見れば自明かと思います。
手順としては、wavelet 変換したものの中から欲しい周波数帯域を選んで、
wavelet 逆変換を行えば bandpass filter になります。

もう一つの時間周波数解析、ヒルベルト変換

wavelet 変換は 1 つの実数の波から実軸と虚軸を分離しました。
でも、そもそも元々の波は実軸に存在します。おかしいですね？

元々の実軸をそのまま実軸にして、新たな虚軸を生み出す変換が
ヒルベルト変換です。これは bandpass filter に応用されたりしていますし、
パワーの算出にも使われます。
wavelet 変換は掛け算をして積分をして算出していましたが
ヒルベルト変換では双曲線を掛け算します。
一部無限大が出てきて気持ち悪さが残りますが、仕様です。



これについては、既に wavelet 変換と同等の性能を持っていることが
分かっている…らしいですが、実際どうなんでしょうね？

コネクティビティ各論

コネクティビティについては色んな method があります。
なので、スペクトルとは章を分けてみました。
ここではコネクティビティそれぞれの method について
僕の考えを述べます。間違ってたらごめんね。

王道の **PLV** と **Coherence** とその問題点

さて、フーリエ変換のところでコネクティビティについてちらりと書きました。
位相の差をとっていけば PLV という「どのくらい波が関連しているか」という
指標になると書きました。式で表すと

$$PLV = |\overline{\frac{Sxy}{|Sxy|}}|$$

という感じです。
この指標は正しいです。が、大きな欠点があります。
脳波にしろ、脳磁図にしろ、脳内に電極をブチ込むやり方ではなく、
漏れて拡散してきた物を捉えることになります。
ということは、何らかの大きな震源が近くにある場合、
影響を受けて似たような波が出た全てのチャンネルは
コネクティビティが「ある」と間違った結果が出てきてしまうのです。
図に沿って言うと、2 つの青い点 (センサー) で、一つの赤い波を同時に測定すると、
繋がっていると勘違いするのです。常識的におかしい。

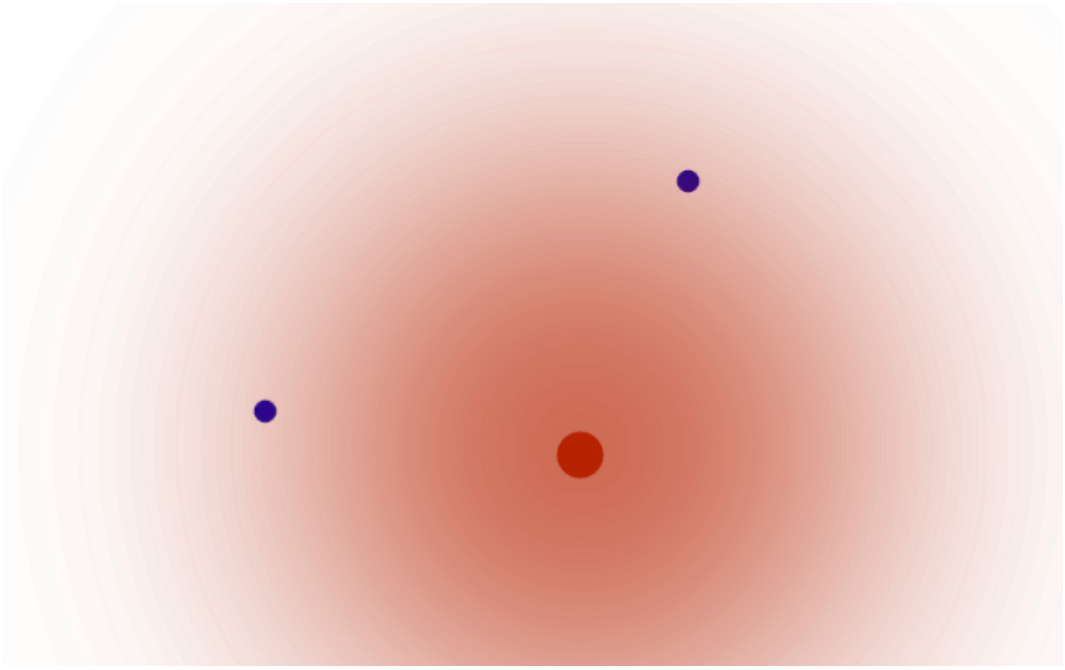


Figure 24: PLV でコネクティビティを計算する場合、この図の青い点が繋がっていると言うことになる。

さて、PLV について語ってきましたが、Coherence という計算の方法もあります。
以下のように計算します。これもまた、違和感のない計算方法ですが、

PLV と同様に拡散やノイズの影響が大きいです。

$$Coherence = \frac{\overline{|Sxy|}}{\sqrt{\overline{|Sxx|} * \overline{|Syy|}}}$$

ちなみに、Coherence に位相情報を残したものが Coherency です。

$$Coherence = \frac{\overline{Sxy}}{\sqrt{\overline{|Sxx|} * \overline{|Syy|}}}$$

こいつの使いみちは僕はあまり知らないです...

PLV や Coherence の欠点の克服

いくつか方法があります。
大きく分けて 2 つ、またはその組み合わせです。

- 拡散する前の電流を推定する
- 拡散しても大丈夫な計算方法を採る

電流源推定

センサーベースで拡散するならソースベースにしちゃえばいいじゃない。
というと、ソースベース解析が出来ない人からするとマリー・アントワネット的に
聞こえるかも知れませんが、本書はソースベース解析を主眼としている本なので...
MNE でも sLORETA でも dSPM でも使ってやればいいんじゃないですかね。

もう一つの方法が CurrentSourceDensity という方法です。
こっちはソースベース解析ほどバリバリに推定するわけではない感じですが、
PLV の弱点を補う方法ですね。
残念ながら MNEpython には未実装です。

PLV の発展系

MNEpython に実装されている有名所として、PLI と WPLI を紹介します。

PLI

PhaseLagIndex という指標があります。
こいつは上記の拡散をキャンセルしてしまう方法で、
MNE では spectral_connectivity 関数に実装されています。
式は下記

$$PLI = \overline{|sign(Im(Sxy))|}$$

ここで、sign は内容が正なら 1、0 なら 0、負なら-1 を返す関数。
Im は虚軸だけを返す関数です。
位相が常に x より進んでいたり、遅れている奴だけ加算していき、
位相のズレが 0 付近をウロウロしてるやつやバラバラなやつは消すメソッドですね。
確かにこれならばさっき挙げた偽のコネクティビティは出ないでしょう。
僕は初見「マジカヨ...」って思いました。
これで Connectivity を Plot したら、隣同士ばかり繋がるような図じゃなくなります。
つまり、成功しているということですかね。

WPLI

上記で、正なら 1、負なら-1 というのについてマジカヨ感が漂うのですが、
そこをもう少しスムーズにしたのが WeightedPhaseLagIndex です。
これは位相のズレが $\pi/2$ に近ければ近いほど大きな値を、
$-\pi/2$ に近ければ近いほど小さな値を入れ込みます。
そうすることによって、ノイズに強くなった...らしいです。
そりゃそうですよね。PLI は一寸ノイズがあったら 1 か-1 に振り切れますもの。
式は以下。

$$WPLI = \frac{|\overline{Sxy}|}{\overline{|Sxy|}}$$

見にくい記法ですみません...

Coherence の発展系

さて、PLI 系は同一の電流源からのラグをとっていたわけですが、
同様のやり方が ImaginaryCoherence です。

これは Coherence 虚軸のみを加算平均したものです。
虚軸を加算平均するということは、位相が $\pi/2$ ずれたら最大になりますね。
式は下記のような感じです。

$$Coherence = \frac{\overline{Im(Sxy)}}{\sqrt{\overline{|Sxx|} * \overline{|Syy|}}}$$

で、こういうのってロバストなの？

分からない…僕には何もわからないんです…

グラフ理論

さらなる解析として、グラフ理論があります。
これは「互いにどんな風に繋がっているかな？」というのを
考えていく理論です。色々なやり方があります。
例えば、一筆書きでどんな風に繋ぐか、ループを作らずにどんな風に繋ぐかとか、
そういうノリのやつです。
ここはまだ詳しくないんだ。ごめんね。

Minimum-norm-estimation の理屈

かなり面倒いので丁寧にはかけません。ここからは線形代数の範囲になります。
超簡単に言いますと、一般化逆行列を使って割り算したやつです。
ノイズについてベイズ推定する方法もありますが、
面倒いのでここでは一般化逆行列だけ使って説明します。

まず、脳のソースから出た磁力…電力でも良いですが、
そのようなものがセンサーに届く時、その強さはソースで発生した電力、磁力に比例するはずです。
多分、距離の二乗には反比例すると思います。
つまり…距離云々を除くと簡単な一次連立方程式になるはずなんです。
センサーで捉えた情報を y とし、ソースで発生したのを x としましょう。
あるソースで発生した電力とセンサーで捉えた電力の関係を下記の式で表せるとします。

$$y = ax$$

これは単純な掛け算なわけですが、1センサー1ソースではなく、多センサー多ソースです。
ここで、沢山になった y,a,x について、下記のように表すとします。

$$Y = y_1, y_2, y_3.....$$

$$X = x_1, x_2, x_3.....$$

ここで A を下記を満たす行列とします。

$$Y = AX$$

この連立方程式を解きます。これがどうしても連立方程式に見えない人は、
行列代数の入門書でも読んで下さい。ネットで検索するよりこれは本の方がいいです。

では解きま...じつは解けません。
一時連立方程式はあんまり数が多くなると解けなくなるのです。
ですが、最も真実に近いっぽいのを推定することは出来ます。
今回、脳全体のある一瞬の波が最も小さくなるような波を推定しましょう。
Minimum-norm-estimation(MNE) と言います。
(他に一つ一つの波が一番小さくなる beamformer 法とか色々あります)
導出は有名なムーアペンローズの逆行列とよく似ていますので、
それを勉強すれば理解が早まると思います。

つまり、条件 $Y = AX$ のもとで X を小さくしたい[44]のです。
X が小さいってことは下記の式が小さいってことです。

$$||X||^2$$

まず、ラグランジュの未定乗数法という公式を使います。[45]

$f(x, y)$ が極値になる x,y は

$$L(x, y, \lambda) = f(x, y) - \lambda g(x, y)$$

の時に

$$\frac{\partial L}{\partial \lambda} = \frac{\partial L}{\partial y} = \frac{\partial L}{\partial x} = 0$$

の解、または

[44] ノルムが小さいということ。中学生風に言うと絶対値。
[45] ムーアペンローズの逆行列ではよくあること。

$$\frac{\partial g}{\partial x} = \frac{\partial g}{\partial y}$$

の解。...という感じの公式です。
行列の微分をしないといけないので、行列の微分の仕方を確認しておきます。

$$\frac{\partial a^t x}{\partial x} = a$$

$$\frac{\partial x^t a}{\partial x} = a$$

今回はこれを微分します。

$$L = ||X||^2 - \lambda||Y - AX||^2$$

$$L = X^T X - \lambda(Y - AX)^T(Y - AX)$$

$$= X^T X - \lambda(Y^T - X^T A^T)(Y - AX)$$

$$= X^T X - \lambda(Y^T Y - X^T A^T Y - Y^T AX + X^T A^T AX)$$

$$\frac{\partial L}{\partial X} = 2X - \lambda(-A^T Y - A^T Y + (A^T A + A^T A)X)$$

$$= 2\lambda(A^T Y - A^T AX + \frac{X}{\lambda})$$

これが 0 になるので

$$(A^T A - \frac{I}{\lambda})X = A^T Y$$

$$X = (A^T A - \frac{I}{\lambda})^{-1} A^T Y$$

ここで $\frac{I}{\lambda}$ を C とおくと

$$X = (A^T A - CI)^{-1} A^T Y$$

これで無事 X を A と Yで表せました。
C というのが出てきましたが、これはまぁ...定数です。

ものすごくざっくりというとこの様な事です。
では、次に重み付けをしてみましょう。

MNE の重み付け

さて、MNE は重み付けを行うことが出来ます。
これは MEG の場合は特に大事です。何故なら、MEG は計測の
方法[46]によっては脳の表面の皮質の信号しか捉えられないからです。
どの値を重視して推定していくかが設定できるわけです。
また、上記の式は不十分な部分があります。
ノルムの種類がこれでは一寸困るところがあるのです。
実際は L2 ノルムというのを使います。
では、重みを付けてみましょう。

$$||X||^2$$

を最小にするようなものでしたが、これは重みが入っていません。
重みの行列を仮に w として、その上で

$$X^T w X$$

が最小になるような条件を設定してあげればいいです。
この w は縦と横の長さが同じ正方行列かつ、対角線上以外全部 0 の行列です。
こういうのを掛け算すると重み付けが出来るんですね。

[46] グラディオメーターのばあい。
さっきは

そりゃそうです。行列 I の要素に順番にスカラー値を掛け算していくわけなので、
重みになります。

ではそのことを踏まえて今回はこれを微分します。

$$L = X^T wX - \lambda ||Y - AX||^2$$

$$L = X^T wX - \lambda (Y - AX)^T (Y - AX)$$

$$= X^T wX - \lambda (Y^T - X^T A^T)(Y - AX)$$

$$= X^T wX - \lambda (Y^T Y - X^T A^T Y - Y^T AX + X^T A^T AX)$$

$$\frac{\partial L}{\partial X} = (w + w^T)X - \lambda (-A^T Y - A^T Y + (A^T A + A^T A)X)$$

$$= 2\lambda (A^T Y - A^T AX + \frac{(w + w^T)X}{2\lambda})$$

$$= 2\lambda (A^T Y - A^T AX + \frac{wX}{\lambda})$$

$$= 2\lambda (A^T Y - (A^T A + \frac{w}{\lambda})X)$$

これが 0 になるので

$$(A^T A + \frac{w}{\lambda})X = A^T Y$$

$$(\lambda A^T A + w)X = \lambda A^T Y$$

$$X = \lambda(\lambda A^T A + w)^{-1} A^T Y$$

凄いですね！ これやこの、重み付き MNE の式です。

MAP 推定

今回はラグランジュの未定乗数法で二乗したのが x だけでしたが、
ここに正規化する変数を入れてやる必要があります。
その操作をしたのが下記です。

また、Y と X についてはベイズ統計学の MAP 推定した結果と同じになるのですが、
これはここに書くのが超絶面倒いので書きません。

これの解説だけで本が一冊書けます。

dSPM の理屈

さて...MNE の式をよく眺めてみましょう。
これは

$$Y = AX$$

の変形であり、シンプルな掛け算なのは言うまでもありません。
そこで、MNE の式を次のように書き直してみます。

$$X = BY$$

ここで、ふとした疑問が出てきます。
B の絶対値が 1 じゃない場合です。
B が 1 じゃない場合で、空室を撮ったと仮定してみて下さい。
センサーが捉えるノイズと空室のノイズは同じ大きさのはずですが、
B が 1 だったら空室がうまく説明できませんね？？？
ということで、これを補正してみます。

$$X' = \frac{BY}{||B||}$$

$$X' = \frac{BY}{\sqrt{BCB^T}}$$

このように、ノルムが 1 になるように割り算してあげることを数学の言葉で
正規化といいます。MNE の結果を正規化したものが dSPM です。
これが僕の dSPM に対する理解です。

ところで、分散を 1 にしてあげる方法もあるのですが、
これを標準化と言います。
かの有名な sLORETA は dSPM に対して正規化ではなく標準化したものです。
式はこうです。

$$X' = \frac{BY}{\sqrt{B}}$$

間違ってたらごめん (´・ω・`)

python での高速化のあれこれ

時に、処理速度が大事になります。特にソースレベル解析ともなると膨大な計算量になります。
その時のやり方をいくつか記しておきます。

for 文とリスト内包表記

python の for 文は絶望的に遅いため、for 文の入れ子はやめましょう…と言われています。
軽い処理なら良いんじゃないかと個人的には思いますが。

代わりと言ってはアレですが、このような python 構文があります。

```
    n=[i+4 for i in range(5) ]
```

この場合、[4,5,6,7,8] が帰ってきます。この書き方はリスト内包表記と言い、
広く使われています。詳しくはググってください。

numpy

numpy は速いので、重い演算の時は使えるなら使いましょう。
python は四則演算とか for 文とかとっても遅いのです。

並列化 (まずます速い)

python での並列化はとても簡単です。

```
    from multiprocessing import Pool
```

この Pool というのがお手軽並列化ツールです。
並列化する時は必ず何か関数を定義して下さい。

```
    def test(i):
        return i*8
```

これを Pool の中にぶちこみます。

```
    p=Pool(5)
    result=p.map(test,[1,2,3,4])
    p.close()
    result.get()
```

p という変数に並列化するための箱をつくりました。
この箱には 5 つの CPU で並列する機能をつけました。
そして、箱の中に並列したい test 関数と、それに入れたい変数を配列の形で
入れました。p の中に map という関数がありますが、こいつが並列化の関数です。
こいつを回せば、[8,16,24,32] という結果が出てくるのです。

ここでは map_async という関数を使う方法もあります。
map_async は map よりも頭の良い並列化関数です。
map は全員一斉にやる感じ、map_async は全員でやるけれど、終わった人は
次の課題をし始める感じです。

今の所引数が 1 つのものじゃないと無理です。
複数ある場合は、wrapper[47] を使わねばなりません。

```
    from multiprocessing import Pool
    def test(i,j):
        return i*j
    def wrap(a):
        return test(*a)
    p=Pool(5)
    result=p.map_async(wrap,[(1,2),(4,6)])
    p.close()
    result.get()
```

[47] 関数を加工する関数みたいなもの。

うわ面倒くせえ…
wrap という関数で test という関数を包み込み、そいつに引数を渡します。

素直に引数一つの関数でやるのが楽で好きです。

クラスタレベルの並列化 (数で押す方法)

ipython はコンピュータクラスタレベルの並列化をサポートしています。
一人二役 (コントローラとエンジン) することで一台でも実現できます。

今回は複数の引数付きでやってみたいと思います。クラスタの作り方については詳しくは述べませんが、準備段階については前述していますから参照してください。

その上で、おなじみのアレです。

```
    pip install ipyparallel
```

クラスタの設定ファイルを作ります。

```
    ipython profile create --parallel --profile=default
```

この設定ファイルのいじり方は公式サイトとか、qiita の記事を見てください。

その後のやり方は色々ありますが、僕のやり方を書きます。
まず、元締めのコンピュータでクラスタを起動します。ターミナルで以下を叩いてください。

```
    ipcontroller --ip=hogehoge --profile default
```

これで default という名前のクラスタのコントローラを ip 指定で起動しました。
…もちろん、1 台だけの場合は ip は要りません。
次に、下記のように各計算機 (子機？) でエンジンを起動していきます。

```
    ipcluster engines --n=4 --profile=default
```

–n は使うコア数です。元締めのコンピュータでも計算するなら同じようにしてください。
これで準備が整いました。

例えば、貴方が下記のような関数を実装したとします。

```
    calc_source_evokedpower(id,person,tmax,tmin):
        d=id+person+tmax+tmin
        return d
```

うん、立派な関数ですね。

この関数はグローバル変数を参照できないことに注意してください。
変数や import 文は全て関数内で宣言するようにしてください。
宗教的な理由で関数内で import 出来ない方はお引き取りください。

このうち、id と person は全ての組み合わせを、tmax と tmin は固定した値を入れたいとします。
これをクラスタレベルでガン回しします。

```
    import itertools
    from ipyparallel import Client
    client=Client(profile='default')
    print(client.ids)

    function=calc_source_evokedpower

    product=zip(*list(itertools.product(id,person)))
    plus1=tuple(['20Hz']*len(arglist[0]))
    plus2=tuple(['50Hz']*len(arglist[0]))

    arglist=product+[plus1]+[plus2]

    view=client.load_balanced_view()
    async=view.map_async(function,*arglist)
    async.wait_interactive()
```

これも面倒くせえ！

まず、ipyparallel をインポートします。一応クライアントを確認しておきます。
function に実行したい関数名を入れます。

その後、itertools の product 関数を使って id と person の全ての組み合わせを作ります。
さらに、固定した 20Hz と 50Hz を後に加えます。そして、配列を足し算していきます。

その後、3 行の呪文を唱えれば出来上がりです。返り値は async[:] で見れます。

Cython(使いこなせば相当強いが、多分不要)

Cpython[48]ではありません。Cython という別ものです。
python を C に変換することで場合によっては python の 100 倍[49]のスピードを

[48] python の正式名称
[49] 誇張ではありません。実際に効率の悪い python コードを最適化すると 100 倍速くなったりします。最適化なしでも 2 倍くらい速くなることもあります。

実現することが可能です。numpy とかは型関係が難しいです。
jupyter は大変優秀なので、下記のようにするだけで Cython を実行することが出来ます。

```
    %load_ext cython
```

これを jupyter で実行した後、関数を実装します。下記は numpy の例です。

```
    %%cython -a
    import numpy as np
    cimport numpy as np
    DINT=np.int
    ctypedef np.int_t DINT_t
    DDOUBLE=np.double
    ctypedef np.double_t DDOUBLE_t

    def u(np.ndarray[DDOUBLE_t,ndim=1] ar):
        cdef int n
        cdef double m
        for n in xrange(5):
            m=np.mean(ar)
            print(m)
```

上 5 行は Cython と numpy を組み合わせた時の特有のおまじないです。
上では、numpy のために int 型と double 型を用意してあげています。また、cdef は型指定です。
関数を宣言するときも黒魔術的に numpy の型を指定してあげねば
なりません。じゃないと動くけど遅いままになります。ndim は numpy 配列の次元数です。

それ以外は C 言語を書いた人からすると型指定が必要なただの python なので、
苦労はあまりないはずです？
ちなみに、新しい python では静的型付け風にコーディングする文法があり、
これはなんと Cython 対応ですが、残念ながら numpy 対応ではありません…

詳しくは cython のホームページをググってください。
http://omake.accense.com/static/doc-ja/cython/index.html

C 言語、C++、FORTRAN(最終兵器)

まぁ…そういうやり方もあります。正統派なやり方なのですが、本書では触れません。
車輪の再発明に気をつけましょう。

graph

ここはまだ僕は詳しくないのでお試しです。
お試しな同人誌の中で更にお試しです。

graph 理論でなにかやりたい場合はこうです。

```
    pip install bctpy
```

これで bctpy がインストールされました。
コネクティビティの結果である三角行列を突っ込みたいですね。
突っ込みます。

```
    import bct
```

例えば conmat という numpy 三角行列があったとして、こいつを放り込むなら
まずは三角行列を普通の行列にしてやるべきでしょう。
(方向ありの行列なら三角行列にはならないのでそのままでいいです)

```
    dcon=conmat+conmat.T
```

global efficiency を重み付けありで計算したいならこうと思います。

```
    bct.efficiency_wei(dcon)
```

すると、スカラー値が算出されます。

おすすめの参考書

ステルスマーケティングです。

- Analyzing Neural Time Series Data:Theory and Practice
  表題見て分かる通り、洋書ですが名著です。英語ですが平易に書かれています。
  amazon でも売ってます。どうすれば良いのかわからなくなった時の道標です。
  MNE-Python はどうすれば良いのか分からなくなる事が多いのです。
  買って下さい。
- 事象関連電位ー事象関連電位と神経情報科学の発展
  脳波関連の和書の名著です。なのですが、絶版です。古本を見つけたらすかさず買いましょう。
  内容的には凄く難しい数学はなく、実践的です。
  通読向けではありますが、やはり手を動かしながらじゃないときついです。

- 意味がわかる線形代数
  これは異色の本です。多くの線形代数の本は文系や医学生には
  冥王星語でも読んでいるように感じるのですが、
  これは日本語で書いてあります。意味がわかります！
  反面、内容は「分かっている人」からすると薄いでしょう。
- 統計的信号処理 信号・ノイズ・推定を理解する
  MNE って行列代数なのです。算数の基礎がわかってないと分かりません。
  でも、基礎がわかってもわからない本が普通にあるので悲しい。
  この本は基礎がわかったら何とか読めます。凄く親切に書いてある名著です。
  アホでも読める感じがしました。買いましょう。(理解できるかは別)
- 完全独習ベイズ統計学
  ベイズ統計学に関して直感的に分かる本です。
  この本は上記「意味がわかる線形代数」と同様、
  冥王星語を関西弁くらいまで下ろしてくれます。
  やはり入門書はざっくりしたものが一番です。
- Electromagnetic Brain Imaging: A Bayesian Perspective 2015
  割りと平易な英語で書いてある応用数学の本です。
  MNE,beamformer,dSPM,sLORETA 等が載っていて重宝します。
  内容は結構噛み砕いてくれていますが、元が僕にとっては難しいものなので大変でした。
  いい本です。算数を理解したいなら買って下さい。
- ゼロから作る　DeepLearning
  コラム 4 参照。ニューラルネットワーク系人工知能本の名著です。
  機械学習に興味がお有りなら買ってください。
  基礎を学ぶには本当に良いです。その後は chainer だとか tensorflow だとか
  触れば良いんじゃないですかね？
- パターン認識と機械学習 (上下)
  通称ぷるむる。
  難しくてとてもつらいですが、機械学習方面では聖書の一つです。
  この領域では機械学習を使うこともあるので、必要になればどうでしょう？

おすすめサイト

高校数学の美しい物語
http://mathtrain.jp
高校数学についてのサイトです。高校数学を復習するにあたって、このサイトは素晴らしい。

フーリエ変換とか行列計算とか複素数とか、そういったことを考える時に
辞書みたいにつかってみては如何でしょうか？

ウェーブレット変換の基礎と応用事例：連続ウェーブレット変換を中心に
https://www.slideshare.net/ryosuketachibana12/ss-42388444
このウェーブレット変換スライドは素晴らしいです。作者が学生の頃に勉強して作ったのだそうですが、
多くの「分かっている人向けの数学的入門書」と違い、直感的に分かるように書いてあるのです。

おすすめ **SNS**

qiita
http://qiita.com
日本のプログラマ用の SNS…というかブログサービスです。
かなり分かりやすい記事が多く、大変重宝します。
反面、コピペプログラマになるのを避ける心がけは大事ですね。
カジュアルな雰囲気漂う気軽なサイトです。

twitter
http://twitter.com
twitter かよ！ と思われるかもですが、学者さんのアカウント、雑誌のアカウント
開発者さんのアカウントは極めて有用かつ濃密です。
カジュアルな感じですが炎上には気をつけましょう。

github
https://github.com
プログラマ用 SNS の中でも最も有名なものでしょう。
qiita はただの記事集ですが、github は開発ツールです。ただし、学習コストが高いですね。
下記の git を中心に据えた web サービスです。
余談ですがマスコットキャラの octocat が可愛いです。

おすすめソフト

これまで散々色んなソフトを紹介してきましたが、
それ以外のツールも紹介しておきましょう。
当然ながら全てフリーウェアです。

- git
  バージョン管理ソフトです。プログラム書くときとか、長文書くときとかに
  どこをどう修正したのか分からなくなったりしませんか？

また、修正したいけど壊しそうで怖いからコピーとって修正したりしていませんか？
そんな貴方に必要なのはバージョン管理ソフトでしょう。
git は最も有名なバージョン管理ソフトの一つです。

- source tree
  git は使うための学習コストが結構高いです。
  こいつは git を簡単に使いこなす為の GUI ツールです。
  他にも github desktop とか git kraken 等色々あります。好きなの使えばいいです。
- pandoc
  markdown という形式で書いた文書をあらゆる形式に変換するソフトです。
  詳しくはコラム参照。これの何が嬉しいかというと、markdown で書いたものを
  word、LaTeX、PDF、HTML 等、あらゆる形式に変換してくれるのです。
  つまり、markdown さえかければ、他はいらなかった！
  (細かい所の調整は LaTeX 書く必要が有ることもありますが…)

参考文献

(まだ途中です…)
- Gramfort, M. Luessi, E. Larson, D. Engemann, D. Strohmeier, C. Brodbeck, R. Goj, M. Jas, T. Brooks, L. Parkkonen, M. Hämäläinen, MEG and EEG data analysis with MNE-Python, Frontiers in Neuroscience, Volume 7, 2013, ISSN 1662-453X
- Margherita Lai, Matteo Demuru, Arjan Hillebrand, Matteo Fraschini, A Comparison Between Scalp- And Source-Reconstructed EEG Networks
- Gramfort, M. Luessi, E. Larson, D. Engemann, D. Strohmeier, C. Brodbeck, L. Parkkonen, M. Hämäläinen, MNE software for processing MEG and EEG data, NeuroImage, Volume 86, 1 February 2014, Pages 446-460, ISSN 1053-8119
- https://surfer.nmr.mgh.harvard.edu/fswiki/FreeSurferWiki

MNEpython 実装時の小技

一応、実装が苦手な人が読者と思っているので、
ありふれた小技ですが紹介します。
object 指向とかは他の本を読んで下さい。得られるものが多いでしょう。

メソッド・チェーン

今回は超手軽に解析してみましょう！
何度もフィルタ掛けるの面倒くさいから、一気にかけちゃう方法です。
メソッド・チェーンを使います。
メソッド・チェーンとはドットで数珠つなぎに処理をしていく技法です。
MNEpython では raw オブジェクト辺りで割とできる感じです。実際見てみましょう。

```
    from mne.io import Raw
    Raw('hoge.fif').filter(1,100).notch_filter(60).save('fuga.fif')
```

どんだけ略してんだよ！ というくらい略されてますね。
このケースでは、読み込んでフィルタを 2 つ掛けて保存しています。
まぁ、使いすぎは色々大変になるので良くないです。

変数を減らしてみる

raw を弄る時 raw.filter 関数などを使うと raw 自体が
書き換わってしまいます。
これ自体は正しい動作なのですが、一寸わかりにくさを感じるかも知れません。
raw は raw としてどっしり構えてもらって、
加工品だけ作って行きたいかも知れません。
そんなときは raw.copy 関数がいいです。

```
    raw2 = raw.copy()
```

これで raw の copy が出来ましたね。しかし、どうも変数が多くなります。
raw2, raw3, raw4 と作るうちに raw ∞ とかなって死にます。
それの対策にはメソッドチェーンがいい味を出すと思っています。

```
    filtered = raw.copy().filter(1,100).notch_filter(60)
```

raw2 など要らなかった。
まぁ、一寸消費メモリとかは多くなるかも知れません。

引数多すぎだろ死ね！

確かに MNE の method は引数が多すぎである。
引数が多すぎて毎回引数入れるのがダルいし、ミスも多くなりそうだ。

だが、落ち着いて聞いてほしい。
python には良い道具があるのだ。

```
    from functools import partial
```

こいつは関数を部分的に解いちゃう関数だ。

今君は、複数の epoch オブジェクトを作りたいとする。
event_id は 1、2、3、4、5、6 だ。その都度入力するのはダルいし、
変数が増えすぎると管理も大変だ。
そんなときはこのようにすればいい。

```
    from mne.io import Raw
    from mne.epochs import Epochs
    from mne import find_events

    raw = Raw('hoge.fif', preload=True)
    events = find_events(raw)
    make_epoch = partial(Epochs, raw, events)
```

これで make_epoch という関数が出来た。以降は例えば

```
    make_epoch(4)
```

とかで event_id が 4 の epoch オブジェクトが返る。
これで君の怒りが少しでもおさまってくれたら嬉しい。

ここまでのまとめ

というわけで、凄く省略すれば、epoching まで下記のように書けるのです。

```
    import mne
    raw = mne.io.Raw('hoge').interpolate_bads().filter(1, 100).\
          notch_filter(60)
    make_epochs = partial(raw, mne.find_events(raw), tmin=-0.2, tmax=5.0)
    epochs = [make_epochs(n) for n in range(1,7)]
```

まぁ、ICA とか省いているから本当はもうちょっと長いです。