Chapter 1

The Algorithm

This section walks through the algorithm step by step, so as to explain which methods are used and how they are implemented. The idea is that the code from step `S00..` on can only accept one well defined structure of data. In earlier versions the approach was to write code that would adapt to different types of data automatically. All of this extra adaptivity turned out to visually and structurally clog the code more than it did offer much of a benefit. The concept was therefor reversed. `S00_prep_data` can be altered to produce required output. Yet, there should be no need to adapt any of the later steps in any way. All input parameters are to be set in `INPUT.m` and `INPUT`*xxx*`.m`.

1.1 Step S00: Prepare Data

`function S00_prep_data`

Before the actual eddy detection and tracking is performed, SSH-, latitude- and longitude-data is extracted from the given data at desired geo-coordinate bounds and saved as structures in the form needed by the next step (S01). This step also builts the file `window.mat` via `GetWindow3` which saves geometric information about the input and output data as well as a cross-referencing index-matrix which is used to reshape all *cuts* to the user defined geo-coordinate-geometry. The code can handle geo-coordinate input that crosses the longitudinal seam of the input data. E.g. say the input data comes in matrices that start and end on some (not necessarily strictly meridional) line straight across the Pacific and it is the Pacific only that is to be analyzed for eddies, the output maps are stitched accordingly. In the zonally continous case i.e. the full-longitude case, an *overlap* in x-direction across the *seam*-meridian of the chosen map is included so that contours across the seam can be detected and tracked across it. One effect is that eddies in proximity to the seam can get detected twice at both zonal ends of the maps. The redundant *ghost*-eddies get filtered out in `S05_track_eddies`.

1.2 Step S01: Find Contours

`function S01_contours`

The sole purpose of this step is to apply MATLAB's `contourc.m` function to the SSH data. It simply saves one file per time-step with all contour indices appended into one vector [1]. The contour intervals are determined by the user defined increment and range from the minimum- to the maximum of given SSH data.
The function `initialise.m`, which is called at the very beginning of every step, here has the purpose of rechecking the *cuts* for consistency and correcting the time-steps accordingly (i.e. when files are missing). `initialise.m` also distributes the files to the threads i.e. parallelization is in time dimension.

1.3 Step S01b: Find Mean Rossby Radii and Phase Speeds

`function S01b_BruntVaisRossby`

This function...

- – ...calculates the pressure $P(z,\phi)$ in order to...
  – ...calculate the Brunt-Väisälä-Frequency according to $N^2(S,T,P,\phi) = -\frac{g(\phi)}{P}\frac{\partial\rho(S,T,P)}{\partial z}$ in order to...
- – ...integrate the Rossby-Radius $\mathrm{L_R} = \frac{1}{\pi f}\int_H N\,\mathrm{d}z$ and ...

[1]see the MATLAB documentation.

- apply the long-Rossby-Wave dispersion relation to found L_R to estimate Rossby-Wave phase-speeds $c = -\frac{\beta}{k^2+(1/L_r)^2} \approx -\beta L_R{}^2$

The 3-dimensional matrices (S and T) are cut zonally into slices which then get distributed to the threads. This allows for matrix operations for all calculations which would otherwise cause memory problems due to the immense sizes of the 3d-data [2].

1.4 Step S02: Calculate Geostrophic Parameters

`function S02_infer_fields`
This step reads the cut SSH data from `S00_prep_data` to perform 2 steps:

1. Calculate a mean over time of $SSH(y, x)$.
2.
   - use one of the files' geo-information to determine f , β , g and the ratio g/f .
   - calculate geostrophic fields from SSH gradients.
   - calculate deformation fields (vorticity, divergence, stretch and shear) via the fields supplied by the last step.
   - calculate O_w .
   - Subtract the mean from step 1 from each $SSH(t)$ to filter out persistent SSH-gradients e.g. across the Gulf-Stream.

1.5 Step S04: Filter Eddies

`function S04_filter_eddies`
Since indices of all possible contour lines at chosen levels are available at this point, it is now time to subject each and every contour to a myriad of tests to decide whether it qualifies as the outline of an eddy as defined by the user input threshold parameters.

Reshape for Filtering and Correct out of Bounds Values

`function eddies2struct`
`function CleanEddies`
In the first step the potential eddies are transformed to a more sensible format, that is, a structure `Eddies(EddyCount)` where `EddyCount` is the number of all contours. The struct has fields for level, number of vertices, exact i.e. interpolated coordinates and rounded integer coordinates.
The interpolation of `contourc.m` sometimes creates indices that are either smaller than 0.5 or larger than $N + 0.5$ [3] for contours that lie along a boundary. After rounding, this seldomly leads to indices of either 0 or $N + 1$. These values get set to 1 and N respectively in this step.

Descent/Ascend Water Column and Apply Checks

`function walkThroughContsVertically`
The concept of this step is a direct adaption of the algorithm described by **?**. It is split into two steps, one for anti-cyclones and one for cyclones. Consider e.g. the anti-cyclone situation. Since all geostrophic anti-cyclones are regions of relative high pressure, all anti-cyclones [4] effect an elevated SSH i.e. a *hill*. The algorithm ascends the full range of SSH levels where contours were found. Consider an approximately Gaussian shaped AC that has a peak SSH of say 5 increments larger than the average surrounding waters. As the algorithm approaches the sea surface from below, it will eventually run into contours that are closed onto themselves and that encompass the AC. At first these contours might be very large and encompass not only one but several ACs and likely also cyclones, but as the algorithm continues upwards found contour will get increasingly circular, describing some outer *edge* of the AC. Once the contour and its interior pass all of the tests the algorithm will decide that an AC was found and write it and all its parameters to disk. The AC's region i.e. the interior of the contour will be flagged from here on. Hence any inner contour further up the water column will not pass the tests. Once all AC's are found for a given time-step, the SSH flags get reset and the entire procedure is repeated only this time *descending* the SSH-range to find cyclones. The tests for cyclones and anti-cyclones are identical except for a factor -1 where applicable. In the following the most important steps of the analysis are outlined.

[2] E.g. the pop data has dimensions $42 \times 3600 \times 1800$.
[3] where N is the domain size
[4] henceforth abbreviated AC

NaN-Check Contour

`function CR_RimNan`
The first and most efficient test is to check whether indices of the contour are already flagged. Contours within an already found eddy get thereby rejected immediately.

Sub-Window

`function get_window_limits`
`function EddyCut_init`
For further analysis a sub-domain around the eddy is cut out of the SSH data. These functions determine the indices of that window and subtract the resultant offset for the contour indices.

Logical Mask of Eddy Interiour

`function EddyCut_mask`
Basically this function creates a flood-fill logical mask of the eddy-interior. This is by far the most calculation intensive part of the whole filtering procedure. A lot more time was wasted on attempting to solve this problem more efficiently than time could have been saved would said attempts have been successful. The current solution is basically just MATLAB's `imfill.m`, which was also used in the very first version of 09/2013. EDIT: `imfill.m` was replaced by using `inpoly.m` to determine which indices lie within the contour-polygon. This method seems to be more exact at determining whether the inside-part of one grid cell (with respect to the smooth, spline-interpolated contour) is larger than the outside part or not.

Sense

`function CR_sense`
All of the interior SSH values must lie either above or below current contour level, depending on whether anti-cyclones or cyclones are sought.

Area

`function getArea`
The main goal here is to determine the most exact area encompassed by the exact coordinates of the contour. It does so via MATLAB's `polyarea` function. This area is not related to the scale σ that is determined in [section][5][1]1.5. It is however the relevant scale for [section][5][1]1.5. `function EddyCircumference`
Circumference e.g. line-length described by the contour. This is the other parameter needed for [section][5][1]1.5. This is however neither related to the actual eddy scale determined in [section][5][1]1.5.

Cirumference

`function EddyCircumference`
Line-length (sum of Euclidean norm of all nodes) of the contour.

Shape

`function CR_Shape`

This is the crucial part of deciding whether the object is *round enough*. A perfect vortex with $\frac{\partial u}{\partial y} = -\frac{\partial v}{\partial x}$ is necessarily a circle. The problem is that eddies get formed, die, merge, run into obstacles, get asymmetrically advected etc. To successfully track them it is therefor necessary to allow less circle-like shapes whilst still avoiding to e.g. count 2 semi merged eddies as one. This is achieved by calculating the isoperimetric quotient, defined as the ratio of a ring's area to the area of a circle with equal circumference. **?** use a similar method. They require:
The distance between any pair of points within the connected region must be less than a specified maximum (**?**).
While this method clearly avoids overly elongated shapes it allows for stronger deformation within its distance bounds.

e both,
ictures

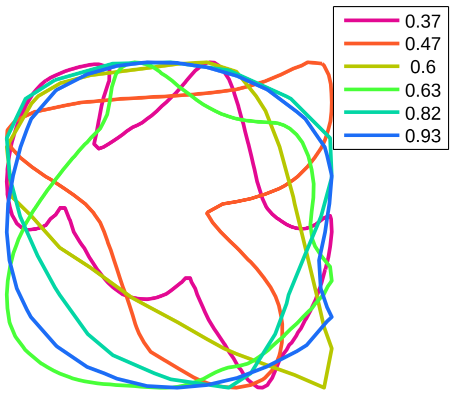

Figure 1.1: Different values of the isoperimetric quotient.

Amplitude

`function CR_AmpPeak`
This function determines the amplitude i.e. the maximum of the absolute difference between SSH and current contour level and the position thereof as well as the amplitude relative to the mean SSH value of the eddy interior as done by **?**. The amplitude is then tested against the user-given threshold. The function also creates a matrix with current contour level shifted to zero and all values outside of the eddy set to zero as well.

Chelton's Scales

`function cheltStuff`
? introduced 4 different eddy-scales.

1. The effective scale L_{eff} as the radius of a circle with its area equal to that enclosed by the contour.
2. The scale $L_e = L(z = \mathrm{e}^{-1}a)$ where a as the amplitude with reference to the original contour and the z-axis zero-shifted to that contour. In other words the effective scale of the contour that is calculated at $1/\mathrm{e}$ of the original amplitude.
3. The scale $L = L_e/\sqrt{2}$.
4. The scale L_s which is *a direct estimate based on the contour of SSH within the eddy interior around which the average geostrophic speed is maximum.* (**?**) It is hence conceptually the same as σ . This scale was not calculated here, as I could not think of an efficient simple way to estimate the area bounded by maximum geostrophic speed i.e. the zero-vorticity contour. To understand why this would be difficult to achieve see also sections... .

ref

pprop
s eg noise
etc

Profiles

`function EddyProfiles`
This step

- saves the meridional and zonal profiles of SSH, U and V through the eddie's peak and spanning the entire sub-domain as described in [section][5][1]1.5.
- creates spline functions from the ssh-profiles and uses them to interpolate the profiles onto 100-piece equi-distant coordinate vectors to buid smooth interpolated versions of ssh-profiles in both directions.
- in turn uses the splined data to create smooth 4-term Fourier-series functions for the profiles.

lots fouri-
fourier

Dynamic Scale (σ)

`function EddyRadiusFromUV`

The contour line that is being used to detect the eddy is not necessarily a good measure of the eddy's *scale* i.e. it doesn't necessarily represent the eddy's outline very well. This becomes very obvious when the area, as inferred by [section][5][1]1.5, is plotted over time for an already successfully tracked eddy. The result is not a smooth curve at all. This is so because at different time steps the eddy usually gets detected at different contour levels. Since its surrounding continuously changes and since the eddy complies with the testing-criteria the better the closer the algorithm gets to the eddy's peak value, the determined area of the contour jumps considerably between time steps. This is especially so for large flat eddies with amplitudes on the order of $1cm$. If the contour increment is on that scale as well, the difference in contour-area between two time steps easily surpasses 100% and more. Since there is no definition for the *edge* of an eddy, it is here defined as the ellipse resulting from the meridional and zonal diameters that are the distances between first minimum and maximum orbital velocity, away from the eddy's peak in positive and negative y and x directions respectively. In the case of a meandering jet with a maximum flow speed at its center, that is shedding off an eddy, this scale corresponds to half the distance between two opposing center-points of the meander. It is also the distance at which a change in vorticity-polarity occurs and is thus assumed to be the most plausible *dividing line* between vortices.

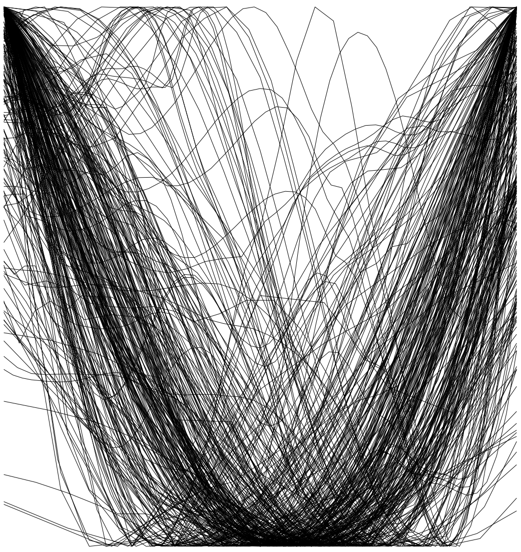
Figure 1.2: Zonal x- and z-normalized cyclone-profiles (early data $\sim$ '13/12).

Trying to determine the location where this sign change in vorticity occurs in the profiles turns out to be very tricky. What we seek are local extremata of the geostrophic speeds i.e. of the ssh-gradients h_x. In a perfect Gaussian-shaped eddy, these would simply correspond to the first local extremata of h_x away from the peak. In *reality* the eddies can be very wobbly with numerous local maxima and minima in the gradients of their flanks. One could argue, that it must be the largest extremata, as it is the highest geostrophic speeds that are sought. In practice [5] multiple superimposed signals of different scales often create very strong gradients locally. But the main issue here is that one weak eddy adjacent to one strong eddy also has the stronger gradients of the stronger one within its domain so that simply looking for the fastest flow speeds along the profiles is insufficient. It is also not possible to restrict the cut domain to the extent of a single eddy only, because at the point where the domain is selected, we do not know yet whether the detection algorithm *took bait* at the eddy's base or later close to the tip.

The best method thus far seems to be to use the Fourier-series functions from [section][5][1]1.5 to determine the first extremata away from the eddy's peak. The Fourier order was chosen to be 4 by trial and error. The effect is that small-scale low-amplitude noise is avoided, allowing for more reliable determinations of $\nabla^2 h_{fou} = 0$ Once the zero crossings in all 4 directions are found, their mean is taken as the eddy's scale.

Dynamic Amplitude

`function EddyAmp2Ellipse`

As mentioned above, the contour that helps to detect the eddy is not representative of its extent. This is also true for the z-direction, for the same reasons. This function therefor takes an SSH-mean at indices of the ellipse created by the determined zonal and meridional *dynamical* diameters, and uses this as the basal value to determine a *dynamic* amplitude.

Center of Volume

`function CenterOfVolume`

Instead of using the geo-position of the eddy's peak in the tracking procedure, it was decided to instead use the center of the volume created by the basal shifted matrix from [section][5][1]1.5 i.e. *the center of volume of the dome (resp. valley) created by*

[5]especially for the high-resolution model data.

capping off the eddy at the contour level. This method was chosen because from looking at animations of the tracking procedure it became apparent that, still using peaks as reference points, the eddy sometimes jumped considerably from one time step to the next if two local maxima existed within the eddy. E.g. in one time-step local maximum A might be just a little bit larger than local maximum B and one time-step later a slight shift of mass pushes local maximum B in pole position, creating a substantial jump in the eddy-identifying geo-position hence complicating the tracking procedure.

Geo Projection

`function ProjectedLocations`
An optional threshold on the distance an eddy is allowed to travel over one time-step is implemented in the tracking algorithm [section][5][1]1.5. This is a direct adaptation of the ellipse-based constraint described by **?**. The maximum distance in western direction traveled by the eddy within one time-step is limited according to $x_{west} = \alpha c \delta t$ where c as the local long-Rossby-wave phase-speed and e.g. $\alpha = 1.75$. In eastern direction the maximum is fixed at a value of e.g. $x_{east} = 150$km. This value is also used to put a lower bound on x_{west} and for half the minor axis (y-direction) of the resultant ellipse.

1.6 Step S05: Track Eddies

`function function S04_track_eddies`
Due to the the relatively fine temporal resolution (daily) of the model data, the tracking procedure turns out to be much simpler than the one described by **?**. There is really no need to project the new position of an eddy, as it generally does not travel further than its own scale in one day. This means that one eddy can be unambigiously tracked from one time step to the next as long both time-steps agree on which eddy from the other time-step is located the least distance away. The algorithm therefor simply builds an arc-length-distance matrix between all old and all new eddies and then determines the minima of that matrix in both directions i.e. one array for the new with respect to the old, and one for the old with respect to the new set. This leads to the following possible situations:

- Old and new agree on a pair. I.e. old eddy O_a has a closest neighbour in the new set N_a and N_a agrees that O_a is the closest eddy from the old set. Hence the eddy is tracked. N_a is O_a at a later time.

- N_a claims O_a to be the closest, but N_b makes the same claim. I.e. two eddies from the new set claim one eddy from the old set to be the closest. In this situation the closer one is decided to be the old one at a later time-step and the other one must be a newly formed eddy.

- At this point all new eddies are either allocated to their respective old eddies or assumed to be *newly born*. The only eddies that have not been taken care of are those from the old set, that *lost* ambigious claims to another old eddy, that was closer to the same claimed new eddy. I.e. there is no respective new eddy available which must mean that the eddy just *died*. In this case the entire track with all the information for each time step is saved as long as the track-length meets the threshold criterium. If it doesn't, the track is deleted.

1.7 Step S05: Cross Reference Old to New Indices

```
function function S05_init_output_maps
```

The main purpose of this step is to allocate all grid nodes of the input data to the correct node of the output map. Since the output map is usually much coarser than the input data there is no need for interpolation.

1.8 Step S06: Make Maps of Mean Parameters

```
function function S06_analyze_tracks
```

1.9 Example

To show a hands-on example an exemplary run for part of the North Atlantic is demonstrated in the following.

1.9.1 Map Parameters

The geo-coordinates for the map window are to be set in `map_vars.m`. In this case a small region in the eddy-rich North Atlnatic from $-90°$ west till $-40°$ west and from $20°$ north till $60°$ north is chosen. This script has an effect only on the very first step (`S_00..`).

```
function MAP=map_vars
    %% user input
    MAP.geo.west=-90;
    MAP.geo.east=-40;
    MAP.geo.south=20;
    MAP.geo.north=60;
    MAP.time.delta_t = 1; % [days]
    MAP.SSH_unitFactor = 100; % eg 100 if SSH data in cm, 1/10 if in deka m etc..
    MAP.pattern.in='SSH_GLB_t.t0.1_42l_CORE.yyyymmdd.nc';
end
```

show map

1.9.2 Other Parameters

All other parameters are to be set in `input_vars.m`.

- The maximum number of *local workers* (threads) is at default settings usually limited to 12 by Matlab. Hence

```
    U.threads.num=12;
```

- The data available at this point in time spans from 1994/04/02 till 2006/12/31. Hence

```
        U.time.from.str='19940402';
        U.time.till.str='20061231';
```

- Data is available per day. Therefor `U.time.delta_t=1;` is set to 1. This could also be changed to another value, if the user wished to skip time-steps.
- The path `U.path.root='../dataXMPL/';` can be set arbitrarily. If it does not yet exist, it will be created. It is suggested to not reuse a path, as this could lead to inconsitencies in the data.

.TempSalt

- `.name='TempSalt/';`
- `U.path.raw.name='/scratch/uni/ifmto/u241194/DAILY/EULERIAN/SSH/';`
  This is where the SSH data is stored.

```
        U.contour.step=0.01; % [SI]
        U.thresh.radius=5e3; % [SI]
        U.thresh.amp=0.02; % [SI]
        U.thresh.shape.iq=0.5; % isoperimetric quotient
        U.thresh.shape.chelt=0.5; % (diameter of circle with equal area)/(maximum
distance between nodes) (if ~switch.IQ)
        U.thresh.corners=6; % min number of data points for the perimeter of an eddy
        U.thresh.dist=1*24*60^2; % max distance travelled per day
        U.thresh.life=20; % min num of living days for saving
```

In this segment the following values are set (all in SI-units, except for time dimension).:

– The contour intervall with which to look for SSH-contours from maximum to minimum SSH-value.

– The minimum radius threshold for an eddy.

– The minimum amplitude threshold.

– Either the minimum IQ value or the minimum ratio of the diameter of a circle with equal area over the maximum distance between nodes, depending on which method is chosen (see).

– The minimum number of grid nodes making up the contour.

– The maximum distance-travelled-per-timestep threshold.

– The minium track-length (in time) threshold for a track to be saved.

```
    U.dim.X=40*1+1;
    U.dim.Y=40*1+1;
    U.dim.west=-90;
    U.dim.east=-50;
    U.dim.south=20;
    U.dim.north=60;
    U.dim.NumOfDecimals=1;
```

These paramters describe the output maps, that will be created in the very end (e.g. maps of mean values). Ideally they are set to the same limits as those set in [subsection][1][1,9]1.9.1. `U.dim.X`/`U.dim.Y` dictate the size of the output maps. They can be set arbitrarilly, but the format e.g. $(east-west)*n+1; \quad n \in \mathrm{N}$ is suggested to avoid long decimals in the coordinate matrices.

-
  ```
  U.switchs.RossbyStuff=false;
  U.switchs.IQ=true;
  ```

Choose whether step `S01b_BruntVaisRossby` is to be run (needs adequate salt and temperature files) and the type of shape-testing (`U.switchs.IQ=true`) for IQ -method.

- The rest are less important technical things that are only relevant if the code itself is modulated (e.g. if modules are added).

1.9.3 Running the Code

The seperate steps can be run all at once via `Sall.m` or one by one, as long as they are started consecutively in the order indicated by their name (`S00..`, then `S01..` etc.). `S01b` is not necessary though. Each step saves its own files which are then read by the next step. All output data is saved in the user given

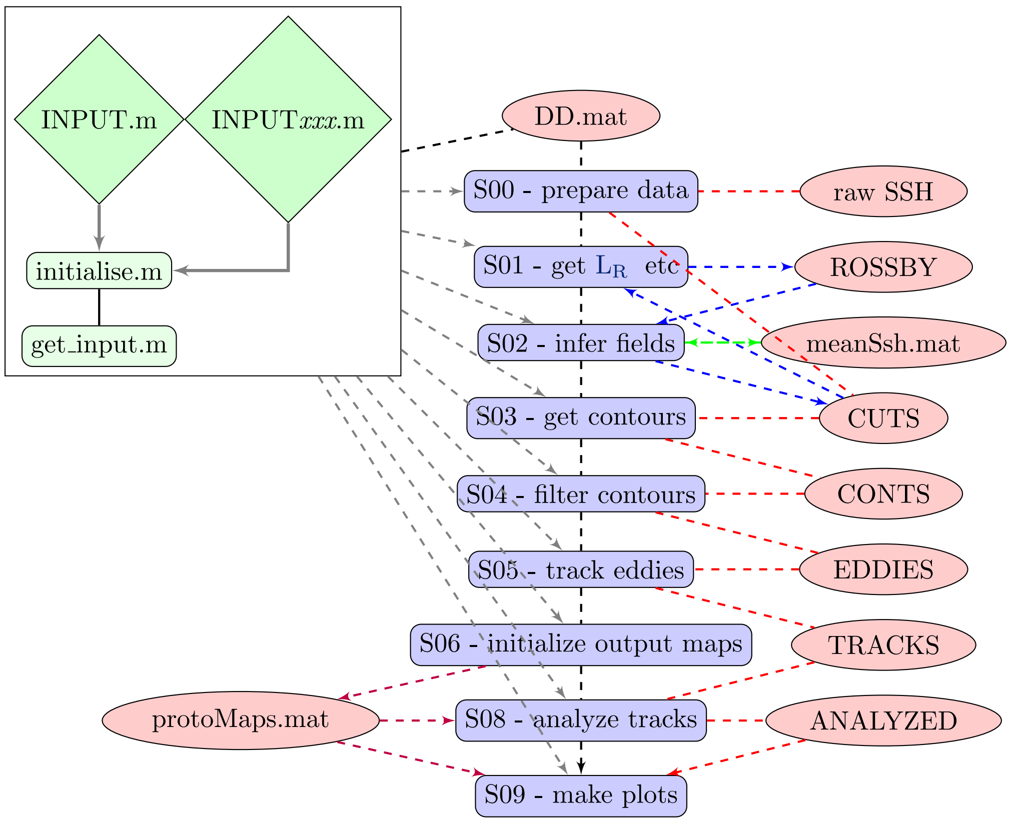

Figure 1.3: Basic code structure. The only files that are to be edited are the INPUT files. INPUT.m is independant of the origin of data, whilst the files INPUTaviso.m, INPUTpop.m etc set source-specific parameters. Each of the SXX-steps initially calls initialise.m, which in turn scans all available data, reads in the INPUT data via get_input.m, corrects for missing data etc and creates DD.mat. The latter is the main meta-data file, which gets updated throughout all steps. All data is built step-by-step along the consecutive SXX-steps (red line). The SXX-steps are the only programs that have to be called (in order) by the user. Missing data is filled automatically in each step (Note that meanSsh.mat should be recalculated if the time span is changed!).

root-path from [subsection][2][1,9]1.9.2. This concept uses quite a lot of disk space and is also quite substantially slowed by all the reading and writing procedures. The benefits, on the other hand, are that debugging becomes much easier. If the code fails at some step, at least all the calculations up to that step are dealt with and do not need to be re-run. The concept also makes it easy to extend the code by further add-ons.

plots to fol

see github for progress