Problem Set 2

⚠ Warning

Functions in this report are experimental, not tested, and may change without notice.

1 Introduction

In this report, I will demonstrate the application of Julia in space data analysis. Due to its highly flexible type system, **multiple dispatch** feature, and seamless interoperability between Python and C packages, Julia enables the implementation of `SPEDAS` complex functionalities in a concise and generalizable manner, often requiring only a few lines of code.

More specifically, we utilize

- `DimensionalData` : which provides an abstract array with named dimensions, facilitating more intuitive indexing and generalized manipulation.
- `Unitful` : which enables seamless unit conversion and supports dimensional analysis

Since Julia features an **abstract type system**, most packages can be used directly and composed seamlessly without unintended side effects. This contrasts with Python, where inheritance and subtyping are commonly used, making it more challenging to share functionality across different classes.

`degap` and `rectify_datetime` are used to clean up the data and rectify the time series (make timestamps uniform).

```julia
function degap(da::DimArray; dim=Ti)
    dims = otherdims(da, dim)
    rows = filter(x -> !any(isnan, x), eachslice(da; dims))
    if !isempty(rows)
        cat(rows...; dims)
    else
        similar(da, (0, size(da, 2)))
    end
end

function degap(ts::TimeArray)
    ts[all.(!isnan, eachrow(values(ts)))]
end

function rectify_datetime(da; tol=2, kwargs...)
    times = dims(da, Ti)
    t0 = times[1]
```

```
    dtime = Quantity.(times.val .- t0)
    new_times = TimeseriesTools.rectify(Ti(dtime); tol)[1]
    set(da, Ti => new_times .+ t0)
end
```

`tplot` could be decomposed into multiple steps and lead to better readability and flexibility.

```
"""
Lay out multiple time series on the same figure across different panels (rows)
"""
function  tplot(f,  tas::AbstractVector;  add_legend=true,  link_xaxes=true,
kwargs...)
    aps = map(enumerate(tas)) do (i, ta)
        ap = tplot(f[i, 1], ta; kwargs...)
        # Hide redundant x labels
       link_xaxes && i != length(tas) && hidexdecorations!(ap.axis, grid=false)
        ap
    end
    axs = map(ap -> ap.axis, aps)
    link_xaxes && linkxaxes!(axs...)
    add_legend && axislegend.(axs)
    FigureAxes(f, axs)
end

"""
Setup the axis on a position and plot multiple time series on it
"""
function tplot(gp::GridPosition, tas::AbstractVector; kwargs...)
    ax = Axis(gp, ylabel=ylabel(ta))
    plots = map(tas) do ta
        tplot!(ax, ta; kwargs...)
    end
    ax, plots
end

"""
Plot a multivariate time series on a position in a figure
"""
function  tplot(gp::GridPosition,  ta::AbstractDimArray;  labeldim=nothing,
kwargs...)
    args, attributes = _series(ta, kwargs, labeldim)
    series(gp, args...; attributes...)
end
```

Note

Some functions in this report have been collected into multiple Julia packages Speasy.jl and SpaceTools.jl, please refer to the package for more information.

2 Energy input and energy dissipation

Compute the total energy input and total energy dissipation in the magnetosphere during a storm, the one which occurred on 17 March, 2015. This was the largest storm of the previous solar cycle. The total energy input rate is: $\varepsilon[W] = (4\pi/\mu_0)VB^2 \sin^4(\theta/2)I_0^2$, also widely known as the Akasofu "epsilon" parameter1,2,3,4,5,6, with $\theta = acos(B_{z,GSM}/B_{yz,GSM})$, $I_0 = 7RE$. Its cumulative integral is: $U_{in} = \int \varepsilon dt$ [in PetaJoules]. The magnetospheric energy dissipation rate in the ionosphere and ring current (in J/s or W) is: $W_{md} = [410^{13}(\partial(-Dst*)/\partial t + (-Dst*)/\tau_R) + 300AE]$ and its integral is given by: $U_{md} = \int W_{md} dt$ [in PJ], where: $\tau_R = 1$ hr is the ring-current decay rate of O+ through charge exchange in the beginning of the storm, and $\tau_R = 6$ hr late in the storm recovery when H+ is the dominant species, and Dst* is the corrected Dst to account for SW pressure variations6. Compute these quantities by following the crib sheet. Scale factors are included. Translate as needed (in Python) and complete (below lines: "; CONSTRUCT…") the crib sheet EPSS_Hwk02.1_crib.pro. Produce the plot below. In your answer, explain each panel in the plot.

2.1 Define and load data sets

We write a simple Julia wrapper around the Speasy, a Python package to deal with main Space Physics WebServices using API instead of downloading files. This allows easier integration between Python and Julia.

```julia
abstract type AbstractDataSet end

@kwdef struct DataSet <: AbstractDataSet
    name::String
    parameters::Vector{String}
end

speasy() = @pyconst(pyimport("speasy"))

struct SpeasyVariable
    py::Py
end

function get_data(args...)
    res = @pyconst(pyimport("speasy").get_data)(args...)
    return apply_recursively(res, SpeasyVariable, is_pylist)
end
```

```julia
function        load_dataset(dataset,        args...;        name=dataset.name,
products=dataset.parameters, provider="cda")
    map(products) do p
        replace_fillval_by_nan!(get_data("$provider/$name/$p", args...))
    end
end
```

```
load_dataset (generic function with 1 method)
```

```julia
OMNI_HRO_PARAMS = [
    "Vx", "Vy", "Vz", "flow_speed",
    "BX_GSE", "BY_GSM", "BZ_GSM", "E",
    "AE_INDEX", "SYM_H", "Pressure"
]

"""
High resolution (1-min), multi-source, near-Earth solar wind magnetic field and
plasma data as shifted to Earth's bow shock nose, plus several 1-min geomagnetic
activity indices.

References:[DOI](https://doi.org/10.48322/45bb-8792)
"""
OMNI_HRO_1MIN = DataSet("OMNI_HRO_1MIN", OMNI_HRO_PARAMS)

"""
Version 2 of OMNI_HRO_1MIN dataset

- [DOI](https://doi.org/10.48322/mj0k-fq60)
"""
OMNI_HRO2_1MIN = DataSet("OMNI_HRO2_1MIN", OMNI_HRO_PARAMS)

OMNI2_H0_MRG1HR = DataSet(
    "OMNI2_H0_MRG1HR",
    ["KP1800", "DST1800", "AE1800"]
)

timespan = ["2015-03-15", "2015-03-22"] # 7 days
omni_hro_ds = load_dataset(OMNI_HRO_1MIN, timespan) |> TimeArray
OMNI2_H0_MRG1HR_ds = load_dataset(OMNI2_H0_MRG1HR, timespan) |> TimeArray
```

```
Can't get OMNI2_H0_MRG1HR/KP1800 without web service, switching to web service
Can't get OMNI2_H0_MRG1HR/DST1800 without web service, switching to web service
Can't get OMNI2_H0_MRG1HR/AE1800 without web service, switching to web service
```

```
168×3 TimeSeries.TimeArray{Float64, 2, Dates.DateTime, Matrix{Float64}}
2015-03-15T00:30:00 to 2015-03-21T23:30:00
┌─────────────────────┬────────┬─────────┬────────┐
│                     │ KP1800 │ DST1800 │ AE1800 │
├─────────────────────┼────────┼─────────┼────────┤
│ 2015-03-15T00:30:00 │   20.0 │   -11.0 │  159.0 │
│ 2015-03-15T01:30:00 │   20.0 │   -11.0 │  147.0 │
│ 2015-03-15T02:30:00 │   20.0 │   -11.0 │  159.0 │
│ 2015-03-15T03:30:00 │   20.0 │   -12.0 │  170.0 │
│ 2015-03-15T04:30:00 │   20.0 │   -11.0 │  232.0 │
│ 2015-03-15T05:30:00 │   20.0 │   -11.0 │  235.0 │
│ 2015-03-15T06:30:00 │   27.0 │    -9.0 │  317.0 │
│ 2015-03-15T07:30:00 │   27.0 │    -9.0 │  446.0 │
│          ⋮          │   ⋮    │    ⋮    │   ⋮    │
│ 2015-03-21T17:30:00 │   17.0 │   -44.0 │  105.0 │
│ 2015-03-21T18:30:00 │   17.0 │   -44.0 │   84.0 │
│ 2015-03-21T19:30:00 │   17.0 │   -44.0 │   72.0 │
│ 2015-03-21T20:30:00 │   17.0 │   -47.0 │   68.0 │
│ 2015-03-21T21:30:00 │   13.0 │   -40.0 │   94.0 │
│ 2015-03-21T22:30:00 │   13.0 │   -43.0 │   55.0 │
│ 2015-03-21T23:30:00 │   13.0 │   -41.0 │   47.0 │
└─────────────────────┴────────┴─────────┴────────┘
                                   153 rows omitted
```

2.2 Akasofu parameter

```
function Akasofu_epsilon(B, V)
    _, By, Bz = B
    I0 = 7 * Re
    Bt = norm([By, Bz])
    θ = acos(Bz / Bt)
    return (4π / μ0) * V * Bt^2 * sin(θ / 2)^4 * I0^2 |> u"GW"
end

B_ts = omni_hro_ds[:BX_GSE, :BY_GSM, :BZ_GSM] .* u"nT"
V_ts = omni_hro_ds[:flow_speed] .* u"km/s"

Akasofu_epsilon_meta = Dict(
    "label" => "Akasofu Epsilon"
)
Akasofu_epsilon_ts = TimeArray(
    timestamp(omni_hro_ds),
    Akasofu_epsilon.(eachrow(values(B_ts)), values(V_ts)),
    [:ε],
    Akasofu_epsilon_meta
)
```

```
10080×1 TimeSeries.TimeArray{Unitful.Quantity{Float64, ?² ? ?⁻³,
Unitful.FreeUnits{(GW,), ?² ? ?⁻³, nothing}}, 1, Dates.DateTime,
Vector{Unitful.Quantity{Float64, ?² ? ?⁻³, Unitful.FreeUnits{(GW,), ?² ?
?⁻³, nothing}}}} 2015-03-15T00:00:00 to 2015-03-21T23:59:00
┌─────────────────────┬──────────────┐
│                     │ ε            │
├─────────────────────┼──────────────┤
│ 2015-03-15T00:00:00 │    53.202 GW │
│ 2015-03-15T00:01:00 │   59.3266 GW │
│ 2015-03-15T00:02:00 │   62.2223 GW │
│ 2015-03-15T00:03:00 │   66.7341 GW │
│ 2015-03-15T00:04:00 │   66.7664 GW │
│ 2015-03-15T00:05:00 │    64.882 GW │
│ 2015-03-15T00:06:00 │   67.7737 GW │
│ 2015-03-15T00:07:00 │   66.3617 GW │
│          ⋮          │      ⋮       │
│ 2015-03-21T23:53:00 │       NaN GW │
│ 2015-03-21T23:54:00 │       NaN GW │
│ 2015-03-21T23:55:00 │       NaN GW │
│ 2015-03-21T23:56:00 │ 0.00733769 GW │
│ 2015-03-21T23:57:00 │  0.0025041 GW │
│ 2015-03-21T23:58:00 │  0.0070985 GW │
│ 2015-03-21T23:59:00 │ 0.00196921 GW │
└─────────────────────┴──────────────┘
                    10065 rows omitted
```

2.3 Dst correction

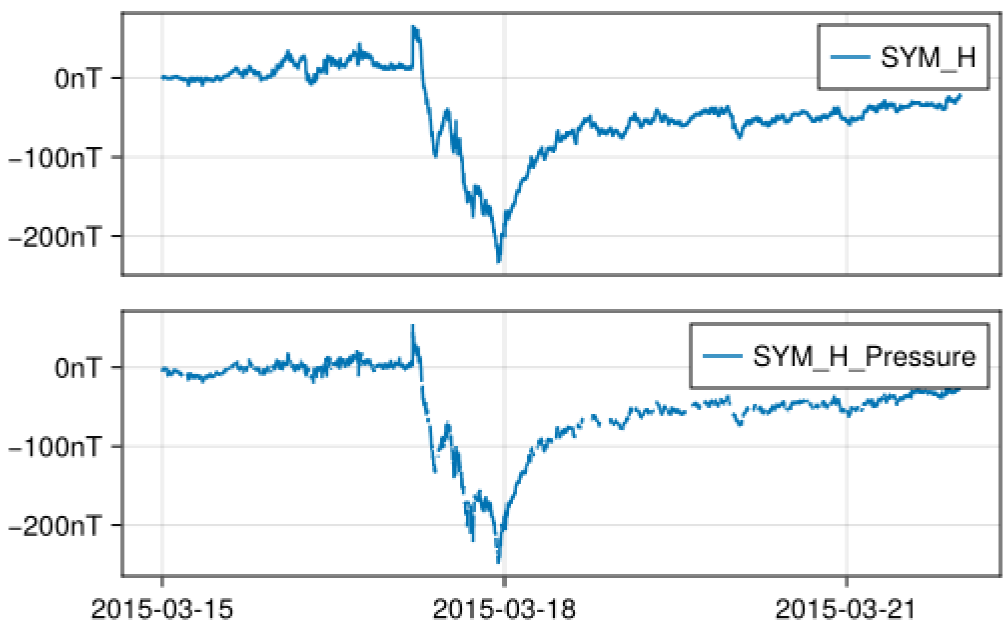
Siscoe etal 1968, JGR found deltaDst = constant*(sqrt(P_after)-sqrt(P_before)) for after/before sudden impulse. We use this here to correct Dst for SW dynamic pressure, relative to prestorm value (Dst=0, Pdyn=2nPa).

```
function correct_Dst(dst, P_after, P_before)
    Siscoe_constant = 13.5u"nT/sqrt(nPa)"
      @. (dst - Siscoe_constant * (sqrt(P_after * u"nPa") - sqrt(P_before * u"nPa")))
end

omni_hro_symh = omni_hro_ds[:SYM_H] .* u"nT"
omni_Dst_corrected = correct_Dst(omni_hro_symh, omni_hro_ds[:Pressure], 2)
```

```
10080×1 TimeSeries.TimeArray{Unitful.Quantity{Float64, ? ?⁻¹ ?⁻²,
Unitful.FreeUnits{(nT,), ? ?⁻¹ ?⁻², nothing}}, 1, Dates.DateTime,
Vector{Unitful.Quantity{Float64, ? ?⁻¹ ?⁻², Unitful.FreeUnits{(nT,), ? ?⁻¹
?⁻², nothing}}}} 2015-03-15T00:00:00 to 2015-03-21T23:59:00
┌─────────────────────┬────────────────┐
```

```
│                     │ SYM_H_Pressure │
├─────────────────────┼────────────────┤
│ 2015-03-15T00:00:00 │    -3.42358 nT │
│ 2015-03-15T00:01:00 │    -5.36862 nT │
│ 2015-03-15T00:02:00 │    -3.73869 nT │
│ 2015-03-15T00:03:00 │     -3.9673 nT │
│ 2015-03-15T00:04:00 │    -3.53824 nT │
│ 2015-03-15T00:05:00 │    -3.13192 nT │
│ 2015-03-15T00:06:00 │    -2.88453 nT │
│ 2015-03-15T00:07:00 │    -2.84302 nT │
│          ⋮          │       ⋮        │
│ 2015-03-21T23:53:00 │         NaN nT │
│ 2015-03-21T23:54:00 │         NaN nT │
│ 2015-03-21T23:55:00 │         NaN nT │
│ 2015-03-21T23:56:00 │    -23.2479 nT │
│ 2015-03-21T23:57:00 │    -23.2479 nT │
│ 2015-03-21T23:58:00 │    -22.5591 nT │
│ 2015-03-21T23:59:00 │    -22.5591 nT │
└─────────────────────┴────────────────┘
                        10065 rows omitted
```

```
tplot([omni_hro_symh, omni_Dst_corrected])
```



2.4 Electric field

```
omni_mVBz = @. -V_ts * B_ts.BZ_GSM |> u"mV/m"
```

```
10080×1   TimeSeries.TimeArray{Unitful.Quantity{Float64,   ?   ?   ?⁻¹   ?⁻³,
Unitful.FreeUnits{(m⁻¹, mV), ? ? ?⁻¹ ?⁻³, nothing}}, 1, Dates.DateTime,
Vector{Unitful.Quantity{Float64, ? ? ?⁻¹ ?⁻³, Unitful.FreeUnits{(m⁻¹, mV),
? ? ?⁻¹ ?⁻³, nothing}}}} 2015-03-15T00:00:00 to 2015-03-21T23:59:00
┌─────────────────────┬───────────────────┐
│                     │ flow_speed_BZ_GSM │
├─────────────────────┼───────────────────┤
│ 2015-03-15T00:00:00 │   0.342721 mV m⁻¹ │
│ 2015-03-15T00:01:00 │    0.65641 mV m⁻¹ │
│ 2015-03-15T00:02:00 │   0.711816 mV m⁻¹ │
│ 2015-03-15T00:03:00 │    0.72822 mV m⁻¹ │
│ 2015-03-15T00:04:00 │    0.74763 mV m⁻¹ │
│ 2015-03-15T00:05:00 │   0.718425 mV m⁻¹ │
│ 2015-03-15T00:06:00 │   0.750825 mV m⁻¹ │
│ 2015-03-15T00:07:00 │     0.7488 mV m⁻¹ │
│          ⋮          │         ⋮         │
│ 2015-03-21T23:53:00 │        NaN mV m⁻¹ │
│ 2015-03-21T23:54:00 │        NaN mV m⁻¹ │
│ 2015-03-21T23:55:00 │        NaN mV m⁻¹ │
│ 2015-03-21T23:56:00 │   -6.13815 mV m⁻¹ │
│ 2015-03-21T23:57:00 │   -6.17108 mV m⁻¹ │
│ 2015-03-21T23:58:00 │   -6.22842 mV m⁻¹ │
│ 2015-03-21T23:59:00 │   -6.19553 mV m⁻¹ │
└─────────────────────┴───────────────────┘
                                10065 rows omitted
```

3 Cumulative energy input and dissipation

```
function integrate(ts)
    ts = degap(ts)
    ε = values(ts)
    times = timestamp(ts)
    dts = Quantity.(diff(times))
    ∫ε_dt = cumsum(ε[1:end-1] .* dts) .|> u"PJ"
    Uin_meta = Dict(
        "label" => "Cumulative energy input"
    )
    TimeArray(times[2:end], ∫ε_dt, [:Uin], Uin_meta)
end

tau_r(t, t0; τr_i=1u"hr", τr_f=6u"hr") = τr_i * (t < t0) + τr_f * (t >= t0)

function compute_Wmd(dDst_dt, Dst, AE, time, time2transition)
    # Time-varying tau_r (assume split between initial and late storm)
```

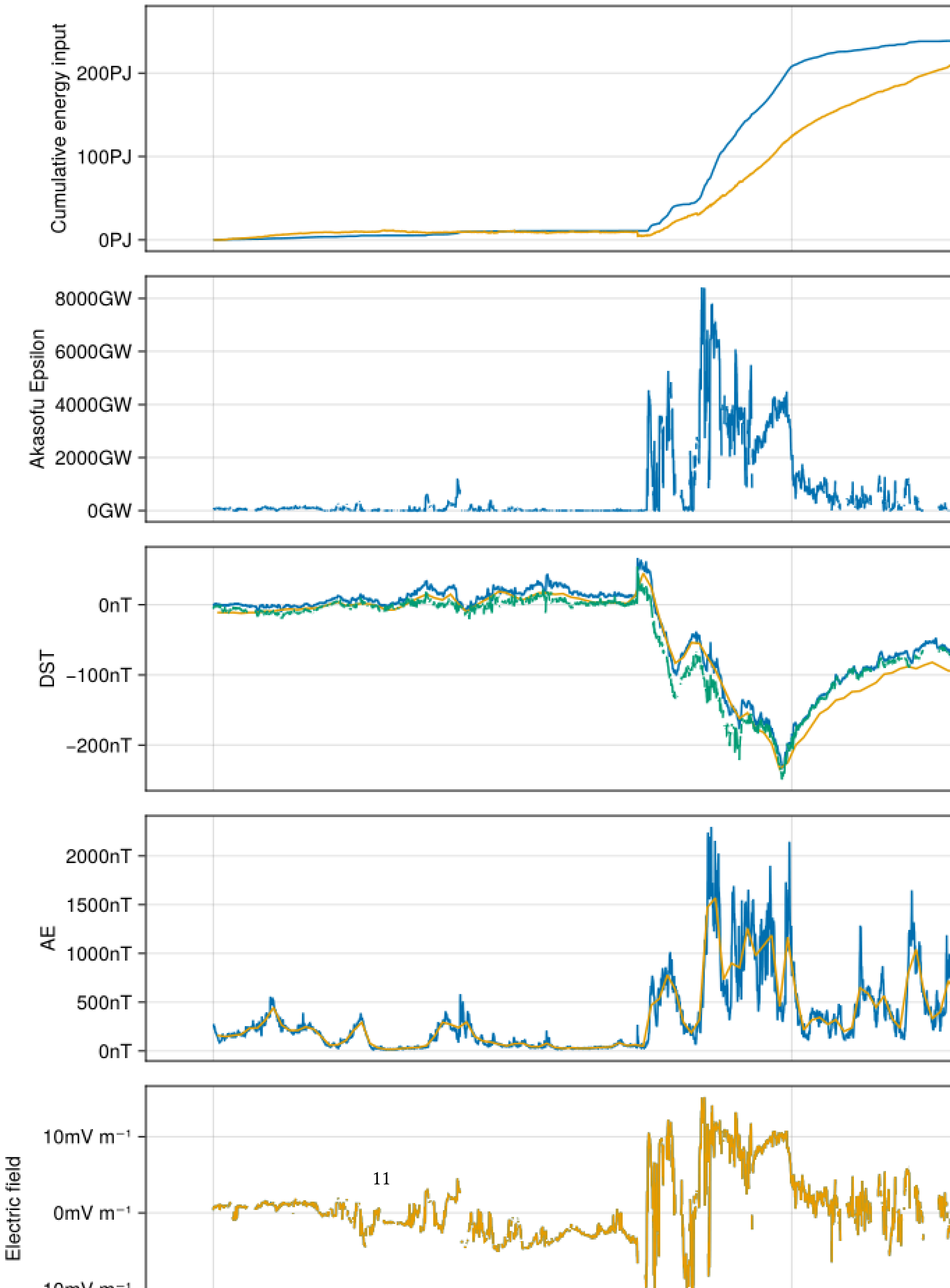
```
    τ_r = tau_r(time, time2transition)
    factor = u"erg/s" / u"nT"
    Wmd_dDstodt = -4e20 * dDst_dt * factor * 1u"s"
    Wmd_dstotau = -4e20 * Dst / τ_r * factor * 1u"s"
    Wmd_AE = 3e15 * AE * factor
    Wmd_all = Wmd_dDstodt + Wmd_dstotau + Wmd_AE
    # return (; Wmd_dDstodt, Wmd_dstotau, Wmd_AE, Wmd_all)
    return Wmd_all
end

item(ts) = values(ts)[1]

omni_AE = omni_hro_ds.AE_INDEX .* u"nT"

Wmd_all_ts = let Dst = omni_Dst_corrected, AE = omni_AE, time2transition =
Date("2015-03-19")
    times = timestamp(Dst)
    dts = Quantity.(diff(times))
    dDst_dt = diff(Dst) ./ dts
    time = times[10]
    # ([dDst_dt[times], Dst[times], AE[times]])
    Wmd_all = map(times[2:end]) do time
            compute_Wmd(item.([dDst_dt[time], Dst[time], AE[time]])..., time,
time2transition)
    end
    TimeArray(times[2:end], Wmd_all, [:Wmd], Dict("label" => "Cumulative energy
dissipation"))
end

Uin = integrate(Akasofu_epsilon_ts)
Uout = integrate(Wmd_all_ts)
rename!(Uout, :Uout)
```

```
6338×1   TimeSeries.TimeArray{Unitful.Quantity{Float64,   ⍰²   ⍰   ⍰⁻²,
Unitful.FreeUnits{(PJ,),  ⍰²  ⍰  ⍰⁻²,  nothing}},  1,  Dates.DateTime,
Vector{Unitful.Quantity{Float64, ⍰² ⍰ ⍰⁻², Unitful.FreeUnits{(PJ,), ⍰² ⍰
⍰⁻², nothing}}}} 2015-03-15T00:02:00 to 2015-03-21T23:59:00
┌─────────────────────┬──────────────┐
│                     │ Uout         │
├─────────────────────┼──────────────┤
│ 2015-03-15T00:02:00 │ 0.0863126 PJ │
│ 2015-03-15T00:03:00 │  0.028522 PJ │
│ 2015-03-15T00:04:00 │ 0.0449733 PJ │
│ 2015-03-15T00:05:00 │ 0.0346158 PJ │
│ 2015-03-15T00:06:00 │  0.024771 PJ │
│ 2015-03-15T00:07:00 │ 0.0211541 PJ │
│ 2015-03-15T00:08:00 │ 0.0256013 PJ │
```

```
│ 2015-03-15T00:09:00 │ 0.0127587 PJ │
│          ⋮          │      ⋮       │
│ 2015-03-21T23:48:00 │   267.107 PJ │
│ 2015-03-21T23:49:00 │    267.11 PJ │
│ 2015-03-21T23:50:00 │   267.073 PJ │
│ 2015-03-21T23:51:00 │    267.09 PJ │
│ 2015-03-21T23:57:00 │   267.109 PJ │
│ 2015-03-21T23:58:00 │   267.112 PJ │
│ 2015-03-21T23:59:00 │   267.088 PJ │
└─────────────────────┴──────────────┘
                    6323 rows omitted
```

3.1 Plot

```
fig = Figure(; size=(1200, 1200))
tvars2plot = [
    [Uin, Uout],
    Akasofu_epsilon_ts,
    [omni_hro_symh, OMNI2_H0_MRG1HR_ds.DST1800 .* u"nT", omni_Dst_corrected],
    [omni_AE, OMNI2_H0_MRG1HR_ds.AE1800 .* u"nT"],
    [omni_mVBz, omni_hro_ds.E .* u"mV/m"],
    [omni_hro_ds.Pressure .* u"nPa"]
]
f, axs = tplot(fig, tvars2plot)
axs[3].ylabel = "DST"
axs[4].ylabel = "AE"
axs[5].ylabel = "Electric field"
axs[6].ylabel = "Pressure"
axislegend.(axs)
f
```


Cumulative energy input
200PJ
100PJ
0PJ
Akasofu Epsilon
8000GW
6000GW
4000GW
2000GW
0GW
DST
0nT
−100nT
−200nT
AE
2000nT
1500nT
1000nT
500nT
0nT
Electric field
10mV m⁻¹
0mV m⁻¹

4 Field line resonances

Obtain magnetic field data from THEMIS-A for a field line resonances observed on 2008-Sep-05 10 – 20 UT. These have periods of 10-30 mHz and can be seen in Figure 1 of Sarris et al., 2010 [1]. It would be sufficient to use spin-period (FGS) data.

i. Show the band-bass filtered data between fmin=1/180s and fmax=1/15s (low-pass using block average function tsmooth2 with window 61 points, subtract it from the original data to get the high-pass, then do tsmooth2 on that with 5 points). Plot the data.

```
tha_l2_fgm_ds = DataSet("THA_L2_FGM", ["tha_fgs_gse"])
tspan = ["2008-09-05T10:00:00", "2008-09-05T22:00:00"]
tha_fgs_gse = load_dataset(tha_l2_fgm_ds, tspan) |> TS

tha_fgs_gse.metadata["long_name"] = "THA FGS GSE"
tha_fgs_gse.metadata["units"] = "nT"
```

```
"nT"
```

The `smooth` (`tsmooth2` in IDL) function is implemented as follows. By using `mapslices`, the function efficiently applies the operation along the desired dimension. This approach is very general and can be used for multiple dimensions.

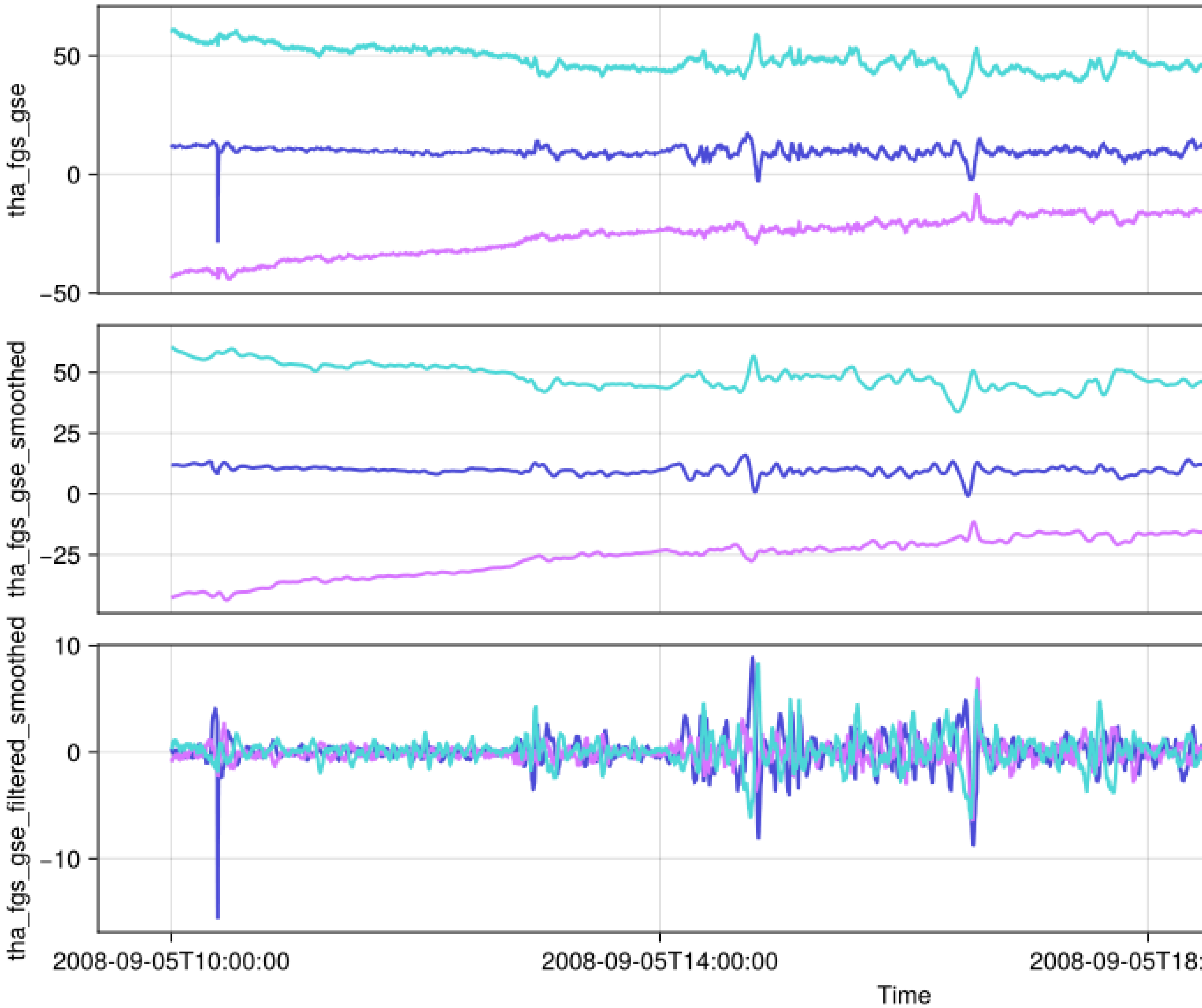
```
function smooth(da::AbstractDimArray, span::Integer; dims=Ti,
suffix="_smoothed", kwargs...)
    new_da = mapslices(da; dims) do slice
        mean.(RollingWindowArrays.rolling(slice, span; kwargs...))
    end
    rebuild(new_da; name=Symbol(da.name, suffix))
end
```

```
function amap(f, a::AbstractDimArray, b::AbstractDimArray)
    shared_selectors = DimSelectors(b)
    data = f(a[shared_selectors], b[shared_selectors])
end
```

```
amap (generic function with 1 method)
```

```
da = tha_fgs_gse
da_smoothed = smooth(da, 61)
da_filtered = amap(-, da, da_smoothed)
da_filtered = rebuild(da_filtered; name=:tha_fgs_gse_filtered)
```

```
da_filtered_smoothed = smooth(da_filtered, 5)

figure = (; size=(1000, 600))
tplot([da, da_smoothed, da_filtered_smoothed]; figure)
```



4.1 Dynamic power spectrum

ii. Do a dynamic power spectrum of the unfiltered data.

In this section, we implement two approaches to represent the time-frequency domain of time series data. The first approach utilizes a window function, while the second employs a wavelet transform. The functions `pspectrum` are dispatched based on the second argument, leveraging Julia's multiple dispatch mechanism.

- Reference: Matlab, PySPEDAS : pytplot.tplot_math.dpwrspc

```
using SignalAnalysis
using DimensionalData: Where

function pspectrum(x::AbstractDimArray, spec::Spectrogram)
    fs = SpaceTools.samplingrate(x)
    y = tfd(x, spec; fs)
    t0 = dims(x, Ti)[1]
    times = Ti(y.time .* 1u"s" .+ t0)
    freqs = ?(y.freq * 1u"Hz")
    y_da = DimArray(y.power', (times, freqs))
end
```

```
pspectrum (generic function with 1 method)
```

```
using ContinuousWavelets

"""
    pspectrum(x, wt; kwargs...)

Returns the power spectrum of `x` in the time-frequency domains
"""
function pspectrum(x::AbstractDimArray, wt::CWT)
    fs = SpaceTools.samplingrate(x)
    res = cwt(x, wt)
    power = abs.(res) .^ 2
    n = length(dims(x, Ti))
    times = dims(x, Ti)
    freqs = ?(getMeanFreq(computeWavelets(n, wt)[1], fs) * 1u"Hz")
    DimArray(power, (times, freqs))
end
```

```
Main.Notebook.pspectrum
```

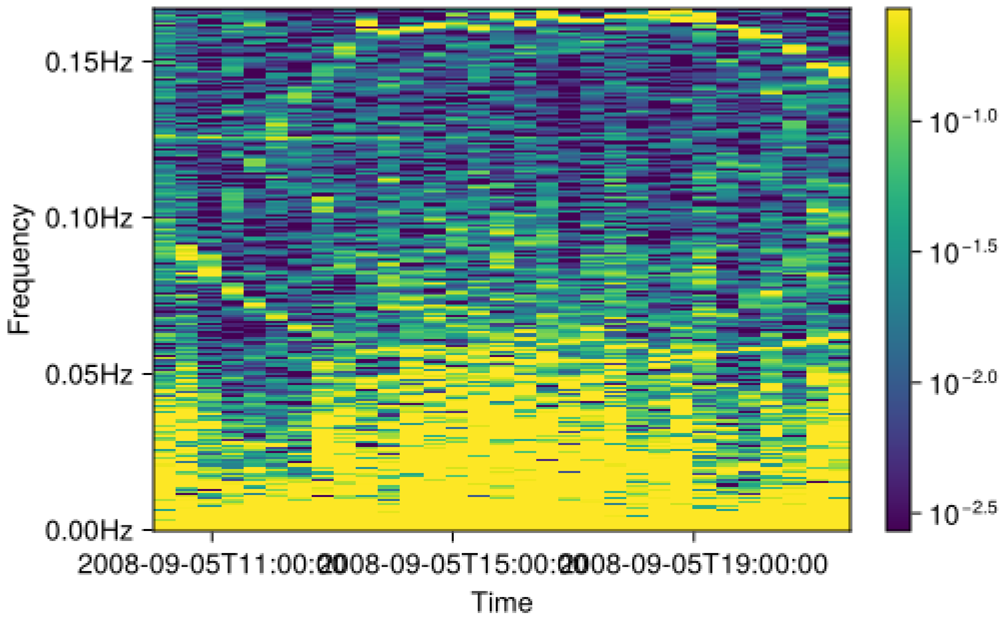
```
const DD = DimensionalData

"""
    plot_tfr(data; kwargs...)

```

```
Displays time frequency representation using Makie.
"""
function      plot_tfr(da::DimArray;      colorscale=log10,      crange=:auto,
figure_kwargs...)
    cmid = median(da)
    cmax = cmid * 10
    cmin = cmid / 10

    fig, ax, hm = heatmap(da; colorscale, colorrange=(cmin, cmax))
    Colorbar(fig[:, end+1], hm)

    # rasterize the heatmap to reduce file size
    if *(size(da)...) > 32^2
        hm.rasterize = true
    end

    fig
end
```

```
Main.Notebook.plot_tfr
```

Applying hamming window and plotting the power spectrum

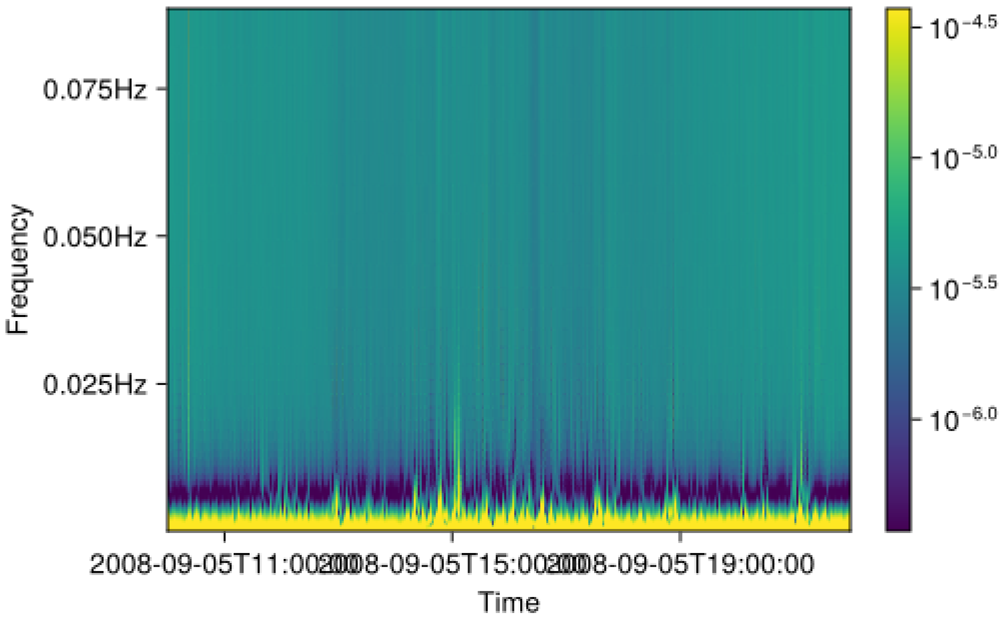
```
da = TS(tha_fgs_gse[:, 3])
da = rectify_datetime(da)

spec1 = Spectrogram(nfft=512, noverlap=64, window=hamming)
res1 = pspectrum(da, spec1)
plot_tfr(res1)
```



Applying Morlet wavelet and plotting the power spectrum

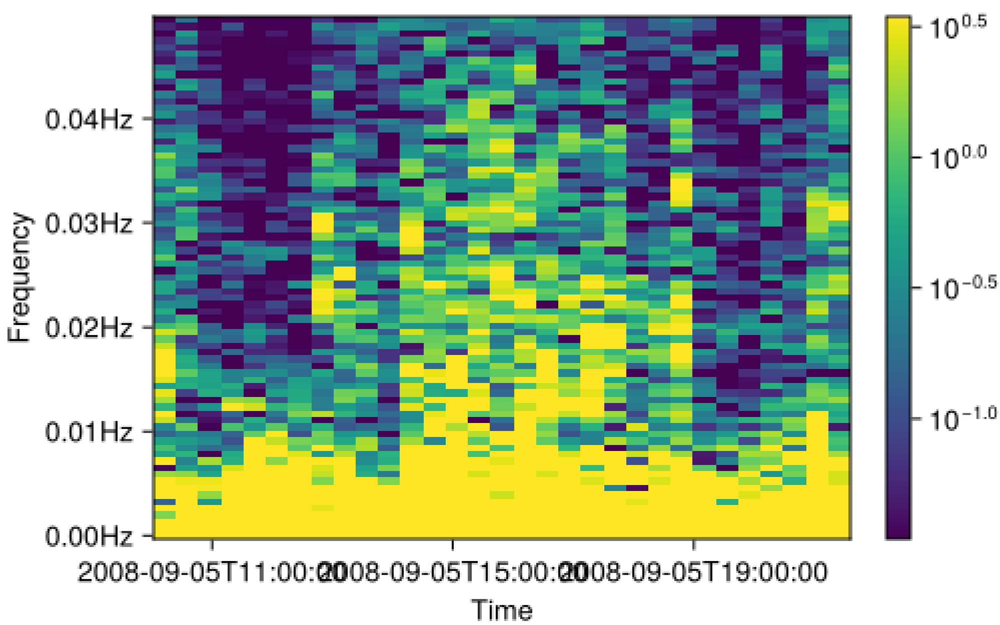
```
wl2 = wavelet(Morlet(π), β=2)
res2 = pspectrum(da, wl2)
plot_tfr(res2)
```

```
┌ Warning: the lowest frequency wavelet has more than 1% its max at zero, so it
may not be analytic. Think carefully
│   lowAprxAnalyt = 0.076815
└    @    ContinuousWavelets    ~/.julia/packages/ContinuousWavelets/eb0df/src/
sanityChecks.jl:7
┌ Warning: the lowest frequency wavelet has more than 1% its max at zero, so it
may not be analytic. Think carefully
│   lowAprxAnalyt = 0.076815
└    @    ContinuousWavelets    ~/.julia/packages/ContinuousWavelets/eb0df/src/
sanityChecks.jl:7
```



Field line resonances are more visible using window function although with lower resolution. Zoom in on the low frequencies

```
res1_s = res1[?=Where(<=(0.05u"Hz"))]
plot_tfr(res1_s)
```



4.2 Field aligned coordinate system

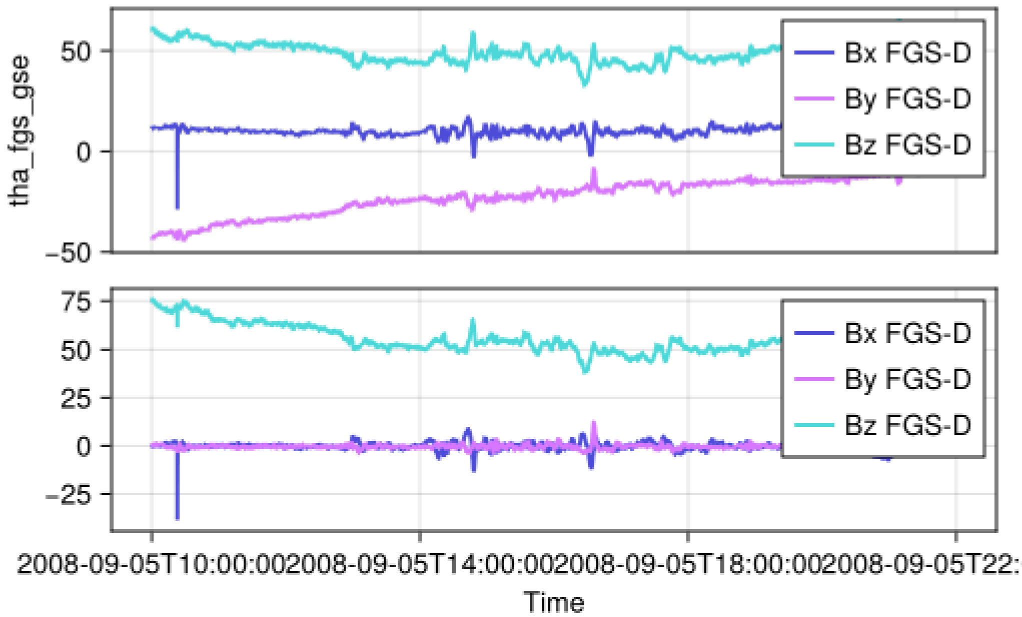
iii. Transform to the field aligned coordinate system, with "Other dimension" being radially inward (minus R). This will give you 3 components: Radially inward, Azimuthal Westward, and Field aligned. Plot and confirm that the compressional component is small.

To confirm that the compressional component of the magnetic field is small, we need to analyze the fluctuations in the magnetic field and determine whether the component along the background field direction is significantly weaker than the perpendicular components.

`fac_matrix_make` and `rotate` could be easily implemented in few lines. Note that we implement only the `array` version (corresponding to one timestamp), and the `matrix` version could be freely got using Julia `broadcast` operators which also align with dimensions.

```
function fac_matrix_make(
    vec::AbstractVector;
    xref=[1.0, 0.0, 0.0]
)
    z0 = normalize(vec)
    y0 = normalize(cross(z0, xref))
    x0 = cross(y0, z0)
    return vcat(x0', y0', z0')
end

function rotate(da, mat)
    da = da[DimSelectors(mats)]
    da_rot = mats .* eachrow(da.data)
    TS(dims(da, Ti), dims(da, 2), hcat(da_rot...)')
end
```

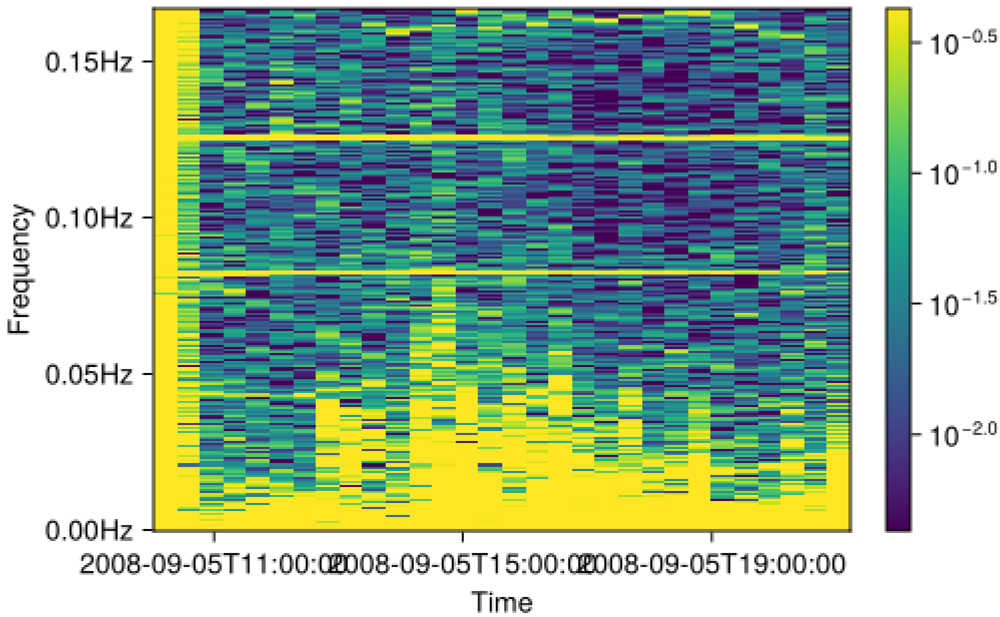
```
da = tha_fgs_gse
# smooth the data
mat_da = smooth(da, 601)
# make the rotation matrix from the smoothed data
mats = fac_matrix_make.(eachslice(mat_da, dims=Ti))
# rotate original data
da_fac = rotate(da, mats)
tplot([da, da_fac])
```



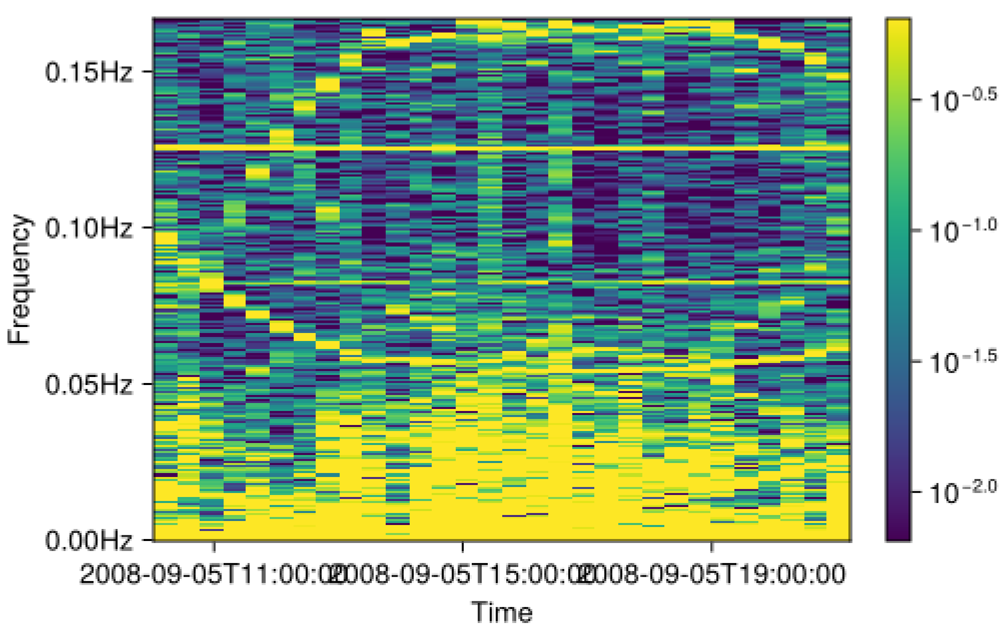
4.3 Poloidal vs. toroidal

iv. Make dynamic power spectra of the radial and azimuthal components. Which of the two components dominates? This determines if the FLR is poloidal or toroidal. (You may use wavelet transform instead of FFT to construct the dynamic power spectrum, if you prefer and can easily do so as in IDL SPEDAS). Report on your plots with explanations. Show all your code in addition to your plots, with comments included for clarification.

```
da_x = da_fac[:, 1]
da_x = rectify_datetime(da_x)

res_x = pspectrum(da_x, spec1)
plot_tfr(res_x)
```




```
da_y = da_fac[:, 2]
da_y = rectify_datetime(da_y)

res_y = pspectrum(da_y, spec1)
plot_tfr(res_y)
```



We can see that the power spectrum of the y (azimuthal) component is much higher than the power spectrum of the x (radial) component. This indicates that the FLR is likely to be toroidal.

Bibliography

[1] T. E. Sarris *et al.*, “THEMIS Observations of the Spatial Extent and Pressure-Pulse Excitation of Field Line Resonances,” *Geophysical Research Letters*, vol. 37, no. 15, 2010, doi: 10.1029/2010GL044125.