Problem Set 4

1 Barycentric coordinates

[1] We discussed barycentric coordinates in Lecture 08, pp. 11-12, and they are covered in Ch. 14 (Chanteur) in the ISSI book pp. 350-351. They are defined based on equation: $\left(\sum_{\alpha=1}^{4}\mu_\alpha\right)\mathbf{r} = \sum_{\alpha=1}^{4}(\mu_\alpha \mathbf{r}_\alpha)$ with the constraint $\sum_{\alpha=1}^{4}\mu_\alpha = 1$. We said that this physically means $\mu_\alpha(\mathbf{r}) = 1 + \mathbf{k}_\alpha \cdot (\mathbf{r} - \mathbf{r}_\alpha)$, a linear decrease of the weight with increasing distance from the vertex r_α of the tetrahedron, scaled according to the reciprocal vector (the inverted distance of vertex to plane across it).

Prove that $\mu_\alpha(\mathbf{r}) = 1 + \mathbf{k}_\alpha \cdot (\mathbf{r} - \mathbf{r}_\alpha)$ by using the inverse transformation: $\boldsymbol{\mu} = \mathbf{T}^{-1}(\mathbf{r} - \mathbf{r}_4)$.

In barycentric coordinates we have

$$\mathbf{r} - \mathbf{r}_4 = \sum_{\alpha=1}^{4}(\mu_\alpha \mathbf{r}_\alpha) - \left(\sum_{\alpha=1}^{4}\mu_\alpha\right)\mathbf{r}_4 = \mu_1\mathbf{d}_1 + \mu_2\mathbf{d}_2 + \mu_3\mathbf{d}_3.$$

A geometric way to see the first row of T^{-1} is to note that

$$\mathbf{k}_1 = \frac{\mathbf{d}_2 \times \mathbf{d}_3}{\mathbf{d}_1 \cdot (\mathbf{d}_2 \times \mathbf{d}_3)}$$

is the unique "reciprocal vector" satisfying $\mathbf{k}_1 \cdot \mathbf{d}_1 = 1$ and $\mathbf{k}_1 \cdot \mathbf{d}_2 = \mathbf{k}_1 \cdot \mathbf{d}_3 = 0$.

so

$$T^{-1} = \begin{pmatrix} \mathbf{k}_1 \\ \mathbf{k}_2 \\ \mathbf{k}_3 \end{pmatrix}$$

with $\mathbf{k}_2, \mathbf{k}_3$ defined similarly by the appropriate cross products and normalization.

Therefore from $\boldsymbol{\mu} = \mathbf{T}^{-1}(\mathbf{r} - \mathbf{r}_4)$, we have $\mu_i = \mathbf{k}_i \cdot (\mathbf{r} - \mathbf{r}_4)$, where $i = 1, 2, 3$.

Notice that $\mathbf{r} - \mathbf{r}_4 = (\mathbf{r} - \mathbf{r}_1) + \underbrace{(\mathbf{r}_1 - \mathbf{r}_4)}_{=\mathbf{d}_1}$ and by construction $\mathbf{k}_1 \cdot \mathbf{d}_1 = 1$.

Therefore

$$\mu_1 = \mathbf{k}_1 \cdot (\mathbf{r} - \mathbf{r}_1) + \mathbf{k}_1 \cdot \mathbf{d}_1 = 1 + \mathbf{k}_1 \cdot (\mathbf{r} - \mathbf{r}_1)$$

By identical reasoning, $\mu_\alpha = 1 + \mathbf{k}_\alpha \cdot (\mathbf{r} - \mathbf{r}_\alpha)$.

2 Cluster magnetopause crossing

[2] For a CLUSTER magnetopause crossing on 2001/06/11 other than the one in the crib... - compute Qsr, Elongation, Planarity and then the magnetopause normal and velocity. - Compute the current density, and field curvature at that location. - Compute the magnetopause thickness. - Compare with table below from Keyser et al. (original from Dunlop and Balogh, 2004). - Create and discuss plots of data, satellite positions, table of results from the "simple" and from either the "symmetric" or the "relative timing" methods. See next page for details.

Details: - I have released a crib sheet (Hwk04_02.pro) to help. It is not to be executed and run. Take it as a cheat-sheet: a collection of command-line steps to aid your construction of your own code. I brought over the Cluster CDFs, which you can read using IDL or Python tools or spd_gui. (The data does not exist in SPDF, I had to get it from the Cluster Active Archive. Perhaps AMDA can help you read it.). Use ChatGPT widely to help you, but review documentation when unsure. - The IDL code above uses helper functions for the linear gradient estimation and timing analysis which we discussed in class, and I include in the zip file. You may rewrite them in Python if you wish or use them directly in IDL (they are very simple). They are called lingradest.pro and mptiming.pro.

> ⚠ Warning
>
> Functions in this report are included in `SpaceTools` and documented in the following page: Multi-spacecraft analysis methods. However, they are experimental, not tested, and may change without notice.

2.1 Loading the data

To load the data from CSA server, we implement a module using `HAPI` protocol. More details can be found in the `HAPI.jl`.

```
using CairoMakie
using HAPI
using Dates
using SpaceTools
using DimensionalData
using Unitful
using DataFrames
using SummaryTables
using PrettyPrinting
```

```
trange = timerange("2001-06-11T20:10:30", "2001-06-11T20:12:30")
B_vars = [
```

```
    "csa/C1_CP_FGM_FULL/B_vec_xyz_gse",
    "csa/C2_CP_FGM_FULL/B_vec_xyz_gse",
    "csa/C3_CP_FGM_FULL/B_vec_xyz_gse",
    "csa/C4_CP_FGM_FULL/B_vec_xyz_gse",
]
pos_vars = [
    "csa/C1_CP_FGM_FULL/sc_pos_xyz_gse",
    "csa/C2_CP_FGM_FULL/sc_pos_xyz_gse",
    "csa/C3_CP_FGM_FULL/sc_pos_xyz_gse",
    "csa/C4_CP_FGM_FULL/sc_pos_xyz_gse",
]
B_data = get_data.(B_vars, trange)
pos_data = get_data.(pos_vars, trange) .|> DimArray
```

```
4-element Vector{DimVector{Vector{Quantity{Float64, ?,
Unitful.FreeUnits{(km,), ?, nothing}}},
Tuple{Ti{DimensionalData.Dimensions.Lookups.Sampled{DateTime,
SentinelArrays.ChainedVector{DateTime, Vector{DateTime}},
DimensionalData.Dimensions.Lookups.ForwardOrdered,
DimensionalData.Dimensions.Lookups.Irregular{Tuple{Nothing, Nothing}},
DimensionalData.Dimensions.Lookups.Points,
DimensionalData.Dimensions.Lookups.NoMetadata}}}, Tuple{},
Vector{Vector{Quantity{Float64, ?, Unitful.FreeUnits{(km,), ?, nothing}}}},
Symbol, Dict{Any, Any}}}:
 Vector{Quantity{Float64, ?, Unitful.FreeUnits{(km,), ?, nothing}}}
[[-17192.3 km, -121540.1 km, 12485.9 km], [-17192.4 km, -121540.1 km, 12485.8
km], [-17192.4 km, -121540.1 km, 12485.8 km], [-17192.4 km, -121540.1 km,
12485.7 km], [-17192.4 km, -121540.1 km, 12485.7 km], [-17192.5 km, -121540.1
km, 12485.6 km], [-17192.5 km, -121540.1 km, 12485.6 km], [-17192.5 km,
-121540.2 km, 12485.6 km], [-17192.5 km, -121540.2 km, 12485.5 km], [-17192.5
km, -121540.2 km, 12485.5 km]  …  [-17244.3 km, -121549.9 km, 12366.8 km],
[-17244.3 km, -121549.9 km, 12366.8 km], [-17244.3 km, -121549.9 km, 12366.7
km], [-17244.3 km, -121549.9 km, 12366.7 km], [-17244.3 km, -121549.9 km,
12366.6 km], [-17244.3 km, -121549.9 km, 12366.6 km], [-17244.4 km, -121549.9
km, 12366.5 km], [-17244.4 km, -121549.9 km, 12366.5 km], [-17244.4 km,
-121549.9 km, 12366.5 km], [-17244.4 km, -121549.9 km, 12366.4 km]]
 Vector{Quantity{Float64, ?, Unitful.FreeUnits{(km,), ?, nothing}}}
[[-19049.1 km, -122091.8 km, 13031.5 km], [-19049.1 km, -122091.8 km, 13031.5
km], [-19049.1 km, -122091.8 km, 13031.4 km], [-19049.1 km, -122091.8 km,
13031.4 km], [-19049.1 km, -122091.8 km, 13031.4 km], [-19049.2 km, -122091.8
km, 13031.3 km], [-19049.2 km, -122091.9 km, 13031.3 km], [-19049.2 km,
-122091.9 km, 13031.2 km], [-19049.2 km, -122091.9 km, 13031.2 km], [-19049.3
km, -122091.9 km, 13031.1 km]  …  [-19100.2 km, -122101.3 km, 12914.8 km],
[-19100.2 km, -122101.3 km, 12914.8 km], [-19100.2 km, -122101.3 km, 12914.7
km], [-19100.2 km, -122101.3 km, 12914.7 km], [-19100.2 km, -122101.3 km,
12914.6 km], [-19100.2 km, -122101.3 km, 12914.6 km], [-19100.3 km, -122101.3
km, 12914.5 km], [-19100.3 km, -122101.3 km, 12914.5 km], [-19100.3 km,
```

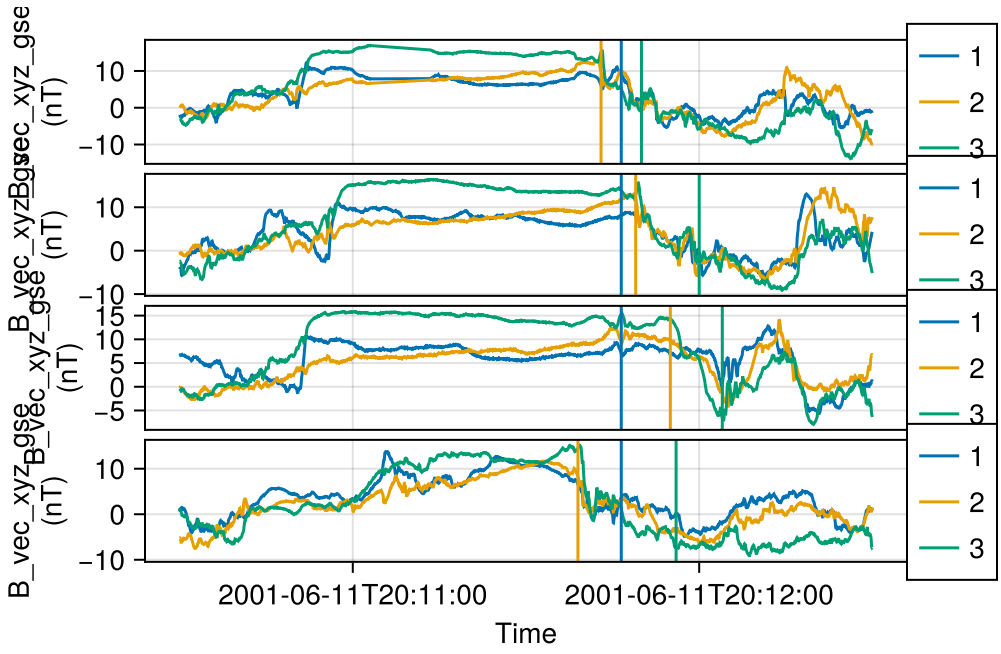
```
-122101.3 km, 12914.4 km], [-19100.3 km, -122101.3 km, 12914.4 km]]
 Vector{Quantity{Float64, ?, Unitful.FreeUnits{(km,), ?, nothing}}}
[[-18506.3 km, -121999.5 km, 11157.3 km], [-18506.3 km, -121999.5 km, 11157.2
km], [-18506.3 km, -121999.5 km, 11157.2 km], [-18506.3 km, -121999.5 km,
11157.2 km], [-18506.3 km, -121999.5 km, 11157.1 km], [-18506.3 km, -121999.6
km, 11157.1 km], [-18506.4 km, -121999.6 km, 11157.0 km], [-18506.4 km,
-121999.6 km, 11157.0 km], [-18506.4 km, -121999.6 km, 11156.9 km], [-18506.4
km, -121999.6 km, 11156.9 km]  …  [-18558.0 km, -122012.7 km, 11040.1 km],
[-18558.0 km, -122012.7 km, 11040.0 km], [-18558.0 km, -122012.7 km, 11040.0
km], [-18558.0 km, -122012.7 km, 11040.0 km], [-18558.0 km, -122012.7 km,
11039.9 km], [-18558.1 km, -122012.7 km, 11039.9 km], [-18558.1 km, -122012.7
km, 11039.8 km], [-18558.1 km, -122012.7 km, 11039.8 km], [-18558.1 km,
-122012.7 km, 11039.7 km], [-18558.2 km, -122012.8 km, 11039.7 km]]
 Vector{Quantity{Float64, ?, Unitful.FreeUnits{(km,), ?, nothing}}}
[[-17608.1 km, -123229.0 km, 12203.8 km], [-17608.1 km, -123229.0 km, 12203.8
km], [-17608.2 km, -123229.0 km, 12203.7 km], [-17608.2 km, -123229.0 km,
12203.7 km], [-17608.2 km, -123229.0 km, 12203.6 km], [-17608.2 km, -123229.0
km, 12203.6 km], [-17608.2 km, -123229.0 km, 12203.6 km], [-17608.2 km,
-123229.0 km, 12203.5 km], [-17608.3 km, -123229.0 km, 12203.5 km], [-17608.3
km, -123229.0 km, 12203.4 km]  …  [-17658.1 km, -123238.3 km, 12089.2 km],
[-17658.1 km, -123238.3 km, 12089.2 km], [-17658.1 km, -123238.3 km, 12089.1
km], [-17658.1 km, -123238.3 km, 12089.1 km], [-17658.2 km, -123238.3 km,
12089.0 km], [-17658.2 km, -123238.3 km, 12089.0 km], [-17658.2 km, -123238.3
km, 12089.0 km], [-17658.2 km, -123238.3 km, 12088.9 km], [-17658.2 km,
-123238.3 km, 12088.9 km], [-17658.2 km, -123238.3 km, 12088.8 km]]
```

```
# Make start time of the field-rotation (start of By, Bz change) fiducial
t0s = [
    "2001-06-11T20:11:43",
    "2001-06-11T20:11:49",
    "2001-06-11T20:11:55",
    "2001-06-11T20:11:39",
] .|> DateTime
t1s = [
    "2001-06-11T20:11:50",
    "2001-06-11T20:12:00",
    "2001-06-11T20:12:04",
    "2001-06-11T20:11:56",
] .|> DateTime

avgtime = tmean(t0s)
```

```
2001-06-11T20:11:46.500
```

```
fig, axs = tplot(B_data)
tlines!.(axs, avgtime)
```

```
tlines!.(axs, t0s)
tlines!.(axs, t1s)
fig
```



2.2 Plot sc positions relative to mesocenter

```
using Statistics

Rs = tinterp.(pos_data, avgtime) # Rs = getindex.(pos_data, Ti=Near(avgtime))
R00 = mean(Rs)
R0s = Rs .- Ref(R00)
```

```
4-element Vector{Vector{Quantity{Float64, ?, Unitful.FreeUnits{(km,), ?,
nothing}}}}:
 [896.0960227272735 km, 675.4250000000029 km, 264.8500000000022 km]
 [-960.0835227272728 km, 123.92500000000291 km, 812.0500000000011 km]
 [-417.60852272727425 km, 213.8249999999971 km, -1062.5499999999993 km]
 [481.5960227272735 km, -1013.1750000000029 km, -14.349999999998545 km]
```

```
function plot_positions(positions::AbstractVector{<:AbstractVector}; axis=(;))
    n = length(positions)
    theme = Theme(
        Scatter=(
            markersize=15,
            cycle=Cycle([:color, :marker], covary=true)
```

```
        )
    )
    base_unit = unit(positions[1][1])
    dim1_conversion = dim2_conversion = Makie.UnitfulConversion(base_unit;
units_in_label=false)

    with_theme(theme) do
        fig = Figure()
        ax1 = Axis(fig[1, 1]; title="XY", ylabel="y", dim1_conversion,
dim2_conversion, axis...)
        ax2 = Axis(fig[2, 1]; title="XZ", xlabel="x", ylabel="z",
dim1_conversion, dim2_conversion, axis...)
        ax3 = Axis(fig[2, 2]; title="YZ", xlabel="y", dim1_conversion,
dim2_conversion, axis...)
        axs = [ax1, ax2, ax3]

        # Plot each positions
        plots = map(positions) do pos
            scatter!(ax1, [pos[1]], [pos[2]])
            scatter!(ax2, [pos[1]], [pos[3]])
            scatter!(ax3, [pos[2]], [pos[3]])
        end
        Legend(fig[1, 2], plots, ["C$i" for i in 1:n]; tellwidth=false,
tellheight=false)
        fig, axs
    end
end

plot_positions(R0s)
```

```
(Scene (600px, 450px):
  0 Plots
  4 Child Scenes:
    ├ Scene (600px, 450px)
    ├ Scene (600px, 450px)
    ├ Scene (600px, 450px)
    └ Scene (600px, 450px), Axis[Axis (4 plots), Axis (4 plots), Axis (4
plots)])
```

2.3 Tetrahedron quality factors: Qsr, elongation, planarity

```
"""Calculate tetrahedron quality factors"""
function tetrahedron_quality(positions::AbstractVector{<:AbstractVector})
    Rvol = volumetric_tensor(positions)
    # Calculate eigenvaluesz and eigenvectors
    F = eigen(ustrip(Rvol), sortby=x -> -abs(x)) # Note: we want descending
order
```

```
    semiaxes = sqrt.(F.values)  # sqrt of eigenvalues
    eigenvectors = F.vectors
    # Calculate quality parameters
    Qsr = 0.5 * (sum(semiaxes) / semiaxes[1] - 1)
    Elongation = 1 - (semiaxes[2] / semiaxes[1])
    Planarity = 1 - (semiaxes[3] / semiaxes[2])

    return (; det=det(Rvol), semiaxes, Qsr, Elongation, Planarity,
eigenvectors)
end
```

```
tetrahedron_quality(R0s) |> pprint
```

```
(det = 9.491477044462083e16 km⁶,
 semiaxes = [739.1443381212663, 675.706718317148, 616.8497731211844],
 Qsr = 0.8743600030297949,
 Elongation = 0.08582575355357791,
 Planarity = 0.08710427704869828,
 eigenvectors =
     [0.9278535982834907 0.35338946097171103 0.11917881114737737;
-0.1663544954859536 0.10616487882162137 0.9803342288915573;
-0.33378718067653923 0.9294325727985404 -0.1572933903159741])
```

2.4 Magnetopause normal and velocity

```
"""
    CVA(positions, times)

Constant Velocity Approach (CVA) for determining boundary normal and velocity.
Solve timing equation: ``D * m = Δts``

Parameters:
- positions: Positions of 4 spacecraft (4×3 array)
- times: Times of boundary crossing for each spacecraft
"""
function CVA(positions, times)
    # Calculate time delays relative to first spacecraft
    Δts = times[2:end] .- times[1]
    # Calculate position differences relative to first spacecraft
    D = reduce(hcat, [r - positions[1] for r in positions[2:end]])'
    m = inv(D) * Δts
    return m2nV(m)
end
```

```
"Convert timing vector ``⍰`` to normal vector and velocity where ``⍰ = ⍰/
V``"
tv_to_nv(m) = SpaceTools.m2nV(m)

"Symmetric timing method using barycenter reference"
function timing_symmetric(positions, times)
    # Calculate times relative to average crossing time
    Δts = times .- tmean(times)
    R = volumetric_tensor(positions)

    # Calculate sum of time-distance products (ensure column vector)
    sum_td = sum(t .* r for (t, r) in zip(Δts, positions)) ./ length(Δts)

    # Solve timing equation: R * M = sum_td
    m = inv(R) * sum_td
    return tv_to_nv(m)
end

"Relative timing method using all spacecraft pairs"
function timing_relative(positions, times)
    n = length(positions)

    # Create arrays of all pairs of time differences
    Δt_matrix = [t_j - t_i for t_i in times, t_j in times]

    # Create arrays of all pairs of position differences
    Δr_matrices = [
        [r_j[i] - r_i[i] for r_i in positions, r_j in positions]
        for i in 1:3  # x, y, z components
    ]

    # Calculate double sum of time-distance products (ensure column vector)
    sum_td = [
        sum(Δt_matrix .* Δr_matrix) for Δr_matrix in Δr_matrices
    ] ./ (2 * n^2)

    # Calculate volumetric tensor and solve timing equation: R * M = sum_td
    R = volumetric_tensor(positions)
    m = inv(R) * sum_td
    return tv_to_nv(m)
end

# Example usage with your data
times = t0s

# Calculate using all three methods
simple_result = (; method="Simple", CVA(Rs, times)...,)
symmetric_result = (; method="Symmetric", timing_symmetric(Rs, times)...,)
```

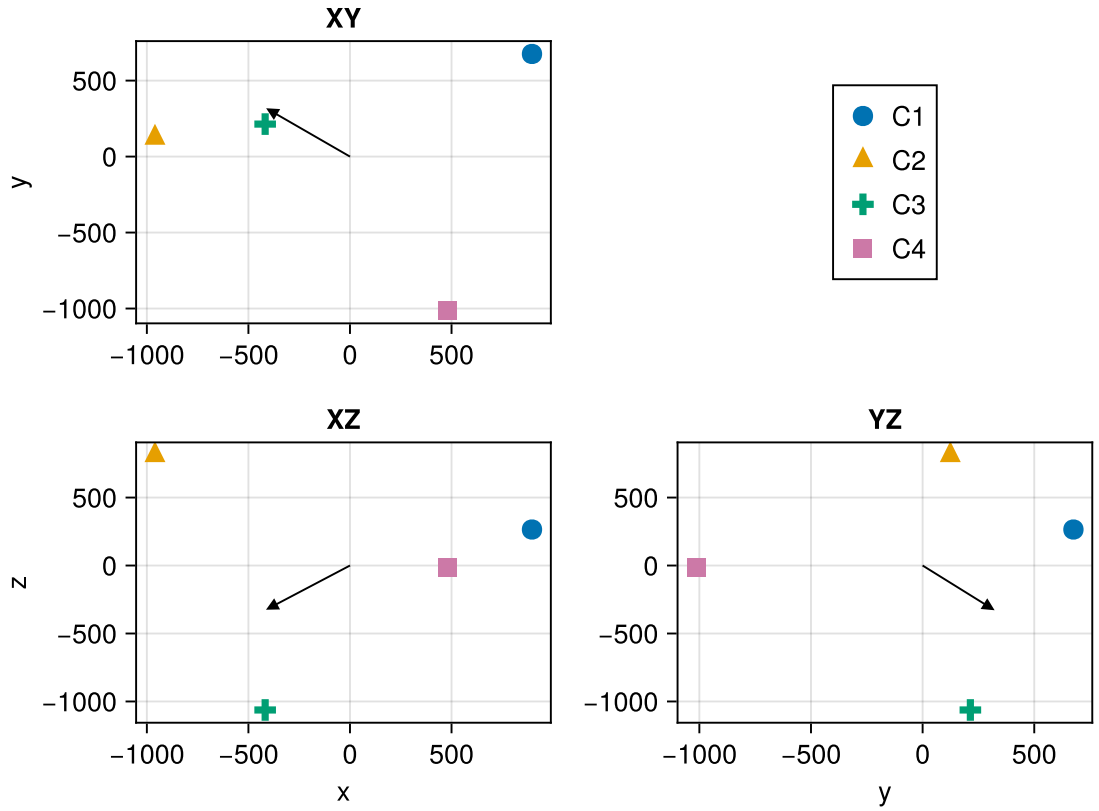
```
relative_result = (; method="Relative", timing_relative(Rs, times)...,)

# Pretty Print
df = DataFrame([simple_result, symmetric_result, relative_result])
round_array(x) = round.(x, digits=3)
df.n = round_array.(df.n)
df.V = round.(u"km/s", df.V)
simple_table(df)
```

method	n	V
Simple	[−0.672, 0.518, −0.529]	112.0 km s^{-1}
Symmetric	[−0.672, 0.518, −0.529]	112.0 km s^{-1}
Relative	[−0.672, 0.518, −0.529]	112.0 km s^{-1}

Within numeric tolerance, the three methods yield the same result. The results of n are different from the table, this may be due to different coordinate systems (not implemented for now).

2.4.a Overplot magnetopause velocity

```
fig, axs = plot_positions(R0s)
res = symmetric_result
factor = 2e-1u"km/s"
arr = res.n * res.V / factor .|> upreferred
arrows!(axs[1], [0.0], [0.0], arr[1:1], arr[2:2])
arrows!(axs[2], [0.0], [0.0], arr[1:1], arr[3:3])
arrows!(axs[3], [0.0], [0.0], arr[2:2], arr[3:3])
fig
```



2.5 Current density, and field curvature

Utilizing Julia's linear algebra library, `lingradest` function can be efficiently implemented in Julia with fewer than 40 lines of code.

```
"""
    lingradest(B1, B2, B3, B4, R1, R2, R3, R4)

Compute spatial derivatives such as grad, div, curl and curvature using
reciprocal vector technique (linear interpolation).

# Arguments
- `B1, B2, B3, B4`: 3-element vectors giving magnetic field measurements at
each probe
- `R1, R2, R3, R4`: 3-element vectors giving the probe positions

# Returns
A named tuple containing:
  • Rbary: Barycenter position
  • Bbc: Magnetic field at the barycenter
  • Bmag: Magnetic field magnitude at the barycenter
```

```
  • LGBx, LGBy, LGBz: Linear gradient estimators for each component
  • LD: Linear divergence estimator
  • LCB: Linear curl estimator
  • curvature: Field-line curvature vector
  • R_c: Field-line curvature radius

# References
Based on the method of Chanteur (ISSI, 1998, Ch. 11).
- [lingradest.pro](https://github.com/spedas/bleeding_edge/blob/master/
projects/mms/common/curlometer/lingradest.pro)
- [lingradest.py](https://github.com/spedas/pyspedas/blob/master/pyspedas/
analysis/lingradest.py#L5)
"""
function lingradest(B1, B2, B3, B4, R1, R2, R3, R4)

    Rs = [R1, R2, R3, R4]
    Bs = [B1, B2, B3, B4]
    Bxs = getindex.(Bs, 1)
    Bys = getindex.(Bs, 2)
    Bzs = getindex.(Bs, 3)

    # Barycenter of the tetrahedron
    Rbary = (R1 .+ R2 .+ R3 .+ R4) ./ 4
    dRs = Ref(Rbary) .- Rs

    # Reciprocal vectors and μ factors
    ks = reciprocal_vectors(R1, R2, R3, R4)
    μs = @. 1 + dot(ks, dRs)

    # Magnetic field at barycenter
    Bbc = sum(μs .* Bs)
    Bmag = norm(Bbc)

    # Linear Gradient estimators
    LGBx = sum(Bxs .* ks)
    LGBy = sum(Bys .* ks)
    LGBz = sum(Bzs .* ks)
    LGB = [LGBx LGBy LGBz]
    # Linear Divergence estimator
    div = sum(dot.(ks, Bs))

    # Linear Curl estimator
    curl = sum(cross.(ks, Bs))
    # Field-line curvature components
    curvature = (LGB' * Bbc) / (Bmag^2)
    R_c = 1 / norm(curvature)

    return (; Rbary, Bbc, Bmag,
```

```
        LGBx, LGBy, LGBz,
        div, curl, curvature, R_c)
end
```

```
B_das = tstack.(DimArray.(B_data))
times = dims(B_das[1], Ti)
B_interp_das = tinterp.(B_das, Ref(times))
pos_interp_das = tinterp.(tstack.(pos_data), Ref(times))
res = lingradest(B_interp_das..., pos_interp_das...)
```

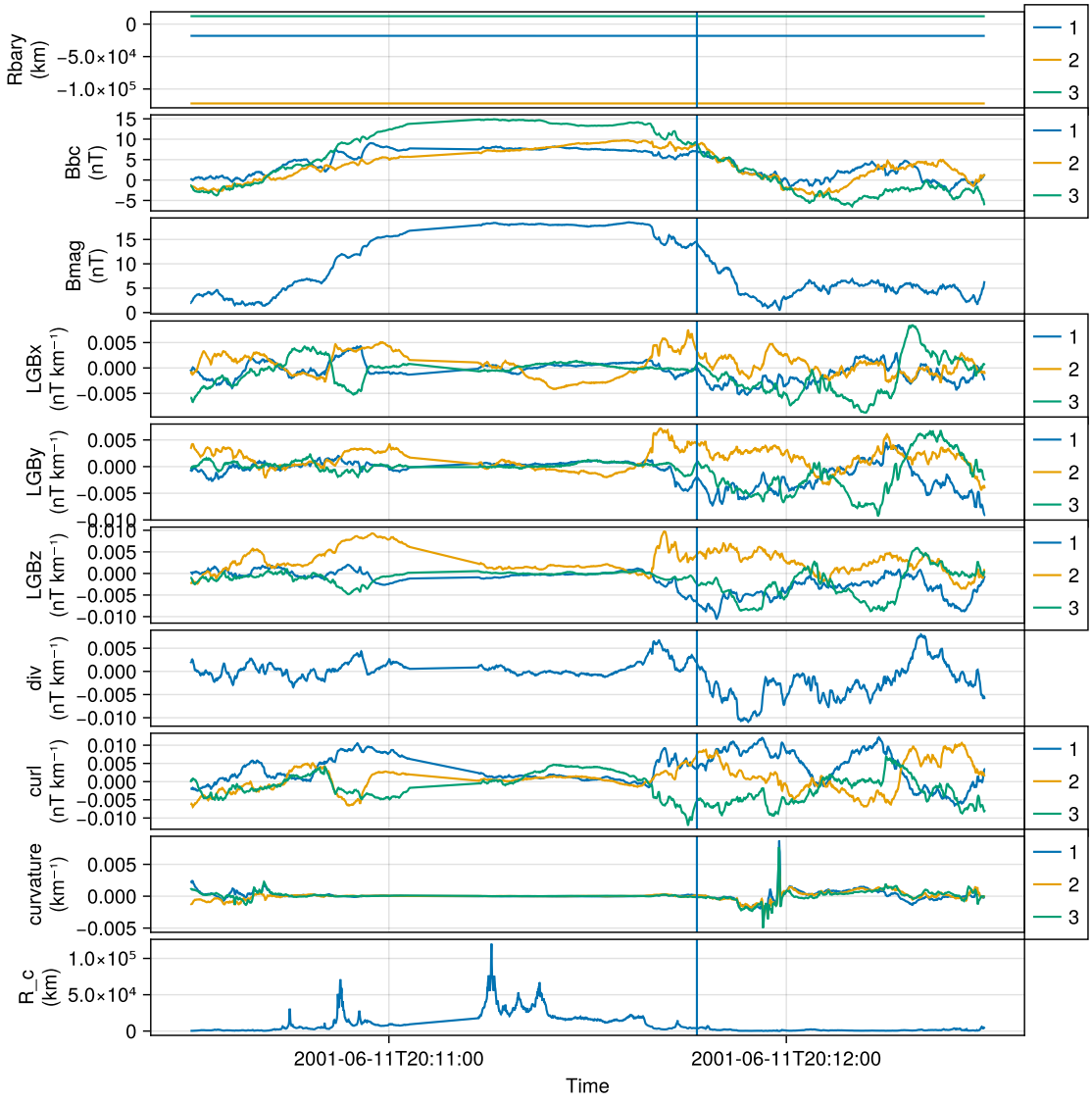
```
┌ 2459-element DimStack ┐
├───────────────────────┴──────────────────────────────────────────────────────── dims
┐
  ↓ Ti Sampled{Dates.DateTime} [2001-06-11T20:10:30.036, …,
2001-06-11T20:12:29.966] ForwardOrdered Irregular Points
├──────────────────────────────────────────────────────────────────────────────── layers
┤
  :Rbary     eltype: Vector{Unitful.Quantity{Float64, ?,
Unitful.FreeUnits{(km,), ?, nothing}}} dims: Ti size: 2459
  :Bbc       eltype: Vector{Unitful.Quantity{Float64, ? ?⁻¹ ?⁻²,
Unitful.FreeUnits{(nT,), ? ?⁻¹ ?⁻², nothing}}} dims: Ti size: 2459
  :Bmag      eltype: Unitful.Quantity{Float64, ? ?⁻¹ ?⁻²,
Unitful.FreeUnits{(nT,), ? ?⁻¹ ?⁻², nothing}} dims: Ti size: 2459
  :LGBx      eltype: Vector{Unitful.Quantity{Float64, ? ?⁻¹ ?⁻¹ ?⁻²,
Unitful.FreeUnits{(km⁻¹, nT), ? ?⁻¹ ?⁻¹ ?⁻², nothing}}} dims: Ti size: 2459
  :LGBy      eltype: Vector{Unitful.Quantity{Float64, ? ?⁻¹ ?⁻¹ ?⁻²,
Unitful.FreeUnits{(km⁻¹, nT), ? ?⁻¹ ?⁻¹ ?⁻², nothing}}} dims: Ti size: 2459
  :LGBz      eltype: Vector{Unitful.Quantity{Float64, ? ?⁻¹ ?⁻¹ ?⁻²,
Unitful.FreeUnits{(km⁻¹, nT), ? ?⁻¹ ?⁻¹ ?⁻², nothing}}} dims: Ti size: 2459
  :div       eltype: Unitful.Quantity{Float64, ? ?⁻¹ ?⁻¹ ?⁻²,
Unitful.FreeUnits{(km⁻¹, nT), ? ?⁻¹ ?⁻¹ ?⁻², nothing}} dims: Ti size: 2459
  :curl      eltype: Vector{Unitful.Quantity{Float64, ? ?⁻¹ ?⁻¹ ?⁻²,
Unitful.FreeUnits{(km⁻¹, nT), ? ?⁻¹ ?⁻¹ ?⁻², nothing}}} dims: Ti size: 2459
  :curvature eltype: Vector{Unitful.Quantity{Float64, ?⁻¹,
Unitful.FreeUnits{(km⁻¹,), ?⁻¹, nothing}}} dims: Ti size: 2459
  :R_c       eltype: Unitful.Quantity{Float64, ?, Unitful.FreeUnits{(km,),
?, nothing}} dims: Ti size: 2459
└──────────────────────────────────────────────────────────────────────────────────┘
```

The current density and field curvature at the average time are

```
pprint(res[Ti=Near(avgtime)])
```

```
(Rbary = [-18121.68409090909 km, -122221.825 km, 12144.868333333334 km],
 Bbc = [7.078833838383838 nT, 8.575880808080807 nT, 8.992595959595958 nT],
 Bmag = 14.301132900522195 nT,
 LGBx = [1.3813172727203307e-5 nT km⁻¹,
         0.003366241172121136 nT km⁻¹,
         0.0007368641222804634 nT km⁻¹],
 LGBy = [-0.002049983588783596 nT km⁻¹,
         0.004280815830114207 nT km⁻¹,
         0.0007956255498070833 nT km⁻¹],
 LGBz = [-0.006685534578927632 nT km⁻¹,
         0.004410837499235884 nT km⁻¹,
         -0.00242034765020941 nT km⁻¹],
 div = 0.0018742813526320009 nT km⁻¹,
 curl = [0.0036152119494287996 nT km⁻¹,
         0.007422398701208097 nT km⁻¹,
         -0.005416224760904732 nT km⁻¹],
 curvature = [0.00017402780896923182 km⁻¹,
              0.00014352956892380598 km⁻¹,
              -0.00015286433126961862 km⁻¹],
 R_c = 3669.7823651585372 km)
```

```
fig, axs = tplot(res; figure=(; size=(800, 800)))
tlines!.(axs, avgtime)
fig
```



2.6 Magnetopause thickness

The thickness can be estimated by multiplying the duration of each center crossing by the normal velocity obtained from the preceding analysis. Since we assume that velocity is constant instead of varying, the estimated thickness is simply the mean of the product of the velocity and the duration.

```
durations = t1s .- t0s
ds = symmetric_result.V .* durations
@info "thickness" ds
@info "mean thickness" mean(ds)
```

```
┌ Info: thickness
│   ds =
│    4-element Vector{Quantity{Float64, ?, Unitful.FreeUnits{(km,), ?,
nothing}}}:
│      785.0844206375767 km
│     1233.7040895733348 km
│     1009.3942551054557 km
└     1906.633592976972 km
┌ Info: mean thickness
└   mean(ds) = 1233.7040895733348 km
```

3 Wave polarization

[3] Show (see Lecture 09 p. 31) that in the plane of polarization of an elliptically polarized wave the ellipticity $\varepsilon = tan(\beta)$, and direction of polarization φ, can be obtained from the amplitudes $B_{(x0)'}, B_{(y0)'}$ and the phase difference, ψ, between the y' and x' components from the equations: $sin(2\beta) = sin(\psi)2B_{(x0)'}B_{(y0)'}/\left({B_{x0}}'^2 + {B_{y0}}'^2\right)$, and: $tan(2\varphi) = cos(\psi)2B_{(x0)'}B_{(y0)'}/\left({B_{x0}}'^2 - {B_{y0}}'^2\right)$ This is the origin of the wave polarization analysis using minimum variance techniques. They are used both in exploration geophysics and in space physics, so it is good to be convinced of their origin.

We drop the primes in the plane of polarization for simplicity.

Consider the magnetic field components in the plane of polarization:

$$B_x(t) = B_{x0}\cos(\omega t),$$
$$B_y(t) = B_{y0}\cos(\omega t + \psi).$$

Eliminating ωt from these equations (using $cos^2(\omega t) + sin^2(\omega t) = 1$ identity) $yields the ellipse:

$$\left(\frac{B_x}{B_{x0}}\right)^2 + \left(\frac{B_y}{B_{y0}}\right)^2 - 2\left(\frac{B_x B_y}{B_{x0}B_{y0}}\right)\cos\psi = \sin^2\psi.$$

Stokes Parameters characterize the polarization state:

$$S_0 = B_{x0}^2 + B_{y0}^2,$$
$$S_1 = B_{x0}^2 - B_{y0}^2,$$
$$S_2 = 2B_{x0}B_{y0}\cos\psi,$$
$$S_3 = 2B_{x0}B_{y0}\sin\psi.$$

Given the Stokes parameters, one can solve for the spherical coordinates with the following equations:

$$2\psi = \arctan\frac{S_2}{S_1}$$

$$2\beta = \arctan\frac{S_3}{\sqrt{S_1^2 + S_2^2}}$$

Using Stokes parameters, the ellipticity $\epsilon = \tan\beta$ relates to the minor-to-major axis ratio:

$$\sin(2\beta) = \sin\arctan\left(\frac{S_3}{\sqrt{S_1^2 + S_2^2}}\right) = \frac{S_3}{S_0} = \frac{2B_{x0}B_{y0}\sin\psi}{B_{x0}^2 + B_{y0}^2}.$$

The angle ψ of the major axis is given by the following equation:

$$\tan(2\phi) = \frac{S_2}{S_1} = \frac{2B_{x0}B_{y0}\cos\psi}{B_{x0}^2 - B_{y0}^2}.$$

Bibliography