Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Bc. Miroslav Černý

Studium vlnově-částicových interakcí v kosmickém plazmatu

Katedra fyziky povrchů a plazmatu

Vedoucí diplomové práce:	doc. RNDr. Ondřej Santolík Dr.
Studijní program:	fyzika
Studijní obor:	teoretická fyzika

Praha 2011

Chtěl bych tímto poděkovat vedoucímu diplomové práce doc. RNDr. Ondřeji Santolíkovi Dr. za odborné vedení, podnětné připomínky a poskytnutí počítačových programů pro numerické výpočty.

Prohlašuji, že jsem tuto diplomovou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Název práce: Studium vlnově-částicových interakcí v kosmickém plazmatu

Autor: Bc. Miroslav Černý

Katedra: Katedra fyziky povrchů a plazmatu

Vedoucí diplomové práce: Doc. RNDr. Ondřej Santolík Dr., katedra fyziky povrchů a plazmatu

Abstrakt: Tato práce se zabývá lineární analýzou plazmatických vln, zejména metodami řešení disperzní relace horkého plazmatu. Uvádí některé výsledky, jichž bylo dosaženo při výzkumu vln v kosmickém plazmatu a mapuje stavající numerické metody pro jejich analýzu. Kromě toho představuje novou numerickou proceduru, program PDRS (Plasma Dispersion Relation Solver), umožňující hledat řešení disperzní funkce studeného, ale i horkého plazmatu s obecnou distribuční funkcí. Použití metod PDRS ukazuje na reálných případech vln v kosmickém plazmatu v okolí Země na základě dat naměřených družicí Cluster.

Klíčová slova: plazmatické vlny, disperzní relace, dielektrický tenzor, Langmuirovy vlny

Title: Analysis of wave-particle interactions in space plasmas

Author: Bc. Miroslav Černý

Department: Department of Surface and Plasma Science

Supervisor: Doc. RNDr. Ondřej Santolík Dr., Department of Surface and Plasma Science

Abstract: This work deals with the linear analysis of plasma waves, especially with the methods of solution of a hot plasma dispersion relation. There are cited some results achieved in the space plasma research and mapped current numerical methods of their analysis. Besides, this work introduces a new numeric procedure, computer code PDRS (Plasma Dispersion Relation Solver), which allows finding solution of a dispersion function of a cold or hot plasma with general distribution function. It also demonstrates the usage of the PDRS methods on real examples of waves in space plasma based on the spacecraft Cluster measurement.

Keywords: plasma waves, dispersion relation, dielectric tensor, Langmuir waves

Obsah

Úvod **3**

1 Teorie šíření vln v plazmatu **4**
1.1 Disperzní relace 4
1.2 Základní vzorce 4
1.3 Dielektrický tenzor 5
1.3.1 Studené plazma 5
1.3.2 Horké plazma 6

2 Některé výsledky výzkumu vln v kosmickém plazmatu **7**
2.1 WHAMP 7
2.2 WDF 8
2.3 HOTRAY 8
2.4 Další 9

3 Cíle práce **10**

4 Nové metody PDRS pro analýzu vlastností vln v plazmatu **11**
4.1 Popis algoritmů 11
4.1.1 Řešení disperzní relace 11
4.1.2 Diskretizace distribuční funkce 12
4.1.3 Integrace 12
4.1.4 Derivace 13
4.1.5 Interpolace 13
4.1.6 Suma Besselových funkcí 13

5 Testování metody PDRS **15**
5.1 Mód volného prostoru 15
5.2 Podélné šíření 16
5.3 Šíření ve směru kolmém na směr magnetického pole 16
5.4 Lamgmuirovy vlny 17

6 Harmonické emise pozorované družicemi Cluster **23**
6.1 Popis modelu plazmatu 23
6.2 Výsledky 23

7 Emise typu chorus **26**
7.1 Popis modelu plazmatu 26
7.2 Výsledky 26

8 Dokumentace programu PDRS **28**
8.1 Vstupy programu PDRS 28
8.1.1 Definice modelu plazmatu 28
8.1.2 Řízení běhu programu 28
8.1.3 Distribuční funkce 28
8.2 Výstupy programu PDRS 29

8.2.1 Data 29
8.2.2 Návratová hodnota 29
8.2.3 Distribuční funkce 29
8.3 Kompilace programu PDRS 29

Závěr **36**

Seznam použité literatury **37**

Seznam tabulek **38**

Seznam použitých zkratek **39**

Příloha 1. Vstupní soubory s definící modelů plazmatu **40**
Vstupní soubor s definící modelu plazmatu použitého v kapitole 6 . . . 40
Vstupní soubor s definící modelu plazmatu použitého v kapitole 7 . . . 41

Příloha 2. Výpis zdrojového kódu programu PDRS **43**
pdrs.cpp 43
bessel.h 46
bessel.cpp 46
calc.h 49
calc.cpp 49
diel.h 53
diel.cpp 54
io.h 69
io.cpp 72

Úvod

Tato práce se zabývá lineární analýzou vln v kosmickém plazmatu. Hlavním předmětem jsou metody řešení disperzní relace horkého plazmatu. Vzhledem ke složitosti a komplexnosti rovnic pro výpočet tvaru dielektrického tenzoru v modelu horkého plazmatu a disperzní funkce, která na něm závisí, se jedná hlavně o metody řešící tento problém numericky za použití výpočetní techniky.

Kromě pokusu o zmapování existujících postupů tato práce představuje vlastní metodu pro výpočet disperzní relace plazmatu. Tato nová metoda, resp. program, nazvaný PDRS (Plasma Dispersion Relation Solver), umožňuje hledat disperzní relace, tj. řešení disperzní funkce studeného, ale i horkého plazmatu s obecnou distribuční funkcí.

Práce je členěna následovně: začíná stručným úvodem do teorie šíření vln v plazmatu, jehož hlavním učelem je shrnout vzorce používané dále. Následuje kapitola mapující současný výzkum vln v kosmickém plazmatu a stávajících metod pro jejich analýzu.

Po definici cílů práce jsou popsány metody zmíněného programu PDRS a podrobně rozebrány použité algoritmy a numerické metody.

Další kapitoly ukazují použití PDRS, nejprve na příkladech některých základních typů vln v přiblížení studeného i horkého plazmatu, a poté na několika reálných příkladech vln v kosmickém plazmatu.

Práci uzavírá kapitola s dokumentací programu PDRS a závěr, ve kterém jsou diskutovány dosažené výsledky.

1. Teorie šíření vln v plazmatu

1.1 Disperzní relace

Při studiu vln v plazmatu je přirozené vyjít z vlnové rovnice

$$\nabla \times \nabla \times \mathbf{E} + \frac{1}{c^2}\frac{\partial^2 \mathbf{E}}{\partial t^2} + \mu_0 \frac{\partial \mathbf{j}}{\partial t} = 0, \tag{1.1}$$

kterou lze snadno odvodit z Maxwellových rovnic. Hustotu elektrického proudu $\mathbf{j}$ v plazmatu vyjádříme pomocí intenzity elektrického pole $\mathbf{E}$ (viz [Stix, 1992, str. 4]):

$$\mathbf{j} = \sum_s \mathbf{j_s} = \sum_s \sigma_\mathbf{s}.\mathbf{E} = \sum_s -i\omega\epsilon_0\chi_\mathbf{s}.\mathbf{E}. \tag{1.2}$$

V této rovnici se sčítá přes všechny druhy s částic přítomných v plazmatu a $\mathbf{j_s}$ je příspěvek částic typu s k el. proudu, $\sigma_\mathbf{s}(\omega, \mathbf{k})$ příspěvek ke konduktivitě $\sigma(\omega, \mathbf{k})$ a $\chi_\mathbf{s}(\omega, \mathbf{k})$ je tenzor susceptibility s-té komponenty plazmatu.

Použijeme-li dále aproximaci rovinné vlny s vlnovým vektorem $\mathbf{k}$ a úhlovou frekvencí ω, dostaneme rovnici

$$\frac{N^2}{k^2}\mathbf{kk} \cdot \mathbf{E} - N^2\mathbf{E} + \epsilon \cdot \mathbf{E} = 0, \tag{1.3}$$

kde ϵ je dielektrický tenzor

$$\epsilon(\omega, \mathbf{k}) = I + \sum_s \chi_\mathbf{s}(\omega, \mathbf{k}), \tag{1.4}$$

I je jednotkový tenzor a $N = \frac{kc}{\omega}$ je index lomu (viz níže).

Z podmínky existence netriviálního řešení pro $\mathbf{E}$,

$$\det(\frac{N^2}{k^2}\mathbf{kk} - N^2 I + \epsilon) = 0, \tag{1.5}$$

obdržíme rovnici, jenž spolu svazuje (úhlovou) frekvenci ω a vlnový vektor $\mathbf{k}$, tzv. disperzní relaci

$$D(\omega, \mathbf{k}) = 0. \tag{1.6}$$

1.2 Základní vzorce

Z disperzní relace lze odvodit mnoho důležitých parametrů vln, zmíním zde fázovou rychlost

$$\mathbf{v_f} = \frac{\omega}{k^2}\mathbf{k}, \tag{1.7}$$

grupovou rychlost

$$\mathbf{v_g} = \frac{\partial \omega}{\partial \mathbf{k}} \tag{1.8}$$

a vektor indexu lomu

$$\mathbf{N} = \frac{\mathbf{k}c}{\omega}. \tag{1.9}$$

Dále uvedu definice dvou veličin, ktere budu v dalším textu často používat: plazmovou frekvenci ω_{ps}

$$\omega_{ps}^2 = \frac{n_s q_s^2}{\epsilon_0 m_s}, \tag{1.10}$$

kde n_s, q_s a m_s jsou hustota, elektrický náboj a hmotnost částice typu s, a cyklotronovou frekvenci částic typu s

$$\omega_{cs} = \frac{q_s B_0}{m_s}, \tag{1.11}$$

B_0 je velikost magnetické indukce.

1.3 Dielektrický tenzor

Abychom získali konkrétní tvar disperzní relace 1.6, je potřeba znát dielektrický tenzor. Pro jeho vyjádření je potřeba studované plazma blíže specifikovat, tzn. použít nějaký model plazmatu. V této práci používam modely studeného a horkého plazmatu.

1.3.1 Studené plazma

Nejjednodušším modelem je studené plazma, v nemž se berou v úvahu pouze první dva momenty Boltzmanovy rovnice (rovnice kontinuity a zákon zachování hybnosti). Předpokládá se nulová teplota (tedy i tepelná rychlost) a distribuční funkce je pak úměrná Diracově delta-distribuci, centrované na hodnotě makroskopické rychlosti. Z toho také plyne použitelnost modelu: fázová rychlost musí být mnohem větší než tepelná rychlost.

Tvar dielektrického tenzoru v přiblížení studeného plazmatu uvádí např. [Stix, 1992, str. 7] a [Swanson, 2003, str.21].

$$\epsilon = \begin{pmatrix} S & -iD & 0 \\ iD & S & 0 \\ 0 & 0 & P \end{pmatrix} = \begin{pmatrix} \mathcal{K}_1 & \mathcal{K}_2 & 0 \\ -K_2 & \mathcal{K}_1 & 0 \\ 0 & 0 & \mathcal{K}_3 \end{pmatrix}, \tag{1.12}$$

kde S, D a P[1] jsou definovány takto:

$$\mathcal{K}_1 \equiv S \equiv \frac{1}{2}(R+L) \tag{1.13}$$

$$i\mathcal{K}_2 \equiv D \equiv \frac{1}{2}(R-L) \tag{1.14}$$

$$R = 1 - \sum_s \frac{\omega_{ps}^2}{\omega(\omega+\omega_{cs})} \tag{1.15}$$

$$L = 1 - \sum_s \frac{\omega_{ps}^2}{\omega(\omega-\omega_{cs})} \tag{1.16}$$

$$\mathcal{K}_3 \equiv P = 1 - \sum_s \frac{\omega_{ps}^2}{\omega^2}. \tag{1.17}$$

[1] Písmena S, D, P, L a R jsou zkratkami pro *sum*, *difference*, *plasma*, *left* a *right*.

Dosazením do rovnice 1.5 lze odvodit disperzní relaci studeného plazmatu (viz např. [Stix, 1992, str. 8] a [Swanson, 2003, str. 21])

$$AN^4 - BN^2 + C = 0, \tag{1.18}$$

kde

$$A = S\sin^2\theta + P\cos^2\theta \tag{1.19}$$
$$B = RL\sin^2\theta + PS(1+\cos^2\theta) \tag{1.20}$$
$$C = PRL \tag{1.21}$$

a θ je úhel mezi vlnovým vektorem $\mathbf{k}$ a osou z, kterou jsme položili ve směru neporušené složky magnetického pole $\mathbf{B_0}$.

Rovnice 1.18 má řešení

$$N^2 = \frac{B \pm F}{2A}, \tag{1.22}$$

kde $F^2 = B^2 - 4AC$.

Pro analýzu vlastností vln v přiblížení studeného plazmatu tedy vystačíme s analytickým řešením a není nutné řešit disperzní relaci numericky.

1.3.2 Horké plazma

Model horkého plazmatu je obecnější a počítá přímo s distribuční funkcí. Rovnice se ale stávají složitejšími a většinou již není možné je řešit analyticky.

Tvar dielektrického tenzoru pro obecnou distribuční funkci $f_0(p_\perp, p_\parallel)$[2] uvádí např. [Stix, 1992, str. 254]. Tento dielektrický tenzor je dán rovnicí 1.4, kde tenzor susceptibility má tvar

$$\chi_{\mathbf{s}} = \frac{\omega_{ps}^2}{\omega\omega_{cs}} \sum_{n=-\infty}^{\infty} \int_0^\infty 2\pi p_\perp dp_\perp \int_{-\infty}^{\infty} dp_\parallel \left(\frac{\Omega}{\omega - k_\parallel v_\parallel - n\Omega}\mathbf{S_n}\right)_s, \tag{1.23}$$

$$\mathbf{S}_n = \begin{pmatrix} \frac{n^2 J_n^2}{z^2} p_\perp U & \frac{inJ_nJ_n'}{z} p_\perp U & \frac{nJ_n^2}{z} p_\perp W \\ -\frac{inJ_nJ_n'}{z} p_\perp U & (J_n')^2 p_\perp U & -iJ_nJ_n' p_\perp W \\ \frac{nJ_n^2}{z} p_\parallel U & iJ_nJ_n' p_\parallel U & J_n^2 p_\parallel W \end{pmatrix}, \tag{1.24}$$

kde $J_n = J_n(z)$ je Besselova funkce argumentu $z = k_\perp v_\perp/\Omega$, $\Omega = \Omega(p_\perp, p_\parallel) = \omega_c\sqrt{1 - \frac{v^2}{c^2}}$ je relativistická cyklotronová frekvence a

$$U = \frac{\partial f_0}{\partial p_\perp} + \frac{k_\parallel}{\omega}\left(v_\perp \frac{\partial f_0}{\partial p_\parallel} - v_\parallel \frac{\partial f_0}{\partial p_\perp}\right), \tag{1.25}$$

$$W = \left(1 - \frac{n\Omega}{\omega}\right)\frac{\partial f_0}{\partial p_\parallel} + \frac{n\Omega p_\parallel}{\omega p_\perp}\frac{\partial f_0}{\partial p_\perp}. \tag{1.26}$$

Jak jsem již zmiňoval, disperzní relaci horkého plazmatu nelze obecně řešit analyticky a je tedy potřeba použít nějakou aproximaci nebo ji řešit numericky.

Některé možné přístupy k řešení disperzní relace horkého plazmatu jsou zmíněny v následující kapitole.

[2] distribuční funkce je normalizovaná k jedničce $\int f_0(\mathbf{p})d\mathbf{p} = \int \sum_s f_{0s}(\mathbf{p})d\mathbf{p} = 1$

2. Některé výsledky výzkumu vln v kosmickém plazmatu

2.1 WHAMP

Jednou z nejznámějších numerických procedur je WHAMP (Waves in Homogeneous Anisotropic Multi component Plasmas). Jak jsem již popsal v práci [Černý, 2006, str. 19], WHAMP "je počítačový kód v jazyce FORTRAN, sestavený v roce 1982 Rönnmarkem [Rönnmark, 1982]. Umožňuje numerické výpočty disperzních relací vln v homogenním anisotropním plazmatu, nacházejícím se v magnetickém poli. Model plazmatu se může skládat až ze šesti složek, kde každá komponenta je určena svoji hustotou, teplotou, hmotou a elektrickým nábojem částic, anisotropií a driftovou rychlostí podél vnějšího magnetického pole."

Přesněji, distribuční funkce každé složky plazmatu v programu WHAMP může nabývat obecně tvaru (vynechávám zde index s značící složku plazmatu)

$$f_0(p_\perp, p_\parallel) = f_0(v_\perp, v_\parallel) = \frac{1}{(\pi^{\frac{1}{2}} v_t)^3} e^{-(\frac{v_\parallel}{v_t} - v_d)^2} \left(\frac{\Delta}{\alpha_1} e^{-\frac{v_\perp^2}{\alpha_1 v_t^2}} + \frac{1-\Delta}{\alpha_1 - \alpha_2}\left(e^{-\frac{v_\perp^2}{\alpha_1 v_t^2}} - e^{-\frac{v_\perp^2}{\alpha_2 v_t^2}}\right)\right), \tag{2.1}$$

kde v_t je tepelná rychlost příslušné složky plazmatu o teplotě $T = \frac{1}{2} m v_t^2$. Parametr v_d je normalizovaná driftová rychlost ve směru vnějšího magnetického pole, Δ je hloubka ztrátového kužele, α_1 anisotropie ztrátového kužele a konečně α_2 teplotní anizotropie.

Pro zjednoduššení notace používá program WHAMP bezrozměrné proměnné

$$p = \frac{k_\perp v}{\Omega}, z = \frac{k_\parallel v}{\Omega}, x = \frac{\omega}{\Omega}, s_n = \frac{x - z v_d - n}{z}. \tag{2.2}$$

Algoritmus procedury WHAMP popisuje autor v původní praci [Rönnmark, 1982] a v [Rönnmark, 1983].

Využívá tzv. Padého aproximaci

$$Z(s) \approx \sum_{l=1}^{L} \frac{1}{b_l (s - c_l)} \tag{2.3}$$

plazmové disperzní funkce

$$Z\left(\frac{\omega - n\Omega}{k_\parallel v}\right) \equiv \pi^{-\frac{1}{2}} \int_{-\infty}^{\infty} \frac{e^{-(\frac{v_\parallel}{v})^2} dv_\parallel}{v_\parallel - \frac{\omega - n\Omega}{k_\parallel}}. \tag{2.4}$$

Koeficienty Padého aproximace b_l a c_l lze spočítat předem a lze je najít (pro $L = 8$) např. v citované práci [Rönnmark, 1982].

Aplikací Padého aproximace a zavedením funkcí

$$\xi(x, z) = (\alpha s_n + \frac{n}{z}) Z(s_n), \tag{2.5}$$

$$\Lambda_n(\lambda) = e^{-\lambda} I_n(\lambda), \tag{2.6}$$

kde $\lambda = \frac{1}{2}(\frac{k_\perp v}{\Omega})^2$ a I_n je modifikovaná Besselova funkce řádu n, a dále

$$\psi_l = (1 + \alpha\frac{zc_l}{y})R(y, \alpha\lambda) \tag{2.7}$$

a

$$\psi_l^{'} = (1 + \alpha\frac{zc_l}{y})R^{'}(y, \alpha\lambda), \tag{2.8}$$

kde

$$R(y, \lambda) = \sum_{n=-\infty}^{\infty} \frac{n^2}{\lambda}\frac{\Lambda_n(\lambda)}{y - n} \tag{2.9}$$

a

$$R^{'}(y, \lambda) = y^2 \sum_{n=-\infty}^{\infty} \frac{\Lambda_n^{'}(\lambda)}{y - n}, \tag{2.10}$$

dospívá Rönnmark k alternativnímu vyjádření složek tenzoru susceptibility

$$\chi_{11} = \alpha + \alpha \textstyle\sum_{l=1}^{L} b_l y \psi_l \tag{2.11}$$

$$\chi_{12} = -\chi_{21} = i\alpha \textstyle\sum_{l=1}^{L} b_l \psi_l^{'} \tag{2.12}$$

$$\chi_{13} = \chi_{31} = \alpha p \textstyle\sum_{l=1}^{L} (c_l + v_d) b_l \psi_l \tag{2.13}$$

$$\chi_{22} = x_{11} - 2\alpha^2 \textstyle\sum_{l=1}^{L} \frac{b_l}{y} \psi_l^{'} \tag{2.14}$$

$$\chi_{23} = -\chi_{32} = i\alpha p \textstyle\sum_{l=1}^{L} \frac{b_l}{y}(c_l + v_d)\psi_l^{'} \tag{2.15}$$

$$\chi_{33} = \alpha + 2\alpha(v_d^2 + \textstyle\sum_{l=1}^{L} \frac{b_l}{y}(c_l + v_d)^2(x - zv_d + \lambda\psi_l)), \tag{2.16}$$

které tvoří základ algoritmu programu WHAMP.

2.2 WDF

Další numerickou proceduru pro řešení disperzní relace horkého plazmatu představil Santolík ve své práci [Santolik, 1995] pod souhrným označením WDF (Wave distribution function). Jde o soubor několika programů v jazyce FORTRAN. Konkrétně WDF sestává z modulu WDFD (dispersion) řešícího disperzní relaci jednotlivých vlnových módů, WDFM (modes), WDFK (kernels), WDFI (inversion) a WDFS (simulation).

Výstupy procedur WDF jsou v jazyce IDL, což umožnuje jejich snadnou grafickou vizualizaci právě v programu IDL.

2.3 HOTRAY

Jiný přístup k numerickému vyšetřování šíření vln v plazmatu představuje metoda sledování paprsku (ray-tracing).

Příkladem může být program HOTRAY vyvinutý Richardem Hornem [Horne, 1989] na sledování paprsku v horkém bezsrážkovém zmagnetizovaném plazmatu. HOTRAY "umožňuje sledovat buzení či útlum široké škály vln během šíření plazmatem. Přesnost výpočtů je omezena linearitou (malými amplitudami vln), WKB[1] aproximací (tj. vlnová délka vlny musí být malá vzhledem k prostorovým

[1] Wentzel, Kramer a Brillouin

gradientům) a na nerelativistické efekty." (Citace je z práce [Černý, 2006, str. 19-20].)

Kromě již zmíněné práce [Horne, 1989], v níž byl program HOTRAY představen při studiu generování záření TMR (terrestrial myriametric radiation), je jeho použití demonstrováno například také v práci [Horne, 2003] při simulaci interakcí elektronů s hvizdovými vlnovými módy.

V roce 2005 představila Sarah Glauert s Richardem Hornem v práci [Glauert, Horne, 2005] pod názvem PADIE (Pitch angle and energy diffusion of ions and electrons) relativistické rozšíření, které již pracuje s plně relativistickou disperzní relací a lze jej použít pro všechny lineární vlnové módy v plazmatu libovolné hustoty.

2.4 Další

Špatnou konvergenci sum Besselových funkcí v klasickém vzorci pro výpočet tenzoru susceptibility v horkém plazmatu 1.23 se pokusili odstranit autoři článku [Qin a kol., 2007] pomocí Newbergerova vzorce

$$\sum_{n=-\infty}^{\infty} \frac{J_n^2(z)}{a-n} = \frac{\pi}{\sin \pi a} J_{-a}(z) J_a(z). \tag{2.17}$$

Díky němu se jim podařilo nahradit nekonečné sumy součinů Besselových funkcí ve vzorci 1.23 jedním součinem Besselových funkcí komplexního řádu. Dospěli tak k alternativnímu vzorci pro výpočet tenzoru susceptibility v horkém plazmatu (opět vynechávám index s značící složku plazmatu)

$$\chi = \frac{\omega_{ps}^2}{\omega\Omega} \int 2\pi p_\perp dp_\perp dp_\parallel (\mathbf{e}_\parallel \mathbf{e}_\parallel \frac{\Omega}{\omega} \frac{p_\parallel}{p_\perp} (p_\perp \frac{\partial f_0}{\partial p_\parallel} - p_\parallel \frac{\partial f_0}{\partial p_\perp}) + p_\perp U \mathbf{T}), \tag{2.18}$$

$$\mathbf{T} = \begin{pmatrix} \frac{a}{z^2}(Q-1) & \frac{-i}{2z}Q' & \frac{1}{z}(Q-1)\frac{p_\parallel}{p_\perp} \\ \frac{-i}{2z}Q' & \frac{\pi}{\sin \pi a} J'_{-a}(z) J'_a(z) + \frac{a}{z^2} & -\frac{ia}{2}Q' \\ \frac{1}{z}(Q-1)\frac{p_\parallel}{p_\perp} & \frac{i}{2a}Q' & \frac{Q}{a}(\frac{p_\parallel}{p_\perp})^2 \end{pmatrix}, \tag{2.19}$$

kde $Q \equiv \frac{\pi}{\sin \pi a} J_{-a}(z) J_a(z)$, $Q' = \frac{\pi}{\sin \pi a} \frac{\partial}{\partial z}(J_{-a}(z) J_a(z))$ a $a \equiv \frac{\omega - k_\parallel v_\parallel}{\Omega}$. Numerický výpočet dielektrického tenzoru pomocí vzorce 2.18 může být oproti vzorci 1.23 efektivnější. Dle slov autorů může výpočetní úspora v případech, kdy argument Besselových funkcí $z \gg 1$ a jejich suma tak konverguje pomalu, dosahovát až nekolika řádů.

3. Cíle práce

V závěru své práce [Černý, 2006] se zamýšlím nad možností upustit, vzhledem k bouřlivému rozvoji výpočetní techniky, od přílišné optimalizace stávajících numerických metod na řešení disperzní relace horkého plazmatu (např. Padého aproximace v programu WHAMP) a dosáhnout tak přesnějších výsledků za cenu zvýšení výpočetní náročnosti.

Hlavním cílem této práce tak je ověřit tuto možnost a vyvinout alternativní numerickou proceduru pro analýzu disperzní rovnice v horkém plazmatu, která by počítala integrand v rovnici pro výpočet tenzoru susceptibility 1.23 přímo bez jeho aproximace vhodnými funkcemi.

To s sebou přináší dvě významné težkosti: nutnost výpočtu nekonečné sumy, jejíž členy obsahují Besselovy funkce, a možnou divergenci integrandu u některých z těchto členů. Důležitými dílčími cíly je proto analýza zmíněných problémů a nalezení vhodných algoritmů, které je řeší.

S vývojem nových procedur souvisí jejich vyzkoušení. Cílem této práce je tedy i aplikace vyvinutých metod na reálné příklady vln v kosmickém plazmatu a diskuse získaných výsledků.

4. Nové metody PDRS pro lineární analýzu vlastností vln v horkém plazmatu

PDRS (Plasma Dispersion Relation Solver) je program, který, jak název napovídá, umožňuje numericky řešit disperzní relaci. To znamená, že pro zadané $\mathbf{k}$ (resp. ω)[1] hledá (obecně komlexní) ω, (resp. $\mathbf{k}$), pro které je splněna rovnice 1.6.

Program umí počítat disperzní relaci plazmatu skládajícího se z více složek a příspěvek každé složky k dielektrickému tenzoru lze počítat dvěma způsoby: v přiblížení studeného plazmatu, tj. použitím rovnice 1.12 a v modelu horkého plazmatu s obecnou distribuční funkcí (rovnice 1.4 a 1.23). V nejobecnějším případě lze tedy řešit disperzní rovnici vícesložkového homogenního anizotropného horkého plazmatu v magnetickém poli.

4.1 Popis algoritmů

V teto kapitole jsou popsány nejdůležitější algoritmy a numerické metody používané programem PDRS.

4.1.1 Řešení disperzní relace

Pro řešení disperzní relace 1.6, nebo ekvivalentně pro hledání kořenů disperzní funkce $D(\omega, \mathbf{k})$ lze v programu PDRS použít tři metody: metodu bisekce, (jednorozměrnou) Newtonovu metodu a globalně konvergentní (vícerozměrnou) Newtonovu metodu.

Metoda bisekce je popsána v knize [Press a kol., 2002, str. 353]. Lze ji použít při hledání jedné skalarní proměnné a v případě, že se podaří nalézt dvě ruzné hodnoty hledané proměnné, pro které má hodnota disperzní funkce ruzné znaménko. To se ale v praxi málokdy podaří a tato metoda je tak použitelná spíše jen pro testovaní progamu při jednoduchém tvaru disperzní funkce.

Použitelnější je již Newtonova metoda (viz. [Press a kol., 2002, str. 362], kde jí uvádějí i pod názvem Newton-Raphsonova metoda).[2] Prakticky ji lze použít vždy, když máme jen jednu (skalární) neznámou, selhává jen pri určitých tvarech funkce, jejíž kořeny hledá. Podrobněji k omezení Newtonovy metody viz též [Press a kol., 2002, str. 363-364].

Nejobecnější je globálně konvergentní Newtonova metoda, kterou lze použít i v případě více proměnných, nebo jako v našem případě jedné vektorové, nebo komplexní proměnné. Konverguje téměř vždy, může ale nastat případ, kdy "spadne" do lokalního minima, kde $D(\omega, \mathbf{k}) > 0$ a konvergence selže. Program v takovém případě ohlásí chybu a je potřeba hledat řešení s jiným počátečním odhadem. Metoda je popsána stejně jako předchozí metody v knize [Press a kol., 2002, str. 383].

[1] Tyto veličiny, které slouží jako vstup výpočtů, budu dale nazývat řídícími proměnnými.

[2] V této práci ji budu občas označovat jako jednorozměrnou Newtonovu metodu pro odlišení od (vícerozměrné) globalně konvergentní Newtonovy metody.

Z výše uvedeného je zřejmé, že kromě globálně konvergentní Newtonovy metody, nelze vždy použít libovolnou metodu. Ale pokud to povaha problému umožňuje a hledáme jen jednu (skalárni) proměnnou, například hledáme jen reálnou složku frekvence pro dané **k** a imaginární část považujeme a priory za nulovou, je vhodné kvuli zvýšení přesnosti výpočtu a snížení výpočetní náročnosti použít jednodušší metodu.

4.1.2 Diskretizace distribuční funkce

Pro výpočet dielektrického tenzoru v modelu horkého plazmatu je potřeba znát distribuční funkci a provádět s ní určité matematické operace (derivace a integrace). Distribuční funkce v našem případě závisí na dvou proměnných $f_0 = f_0(p_\perp, p_\parallel)$, v programu je tak reprezentována svými hodnotami na dvourozměrné ekvidistanční mřížce o $N_{grid} \times N_{grid}$ bodech. Hodnotu N_{grid}, se kterou má program pracovat, lze volit a je jedním z nejdůležitějších parametrů určujících výpočetní náročnost, protože díky dvourozměrnosti mřížky roste počet potřebných výpočtů kvadraticky s velikostí N_{grid}.

4.1.3 Integrace

Integrace (je potřeba při výpočtu tenzoru susceptibility 1.23) se v programu PDRS provádí zejména pomocí Simpsonova složeného pravidla (viz [Press a kol., 2002, str. 134], kde je uváděno jako "extended Simpson's rule"):

$$\int_{x_1}^{x_N} f(x)dx = h[\frac{1}{3}f_1 + \frac{4}{3}f_2 + \frac{2}{3}f_3 + \frac{4}{3}f_4 + \cdots + \frac{2}{3}f_{N-2} + \frac{4}{3}f_{N-1} + \frac{1}{3}f_N] + O(\frac{1}{N^4}) \tag{4.1}$$

(h je vzdálenost dvou sousedních bodů mřížky a f_i hodnota funkce v i-tem bodě).

Použitelnost této metody je ale v případě výpočtu tenzoru susceptibility omezená. Jak je vidět ze vzorce 1.23, integrand pro hodnoty $p_\parallel$ splňující podmínku

$$p_\parallel = m\frac{\omega - n\Omega}{k_\parallel} \tag{4.2}$$

diverguje. Pokud tento případ nastane pro takové n, $p_\parallel$ a $p_\perp$, pro které ještě nezanedbáváme Besselovy funkce, ani distribuční funkci (viz níže), je integrace nepřesná.

Proto je v programu PDRS implementován i následující algoritmus pro výpočet integrace přes $p_\parallel$. Spočteme dopředu všechna celá čísla n, pro které je splněna podmínka 4.2 pro $p_\parallel$ z integračního intervalu (ten je v důsledku konečnosti diskretizační mřížky také konečný). Integraci pak provádíme zvlášť pro každý interval (p_n, p_{n+1}) (označil jsem $p_n \equiv m\frac{\omega - n\Omega}{k_\parallel}$) a použijeme k tomu nějakou otevřenou metodu (tj. metodu, která nevyužívá krajní body intervalu). Konkrétně program PDRS využívá vzorce (viz [Press a kol., 2002, str. 135])

$$\int_{x_1}^{x_N} f(x)dx = h[\frac{27}{12}f_2 + 0 + \frac{13}{12}f_4 + \frac{4}{3}f_5 + \cdots \tag{4.3}$$

$$+\frac{4}{3}f_{N-4} + \frac{13}{12}f_{N-3} + 0 + \frac{27}{12}f_N - 1] + O(\frac{1}{N^4}). \tag{4.4}$$

(U vnitřních bodů se střídá koeficient 4/3 a 2/3.)

Určitou komplikací je, že je potřeba znát hodnotu integrandu a tedy derivací distribuční funkce i jinde než v bodech mřížky. Ty metoda PDRS získává interpolací hodnot z nejbližších bodů mřížky (viz níže).

Hlavní nevýhodou tohoto algoritmu oproti Simpsonově metodě je větší výpočetní náročnost.

Z použitých metody integrace vyplývá určité omezení programu PDRS, resp. podmínky, které je potřeba klást na distribuční funkci. Ve vzorci 1.23 vystupují nevlastní integraly, které ale dle uvedeného vzorce nahrazujeme (vlastními) určitými integraly. Je tedy nutné, aby integrand v bodech mimo uvažovanou mřížku byl dostečně malý, aby jeho příspěvky k integraci bylo možné zanedbat. To bude splněno, pokud tam budou dostatečně malé příslušné derivace distribuční funkce. Výše uvedené je například dobře splněno pro Maxwellovské rozložení distribuční funkce, samozřejmě za předpokladu vhodné normalizace, tj. velikosti rozdílu hybnosti v sousedních bodech mřížky $\Delta p = p_i - p_{i-1}$.

Dále je vhodné, jak je vidět ze vzorce 4.1, aby počet bodů mřížky N_{grid} byl lichý. Nicméně, vhledem k pravidlům kladeným na distribuční funkci, lze chybu způsobenou případnou sudostí N_{grid} zanedbat.

4.1.4 Derivace

Pro výpočet tenzoru susceptibility v modelu horkého plazmatu je rovněž potřeba znát (parciální) derivace distribuční funkce $\frac{\partial f_0}{\partial p_\perp}$ a $\frac{\partial f_0}{\partial p_\parallel}$.

Pro jejich výpočet se používá symetrická dvou nebo 5-ti bodová metoda (viz vzorce 4.5, resp. 4.6) - lze mezi nimi volit.

$$f_i' = \frac{f_{i+1} - f_{i-1}}{2h} \tag{4.5}$$

$$f_i' = \frac{f_{i-2} - 8f_{i-1} + 8f_{i+1} - f_{i+2}}{12h} \tag{4.6}$$

Výhodou je, ze tyto derivace nezávisí na frekvenci ani vlnovém vektoru, a lze je tedy spočítat jednorázově dopředu před vlastním řešením disperzní relace.

4.1.5 Interpolace

Pro výpočet integrandu ve vzorci 1.23 dle výše popsaného algoritmu využívajícího otevřenou motodu je potřeba znát hodnoty derivací distribuční funkce i mimo body mřížky. Ty lze odhadnout pomocí interpolace hodnot v okolních bodech mřížky. Konkrétně je pro tento učel v programu PDRS implementován Nevillův algoritmus pro polynomiální interpolaci, který je popsán např. v [Press a kol., 2002, str. 108]. Řád použitého interpolačního polynomu lze zadat v řídících parametrech programu, jako výchozí je nastaven rád 1, tj. lineární interpolace.

4.1.6 Suma Besselových funkcí

Zásadní otázkou při výpočtu dielektrického tenzoru (resp. tenzoru susceptibility) horkého plazmatu pomocí vzorce 1.23 je výpočet nekonečné sumy Besselových funkcí.

Situace je ještě poměrně jednoduchá v případě, že velikost argumentu Besselových funkcí $z = k_\perp v_\perp/\Omega$ je rovna nule (v rámci výpočetní přesnosti). V sumě pak zůstává nenulovými konečné množství členů a není potřeba žádné zanedbávat. Vzorec 1.23 má v limitě pro $z \to 0$ tvar (vynechávám zde index s značící složku plazmatu)

$$\lim_{z\to 0}\chi = \lim_{z\to 0}\frac{\omega_p^2}{\omega\omega_{cs}}\sum_{n=-\infty}^{\infty}\int_0^\infty 2\pi p_\perp dp_\perp \int_{-\infty}^{\infty} dp_\parallel \frac{\Omega}{\omega - k_\parallel v_\parallel - n\Omega}\mathbf{S_n} \tag{4.7}$$

$$\lim_{z\to 0}\chi = \frac{\omega_p^2}{\omega\omega_{cs}}\int_0^\infty 2\pi p_\perp dp_\perp \int_{-\infty}^{\infty} \Omega\mathbf{I_0} dp_\parallel, \tag{4.8}$$

$$\mathbf{I}_0 = \begin{pmatrix} \frac{1}{4}(\frac{1}{\omega-k_\parallel v_\parallel-\Omega}+\frac{1}{\omega-k_\parallel v_\parallel+\Omega})p_\perp U & \frac{1}{4}(\frac{1}{\omega-k_\parallel v_\parallel-\Omega}-\frac{1}{\omega-k_\parallel v_\parallel+\Omega})p_\perp U & 0 \\ \frac{1}{4}(\frac{1}{\omega-k_\parallel v_\parallel+\Omega}-\frac{1}{\omega-k_\parallel v_\parallel-\Omega})p_\perp U & \frac{1}{4}(\frac{1}{\omega-k_\parallel v_\parallel-\Omega}+\frac{1}{\omega-k_\parallel v_\parallel+\Omega})p_\perp U & 0 \\ 0 & 0 & \frac{p_\parallel W}{\omega-k_\parallel v_\parallel} \end{pmatrix}. \tag{4.9}$$

V opačném případě [3] má suma nekonečně členů a je potřeba nějaké aproximace. Program PDRS počítá sumu přímo, přičemž používá dvě možnosti, jak určit, kde sumu ukončit, tj. kdy už může vyšší členy zanedbat. První způsob je, že zanedbává členy obsahující Besselovy funkce vyššího, než určitého, dopředu zadaného, řádu (v absolutní hodnotě). Zároveň při výpočtu sleduje velikost členů každého řádu a výpočet ukončíme, klesne-li pod určitou mez. Oba parametry, tj. maximální řád člena sumy a jeho minimální velikost lze zadat na vstupu programu.

Algoritmus pro výpočet sumy Besselových funkcí použitý v programu PDRS vychází z algoritmu pro výpočet Besselovy funkce n-tého řádu, popsaného v [Press a kol., 2002, str. 235]. Využívá známého rekurentního vzorce

$$J_{n+1}(x) = \frac{2n}{x}J_n(x) - J_{n-1}(x), \tag{4.10}$$

díky němuž lze spočítat n-tý člen sumy pomocí dvou předchozích. Pro výpočet derivací Besselových funkcí se využívá vztahu

$$J'_{n+1}(x) = \frac{1}{2}(J_{n-1}(x) - J_{n+1}(x)). \tag{4.11}$$

[3] tj. pro $z > 0$, neboť, jak je zřejmé z definice, z nemůže být záporné

5. Testování metody PDRS

5.1 Mód volného prostoru

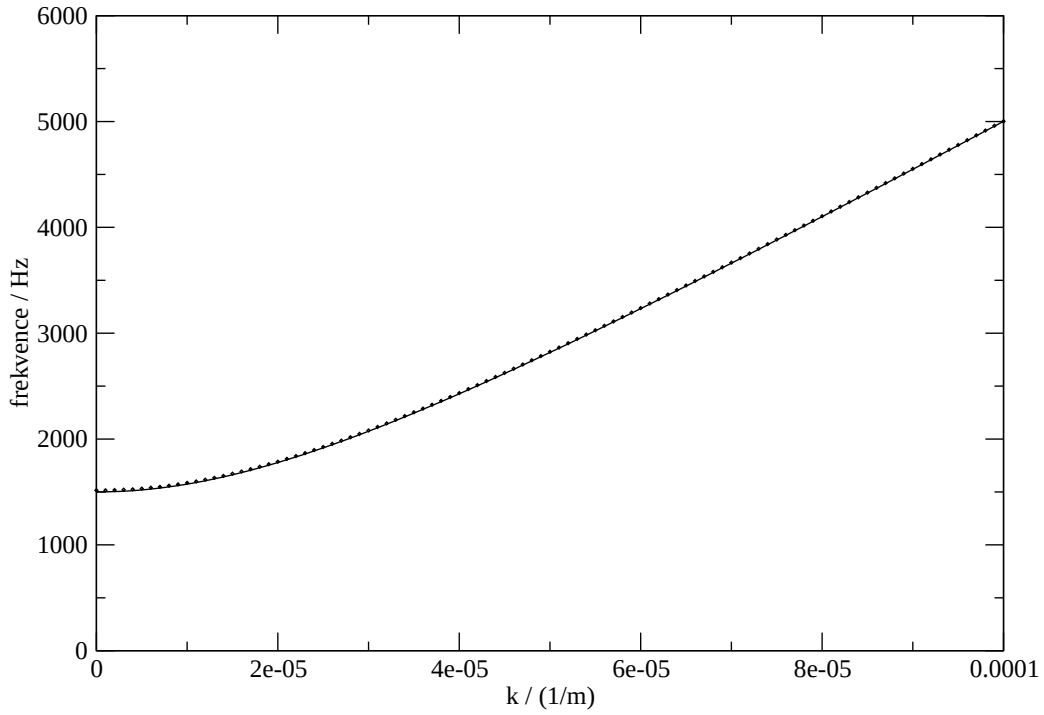
Nejednodušším případem šíření je světelná vlna s disperzní relací

$$\omega^2 = \omega_p^2 + k^2c^2 \tag{5.1}$$

(c je rychlost světla ve vákuu), z které je vidět, že se vlna šíří nezávisle na směru a velikosti magnetického pole.

Výpočet jsem provedl pro plazma skladajícího se z elektronů s plazmovou frekvencí $f_{pe} = \frac{\omega_{pe}}{2\pi}$=1500Hz a v přiblížení horkého plazmatu. Vypočtená disperzní relace je společně s teoretickou zobrazena v grafu 5.1.

Obrázek 5.1: Disperzní relace světelné vlny pro plazma skládající se z elektronů s f_p=1500Hz. Tečkami jsou vynešeny hodnoty vypočtené programem PDRS v přiblížení horkého plazmatu. Plnou čarou je znázorněna teoretická disperzní relace.



5.2 Podélné šíření

Podélným šířením se myslí to, že vlnový vektor zůstává rovnoběžný se siločárami magnetického pole, tj. $\mathbf{k} \parallel \mathbf{B}$. V programu jsem proto držel úhel θ nulový a hledal tak závislost jediné proměnné - velikosti vlnového vektoru k - na frekvenci. Výpočty v tomto testu jsem prováděl jen pro reálnou frekvenci, tedy s nulovou imaginární složku. Díky tomu jsem mohl pro hledání kořenu disperzní funkce použít (jednorozměrnou) Newtonovu metodu.

Plazma jsem uvažoval dvousložkové, složené z iontů (protonů) a elektronů. Cyklotronovou frekvenci elektronů jsem nastavil na $f_{ce} = \frac{\omega_{ce}}{2\pi}$=100Hz a plazmovou frekvenci na $f_{pe} = \frac{\omega_{pe}}{2\pi}$=200Hz.

Dielektrický tenzor jsem počítal oběma způsoby, tedy jak v přiblížení studeného plazmatu, tak horkého plazmatu. Pro horké plazma jsem počítal s distribuční funkcí s maxwellovským rozdělením odpovídajícím teplotě T=240K (u obou složek plazmatu).

V obou případech výsledky v rámci zadané přesnosti (10^{-5}) odpovídaly teoretickým disperzním relacím levotočivého a pravotočivého módu v přiblížení studeného plazmatu (viz např. [Swanson, 2003, str. 23]) Vypočtané disperzní relace jsou vynešeny v grafech 5.2 a 5.3.

Velká přesnost programu PDRS v případě paralelního šíření je dána tím, že není potřeba žádné aproximace pro výpočet sumy Besselových funkcí. Jejich argument je totiž roven nule a suma pak má konečné (a malé) množství nenulových členů.

5.3 Šíření ve směru kolmém na směr magnetického pole

Pro vlny šířící se ve směru kolmém na směr magnetického pole, tj. $\mathbf{k} \perp \mathbf{B}$, lze v přiblížení studeného plazmatu odvodit dvě disperzní relace ([Swanson, 2003, str. 24]). Disperzní relaci řádné vlny

$$N^2 = 1 - \frac{\omega_{pi}^2}{\omega^2} - \frac{\omega_{pe}^2}{\omega^2}. \tag{5.2}$$

a mimořádné vlny.

$$N^2 = 1 - \frac{[(\omega + \omega_{ci})(\omega - \omega_{ce}) - \omega_p^2][(\omega - \omega_{ci})(\omega + \omega_{ce}) - \omega_p^2]}{(\omega^2 - \omega_{ci}^2)(\omega^2 - \omega_{ce}^2) + \omega_p^2(\omega_{ce}\omega_{ci} - \omega^2)}, \tag{5.3}$$

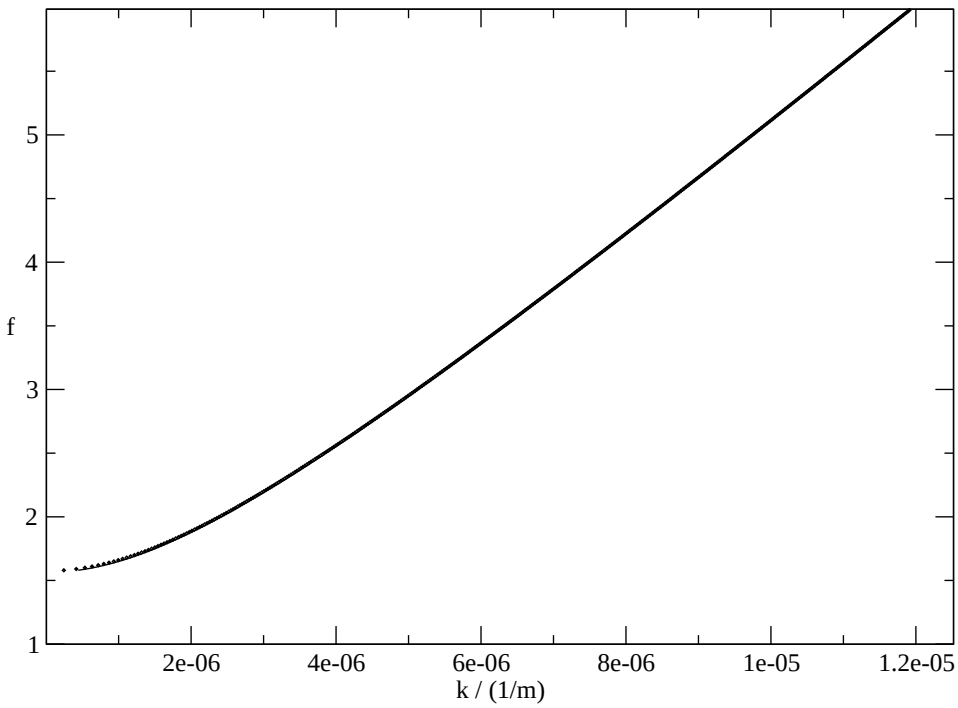
kde $\omega_p^2 \equiv \omega_{pe}^2 + \omega_{pi}^2$.

Výpočet jsem prováděl podobně jako v předchozí kapitole pro dvousložkové plazma. Cyklotronovou frekvenci elektronů jsem nastavil na f_{ce}=80Hz a plazmovou frekvenci na f_{pe}=100Hz. Distribuční funkce obou složek plazmatu odpovídala maxwellovskému rozdělení s teplotou T=240000K.

Hodnoty vypočtené v přiblížení studeného plazmatu opět v rámci zadané přesnosti odpovídaly teorii.

Výsledky počítané v modelu horkého plazmatu jsou vyneseny v grafech 5.4 a 5.5. Povedlo se nalézt řádný i mimořádný vlnový mód. Z vypočtené disperzní

Obrázek 5.2: Disperzní relace levotočivé vlny pro plazma skládající se z iontů a elektronů s f_{ce}=100Hz a f_{pe}=200Hz. V grafu je vynešena normalizovaná frekvence $f = \omega/2\pi f_{ce}$ v závislosti na vlnovém vektoru k. Tečkami jsou vynešeny hodnoty vypočtené programem PDRS v přiblížení horkého plazmatu s teplotou T=240000K. Plnou čarou je znázorněna teoretická disperzní relace.



relace mimořádného módu lze i odečíst hodnotu horní hybridní rezonance $\omega_{UH} \doteq 128Hz$, která dobře odpovídá teoretické hodnotě

$$\omega_{UH}^2 \equiv \omega_{pe}^2 + \omega_{ce}^2. \tag{5.4}$$

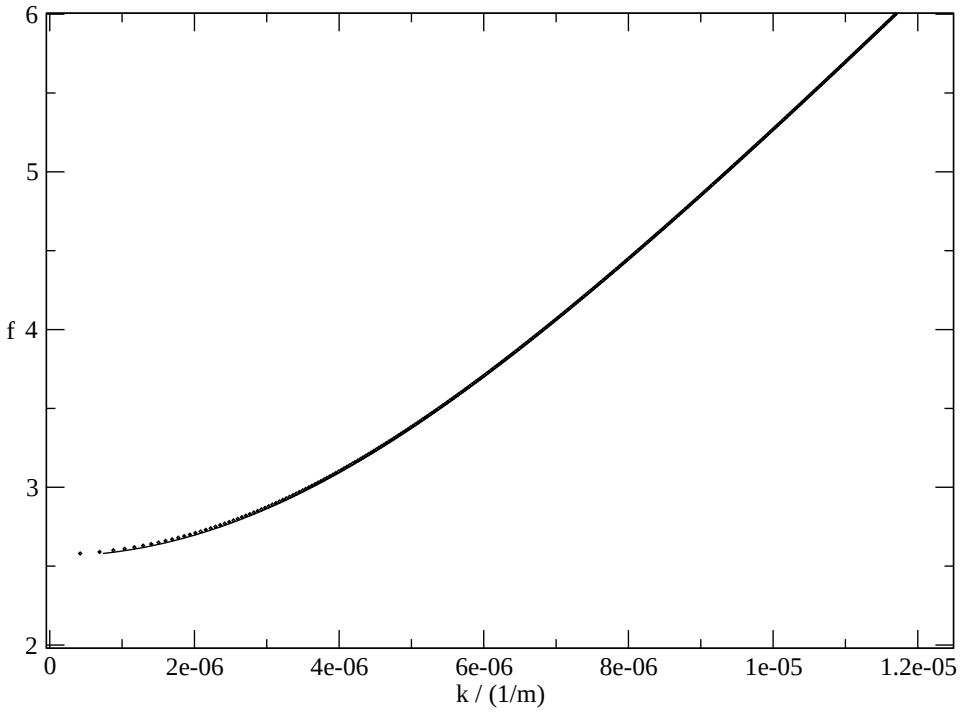
5.4 Lamgmuirovy vlny

Příkladem vln typických pro horké plazma můžou být Langmuirovy vlny. Jejich disperzní relaci pro $\omega_c \ll \omega_p$ uvádí např. [Willes, Cairns, 2000].

$$\omega_L^2(k,\theta) = \omega_p^2 + \omega_c^2 \sin^2\theta + 3k^2v^2. \tag{5.5}$$

Výpočty jsem prováděl pro jednosložkové plazma skládající se z elektronů s f_{ce}=450Hz a f_{pe}=1500Hz a teplotou 500eV.

Obrázek 5.3: Disperzní relace pravotočivé vlny pro plazma skládající se z iontů a elektronů s f_{ce}=100Hz a f_{pe}=200Hz. V grafu je vynešena normalizovaná frekvence $f = \omega/2\pi f_{ce}$ v závislosti na vlnovém vektoru k. Tečkami jsou vynešeny hodnoty vypočtené programem PDRS v přiblížení horkého plazmatu s teplotou T=240000K. Plnou čarou je znázorněna teoretická disperzní relace.



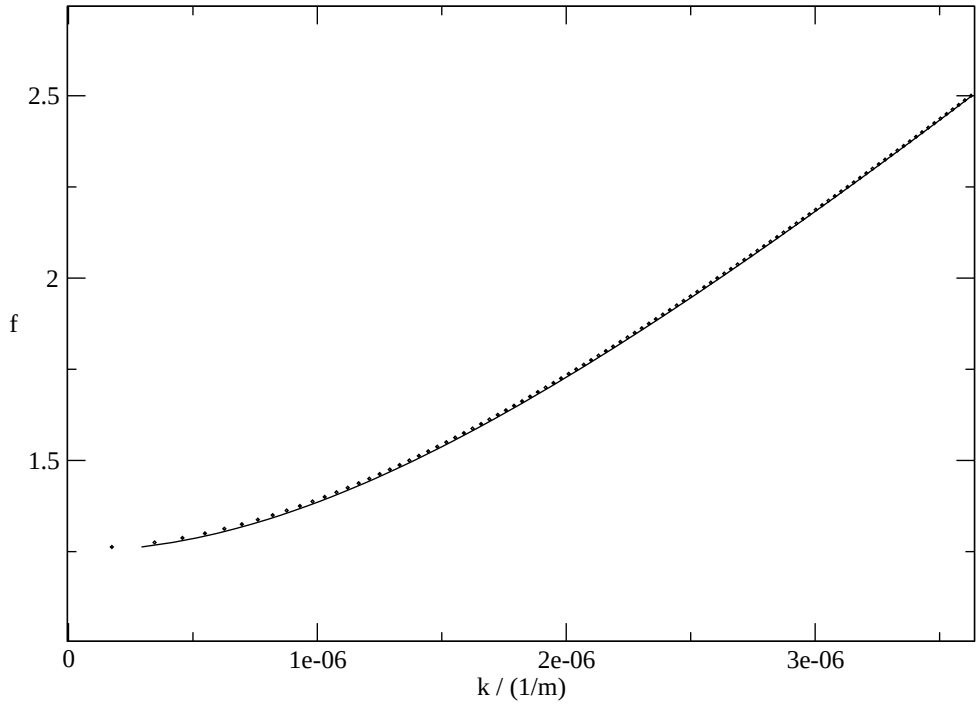
Výsledky jsou shrnuty v grafu 5.6, kde je vynešena normalizovaná frekvence $f = \omega/2\pi f_{ce}$ v závislosti na vlnovém vektoru k při paralelním šíření a grafu 5.7 se závislostí normalizované frekvence na směru šíření θ při konstantní velikosti vlnového vektoru $k = 0.001m^{-1}$.

Na těchto výsledcích, zejména na druhém z grafů (5.7) je již vidět určitá odchylka vypočtených hodnot od teoretických. To je dáno zejména nutností použít složitější integrační algoritmus, jak je zmíněno v kapitole 4.1.3, protože Simpsonova integrace zde již není, na rozdíl od předchozích příkladů, vhodná.

I proto výpočty programu PDRS již v tomto případě podstatně závisí na použitých řídicích parametrech. Jejich hodnoty uvádím v tabulce 5.1.

Přesnost výsledků spolu s dalšími je podrobněji diskutována v závěru práce.

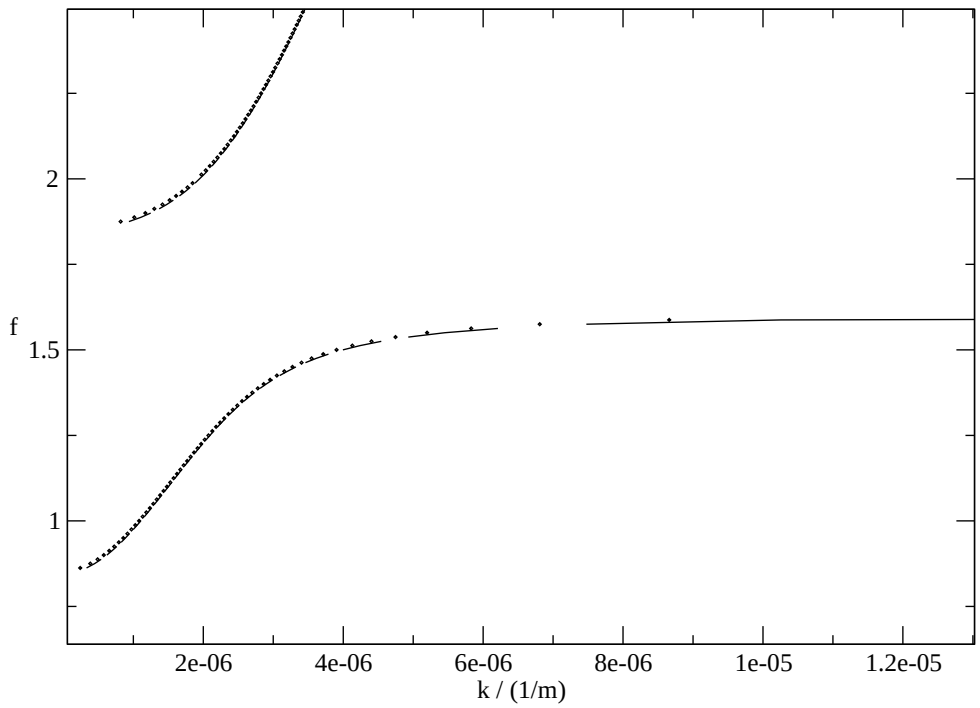
Obrázek 5.4: Disperzní relace řádné vlny pro plazma skládající se z iontů a elektronů s f_{ce}=80Hz a f_{pe}=100Hz. V grafu je vynešena normalizovaná frekvence $f = \omega/2\pi f_{ce}$ v závislosti na vlnovém vektoru k. Tečkami jsou vynešeny hodnoty vypočtené programem PDRS v přiblížení horkého plazmatu s teplotou T=240000K. Plnou čarou je znázorněna teoretická disperzní relace.



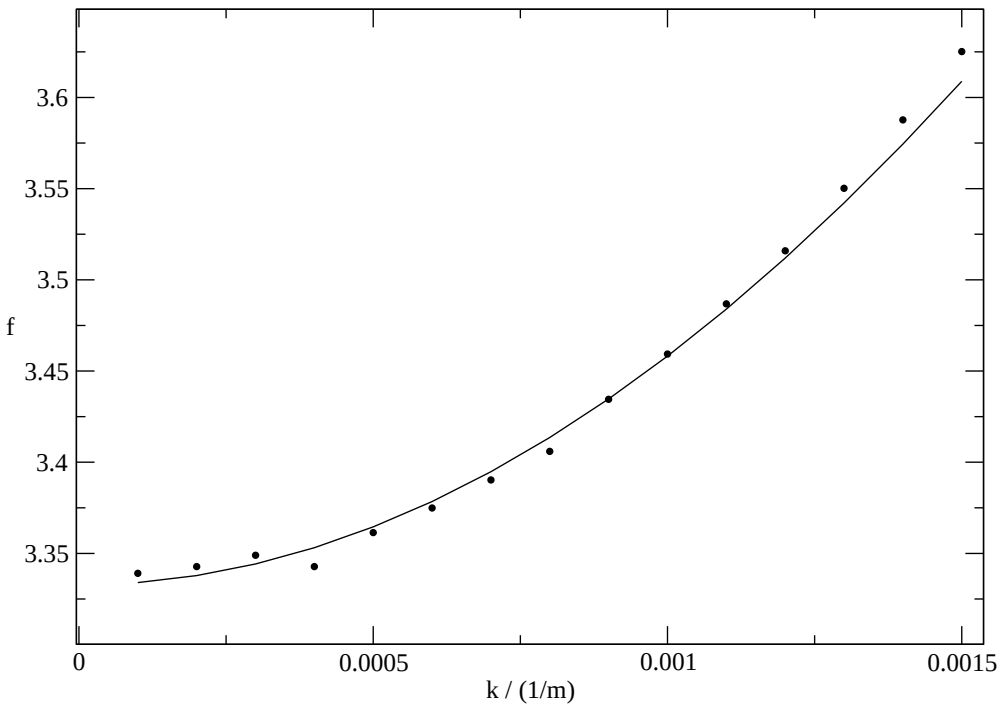
Tabulka 5.1: Hodnoty řídících parametrů použitých při výpočtu disperzní relace Langmuirových vln.

parametr	hodnota
grid size	127
integration method	open
root searching method	global newton
accuracy	1.0e-4
max iter. loops	20
max bessel order	100
bessel sum accuracy	1.0e-5

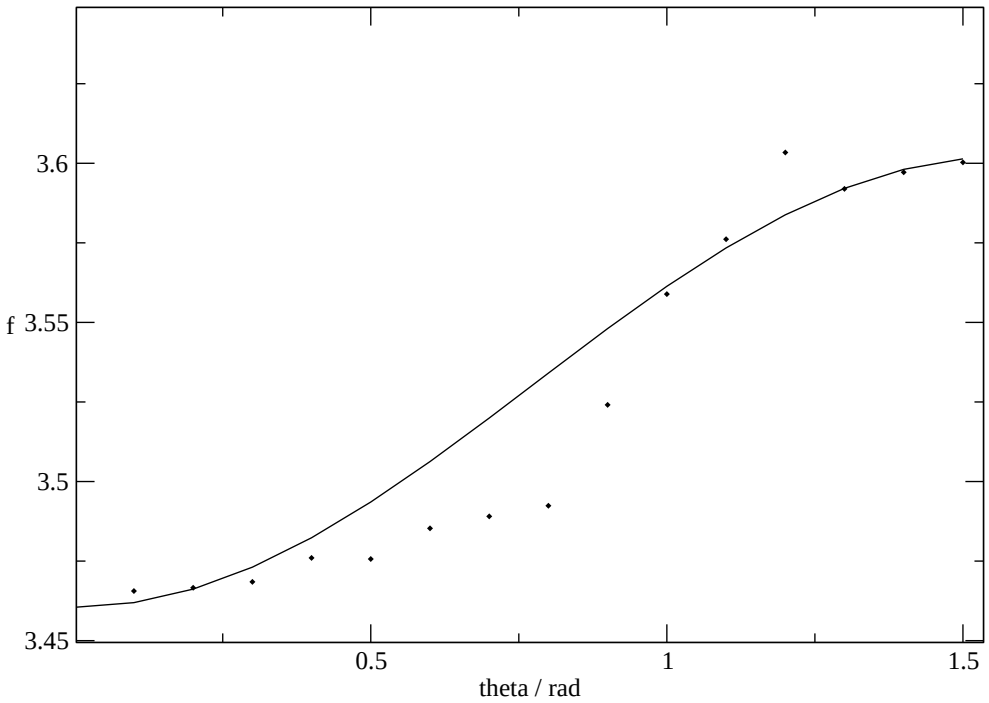
Obrázek 5.5: Disperzní relace řádné vlny pro plazma skládající se z iontů a elektronů s f_{ce}=80Hz a f_{pe}=100Hz. V grafu je vynešena normalizovaná frekvence $f = \omega/2\pi f_{ce}$ v závislosti na vlnovém vektoru k. Tečkami jsou vynešeny hodnoty vypočtené programem PDRS v přiblížení horkého plazmatu s teplotou T=240000K. Plnou čarou je znázorněna teoretická disperzní relace.



Obrázek 5.6: Disperzní relace Langmuirovy vlny pro plazma skládající se z elektronů s f_{ce}=450Hz a f_{pe}=1500Hz a teplotou 500eV. V grafu je vynešena normalizovaná frekvence $f = \omega/2\pi f_{ce}$ v závislosti na vlnovém vektoru k při směru šíření $\theta = 0$. Tečkami jsou vynešeny hodnoty vypočtené programem PDRS v přiblížení horkého plazmatu. Plnou čarou je znázorněna teoretická disperzní relace.



Obrázek 5.7: Disperzní relace Langmuirovy vlny pro plazma skládající se z elektronů s f_{ce}=450Hz a f_{pe}=1500Hz a teplotou 500eV. V grafu je vynešena normalizovaná frekvence $f = \omega/2\pi f_{ce}$ v závislosti na směru šíření θ při velikosti vlnového vektoru $k = 0.001m^{-1}$. Tečkami jsou vynešeny hodnoty vypočtené programem PDRS v přiblížení horkého plazmatu. Plnou čarou je znázorněna teoretická disperzní relace.



6. Harmonické emise pozorované družicemi Cluster v blízkosti plazmapauzy

V této kapitole se pokusím o reprodukci některých výsledků, k nimž došli autoři v práci [Grimald, Santolik, 2010].

Zmíněná práce se zabývá elektromagnetickým zářením netepelného kontinua (NTC - nonthermal continuum) v oblasti plasmapauzy[1] Země. Konkrétně studuje záření odpovídající emisním čarám ve spektrogramu naměřeným družicemi Cluster v oblasti, kde lokální horní hybridní frekvence plazmatu je blízká násobkům elektronové cyklotronové frekvence a zejména pak emisi odpovídající osmé harmonické frekvenci.

6.1 Popis modelu plazmatu

Model plazmatu zde budu uvažovat stejný, jako je použit v práci [Grimald, Santolik, 2010]. Její autoří vycházejí z rozdělení elektronů naměřeného přístrojem PEACE/HEEA[2] mezi časy 0956:57 a 0957:18 UT v energetickém rozmezí 40eV-26keV a z celkové hustoty elektronů naměřené přístrojem WHISPER[3].

Plasma charakterizovali čtyřmi složkami - tři elektronové komponenty s bi-Maxwellovským rozdělením a studené ionty (H^+). Parametry distribučních funkcí složek plazmatu získali interpolací metodou nejmenších čtverců naměřené hustoty fázového prostoru.

Parametry jednotlivých složek plazmatu uvádím v tabulce 6.1 (n je hustota, T teplota a značení ostatních parametrů je stejné jako v rovnici 2.1).

Tabulka 6.1: Parametry distribučních funkcí jednotlivých složek plazmatu modelujících rozdělení částic naměřeného přístrojem PEACE/HEEA mezi 0956:57 a 0957:18 UT v energetickém rozmezí 40eV-26keV (značení viz text).

komponenta	částice	$n[cm^{-3}]$	$T[eV]$	v_d	Δ	α_1	α_2
1	elektrony	47.24	10	0	1	0.08	0
2	elektrony	0.82	1180	0	1	0.98	0
3	elektrony	0.41	1890	0	0	1.16	0.232
4	ionty (H^+)	48.47	0	0	1	1	0

6.2 Výsledky

Podobně jako v práci [Grimald, Santolik, 2010] jsem se zaměřil na hledání vlnových módů v okolí osmé harmonické frekvence $f \sim 8f_{ce}$.

[1] Oblast na hranici plasmasféry, kde prudce (řádově) klesá hustota plasmatu.

[2] PEACE - Plasma Electron And Current Experiment

[3] WHISPER - Waves of High frequency and Sounder for Probing of Electron density by Relaxation

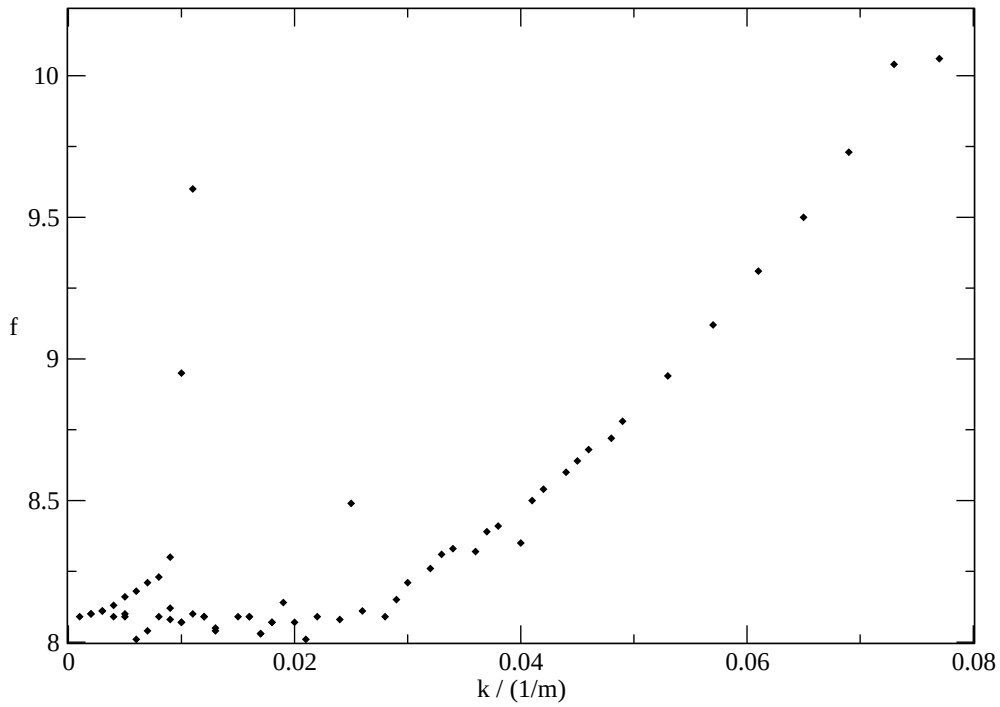
Výsledky numerického řešení disperzní relace pro paralelní šíření jsou uvedeny v grafu 6.1, kde je zřetelný Langmuirův vlnový mód.

Vypočtené disperní relace pro šíření ve směru kolmém na směr magnetického pole jsou vynešeny v grafu 6.2. Podařilo se nalézt Bernsteinovy vlnové módy kolem osmé a deváté harmonické frekvence.

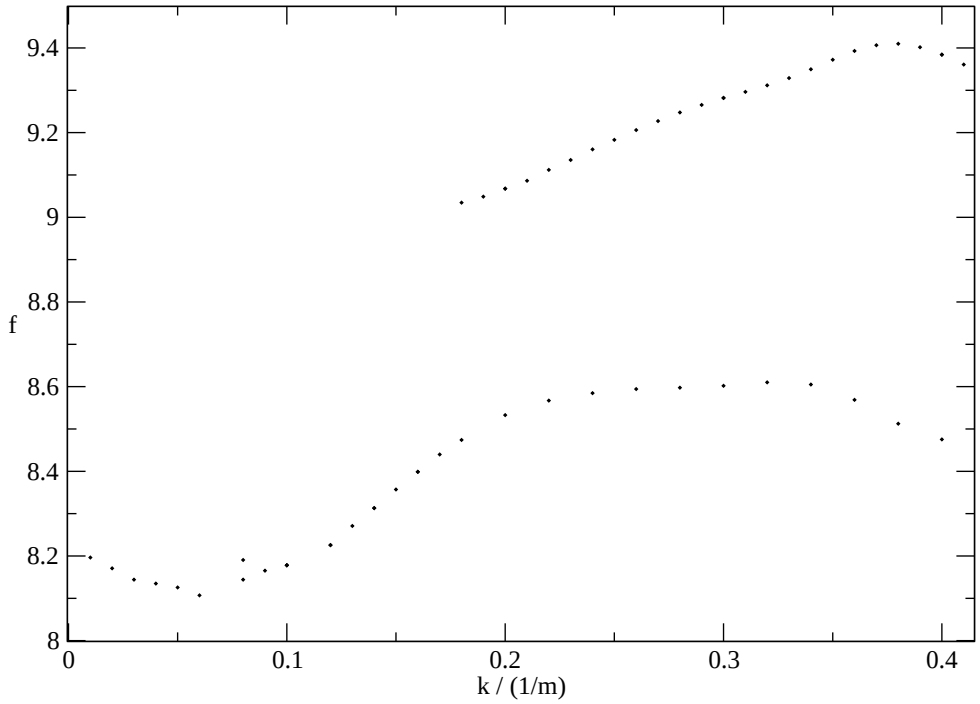
Zmiňované výsledky získané programem PDRS odpovídají výsledkům, k nimž došli autoři v práci [Grimald, Santolik, 2010], jsou ale méně přesné. Důvody jsou stejné, jako v předchozí kapitole a jsou diskutovány v závěru práce.

Řídící parametry programu, použité pro simulace v této kapitole, jsou uvedeny v tabulce 6.2.

Obrázek 6.1: Výsledky numerického řešení disperzní relace pro paralelní šíření pro frekvence blízké $8f_{ce}$. V grafu je vynešena normalizovaná frekvence $f = \omega/2\pi f_{ce}$ v závislosti na velikosti vlnového vektoru k.



Obrázek 6.2: Výsledky numerického řešení disperzní relace pro šíření ve směru kolmém na směr magnetického pole pro frekvence blízké $8f_{ce}$ a $9f_{ce}$. V grafu je vynešena normalizovaná frekvence $f = \omega/2\pi f_{ce}$ v závislosti na velikosti vlnového vektoru k.



Tabulka 6.2: Hodnoty řídících parametrů použitých při výpočtu disperzních relací v kapitole 6.

parametr	hodnota
grid size	63
integration method	open
root searching method	global newton
accuracy	1.0e-4
max iter. loops	20
max bessel order	50
bessel sum accuracy	1.0e-4

7. Emise typu chorus

7.1 Popis modelu plazmatu

V této kapitole používám model plazmatu, s nímž pracují autoři práce [Santolik, 2010] a který vytvořili na základě měření přístroje PEACE na palubě sondy Cluster. Měření zachycuje emisi typu chorus v rovníkové oblasti zemské magnetosféry.

Parametry jednotlivých složek plazmatu uvádím v tabulce 7.1. Elektronová cyklotronova frekvence je $f_{ce} = 9.2kHz$.

Tabulka 7.1: Parametry distribučních funkcí jednotlivých složek plazmatu modelujících rozdělení částic naměřeného přístrojem PEACE během čtyř sekund kolem 0122:54.

komponenta	částice	$n[cm^{-3}]$	$T[eV]$	α_1
1	elektrony	31.25	3.5	1.0
2	elektrony	1.96	15.6	2.1
3	elektrony	0.31	35.3	13.5
4	elektrony	0.03	7000	4.0
5	ionty (H^+)	33.54	1	1.0

7.2 Výsledky

Výsledky numerického řešení disperzní relace jsou shrnuty v grafu 7.1. Jde o graf závislosti normalizované frekvence $f = \omega/2\pi f_{ce}$ na podélné složce vlnového vektoru $k_{\|}$ pro různé hodnoty složky vlnového vektoru $k_{\perp}$ kolmé ke směru magnetického pole.

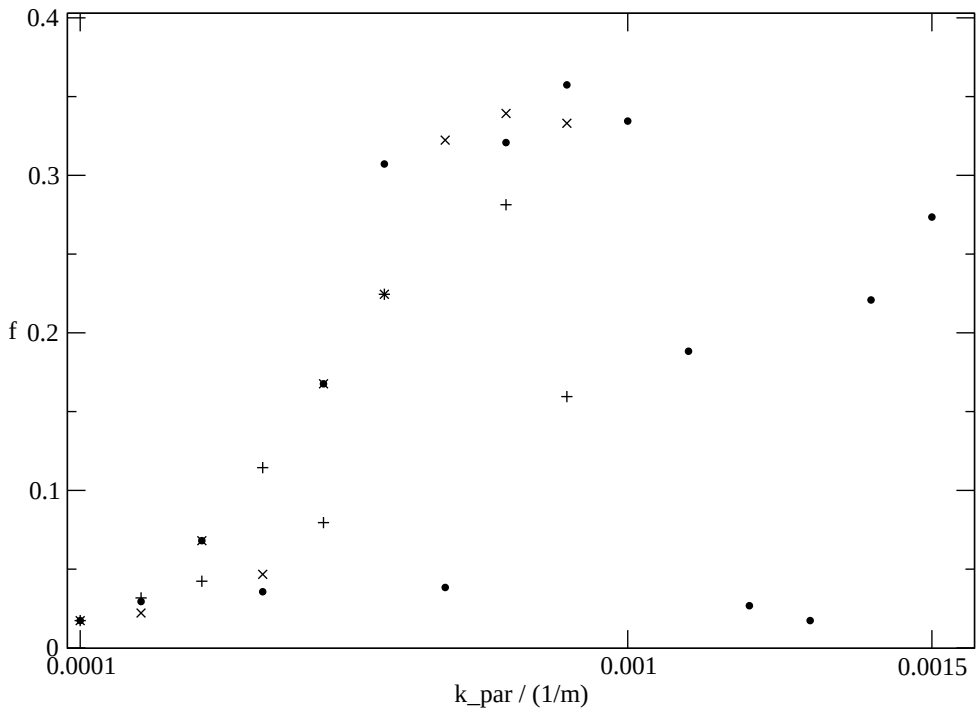
Bohužel se nepodařilo v tomto případě najít souvislejší množiny řešení příslušejících konkrétním vlnovým módům. Důvodem je, kromě nepřesností zmíněných v předchozích kapitolách, velká citlivost nalezeného řešení na počátečním odhadu. Tj. problém určit takový počáteční odhad pro rešení disperzní relace, aby algoritmus hledající kořeny disperzní funkce nalezl právě kořen odpovídající vlnovému módu, jenž nás právě zajímá.

Příslušné řídící parametry programu jsou uvedeny v tabulce 7.2

Tabulka 7.2: Hodnoty řídících parametrů použitých při výpočtu disperzních relací v kapitole 7.

parametr	hodnota
grid size	63
integration method	open
root searching method	global newton
accuracy	1.0e-4
max iter. loops	40
max bessel order	50
bessel sum accuracy	1.0e-4

Obrázek 7.1: Výsledky numerického řešení disperzní relace - graf závislosti normalizované frekvence $f = \omega/2\pi f_{ce}$ v závislosti na podélné složce vlnového vektoru $k_{\parallel}$ pro různé hodnoty složky vlnového vektoru $k_{\perp}$ kolmé ke směru magnetického pole. Plusem jsou označeny hodnoty s $k_{\perp} = 0$, křížkem pak hodnoty pro $k_{\perp} = 0.0001m^{-1}$ a kolečkem pro pro $k_{\perp} = 0.0005m^{-1}$.



8. Dokumentace programu PDRS

8.1 Vstupy programu PDRS

Program v pracovním adresáři očekává dva soubory: *plasma_data.csv* a *control_data.csv*. Z prvního souboru načítá parametry modelu plazmatu, z druhého pak parametry ovlivňující běh programu (rozsah řídících proměnných, zvolené metody výpočtu apod.).

U obou zmíněných souborů platí, že začíná-li nějaký řádek znakem #, program jej ignoruje a lze jej tak použít jako komentář. Ostatní řádky by měly mít formát *parametr:hodnota*, jinak program ohlasí chybu.

8.1.1 Definice modelu plazmatu

Jak již bylo zmíněno, ze souboru *plasma_data.csv* program načítá parametry týkající se modelu plazmatu. Seznam parametrů, ktere je možno/nutno zadat a jejich možných hodnot je v tabulce 8.1. U nepovinných parametrů je uvedena i jejich výchozí hodnota. [1]

V příloze číslo 1 dále je vypsán obsah těchto souborů definujících modely plazmatu použitých v této práci.

8.1.2 Řízení běhu programu

Parametry potřebné pro řízení běhu programu se načítají ze souboru *control_data.csv*. Jejich seznam je v tabulce 8.2

8.1.3 Distribuční funkce

Distruční funkci plazmatu dokaže program PDRS získat dvojím způsobem - může si ji sám spočítat ze zadaných parametrů, nebo ji načíst ze souborů.

Je-li pro nějakou (řekněme i-tou) složku plazmatu[2] definována hodnota parametru *distr. func. file for comp i* v definičním souboru *control_data.csv* (viz výše), načte se distribuční funkce i-té komponenty ze zadaného souboru. Požadovaný formát souborů s distribuční funkcí je dán velikostí N_{grid} diskretizační mřížky. Musí obsahovat N_{grid} řádků a každý z nich N_{grid} hodnot (reálných čísel) oddělených středníkem.

V opačném případě, tj. pokud není zmíněný parametr definován, program disperzní relaci dané složky plazmatu spočítá[3] pomocí vzorce 2.1. Parametry vystupující v této rovnici lze zadávat v definičním souboru *plasma_data.csv*.

[1] Hodnota parametru *DC mag. field [nT]* se využije pro výpočet elektronové cyklotronové frekvence pomocí vzorce 1.11, není-li zadána přímo jiným parametrem.

Podobně parametr *Num. density [1/cm3]* se využije pro výpočet elektronové plazmové frekvence pomocí vzorce 1.10, není-li zadána přímo a naopak: ze zadané plazmové frekvence se vypočte hustota.

Plazmové a cyklotronové frekvence všech složek plazmatu se pak, při znalosti jejich relativních hustot, nabojů a hmotností, vypočtou z elektronové plazmové a cyklotronové frekvence.

[2] Předpokládám zde, že se pro tuto komponentu vyžadují výpočty v přiblížení horkého plazmatu. Pro studené plazma je tato kapitola bezpředmětná.

[3] Stejně jako program WHAMP.

8.2 Výstupy programu PDRS

8.2.1 Data

Vypočtená data program vypisuje zároveň na standartní výstup a do souboru[4] s názvem *output.csv*. Každé množině hodnot řídících proměnných odpovídá jeden řádek, jehož obsah lze definovat parametrem *output* v definičním souboru *control_data.csv* (viz výše).

Formát výstupu na standartní výstup je *proměnná 1=hodnota proměnné 1;proměnná 2=hodnota proměnné 2;*$\cdots$. Formát výstupu do souboru je od druhého řádku *hodnota proměnné 1;hodnota proměnné 2;*$\cdots$, první řádek obsahuje názvy proměnných (rovněž oddělené středníkem).

Seznam možných hodnot parametrů *output* je v tabulce 8.3. [5]

8.2.2 Návratová hodnota

Jedním z možných výstupů programu PDRS je tzv. návratová hodnota, která vypovídá o (ne)úspěšnosti hledání řešení disperzní relace. Její význam je následující: absolutní hodnota udává počet počet uskutečněných iteračních cyklů, kladná hodnota znamená, že metoda v rámci zadané přesnosti nalezla řešení, a záporná, že řešení nalezeno nebylo, případně že byl dosažen maximální počet iteračních cyklů.

8.2.3 Distribuční funkce

Program PDRS ukládá celkovou ditribuční funkci $f_0(p_\perp, p_\parallel)$, a to do souboru[6] s názvem *distr_f.csv*. Formát dat je stejný jako je vyžadován pro načítání distribuční funkce (viz výše).[7]

8.3 Kompilace programu PDRS

Zdrojový kód programu PDRS se skládá z devíti souborů: pdrs.cpp io.h io.cpp calc.h calc.cpp diel.h diel.cpp bessel.h a bessel.cpp. Je napsán v programovacím jazyce c++. Odladěn je pod operačním systémem linux za použití překladače g++ verze 4.1. Přeložit jej lze např. příkazem "*g++ -Wno-deprecated -fno-stack-protector -g -o pdrs pdrs.cpp io.cpp calc.cpp diel.cpp bessel.cpp -I.*".

Výpis zdrojového kódu je v příloze č. 2.

[4] v pracovním adresáři

[5] Normalizované proměnné p, z a x jsou definovane vzorcem 2.2.

[6] v pracovním adresáři

[7] Tato funkcionalita je implementována zejména pro usnadnění tvorby vstupních distribučních funkcí. Lze tedy například tímto způsobem vygenerovat soubor s distribuční funkcí s Maxwellovským rozdělením s určitými parametry, ten posléze editovat a vzniklý soubor s pozměněnou distribuční funkcí použít jako vstup pro další běh programu.

Tabulka 8.1: Parametry pro definici modelu plazmatu v programu PDRS.

parametr	popis parametru	hodnota	popis hodnoty	výchozí hodnota
Num. density [1/cm3]	číselná hustota v cm^{-3}	kladné reálné číslo		0 (nedefinováno)
DC mag. field [nT]	velikost magnetické indukce v nT	reálné číslo		0
Electron gyro freq. [Hz]	elektronová cyklotronová frekvence v Hz	kladné reálné číslo		0 (nedefinováno)
Electron gyro freq. [rad/s]	elektronová cyklotronová frekvence v rad/s	kladné reálné číslo		0 (nedefinováno)
Electron plasma freq. [rad/s]	elektronová plazmová frekvence v rad/s	kladné reálné číslo		0 (nedefinováno)
Electron plasma freq. [Hz]	elektronová plazmová frekvence v Hz	kladné reálné číslo		0 (nedefinováno)
No. of plasma components	počet komponent plazmatu	přirozené číslo ≤ 8		0 (nedefinováno)
Comp i type of dielectric tensor	způsob výpočtu dielektrického tenzoru i-té složky	0 1	studené plazma horké plazma	0
Comp i num. density %	relativní číselná hustota i-té složky	reálné číslo v rozmezí 0 - 100		0
Comp i signed charge / charge of el.	náboj částic i-té složky v jednotkách náboje elektronu	reálné číslo		0

Comp i mass / mass of electron	hmotnost částic i-té složky v jednotkách hmotnosti elektronu	reálné číslo		0
Comp i temperature [eV]	teplota i-té složky v eV	kladné reálné číslo		0 (nedefinováno)
Comp i norm. parallel drift	normalizovaná driftová rychlost ve směru vnějšího magnetického pole	reálné číslo		0
Comp i loss cone depth	hloubka ztrátového kužele	reálné číslo v intervalu $< 0, 1 >$		1
Comp i temperature anisotropy	teplotní anisotropie	reálné číslo		1
Comp i loss cone anisotropy	anisotropie ztrátového kužele	reálné číslo		1

Tabulka 8.2: Parametry pro řízení běhu programu PDRS.

parametr	popis parametru	hodnota	popis hodnoty	výchozí hodnota
grid size	velikost mřížky N_{grid}	přirozené číslo ≥ 7 a ≤ 239		127
accuracy	nepřesnost při výpočtu disperzní relace	kladné reálné číslo		10^{-5}
difference accuracy	relativní velikost kroku při výpočtu disperzní relace	kladné reálné číslo		10^{-2}
max iter. loops	maximalní počet iterací při výpočtu disperzní relace	přirozené číslo		30
max bessel order	nejvyšší řád, který se ještě nezanedbává při výpočtu sumy Besselových funkcí	přirozené číslo		100
bessel sum accuracy	hodnota, pod kterou musí klesnout absolutní velikost člena sumy Besselových funkcí, aby se vyšší členy zanedbaly	kladné reálné číslo		10^{-6}
distr. func. file for comp i	název souboru, ze kterého se má načíst distribuční funkce i-té komponenty	název souboru		0 (nedefinováno)

root searching method	metoda výpočtu disperzní relace	bisection newton global newton	bisekce Newtonova metoda globálně konvergentní Newtonova metoda	global newton
derivation method	metoda výpočtu derivací	2 point 5 point	symetrická 2-bodová symetrická 5-ti bodová	5 point
integration method	metoda výpočtu integrací	simpson open	Simpsonovo složené pravidlo otevřená metoda	simpson
interpolation polynom order	řád polynomu při interpolaci derivací	kladné celé číslo		1 (lineární interpolace)
variable i	i-tá proměnná (i=1,2)	k theta k_par k_per frequency frequency dumping	velikost vlnového vektoru θ $k_{\parallel}$ $k_{\perp}$ reálná část frekvence imaginarní část frekvence	
init value i	počáteční hodnota i-té proměnné	reálné číslo		0
final value i	konečná hodnota i-té proměnné	reálné číslo		-1
step i	krok i-té proměnné	reálné číslo		0

follow mode	sledování módu	yes cokoliv kromě "yes"	sledovat, tzn. jako počáteční odhad při hledání kořene disperzní funkce bude použita hodnota extrapolovaná z výsledků předchozích kroků nesledovat - jako počáteční odhad bude použita zadaná hodnota	yes
polar coordinates	používat pro vlnový vektor polarní souřadnice	yes cokoliv kromě "yes"	používá se k a θ používá se $k_{\parallel}$ a $k_{\perp}$	používá se $k_{\parallel}$ a $k_{\perp}$
log scale	používat logaritmickou škálu	yes cokoliv kromě "yes"	proměnné se krokem násobí k proměnným se krok přičítá	k proměnným se krok přičítá
init guess i	počáteční odhad i-té neznámé	reálné číslo		0

Tabulka 8.3: Seznam možných hodnot parametrů *output*.

hodnota	popis
k	$k[m^{-3}]$
size of wave vector	$k[m^{-3}]$
norm k	$\frac{kv}{\Omega}$
normalized k	$\frac{kv}{\Omega}$
theta	$\theta[rad]$
k_par	$k_{\parallel}[m^{-3}]$
parallel k	$k_{\parallel}[m^{-3}]$
norm_k_par	z
normalized parallel k	z
k_per	$k_{\perp}[m^{-3}]$
perpendicular k	$k_{\perp}[m^{-3}]$
norm_k_per	p
normalized perpendicular k	p
omega	$Re(\omega)[\frac{rad}{s}]$
w	$Re(\omega)[\frac{rad}{s}]$
frequency	$Re(f)[Hz]$
imaginary omega	$Im(\omega)[\frac{rad}{s}]$
imaginary w	$Im(\omega)[\frac{rad}{s}]$
imaginary part of frequency	$Im(f)[Hz]$
normalized omega	$Re(x)$
norm_w	$Re(x)$
normalized frequency	$Re(x)$
N	N
result	návratová hodnota

Závěr

V této práci byla představena nová numerická procedura pro výpočet disperzní relace vln v horkém plazmatu. Při jejím vývoji se povedlo vyřešit problémy předeslané v kapitole o cílech práce, tj. nutnost výpočtu nekonečné sumy Besselových funkcí a divergenci integrandu.

Bohužel tato řešení s sebou přináší náklady ve formě zvýšené výpočetní náročnosti. To má veliký vliv na komfort při používání tohoto programu a může vést, zvláště v případech, že je vyžadováno řešení disperzní relace plazmatu skladajíciho se z více složek s obecnou distribuční funkcí, až k praktické nepoužitelnosti.

Nicméně ve většině případů lze získat pomocí této metody výsledky relevantní pro studium vln v kosmickém plazmatu, což bylo i ukázáno na reálných příkladech.

To je podpořeno i velkou univerzalitou programu PDRS, např. možností zadat na vstupu distribuční funkci témeř libovolného tvaru (jediné požadavky jsou na okrajové podmínky) a také množstvím různých řídících parametrů. Uživatel tak může běh programu do značné míry optimalizovat.

Nejzřejmějším parametrem ovlivňujícím přesnost výsledků je velikost diskretizační mřížky. Na ní nepřímo úměrně závisí velikost integračního kroku, integrace se tak s velikostí mřížky zpřesňuje. Výpočetní náročnost ale vhledem k tomu, že je mřížka dvourozměrná, roste s její velikostí kvadraticky.

Dalším takovým příkladem je způsob ukončování výpočtu nekonečných sum. Také zde jde přesnost a výpočetní náročnost proti sobě. Nicméně situace zde velmi závisí na vyšetřovaných vlnových módech. Např. suma Besselových funkcí dobře konverguje pro $z = k_\perp v_\perp / \Omega \ll 1$, tj. pro směr síření vln přibližně paralelním se směrem magnetického pole a při podélném šíření se nekončná suma dokonce redukuje na konečnou.

Méně již přesnost a výpočetní náročnost závisí na parametrech algoritmu hledajícího kořeny disperzní funkce, protože použité metody většinou konvergují dobře. Problém zde ale může nastat jinde, metoda nemusí konvergovat právě k hledanému kořenu, ale k úplně jinému řešení. Jde ale o principiální problém a nejde jej řešit jinak než vhodným nastavením počátečního odhadu hledaného kořenu (například pomocí teoretické disperzní relace, známe-li ji).

Přesto, že se ukázala cesta přímého výpočtu dielektrického tenzoru a disperzní relace schůdná, jde o metodu výpočetně velmi náročnou a stále zde tedy zůstává prostor pro různé aproximace a optimalizace. A určitě lze nalézt možné optimalizace a vylepšení i přímo pro metodu PDRS.

Seznam použité literatury

Chen F. F.: *Úvod do fyziky plazmatu*, Academia, Praha, 1984.

Černý M.: *Analýza disperze a stability vln v kosmickém plazmatu*, Bakalářská práce, MFF UK, Praha, 2006.

Glauert S. A., Horne R. B.: *Calculation of pitch angle and energy diffusion coefficients with the PADIE code*, Journal of Geophysical Research **110** (2005) A04206

Grimald S., Santolík O.: *Possible wave modes of wideband nonthermal continuum radiation in its source region*, Journal of Geophysical Research **115** (2010) A06209.

Gurnett D. A., Bhattacharjee A.: *Introduction to Plasma Physics: With Space and Laboratory Applications*, Cambridge University Press, Cambridge, 2005.

Horne R. B.: *Path-integrated growth of electrostatic waves - The generation of terrestrial myriametric radiation*, Journal of Geophysical Research **94** (1989) 8895-8909.

Horne R. B., Thorne R. M.: *Relativistic electron acceleration and precipitation during resonant interactions with whistler-mode chorus*, Geophysical Research Letters **30(10)** (2003) 1527.

Press W. H., Teukolsky S. A., Vetterling W. T., Flannery B. P.: *Numerical recipes in C: the art of scientific computing - 2nd edition*, Cambridge University Press, New York, 2002.

Qin H., Phillips C. K., Davidson R. C.: *A new derivation of the plasma susceptibility tensor for a hot magnetized plasma without infinite sums of products of Bessel functions*, Physics of Plasmas **14** (2007) 092103.

Rönnmark K.: *Waves in Homogeneous, anisotropic, multi component plasmas*, Rep. 179, Kiruna Geophys. Inst., Kiruna, Sweden, 1982.

Rönnmark K.: *Computation of the dielectric tensor of a Maxwellian plasma*, Plasma Phys., **25** (1983) 699–701.

Santolík O.: *Etude de la fonction de distribution des ondes dans un plasma chaud*, PhD thesis, Universite d'Orleans, France, 1995.

Santolík O., a kol.: *Wave-particle interactions in the equatorial source region of whistler-mode emissions*, Journal of Geophysical Research **115** (2010) A00F16.

Stix T. H.: *Waves in Plasmas*, American Institute of Physics, New York, 1992.

Swanson D.: *Plasma Waves, Second Edition*, Institute of Physics Publishing, London, 2003.

Willes A. J., Cairns I. H.: *Generalized Langmuir waves in magnetized kinetic plasmas*, Physics of Plasmas **7** (2000) 3167-3180.

Seznam tabulek

- 5.1 Hodnoty řídících parametrů použitých při výpočtu disperzní relace Langmuirových vln.
- 6.1 Parametry distribučních funkcí jednotlivých složek plazmatu modelujících rozdělení částic naměřeného přístrojem PEACE/HEEA mezi 0956:57 a 0957:18 UT v energetickém rozmezí 40eV-26keV.
- 6.2 Hodnoty řídících parametrů použitých při výpočtu disperzních relací v kapitole 6.
- 7.1 Parametry distribučních funkcí jednotlivých složek plazmatu modelujících rozdělení částic naměřeného přístrojem PEACE během čtyř sekund kolem 0122:54.
- 7.2 Hodnoty řídících parametrů použitých při výpočtu disperzních relací v kapitole 7.
- 8.1 Parametry pro definici modelu plazmatu v programu PDRS.
- 8.2 Parametry pro řízení běhu programu PDRS.
- 8.3 Seznam možných hodnot parametrů *output.*

Seznam použitých zkratek

- IDL (*Interactive Data Language*) Software pro analýzu a vizualizaci dat.
- PADIE (*Pitch Angle and energy Diffusion of Ions and Electrons*) Program pro sledování paprsku v horkém bezsrážkovém zmagnetizovaném plazmatu, představený Sarah Glauert s Richardem Hornem v práci [Glauert, Horne, 2005].
- PDRS (*Plasma Dispersion Relation Solver*) Program pro lineární analýzu vlastností vln v horkém plazmatu popsaný v této práci.
- PEACE (*Plasma Electron And Current Experiment*) Přístroj na palubě sondy Cluster měřící rozdělení, směr a tok elektronů.
- TMR (*Terrestrial Myriametric Radiation*) Záření o vlnové délce $\sim 10km$ vznikající v oblasti Země.
- WDF (*Wave Distribution Function*) Soubor programů pro řešení disperzní relace horkého plazmatu, představený Santolíkem v práci [Santolik, 1995].
- WHAMP (*Waves in Homogeneous Anisotropic Multi component Plasmas*) Program pro numerické výpočty disperzních relací vln v homogenním anisotropním plazmatu v magnetickém poli, sestavený v roce 1982 Rönnmarkem [Rönnmark, 1982].
- WHISPER (*Waves of High frequency and Sounder for Probing of Electron density by Relaxation*) Přístroj na palubě sondy Cluster měřící hustotu horkého plazmatu.
- WKB (*Wentzel, Kramer a Brillouin*) Metoda pro hledání přibližných řešení parciálních diferenciálních rovnic s prostorově závislými koeficienty.

Příloha 1. Vstupní soubory s definící modelů plazmatu

Vstupní soubor s definící modelu plazmatu použitého v kapitole 6

```
# Plasma definition
# Cluster
Num. density [1/cm3̂]:96.94
Electron gyro freq. [Hz]:7870
No. of plasma components:4
# core population of electrons:
Comp 1 type of dielectric tensor:1
Comp 1 num. density %:48.73
Comp 1 signed charge / charge of el.:-1.0
Comp 1 mass / mass of electron:1.0
Comp 1 temperature [eV]:10
Comp 1 norm. parallel drift:0
Comp 1 loss cone depth (delta):1
Comp 1 temperature anisotropy:0.08
Comp 1 loss cone anisotropy:0.0
# warm population of electrons:
Comp 2 type of dielectric tensor:1
Comp 2 num. density %:0.85
Comp 2 signed charge / charge of el.:-1.0
Comp 2 mass / mass of electron:1.0
Comp 2 temperature [eV]:1180
Comp 2 norm. parallel drift:0
Comp 2 loss cone depth (delta):1
Comp 2 temperature anisotropy:0.98
Comp 2 loss cone anisotropy:0.0
# hot electrons:
Comp 3 type of dielectric tensor:1
Comp 3 num. density %:0.42
Comp 3 signed charge / charge of el.:-1.0
Comp 3 mass / mass of electron:1.0
Comp 3 temperature [eV]:1890
Comp 3 norm. parallel drift:0
Comp 3 loss cone depth (delta):0
Comp 3 temperature anisotropy:1.16
Comp 3 loss cone anisotropy:0.232
```

ions:
Comp 4 type of dielectric tensor:0
Comp 4 num. density %:50.0
Comp 4 signed charge / charge of el.:1.0
Comp 4 mass / mass of electron:1837
Comp 4 temperature [eV]:0.001
Comp 4 norm. parallel drift:0
Comp 4 loss cone depth (delta):1
Comp 4 temperature anisotropy:1
Comp 4 loss cone anisotropy:0

Vstupní soubor s definící modelu plazmatu použitého v kapitole 7

Plasma definition
Chorus type emission
Num. density [1/cm3̂]:67.08
Electron gyro freq. [Hz]:9200
No. of plasma components:5
cold electrons:
Comp 1 type of dielectric tensor:1
Comp 1 num. density %:46.58
Comp 1 signed charge / charge of el.:-1.0
Comp 1 mass / mass of electron:1.0
Comp 1 temperature [eV]:3.5
Comp 1 norm. parallel drift:0
Comp 1 loss cone depth (delta):1
Comp 1 temperature anisotropy:1.0
Comp 1 loss cone anisotropy:0.0
core population of electrons:
Comp 2 type of dielectric tensor:1
Comp 2 num. density %:2.92
Comp 2 signed charge / charge of el.:-1.0
Comp 2 mass / mass of electron:1.0
Comp 2 temperature [eV]:15.6
Comp 2 norm. parallel drift:0
Comp 2 loss cone depth (delta):1
Comp 2 temperature anisotropy:2.1
Comp 2 loss cone anisotropy:0.0

injected electrons:
Comp 3 type of dielectric tensor:1
Comp 3 num. density %:0.46
Comp 3 signed charge / charge of el.:-1.0
Comp 3 mass / mass of electron:1.0
Comp 3 temperature [eV]:35.3
Comp 3 norm. parallel drift:0
Comp 3 loss cone depth (delta):1
Comp 3 temperature anisotropy:13.5
Comp 3 loss cone anisotropy:0.0
trapped electrons:
Comp 4 type of dielectric tensor:1
Comp 4 num. density %:0.04
Comp 4 signed charge / charge of el.:-1.0
Comp 4 mass / mass of electron:1.0
Comp 4 temperature [eV]:7000
Comp 4 norm. parallel drift:0
Comp 4 loss cone depth (delta):1
Comp 4 temperature anisotropy:4
Comp 4 loss cone anisotropy:0.0
ions:
Comp 5 type of dielectric tensor:0
Comp 5 num. density %:50.0
Comp 5 signed charge / charge of el.:1.0
Comp 5 mass / mass of electron:1837
Comp 5 temperature [eV]:1
Comp 5 norm. parallel drift:0
Comp 5 loss cone depth (delta):1
Comp 5 temperature anisotropy:1
Comp 5 loss cone anisotropy:0

Příloha 2. Výpis zdrojového kódu programu PDRS

pdrs.cpp

```
// g++ -o pdrs pdrs.cc io.cc calc.cc diel.cpp bessel.cpp -I.
#include <diel.h>

void FillGuess(cmplx& Omega, r_vector& K,
               double& k, double& theta,
               double cth, double sth,
               ControlDataStruct& CD,
               bool extrapolation,
               cmplx& last_Omega, r_vector& last_K)
{
  if (extrapolation) {
    if (CD.UnknownIsOmega) {
      cmplx dW = Omega - last_Omega;
      last_Omega = Omega;
      Omega += dW;
    } else {
      double dk;
      for (int i = 0; i < 3; i++) {
        dk =  K[i] - last_K[i];
        last_K[i] = K[i];
        K[i] += dk;
      }
      k = sqrt(K[0] * K[0] + K[1] * K[1]);
      theta = atan(K[1] / K[0]);
      cth = cos(theta);
      sth = sin(theta);
    }
  } else {
    if (CD.InitGuessDefined) {
      // user defined init guess
      if (CD.UnknownIsOmega)
        Omega = cmplx(CD.InitGuess[0], CD.InitGuess[1]);
      else {
        if (CD.Polar) {
          k = CD.InitGuess[0];
          theta = CD.InitGuess[1];
          cth = cos(theta);
          sth = sin(theta);
          K[0] = k * cth;
          K[1] = k * sth;
        } else {
          K[0] = CD.InitGuess[0];
          K[1] = CD.InitGuess[1];
          k = sqrt(K[0] * K[0] + K[1] * K[1]);
          theta = atan(K[1] / K[0]);
          cth = cos(theta);
          sth = sin(theta);
        }
      }
    } else {
      // for init guess let's use N = 1
      if (CD.UnknownIsOmega)
        last_Omega = Omega = cmplx(c_light * k, 0.0);
      else {
        k = real(Omega) / c_light;
        theta = 0.0;
        cth = cos(theta);
        sth = sin(theta);
        K[0] = k * cth;
        K[1] = k * sth;
        for (int i = 0; i < 3; i++) last_K[i] = K[i];
      }
    }
```

```
    last_Omega = Omega;
    for (int i = 0; i < 3; i++) last_K[i] = K[i];
  }
}

bool Finish(cmplx Omega, r_vector K,
            double k, double theta,
            ControlDataStruct& CD)
{
  const int up0 = (CD.Step[0] >= 0) ? 1 : -1;
  const int up1 = (CD.Step[1] >= 0) ? 1 : -1;
  if (CD.UnknownIsOmega) {
    if (CD.Polar) {
      if ((up0 * k) > (up0 * CD.FinalValue[0]) &&
          (up1 * theta) > (up1 * CD.FinalValue[1]))
        return true;
    } else {
      if ((up0 * K[0]) > (up0 * CD.FinalValue[0]) &&
          (up1 * K[1]) > (up1 * CD.FinalValue[1]))
        return true;
    }
  } else
    if ((up0 * real(Omega)) > (up0 * CD.FinalValue[0]) &&
        (up1 * imag(Omega)) > (up1 * CD.FinalValue[1]))
      return true;

  return false;
}

bool Test(cmplx Omega, r_vector K,
          double k, double theta)
{
  if (k < 0) {
    PrintError("Negative k.");
    return false;
  }
  if (K[1] < 0) {
    PrintError("Negative k_perpendicular.");
    return false;
  }
  if (real(Omega) < 0) {
    PrintError("Negative omega.");
    return false;
  }
  return true;
}

int main()
{
  PrintHead();
  PrintMessage("Reading program control data...");
  ControlDataStruct CD;
  if (ReadControlData(CD) != 0) return 0;
  PrintMessage("Reading plasma definition data...");
  PlasmaDefinitionStruct PD(CD);
  if (ReadPlasmaDefs(PD) != 0) return 0;
  PrintMessage("Reading plasma distribution_function...");
  if (ReadPlasmaDistribution(PD, CD.DistrFFile) != 0) return 0;
  PrintMessage("Calculating derivations of distribution functions...");
  if (CalcDerivations(PD, CD.DerivationMethod) != 0) return 0;

  PrintMessage("Calculating...");

  OutputHeadline(CD);

  r_vector K;  // wave vector (K_par, K_per, 0)
  K[2] = 0.0;
  cmplx Omega;
  double k, theta;
  double cth, sth;
  // Initial values
  if (CD.UnknownIsOmega) {
```

```
  if (CD.Polar) {
    k = CD.InitValue[0];
    theta = CD.InitValue[1];
    cth = cos(theta);
    sth = sin(theta);
    K[0] = k * cth;
    K[1] = k * sth;
  } else {
    K[0] = CD.InitValue[0];
    K[1] = CD.InitValue[1];
    k = sqrt(K[0] * K[0] + K[1] * K[1]);
    theta = atan(K[1] / K[0]);
    cth = cos(theta);
    sth = sin(theta);
  }
} else
  Omega = cmplx(CD.InitValue[0], CD.InitValue[1]);

cmplx last_Omega = Omega;
r_vector last_K;
// Initial guess
FillGuess(Omega, K, k, theta, cth, sth, CD,
          false, last_Omega, last_K);

int counter = 0;
while (!Finish(Omega, K, k, theta, CD)) {
  counter++;

  if (!Test(Omega, K, k, theta)) break;

  int res;
  switch (CD.Method) {
  case 0: {
    res = Bisection(Omega, theta, k, PD, CD);
    K[0] = k * cth;
    K[1] = k * sth;
    K[2] = 0.0;
    break;
  }
  case 1: {
    res = Newton1D(Omega, theta, k, PD, CD);
    K[0] = k * cth;
    K[1] = k * sth;
    K[2] = 0.0;
    break;
  }
  case 2 : {
    res = GlobalNewton(K, Omega, PD, CD);
    if (!CD.UnknownIsOmega) {
      k = sqrt(K[0] * K[0] + K[1] * K[1]);
      theta = atan(K[1] / K[0]);
      cth = cos(theta);
      sth = sin(theta);
    }
    break;
  }
    break;
  default : {
    PrintError("Wrong method!");
    return 0;
  }
  }

  Output(res, Omega, K, k, theta, PD, CD);

  // Step
  if (CD.UnknownIsOmega) {
    if (CD.Polar) {
      if (CD.LogScale) {
        k *= CD.Step[0];
        theta *= CD.Step[1];
      } else {
        k += CD.Step[0];
```

```
          theta += CD.Step[1];
        }
        cth = cos(theta);
        sth = sin(theta);
        K[0] = k * cth;
        K[1] = k * sth;
      } else {
        if (CD.LogScale) {
          K[0] *= CD.Step[0];
          K[1] *= CD.Step[1];
        } else {
          K[0] += CD.Step[0];
          K[1] += CD.Step[1];
        }
        k = sqrt(K[0] * K[0] + K[1] * K[1]);
        theta = atan(K[1] / K[0]);
        cth = cos(theta);
        sth = sin(theta);
      }
    } else
      if (CD.LogScale)
        Omega = cmplx(real(Omega) * CD.Step[0],
                      imag(Omega) * CD.Step[1]);
      else
        Omega = cmplx(real(Omega) + CD.Step[0],
                      imag(Omega) + CD.Step[1]);

    // Guess for another step
    if (counter <= 1) {
      last_Omega = Omega;
      for (int i = 0; i < 3; i++) last_K[i] = K[i];
    }
    FillGuess(Omega, K, k, theta, cth, sth, CD,
              CD.FollowMode, last_Omega, last_K);
  }

  return 0;
}
```

bessel.h

```
// -*- c++ -*-
// Bessel functions

// modified Bessel functions
float bessi0(float x);
float bessi1(float x);
float bessin(int n, float x);
float bessi(int n, float x);

// returns the Bessel function J0(x) for any real x.
float bessel0(float x);
// returns the Bessel function J1(x) for any real x.
float bessel1(float x);
```

bessel.cpp

```
#include <math.h>
#include <io.h>

#define ACC 40.0
#define BIGNO 1.0e10
#define BIGNI 1.0e-10


float bessi0(float x)
  // Returns the modified Bessel function I0 (x) for any real x.
```

```
{
  float ax, ans;
  double y;

  if ((ax = fabs(x)) < 3.75)
    {
      y = x / 3.75;
      y *= y;
      ans = 1.0 + y * (3.5156229 + y * (3.0899424 + y * (1.2067492
            + y * (0.2659732 + y * (0.360768e-1 + y * 0.45813e-2)))));
    }
  else
    {
      y = 3.75 / ax;
      ans = (exp(ax) / sqrt(ax)) * (0.39894228 + y * (0.1328592e-1
            + y * (0.225319e-2 + y * (-0.157565e-2 + y * (0.916281e-2
            + y * (-0.2057706e-1 + y * (0.2635537e-1 + y * (-0.1647633e-1
            + y * 0.392377e-2))))))));
    }
  return ans;
}

float bessi1(float x)
  // Returns the modified Bessel function I1 (x) for any real x.
{
  float ax, ans;
  double y;

  if ((ax = fabs(x)) < 3.75)
    {
      y = x / 3.75;
      y *= y;
      ans = ax * (0.5 + y * (0.87890594 + y * (0.51498869 + y * (0.15084934
            + y * (0.2658733e-1 + y * (0.301532e-2 + y * 0.32411e-3))))));
    }
  else
    {
      y = 3.75 / ax;
      ans = 0.2282967e-1 + y * (-0.2895312e-1 + y * (0.1787654e-1 - y * 0.420059e-2));
      ans = 0.39894228 + y * (-0.3988024e-1 + y * (-0.362018e-2
            + y * (0.163801e-2 + y * (-0.1031555e-1 + y * ans))));
      ans *= (exp(ax) / sqrt(ax));
    }
  return x < 0.0 ? -ans : ans;
}

float bessin(int n, float x)
  // Returns the modified Bessel function In (x) for any real x and n >= 2.
{
  int j;
  float bi, bim, bip, tox, ans;

  if (n < 2) PrintError("Index n less than 2 in bessi");
  if (x == 0.0)
    return 0.0;
  else {
    tox = 2.0/fabs(x);
    bip = ans = 0.0;
    bi = 1.0;

    for (j = 2 * (n + int(sqrt(ACC * n))); j > 0; j--)
      {
        bim = bip + j * tox * bi;
        bip = bi;
        bi = bim;

        if (fabs(bi) > BIGNO)
          {
            ans *= BIGNI;
            bi *= BIGNI;
            bip *= BIGNI;
          }
        if (j == n) ans = bip;
```

```
      }

    ans *= bessi0(x) / bi;
    return x < 0.0 && (n & 1) ? -ans : ans;
  }
}

float bessi(int n, float x)
  // Returns the modified Bessel function In (x) for any real x and n >= 0.
{
  if (n == 0) return bessi0(x);
  if (n == 1) return bessi1(x);
  return bessin(n, x);
}

float bessel0(float x)
// returns the Bessel function J0(x) for any real x.
{
    float ax, z;
    double xx, y, ans, ans1, ans2;
    if ((ax = fabs(x)) < 8.0) {
        y = x * x;
        ans1=57568490574.0+y*(-13362590354.0+y*(651619640.7
        +y*(-11214424.18+y*(77392.33017+y*(-184.9052456)))));
        ans2=57568490411.0+y*(1029532985.0+y*(9494680.718
        +y*(59272.64853+y*(267.8532712+y*1.0))));
        ans=ans1/ans2;
    } else {
        z=8.0/ax;
        y=z*z;
        xx=ax-0.785398164;
        ans1=1.0+y*(-0.1098628627e-2+y*(0.2734510407e-4
        +y*(-0.2073370639e-5+y*0.2093887211e-6)));
        ans2 = -0.1562499995e-1+y*(0.1430488765e-3
        +y*(-0.6911147651e-5+y*(0.7621095161e-6
        -y*0.934945152e-7)));
        ans=sqrt(0.636619772/ax)*(cos(xx)*ans1-z*sin(xx)*ans2);
    }
    return ans;
}

float bessel1(float x)
// returns the Bessel function J1(x) for any real x.
{
    float ax,z;
    double xx,y,ans,ans1,ans2;
    if ((ax=fabs(x)) < 8.0) {
        y=x*x;
        ans1=x*(72362614232.0+y*(-7895059235.0+y*(242396853.1
        +y*(-2972611.439+y*(15704.48260+y*(-30.16036606))))));
        ans2=144725228442.0+y*(2300535178.0+y*(18583304.74
        +y*(99447.43394+y*(376.9991397+y*1.0))));
        ans=ans1/ans2;
    } else {
        z=8.0/ax;
        y=z*z;
        xx=ax-2.356194491;
        ans1=1.0+y*(0.183105e-2+y*(-0.3516396496e-4
        +y*(0.2457520174e-5+y*(-0.240337019e-6))));
        ans2=0.04687499995+y*(-0.2002690873e-3
        +y*(0.8449199096e-5+y*(-0.88228987e-6
        +y*0.105787412e-6)));
        ans=sqrt(0.636619772/ax)*(cos(xx)*ans1-z*sin(xx)*ans2);
        if (x < 0.0) ans = -ans;
    }
    return ans;
}
```

calc.h

```
// -*- c++ -*-
/* some mathematical declarations */
#include <math.h>
#include <complex.h>
#include <stdio.h>

typedef complex<double> cmplx;
typedef double r_vector[3];
typedef cmplx i_vector[3];
typedef double matrix2x2[2][2];
typedef double matrix[3][3];

struct tensor
{
  cmplx data[3][3];

  tensor();  // sets to zero
  void set(int i, int j, cmplx x);  // set value x to i,j position
  void clear();  // sets to zero
  void set_to_one();  // sets to unit matrix
};

// scalar product
double scalar(const r_vector& a, const r_vector& b);
// tensor product
tensor tensor_product(const r_vector& a, const r_vector& b);
// other tensor operators
tensor operator + (const tensor& a, const tensor& b);
tensor operator - (const tensor& a, const tensor& b);
void operator += (tensor& a, const tensor& b);
tensor operator * (const double& a, const tensor& t);
tensor operator * (const tensor& t, const cmplx& c);
void operator *= (tensor& t, const cmplx& c);
void operator /= (tensor& t, const cmplx& c);

// prints tensor to standart output
void PrintTensor(const tensor& t);
// prints tensor to file
void PrintTensor(FILE* f, const tensor& t);
// determinant of tensor
cmplx Determinant(const tensor& t);

// converts vector of real number to complex
cmplx r2c(const double* r);

// methods for solving systems of linear equations
// needed by global newton root searching method
int LU_decomposition(double** A,const int N, int* idx, float* d);
void LU_backsubstitution(double** A,const int N, const int* idx, double b[]);

double max(double a, double b);  // maximum of a and b
double min(double a, double b);  // minimum of a and b

// interpolation by polynom of an order N
double polynom_interpolation(double* x, double* y,
                             const int N, const double t,
                             double& error);
```

calc.cpp

```
#include <io.h>

tensor::tensor()
{
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      data[i][j] = cmplx(0.0, 0.0);
}
```

```
void tensor::set(int i, int j, cmplx x)
{
  data[i][j] = x;
}

void tensor::clear()
{
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      data[i][j] = cmplx(0.0, 0.0);
}

void tensor::set_to_one()
{
  clear();
  for (int i = 0; i < 3; i++)
    data[i][i] = cmplx(1.0, 0.0);
}

double scalar(const r_vector& a, const r_vector& b)
{
  return a[0] * b[0] + a[1] * b[1] + a[2] * b[2];
}

tensor tensor_product(const r_vector& a, const r_vector& b)
{
  tensor result;
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      result.set(i, j, a[i] * b[j]);
  return result;
}

tensor operator * (const double& a, const tensor& t)
{
  tensor result;
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      result.set(i, j, a * t.data[i][j]);
  return result;
}

tensor operator + (const tensor& a, const tensor& b)
{
  tensor result;
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      result.set(i, j, a.data[i][j] + b.data[i][j]);
  return result;
}

tensor operator - (const tensor& a, const tensor& b)
{
  tensor result;
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      result.set(i, j, a.data[i][j] - b.data[i][j]);
  return result;
}

void operator += (tensor& a, const tensor& b)
{
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      a.set(i, j, a.data[i][j] + b.data[i][j]);
}

tensor operator * (const tensor& t, const cmplx& c)
{
  tensor result;
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      result.set(i, j, t.data[i][j] * c);
```

```
  return result;
}

void operator *= (tensor& t, const cmplx& c)
{
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      t.set(i, j, t.data[i][j] * c);
}

void operator /= (tensor& t, const cmplx& c)
{
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      t.set(i, j, t.data[i][j] / c);
}

// prints tensor t to standart output
void PrintTensor(const tensor& t)
{
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      printf("(%d,%d): %.16e + %.16e i\n",
             i + 1, j + 1,
             real(t.data[i][j]), imag(t.data[i][j]));
}

// prints tensor t to file f
void PrintTensor(FILE* f, const tensor& t)
{
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      fprintf(f, "(%d,%d): %.16e + %.16e i\n",
              i + 1, j + 1,
              real(t.data[i][j]), imag(t.data[i][j]));
}

// determinant of tensor
cmplx Determinant(const tensor& t)
{
  return t.data[0][0] * t.data[1][1] * t.data[2][2]
    - t.data[0][2] * t.data[1][1] * t.data[2][0]
    + t.data[0][1] * t.data[1][2] * t.data[2][0]
    - t.data[0][0] * t.data[1][2] * t.data[2][1]
    + t.data[0][2] * t.data[1][0] * t.data[2][1]
    - t.data[0][1] * t.data[1][0] * t.data[2][2];
}

// converts vector of real number to complex
cmplx r2c(const double* r)
{
  return cmplx(r[0], r[1]);
}

int LU_decomposition(double** A,const int N, int* idx, float* d)
{
  double vv[N];
  *d = 1.0;
  for (int i = 0; i < N; i++) {
    double big = 0.0;
    double temp = 0.0;
    for (int j = 0; j < N; j++)
      if ((temp = fabs(A[i][j])) > big)
        big = temp;
    if (big == 0.0) {
      PrintError("Singular matrix in LU decomposition.");
      return -1;
    }
    vv[i] = 1.0/big;
  }
  int imax;
  double sum = 0.0;
  for (int j = 0; j < N; j++) {
```

```
    for (int i = 0; i < j; i++) {
      sum = A[i][j];
      for (int k = 0; k < i; k++) sum -= A[i][k] * A[k][j];
      A[i][j] = sum;
    }
    double big = 0.0;
    for (int i = j; i < N; i++) {
      sum = A[i][j];
      for (int k = 0; k < j; k++) sum -= A[i][k] * A[k][j];
      A[i][j] = sum;
      double dum = 0.0;
      if ((dum = vv[i] * fabs(sum)) >= big) {
        big = dum;
        imax = i;
      }
    }
    if (j != imax) {
      for (int k = 0; k < N; k++) {
        double dum = A[imax][k];
        A[imax][k] = A[j][k];
        A[j][k] = dum;
      }
      *d = -(*d);
      vv[imax] = vv[j];
    }
    idx[j] = imax;
    if (A[j][j] == 0.0)  {
      PrintWarning("Division by zero in LU decomposition.");
      double small = 1.0;
      for (int k = 0; k < N; k++)
        for (int l = 0; l < N; l++)
          if (small > fabs(A[k][l]) && A[k][l] != 0.0)
            small = fabs(A[k][l]);
      A[j][j] = small * 1.0e-20;
    }
    if (j != N) {
      double dum = 1.0/A[j][j];
      for (int i = j + 1; i < N; i++) A[i][j] *= dum;
    }
  }
  return 0;
}

void LU_backsubstitution(double** A, const int N,
                         const int* idx, double B[])
{
  int ii = -1;
  for (int i = 0; i < N; i++) {
    int ip = idx[i];
    double sum = B[ip];
    B[ip] = B[i];
    if (ii >= 0)
      for (int j = ii; j < i; j++) sum -= A[i][j] * B[j];
    else if (sum) ii = i;
    B[i] = sum;
  }
  for (int i = N - 1; i >= 0; i--) {
    double sum = B[i];
    for (int j = i + 1; j < N; j++) sum -= A[i][j] * B[j];
    B[i] = sum / A[i][i];
  }
}

double max(double a, double b)
{
  if (a >= b) return a;
  return b;
}

double min(double a, double b)
{
  if (a <= b) return a;
  return b;
```

```
}

double polynom_interpolation(double* x, double* y,
                             const int N, const double t,
                             double& error)
{
  double dif = fabs(t - x[0]);
  double c[N], d[N];
  double dift;

  int ns = 0;
  for (int i = 0; i < N; i++) {
    if ((dift = fabs(t - x[i])) < dif) {
      ns = i;
      dif = dift;
    }
    c[i] = y[i];
    d[i] = y[i];
  }
  double res = y[ns--];
  for (int m = 1; m < N; m++) {
    for (int i = 1; i <= N - m; i++) {
      double ho = x[i-1] - t;
      double hp = x[i+m-1] - t;
      double w = c[i] - d[i-1];
      double den = ho - hp;
      den = w / den;
      d[i-1] = hp * den;
      c[i-1] = ho * den;
    }
    res += (error = (2*(ns+1) < (N - m) ? c[ns+1] : d[ns--]));
  }
  return res;
}
```

diel.h

```
// -*- c++ -*-
/* functions needed for calculation of diel. tensor and DR */
#include <bessel.h>
#include <io.h>

// precalculates derivations of distribution functions
int CalcDerivations(PlasmaDefinitionStruct& pdd,
                    int derivation_method);

// aid structure for integrand calculating
struct integral_data
{
  cmplx ome;  // frequency
  double Omega;  // relativistic cyklotron frequency
  r_vector k;  // wave vector (K_par, K_per, 0)
  double z;  // argument of bessel's functions
  double p_par, p_per,  // parallel and perpendicular momentum
    v_par, v_per,  // parallel and perpendicular velocity
    deriv1,  // derivation of distr. function by p_parallel
    deriv2;  // derivation of distr. function by p_perpendicular
  cmplx U;
};

// calculates coeficient of n-th order of bessel functions sum
cmplx Sum_coef(int n, int type, integral_data& cd);
// calculates sum of bessel functions
tensor Sum(integral_data& cd, ControlDataStruct& CDS);
// calculates integrand (in eq. for susceptibility tensor)
tensor Integrand(double p_par, double p_per,
                 double deriv1, double deriv2,
                 cmplx ome, r_vector k, double gyro_w,
                 double mass, double beta_coef,
                 ControlDataStruct& CDS);
```

```
// adds contribution of cold plasma component to dielectric tensor
int AddColdToDie(cmplx ome, double GyroW,
                 double PlasmaW, tensor& die, double A);

// adds contribution of hot plasma component to dielectric tensor
int AddHotToDie(cmplx ome, r_vector k, int component,
                PlasmaDefinitionStruct& pdd,
                ControlDataStruct& CD,
                tensor& die, double A);

// composes dielectric tensor
int ComposeDie(cmplx ome, r_vector k,
               PlasmaDefinitionStruct& pdd,
               ControlDataStruct& CD,
               tensor& die);

// calculates dispersion function
cmplx getDF(cmplx Omega, r_vector& K, const double norm,
            PlasmaDefinitionStruct& PD,
            ControlDataStruct& CD);

// searching roots of dispersion function by bisection method
int Bisection(cmplx& Omega, double theta, double& k,
              PlasmaDefinitionStruct& PD,
              ControlDataStruct& CD);

// searching roots of dispersion function by Newton method
int Newton1D(cmplx& Omega, double theta, double& _k,
             PlasmaDefinitionStruct& PD,
             ControlDataStruct& CD);

// searching roots of dispersion function by globally convergent Newton method
int GlobalNewton(r_vector& k, cmplx& omega,
                 PlasmaDefinitionStruct& PD,
                 ControlDataStruct& CD);
```

diel.cpp

```
#include <diel.h>

int CalcDerivations(PlasmaDefinitionStruct& pdd, int method)
  // precalculates derivations of distribution functions
{
  for (int i = 0; i < pdd.NComp; i++)
    if (pdd.CompKind[i] == 1) {  // just for hot plasma
      // initialization
      pdd.CompDistFuncDeriv1[i] = new double*[pdd.PDGridSize];
      pdd.CompDistFuncDeriv2[i] = new double*[pdd.PDGridSize];
      for (int j = 0; j < pdd.PDGridSize; j++) {
        pdd.CompDistFuncDeriv1[i][j] = new double[pdd.PDGridSize];
        pdd.CompDistFuncDeriv2[i][j] = new double[pdd.PDGridSize];
      }

      // grid interval length
      double h = pdd.CompDistLength[i] / (pdd.PDGridSize - 1);

      switch (method) {  // type of derivative method
      case 0:  // symetric 2 point method
        {
          for (int j = 0; j < pdd.PDGridSize; j++)
            for (int k = 0; k < pdd.PDGridSize; k++) {
              // d / d p_par
              if (k > 0 && k < pdd.PDGridSize - 1) {
                double diff = pdd.CompDistFunc[i][j][k + 1] - pdd.CompDistFunc[i][j][k - 1];
                pdd.CompDistFuncDeriv1[i][j][k] = diff / (2 * h);
              } else
                pdd.CompDistFuncDeriv1[i][j][k] = 0.0;

              // d / d p_pen
              if (j > 0 && j < pdd.PDGridSize - 1) {
```

```
                double diff = pdd.CompDistFunc[i][j + 1][k] - pdd.CompDistFunc[i][j - 1][k];
                pdd.CompDistFuncDeriv2[i][j][k] = diff / (2 * h);
              } else
                pdd.CompDistFuncDeriv2[i][j][k] = 0.0;
            }
          break;
        }
      case 1:  // symetric 5 point method
        {
          for (int j = 0; j < pdd.PDGridSize; j++)
            for (int k = 0; k < pdd.PDGridSize; k++) {
              if (k > 1 && k < pdd.PDGridSize - 2) {
                double diff = pdd.CompDistFunc[i][j][k - 2] - 8 * pdd.CompDistFunc[i][j][k - 1]
                  + 8 * pdd.CompDistFunc[i][j][k + 1] - pdd.CompDistFunc[i][j][k + 2];
                pdd.CompDistFuncDeriv1[i][j][k] = diff / (12 * h);
              } else
                pdd.CompDistFuncDeriv1[i][j][k] = 0.0;

              if (j > 1 && j < pdd.PDGridSize - 2) {
                double diff = pdd.CompDistFunc[i][j - 2][k] - 8 * pdd.CompDistFunc[i][j - 1][k]
                  + 8 * pdd.CompDistFunc[i][j + 1][k] - pdd.CompDistFunc[i][j + 2][k];
                pdd.CompDistFuncDeriv2[i][j][k] = diff / (12 * h);
              } else
                pdd.CompDistFuncDeriv2[i][j][k] = 0.0;
            }
          break;
        }
      }
    }
  return 0;
}

cmplx Sum_coef(int n, int type, integral_data& cd)
{
  const cmplx i(0.0, 1.0);
  cmplx res = 0.0;

  switch (type) {
  case 0:  // 1,1
    {
      res = double(n * n) * cd.p_per * cd.U / (cd.z * cd.z);
      break;
    }
  case 1:  // 1,2
    {
      res = i * double(n) * cd.p_per * cd.U / cd.z;
      break;
    }
  case 2:  // 1,3
    {
      cmplx W = (cmplx(1.0) - n * cd.Omega / cd.ome) * cd.deriv1
        + n * cd.Omega * cd.p_par * cd.deriv2 / (cd.ome * cd.p_per);
      res = double(n) * cd.p_per * W / cd.z;
      break;
    }
  case 3:  // 2,1
    {
      res = -1.0 * i * double(n) * cd.p_per * cd.U / cd.z;
      break;
    }
  case 4:  // 2,2
    {
      res = cd.p_per * cd.U;
      break;
    }
  case 5:  // 2,3
    {
      cmplx W = (cmplx(1.0) - n * cd.Omega / cd.ome) * cd.deriv1
        + n * cd.Omega * cd.p_par * cd.deriv2 / (cd.ome * cd.p_per);
      res = -i * cd.p_per * W;
      break;
    }
  case 6:  // 3,1
```

```
    {
      res = double(n) * cd.p_par * cd.U / cd.z;
      break;
    }
  case 7:  // 3,2
    {
      res = i * cd.p_par * cd.U;
      break;
    }
  case 8:  // 3,3
    {
      cmplx W = (cmplx(1.0) - n * cd.Omega / cd.ome) * cd.deriv1
        + n * cd.Omega * cd.p_par * cd.deriv2 / (cd.ome * cd.p_per);
      res = cd.p_par * W;
      break;
    }
  }

  cmplx denominator = cd.ome - cd.k[0] * cd.v_par - n * cd.Omega;
  if (abs(denominator / cd.ome) < 1.0e-10) {
    if (abs(res) < 1.0e-10) return cmplx(0.0);
    else PrintWarning("infinite sum coefficient");
  }

  res *= cd.Omega / denominator;

  return res;
}

tensor Sum(integral_data& cd, ControlDataStruct& CDS)
{
  tensor result;

  // the highest order of Bessel functions counted to sum
  const int N = CDS.MaxBesselOrder;
  // the highest order of Bessel functions in recurrence (must be even)
  const int MaxN = 10 * N;
  const double Accuracy = CDS.BesselAccuracy;
  const double MaxNumber = 1.0e100;
  const double MinNumber = 1.0e-100;

  int sgn_n, sgn_x;
  double j0, j1, j2, dj;

  if (cd.z < 0.0) PrintWarning("z is not positive");
  double ax = cd.z;
  double coef = 2.0 / ax;

  for (int type = 0; type < 9; type++) {
    cmplx value(0.0, 0.0);

    switch (type) {
    case 0:  // 1,1
    case 2:  // 1,3
    case 6:  // 3,1
    case 8:  // 3,3
      {
        if (ax < MinNumber) {
          // We cannot use function Sum_coef() because of division by zero z.
          if (type == 0) {  // 1,1
            // only non-zero member has n = 1 and n = -1
            cmplx denominator1 = cd.ome - cd.k[0] * cd.v_par - cd.Omega;
            cmplx denominator2 = cd.ome - cd.k[0] * cd.v_par + cd.Omega;
            cmplx coef = cd.Omega * (1.0 / denominator1 + 1.0 / denominator2);
            value = coef * cd.p_per * cd.U * 0.25;
          } else if (type == 8) {  // 3,3
            // only non-zero member has n = 0
            cmplx W = cmplx(1.0) * cd.deriv1;
            value = cd.p_par * W;
            cmplx denominator = cd.ome - cd.k[0] * cd.v_par;
            if (abs(denominator / cd.ome) < 1.0e-10) {
              if (abs(value) < 1.0e-10) {
                value = cmplx(0.0);
```

```
              break;
            }
            else PrintWarning("infinite sum coefficient");
          }
          value *= cd.Omega / denominator;
}
        // else value remains zero
        break;
      }

      cmplx sum(0.0, 0.0);
      if (ax > (double)N) {  // upwards recurrence
        j0 = bessel0(ax);
        j1 = bessel1(ax);
        sum += Sum_coef(0, type, cd) * j0 * j0;
        sum += (Sum_coef(-1, type, cd) + Sum_coef(1, type, cd)) * j1 * j1;
        for (int l = 1; l < N; l++) {
          j2 = l * coef * j1 - j0;
          cmplx summand = (Sum_coef(l + 1, type, cd) + Sum_coef(-l - 1, type, cd)) * j2 * j2;
          sum += summand;
          if (real(summand) < Accuracy * real(sum) &&
              imag(summand) < Accuracy * imag(sum))
            break;
          j0 = j1;
          j1 = j2;
        }
      } else { // downwards recurrence
        bool jsum = false;
        double sum_tem = 0.0;
        j0 = 0.0;
        j1 = 1.0;
        for (int l = MaxN; l > 0; l--) {
          j2 = l * coef * j1 - j0;
          if (fabs(j2) > MaxNumber) {
            j1 *= MinNumber;
            j2 *= MinNumber;
            sum_tem *= MinNumber;
            sum *= MinNumber * MinNumber;
          }
          if (l <= N) {
            sum += Sum_coef(l - 1, type, cd) * j2 * j2;
            if (l > 1) sum += Sum_coef(-l + 1, type, cd) * j2 * j2;
          }
          if (jsum) sum_tem += j2;
          jsum = !jsum;

          j0 = j1;
          j1 = j2;
        }
        sum_tem = 2.0 * sum_tem - j2;
        sum /= sum_tem * sum_tem;
      }
      value = sum;
      break;
    }
  case 4:  // 2,2
    {
      if (ax < MinNumber) {
        value = (Sum_coef(1, type, cd) + Sum_coef(-1, type, cd)) * 0.25;
        break;
      }

      cmplx sum(0.0, 0.0);
      if (ax > (double)N) {  // upwards recurrence
        j0 = bessel0(ax);
        j1 = bessel1(ax);
        double dj = -j1;
        sum += Sum_coef(0, type, cd) * dj * dj;
        dj = (ax * j0 - j1) / ax;
        // dj-1 = - dj1
        sum += (Sum_coef(-1, type, cd) + Sum_coef(1, type, cd)) * dj * dj;
        for (int l = 1; l < N; l++) {
          j2 = l * coef * j1 - j0;
```

```
        dj = (ax * j1 - (l + 1) * j2) / ax;
        cmplx summand = (Sum_coef(l + 1, type, cd) + Sum_coef(-l - 1, type, cd)) * dj * dj;
        sum += summand;
        if (real(summand) < Accuracy * real(sum) &&
            imag(summand) < Accuracy * imag(sum))
          break;
        j0 = j1;
        j1 = j2;
      }
    } else {  // downwards recurrence
      bool jsum = false;
      double sum_tem = 0.0;
      j0 = 0.0;
      j1 = 1.0;
      for (int l = MaxN; l > 0; l--) {
        j2 = l * coef * j1 - j0;
        if (fabs(j2) > MaxNumber) {
          j1 *= MinNumber;
          j2 *= MinNumber;
          sum_tem *= MinNumber;
          sum *= MinNumber * MinNumber;
        }
        if (l <= N) {
          dj = (l - 1) * j2 / ax - j1;  // dj is of order l - 1
          sum += Sum_coef(l - 1, type, cd) * dj * dj;
          if (l > 1) sum += Sum_coef(-l + 1, type, cd) * dj * dj;
        }
        if (jsum) sum_tem += j2;
        jsum = !jsum;

        j0 = j1;
        j1 = j2;
      }
      sum_tem = 2.0 * sum_tem - j2;
      sum /= sum_tem * sum_tem;
    }
    value = sum;
    break;
  }
case 1:  // 1,2
case 3:  // 2,1
case 5:  // 2,3
case 7:  // 3,2
  {
    if (ax < MinNumber) {
      if (type == 1 || type == 3) {
        cmplx denominator1 = cd.ome - cd.k[0] * cd.v_par - cd.Omega;
        cmplx denominator2 = cd.ome - cd.k[0] * cd.v_par + cd.Omega;
        value = cmplx(0, 1) * cd.Omega * (cd.p_per * cd.U / denominator1 -
                                          cd.p_per * cd.U / denominator2) * 0.25;
        if (type == 3) value *= -1;
      }
      // else value remains zero
      break;
    }

    cmplx sum(0.0, 0.0);
    if (ax > (double)N) {  // upwards recurrence
      j0 = bessel0(ax);
      j1 = bessel1(ax);
      double dj = -j1;
      sum += Sum_coef(0, type, cd) * dj * j2;
      dj = (ax * j0 - j1) / ax;
      // dj-1 = - dj1
      sum += (Sum_coef(-1, type, cd) + Sum_coef(1, type, cd)) * dj * j1;
      for (int l = 1; l < N; l++) {
        j2 = l * coef * j1 - j0;
        dj = (ax * j1 - (l + 1) * j2) / ax;
        cmplx summand = (Sum_coef(l + 1, type, cd) + Sum_coef(-l - 1, type, cd)) * dj * j2;
        sum += summand;
        if (real(summand) < Accuracy * real(sum) &&
            imag(summand) < Accuracy * imag(sum))
          break;
```

```
            j0 = j1;
            j1 = j2;
          }
        } else {  // downwards recurrence
          bool jsum = false;
          double sum_tem = 0.0;
          j0 = 0.0;
          j1 = 1.0;
          for (int l = MaxN; l > 0; l--) {
            j2 = l * coef * j1 - j0;
            if (fabs(j2) > MaxNumber) {
              j1 *= MinNumber;
              j2 *= MinNumber;
              sum_tem *= MinNumber;
              sum *= MinNumber * MinNumber;
            }
            if (l <= N) {
              dj = ((l - 1) * j2 - ax * j1) / ax;  // dj is of order l - 1
              sum += Sum_coef(l - 1, type, cd) * dj * j2;
              if (l > 1) sum += Sum_coef(-l + 1, type, cd) * dj * j2;
            }
            if (jsum) sum_tem += j2;
            jsum = !jsum;

            j0 = j1;
            j1 = j2;
          }
          sum_tem = 2.0 * sum_tem - j2;
          sum /= sum_tem * sum_tem;
        }
        value = sum;
        break;
      }
    }

    result.set(type / 3, type % 3, value);
  }

  return result;
}

tensor Integrand(double p_par, double p_per,
                 double deriv1, double deriv2,
                 cmplx ome, r_vector k, double gyro_w,
                 double mass, double beta_coef,
                 ControlDataStruct& CDS)
{
  if (p_per == 0.0) {
    // integrand is zero when p_per = 0
    return tensor();  // zero
  }

  integral_data cd;
  cd.ome = ome;
  for (int i = 0; i < 3; i++) cd.k[i] = k[i];

  cd.p_par = p_par;
  cd.p_per = p_per;

  double beta = p_par * p_par + p_per * p_per;
  beta *= beta_coef;
  double gamma = 1.0 / sqrt(1 + beta);

  cd.v_par = cd.p_par / (gamma * mass);
  cd.v_per = cd.p_per / (gamma * mass);
  cd.Omega = gyro_w * gamma;

  cd.z = cd.k[1] * cd.v_per / cd.Omega;
  if (cd.z < 0.0) PrintWarning("negative z");

  cd.deriv1 = deriv1;
  cd.deriv2 = deriv2;
```

```
  cd.U = cd.deriv2 + (cd.k[0] / cd.ome) *
    (cd.v_per * cd.deriv1 - cd.v_par * cd.deriv2);

  return Sum(cd, CDS);
}

int AddHotToDie(cmplx ome, r_vector k, int component,
                PlasmaDefinitionStruct& PDS,
                ControlDataStruct& CDS,
                tensor& die, double A)
  // adds contribution of hot plasma component to dielectric tensor
{
  const double& len = PDS.CompDistLength[component];
  const double& p_norm = PDS.CompPNorm[component];
  const double& gyro_w = PDS.GyroW[component];
  const double& plasma_w = PDS.PlasmaW[component];
  const double& mass = PDS.CompPars[component][2];

  double k_pi[3];
  for (int i = 0; i < 3; i++) k_pi[i] = k[i] / (2 * M_PI);

  double beta_coef = 1.0 / (mass * mass * sqr_c_l);

  int grid_size = CDS.GridSize;  // number of grid points
  if (grid_size % 2 == 0) grid_size--;  // should be odd
  double h = len / (grid_size - 1);  // grid interval length

  const tensor zero;
  tensor integrand2[grid_size];
  tensor I;

  if (CDS.IntegrationMethod == 0) {
    // integration by compound Simpson rule
    tensor integrand[grid_size][grid_size];
    for (int i = 0; i < grid_size; i++) {
      for (int j = 0; j < grid_size; j++) {
        double p_par = h * (j - 0.5 * grid_size);  // p_parallel
        double p_per = h * i;  // p_perpendicular

        double d1 = PDS.Derivation1(component, i, j);
        double d2 = PDS.Derivation2(component, i, j);

        integrand[i][j] = Integrand(p_par, p_per, d1, d2, ome, k_pi,
                                    gyro_w, mass, beta_coef, CDS);
      }
    }

    for (int i = 0; i < grid_size; i++) {
      tensor I_odd;
      for (int j = 1; j < grid_size - 1; j += 2)
        I_odd += integrand[i][j];
      I_odd *= 2;
      tensor I_even;
      for (int j = 2; j < grid_size - 2; j += 2)
        I_even += integrand[i][j];
      tensor I = (I_even + I_odd) * 2.0;
      I += integrand[i][0] + integrand[i][grid_size - 1];
      I  /= 3;
      integrand2[i] = I;
    }

  } else if (CDS.IntegrationMethod == 1) {
    for (int i = 0; i < grid_size; i++) {
      integrand2[i] = zero;
      if (i == 0) continue;  // p_per = 0 and integral is zero
      double p_per = h * i;  // p_perpendicular

      // integration over p_parallel (-oo ... +oo)
      // parallel p's
      double p_min = - 0.5 * grid_size * h;
      double p_max = 0.5 * grid_size * h;

      double p = 0.5 * (p_min + p_max);
```

```
      double beta = p * p + p_per * p_per;
      beta *= beta_coef;
      double gamma = 1.0 / sqrt(1 + beta);
      double v = p_min / (gamma * mass);
      int n_max = int((real(ome) - k_pi[0] * v) / (gyro_w * gamma));
      v = p_max / (gamma * mass);
      int n_min = int((real(ome) - k_pi[0] * v) / (gyro_w * gamma)) + 1;

      int number_of_intervals = n_max - n_min + 1;
      if (number_of_intervals < 1 ||
          n_min > CDS.MaxBesselOrder ||
          n_max < -CDS.MaxBesselOrder)
        number_of_intervals = 1;
      if (number_of_intervals > grid_size) {
PrintWarning("Too many integrations intervals, setting to one.");
        number_of_intervals = 1;
      }

      tensor I_old;
      I = zero;
      I_old = zero;
      int size = grid_size / number_of_intervals;
      if (size < 8) size = 8;

      do {
        if (size > 2 * grid_size) {
          if (real(Determinant(I)) == 0 && real(Determinant(I_old)) == 0) break;
  PrintWarning("Inaccurate integral over p_parallel!");
          // What should we return? What about average?
          I = 0.5 * (I + I_old);
          break;
        }

        I_old = I;
        I = zero;

        for (int n = 0; n < number_of_intervals; n++) {
          // inproper integral for every interval
          double a, b;
          if (n == 0) a = p_min;
          else {
            a = (real(ome) - (n_max - n) * (gyro_w * gamma));
            a /= k_pi[0];  // v
            a *= gamma * mass;
          }
          if (n == number_of_intervals - 1)
            b = p_max;
          else {
            b = (real(ome) - (n_max - n - 1) * (gyro_w * gamma));
            b /= k_pi[0];  // v
            b *= gamma * mass;
          }

          double h2 = (b - a) / size;

          for (int j = 0; j < size; j++) {
            if (j == 0 || j == 2 ||
                j == size - 3 || j == size - 1)
              continue;
            double integration_coef;
            if (j == 1 || j == size - 2)
              integration_coef = 2.25;  // 27/12
            else if (j == 3 || j == size - 4)
              integration_coef = 13.0 / 12.0;
            else if (j % 2 == 0)
              integration_coef = 4.0 / 3.0;
            else
              integration_coef = 2.0 / 3.0;

            double p_par = a + h2 * j;  // p_parallel
            double d1 = PDS.Derivation1(component, i, p_par);
            double d2 = PDS.Derivation2(component, i, p_par);
            I += (h2 / h) * integration_coef *
```

```
      Integrand(p_par, p_per, d1, d2, ome, k_pi,
gyro_w, mass, beta_coef, CDS);
          }
        }
        size *= 2;
      }
      while (real(Determinant(I_old)) == 0 ||
             fabs(1 - real(Determinant(I) / Determinant(I_old))) > 0.01);

      integrand2[i] += I;  // for every interval
    }
  } else {
    PrintError("Unknown integration method!");
    return -1;
  }

  for (int i = 0; i < grid_size; i++)
    integrand2[i] *= i;  // ... *= p_per

  // integration over p_perpendicular (0 ... +oo)
  tensor I_odd;
  for (int i = 1; i < grid_size - 1; i += 2)
    I_odd += integrand2[i];
  I_odd *= 2;
  tensor I_even;
  for (int i = 2; i < grid_size - 2; i += 2)
    I_even += integrand2[i];
  I = (I_even + I_odd) * 2.0;
  I += integrand2[0] + integrand2[grid_size - 1];
  I /= 3;

  I *= h * h * h;
  I *= p_norm * p_norm * p_norm;

  cmplx w_coef = cmplx(plasma_w) / (gyro_w * ome);  // plasma freq. is already squared
  w_coef *= 2 * M_PI;  // 2pi * p_per * dp_per ...
  I *= w_coef;
  die += I * A;

  return 0;
}

int AddColdToDie(cmplx ome, double GyroW,
                 double PlasmaW, tensor& die, double A)
  // adds contribution of cold plasma component to dielectric tensor
{
  double sqr_PlW = PlasmaW;  // already squared

  cmplx R = -1 * sqr_PlW / (ome * (ome + GyroW));
  cmplx L = -1 * sqr_PlW / (ome * (ome - GyroW));
  cmplx S = 0.5 * (R + L);
  cmplx D = 0.5 * (R - L);
  cmplx P = -1 * sqr_PlW / (ome * ome);

  tensor eps;
  eps.set(0, 0, S);
  eps.set(1, 1, S);
  eps.set(0, 1, -1.0 * cmplx(0, 1) * D);
  eps.set(1, 0, cmplx(0, 1) * D);
  eps.set(2, 2, P);

  die += eps * A;

  return 0;
}

int ComposeDie(cmplx ome, r_vector k,
               PlasmaDefinitionStruct& pdd,
               ControlDataStruct& cd,
               tensor& die)
  // composes dielectric tensor
{
  int result = 0;
```

```
  // initialisation
  die.set_to_one();

  // composition
  for (int i = 0; i < pdd.NComp; i++) {
    switch (pdd.CompKind[i])
      {
      case 0:  // cold plasma
        {
          if (AddColdToDie(ome, pdd.GyroW[i], pdd.PlasmaW[i],
                           die, 1) != 0)
            {
              char str[100];
              sprintf(str, "Cannot calculate diel. tensor for component %d.", i + 1);
              PrintError(str);
              result = -1;
            }
          break;
        }
      case 1:  // hot plasma
        {
          if (AddHotToDie(ome, k, i, pdd, cd, die, 1) != 0)
            {
              char str[100];
              sprintf(str, "Cannot calculate diel. tensor for component %d.", i + 1);
              PrintError(str);
              result = -1;
            }
          break;
        }
      default:
        {
          char str[100];
          sprintf(str, "Unknown plasma type %d, component %d.",
                  pdd.CompKind[i], i + 1);
          PrintError(str);
          result = -1;
          break;
        }
      }
  }

  return result;
}

cmplx getDF(cmplx Omega, r_vector& K, const double norm,
            PlasmaDefinitionStruct& PD,
            ControlDataStruct& CD)
{
  r_vector k;
  cmplx w;
  for (int i = 0; i < 3; i++) k[i] = K[i];
  if (CD.UnknownIsOmega) {
    w = norm * Omega;
  } else {
    w = Omega;
    for (int i = 0; i < 3; i++) k[i] *= norm;
  }
  double sqr_k = scalar(k, k);
  // we need to exchange k1 and k3,
  // because here k paralel is in z-direction
  r_vector k_exchanged;
  k_exchanged[0] = k[2];
  k_exchanged[1] = k[1];
  k_exchanged[2] = k[0];
  tensor kk = tensor_product(k_exchanged, k_exchanged);
  tensor tensor_one; tensor_one.set_to_one();

  double omega = real(w);  // real part of frequency
  // squared index of refraction
  double sqr_N = sqr_c_l * sqr_k / (omega * omega);
```

```
  tensor t = sqr_N * ((1.0 / sqr_k) * kk + (-1.0 * tensor_one));
  tensor Die;  // dielectric tensor
  Die.clear();
  if (ComposeDie(w, k, PD, CD, Die) != 0) PrintError("");
  t += Die;
  return Determinant(t);
}

int Bisection(cmplx& Omega, double theta, double& k,
              PlasmaDefinitionStruct& PD,
              ControlDataStruct& CD)
{
  const double norm = 1.0;
  int counter = 0;
  const double cth = cos(theta);
  const double sth = sin(theta);
  r_vector K;  // wave vector (K_par, K_per, 0)
  K[0] = k * cth;
  K[1] = k * sth;
  K[2] = 0.0;
  // imaginary part of frequency
  // is always supposed to be zero in this method.
  Omega = cmplx(real(Omega), 0);

  cmplx F1 = getDF(Omega, K, norm, PD, CD);
  cmplx F2 = 0;

  if (CD.UnknownIsOmega) {
    double w1 = real(Omega);
    double w2 = w1;
    do {
      w2 *= 10;
      F2 = getDF(cmplx(w2, 0), K, norm, PD, CD);
      if (w2 > w1 * 1.0e10) return -1;
    } while (real(F1) * real(F2) > 0);

    do {
      counter++;
      double w = 0.5 * (w1 + w2);
      F2 = getDF(cmplx(w, 0), K, norm, PD, CD);
      if (real(F1) * real(F2) < 0)
        w2 = w;
      else
        w1 = w;

      if (fabs(w1 - w2) < fabs(w1) * CD.Accuracy) {
        Omega = cmplx(w2, 0);
        return counter;
      }

      if (counter > CD.MaxIterLoops) return -1;
    } while (fabs(real(F2)) > CD.Accuracy);
    Omega = cmplx(w2, 0);
  } else {
    double k1 = k;  // init guess
    double k2 = k;
    double dk = k;
    do {
      k1 += dk;
      K[0] = k1 * cth;
      K[1] = k1 * sth;
      K[2] = 0.0;
      F1 = getDF(Omega, K, norm, PD, CD);
      k2 -= dk / 100;
      K[0] = k2 * cth;
      K[1] = k2 * sth;
      K[2] = 0.0;
      F2 = getDF(Omega, K, norm, PD, CD);

      if (k2 <= 0.0) return -1;  // failed
    } while (real(F1) * real(F2) > 0);

    do {
```

```
      counter++;
      k = 0.5 * (k1 + k2);
      K[0] = k * cth;
      K[1] = k * sth;
      K[2] = 0.0;
      F2 = getDF(Omega, K, norm, PD, CD);

      if (real(F1) * real(F2) < 0)
        k2 = k;
      else
        k1 = k;

      if (fabs(k1 - k2) < fabs(k1) * CD.Accuracy) return counter;

      if (counter > CD.MaxIterLoops) return -1;
    } while (fabs(real(F2)) > CD.Accuracy);
  }

  return counter;
}

int Newton1D(cmplx& Omega, double theta, double& k,
             PlasmaDefinitionStruct& PD,
             ControlDataStruct& CD)
{
  const double norm = 1.0;
  int counter = 0;
  const double cth = cos(theta);
  const double sth = sin(theta);
  r_vector K;  // wave vector (K_par, K_per, 0)
  // imaginary part of frequency
  // is always supposed to be zero in this method.
  Omega = cmplx(real(Omega), 0);

  if (CD.UnknownIsOmega) {
    K[0] = k * cth;
    K[1] = k * sth;
    K[2] = 0.0;

    double w = real(Omega);
    double dw = w * CD.DifferenceAccuracy;
    while (true) {
      counter++;
      cmplx F = getDF(cmplx(w, 0), K, norm, PD, CD);
      cmplx F2 = getDF(cmplx(w + dw, 0), K, norm, PD, CD);

      if (fabs(real(F)) < CD.Accuracy) {
        // has convergated
        Omega = cmplx(w, 0);
        return counter;
      }

      double dF = real(F2 - F) / dw;
      double w_old = w;
      w -= real(F) / dF;

      if (counter > CD.MaxIterLoops) return -counter;  // convergation failed
    }
  } else {
    double dk = k * CD.DifferenceAccuracy;
    while (true) {
      counter++;
      K[0] = k * cth;
      K[1] = k * sth;
      K[2] = 0.0;
      cmplx F = getDF(Omega, K, norm, PD, CD);

      K[0] = (k + dk) * cth;
      K[1] = (k + dk) * sth;
      K[2] = 0.0;
      cmplx F2 = getDF(Omega, K, norm, PD, CD);

      if (fabs(real(F)) < CD.Accuracy)
```

```
        // has convergated
        return counter;

      double dF = real(F2 - F) / dk;
      k -= real(F) / dF;

      if (counter > CD.MaxIterLoops) return -counter;  // convergation failed
    }
  }
}

double CalcF(cmplx omega, double F[], r_vector& k,
             const double normalization,
             PlasmaDefinitionStruct& PD,
             ControlDataStruct& CD)
{
  cmplx tem = getDF(omega, k, normalization, PD, CD);
  F[0] = real(tem);
  F[1] = imag(tem);
  return 0.5 * norm(tem);
}

void CalcJacobian(cmplx omega, double F[], double** J,
                  r_vector& K, const double norm,
                  PlasmaDefinitionStruct& PD,
                  ControlDataStruct& CD)
{
  cmplx w = omega;
  r_vector k;
  k[0] = K[0];
  k[1] = K[1];
  k[2] = 0.0;
  const int N = 2;
  double x[N];
  if (CD.UnknownIsOmega) {
    x[0] = real(omega) / norm;
    x[1] = imag(omega) / norm;
  } else {
    x[0] = K[0] / norm;
    x[1] = K[1] / norm;
  }
  const double precision = CD.DifferenceAccuracy;
  double f[N];
  for (int j = 0; j < N; j++) {
    double temp = x[j];
    double h = precision * fabs(temp);
    if (h == 0.0) h = precision;
    x[j] = temp + h;
    h = x[j] - temp;
    if (CD.UnknownIsOmega)
      w = r2c(x) * norm;
    else {
      k[0] = x[0] * norm;
      k[1] = x[1] * norm;
    }
    cmplx ftemp = getDF(w, k, 1.0, PD, CD);
    f[0] = real(ftemp); f[1] = imag(ftemp);
    x[j] = temp;
    for (int i = 0; i < N; i++)
      J[i][j] = (f[i] - F[i]) / h;
  }
  for (int i = 0; i < N; i++)
    if ((fabs(J[i][0]) == 0.0) &&
        (fabs(J[i][1]) == 0.0))
      for (int j = 0; j < N; j++)
        J[i][j] = precision;
}

int lnsearch(const int N, double x_old[], double f_old, double g[],
             double p[], double x[], double* f, double F[],
             const double max_step,
             cmplx omega, r_vector& k, const double norm,
             PlasmaDefinitionStruct& PD,
```

```
            ControlDataStruct& CD)
{
  const double ALF = CD.Accuracy * 10;
  double sum = 0.0;
  for (int i = 0; i < N; i++) sum += p[i] * p[i];
  sum = sqrt(sum);
  if (sum > max_step)
    for (int i = 0; i < N; i++) p[i] *= max_step / sum;
  double slope = 0.0;
  for (int i = 0; i < N; i++) slope += g[i] * p[i];
  if (slope == 0.0) {
    PrintError("Roundoff problem in lnsearch.");
    return -1;
  }
  double test = 0.0;
  for (int i = 0; i < N; i++) {
    double temp = fabs(p[i]) / max(fabs(x_old[i]), 1.0);
    if (temp > test) test = temp;
  }
  double alamin = 10.e-7 * CD.Accuracy / test;
  double alam = 1.0;
  double alam2 = 0.0;
  double f2 = 0.0;
  while (true) {
    for (int i = 0; i < N; i++)
      x[i] = x_old[i] + alam * p[i];
    if (CD.UnknownIsOmega) {
      omega = r2c(x);
    } else {
      k[0] = x[0];
      k[1] = x[1];
    }
    *f = CalcF(omega, F, k, norm, PD, CD);
    double tmplam = 0.0;
    if (alam < alamin) {
      for (int i = 0; i < N; i++) x[i] = x_old[i];
      return 1;
    } else if (*f <= f_old + ALF * alam * slope) return 0;
    else {
      if (alam == 1.0)
        tmplam = -slope / (2.0 * (*f - f_old - slope));
      else {
        double rhs1 = *f - f_old - alam * slope;
        double rhs2 = f2 - f_old - alam2 * slope;
        double a = (rhs1 / (alam * alam) - rhs2 / (alam2 * alam2)) / (alam - alam2);
        double b = (-alam2 * rhs1 / (alam * alam) + alam * rhs2 / (alam2 * alam2)) /
          (alam - alam2);
        if (a == 0.0) tmplam = -slope / (2.0 * b);
        else {
          double disc = b * b - 3.0 * a * slope;
          if (disc < 0.0) tmplam = 0.5 * alam;
          else if (b <= 0.0) tmplam = (-b + sqrt(disc)) / (3.0 * a);
          else tmplam = -slope / (b + sqrt(disc));
        }
        if (tmplam > 0.5 * alam)
          tmplam = 0.5 * alam;
      }
    }
    alam2 = alam;
    f2 = *f;
    alam = max(tmplam, 0.1 * alam);
  }
}

void delete_matrix(const int N, double** J)
{
  for (int i = 0; i < N; i++)
    delete[] J[i];
  delete[] J;
}

int GlobalNewton(r_vector& k, cmplx& omega,
                 PlasmaDefinitionStruct& PD,
```

```
                 ControlDataStruct& CD)
// With given k, looking for omega (or vice versa)
// satisfying dispersion relation (getDF() == 0)
// by globally convergent Newton method.
// Value in omega (or k) will be used as a initial guess
{
  double x[2];
  if (CD.UnknownIsOmega) {
    x[0] = real(omega);
    x[1] = imag(omega);
  } else {
    x[0] = k[0];
    x[1] = k[1];
  }
  // normalization x to 1 because of roundoff errors
  const double norm = max(fabs(x[0]), fabs(x[1]));
  if (norm == 0.0) {
    PrintError("Bad init guess.");
    return -1;
  }
  for (int i = 0; i < 2; i++) x[i] /= norm;

  double F[2];
  double f = CalcF(omega, F, k,
                   1.0,  // only x is normalized
                   PD, CD);

  printf("x=%.16g, y=%.16g, F1=%.16g, F2=%.16g\n",
         x[0], x[1], F[0], F[1]);
  if (fabs(F[0]) < CD.Accuracy && fabs(F[1]) < CD.Accuracy)
    return 0;  // allready have a root

  // Jacobian
  double** J = new double*[2];
  for (int i = 0; i < 2; i++)
    J[i] = new double[2];

  const double max_step = 1.0e4;

  int counter = 0;
  while (counter < CD.MaxIterLoops) {
    counter++;

    CalcJacobian(omega, F, J, k, norm, PD, CD);

    double g[2];
    for (int i = 0; i < 2; i++) {
      double sum = 0.0;
      for (int j = 0; j < 2; j++)
        sum += J[j][i] * F[j];
      g[i] = sum;
    }

    double x_old[2];
    for (int i = 0; i < 2; i++) x_old[i] = x[i];
    double f_old = f;

    double p[2];
    for (int i = 0; i < 2; i++) p[i] = -F[i];
    int idx[2];
    float d;
    if (LU_decomposition(J, 2, idx, &d) != 0) {
      delete_matrix(2, J);
      return -counter;
    }
    LU_backsubstitution(J, 2, idx, p);

    int res = lnsearch(2, x_old, f_old, g, p, x, &f, F,
                       max_step, omega, k, norm, PD, CD);


    if (res > 0) {  //  convergence in x
      double f = CalcF(omega, F, k,
```

```
                      1.0,  // only x is normalized
                      PD, CD);
      CalcJacobian(omega, F, J, k, norm, PD, CD);
    }

    if (res < 0) {
      delete_matrix(2, J);
      return -counter;
    }
    if (CD.UnknownIsOmega)
      omega = norm * r2c(x);
    else {
      k[0] = norm * x[0];
      k[1] = norm * x[1];
    }

    if (k[1] < 0.0) {
      k[1] = 0.0;
      x[1] = 0.0;
      continue;
    }

    if (fabs(F[0]) < CD.Accuracy && fabs(F[1]) < CD.Accuracy) {
      delete_matrix(2, J);
      return counter;  // has convergated
    }

    if (res > 0) {  //  convergence in x
      double test = 0.0;
      double den = max(f, 1.0);
      for (int i = 0; i < 2; i++) {
        double temp = fabs(g[i]) * max(fabs(x[i]), 1.0) / den;
        if (temp > test) test = temp;
      }
      delete_matrix(2, J);
      if (test > CD.Accuracy)  // zero gradient
        return -counter;
      else return counter;
      //      return -counter;
    }
  }
  delete_matrix(2, J);
  return -counter;  // convergation failed
}
```

io.h

```
// -*- c++ -*-
/* some input/output declaration */
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <complex.h>
#include <calc.h>

#define MaxPlasmaComp 8  // maximum number of plasma components
#define MaxPars 8  // if smaller then 8, check PlasmaDefinitionStruct constructor
#define MaxOutputColumn 20  // maximum number of columns in output

const double c_light = 299792458.0;  // speed of light
const double sqr_c_l = 8.987551787e16;  // squared speed of light
const double m_elect = 9.109382e-31;  // mass of electron
const double e_elect = 1.602177e-19;  // charge of electron
const double eps_0 = 8.854188e-12;  // permitivity of vacuum
//const double k_B = 1.380658e-23;  // Boltzmann constant in J/K
const double k_B = 8.617342e-5;  // Boltzmann constant in eV/K

// Program control data
struct ControlDataStruct
{
```

```
  // size of integration matrix (should be odd - number of intervals even)
  int GridSize;
  double Accuracy;  // (in)accuracy of DR solution
  double DifferenceAccuracy;  // relative size of step when solving DR
  int MaxIterLoops;  // maximum number of iterations when solving DR
  int MaxBesselOrder;  // the highest order of Bessel functions counted to sum
  double BesselAccuracy;  // max error of sum of Bessels

  char** DistrFFile;  // files with distribution functions
  int Method;  // method of solving DR (root searching method)
  int DerivationMethod;  // derivation method
  int IntegrationMethod;  // integration method
  int N_polynom;  // order of interpolation polynom

  bool UnknownIsOmega;  // if true, we are looking for frequency
                        // else we are looking for wave vector
  bool Polar;  // do we use polar coordinates for wave vector?
  bool LogScale;  // do we use logarithmic scale for variables?
                  // using k and theta instead k_par and k_per
  double InitValue[2];  // initial values of variables
  double FinalValue[2];  // final values of variables
  double Step[2];  // step of variables
  double InitGuess[2];  // initial guess of solution (root of DF)
  bool InitGuessDefined;  // if true, initial guess is defined by user
  bool FollowMode;  // if true, we use as init guess recently calculated value

  char* Output[MaxOutputColumn];  // what has to be in output

  ControlDataStruct()
  {
    // default values
    GridSize = 127;
    Accuracy = 1.0e-5;
    DifferenceAccuracy = 1.0e-2;
    MaxIterLoops = 30;
    MaxBesselOrder = 100;
    BesselAccuracy = 1.0e-6;
    DistrFFile = new char*[MaxPlasmaComp];
    for (int i = 0; i < MaxPlasmaComp; i++)
      DistrFFile[i] = 0;
    Method = 2;  // global Newton
    DerivationMethod = 1;  // symetric 5 point method
    IntegrationMethod = 0;
    N_polynom = 1;  // linear
    UnknownIsOmega = true;
    Polar = false;
    LogScale = false;
    for (int i = 0; i < 2; i++) {
      InitValue[i] = 0.0;
      FinalValue[i] = -1.0;
      Step[i] = 0.0;
      InitGuess[i] = 0.0;
    }
    InitGuessDefined = false;  // undefined
    FollowMode = true;
    for (int i = 0; i < MaxOutputColumn; i++)
      Output[i] = 0;
  }

  ~ControlDataStruct()
  {
    for (int i = 0; i < MaxPlasmaComp; i++)
      if (DistrFFile[i] != 0)
        delete[] DistrFFile[i];
    delete[] DistrFFile;
    for (int i = 0; i < MaxOutputColumn; i++)
      if (Output[i] != 0) delete[] Output[i];
  }

  // checking parameters and their consistency
  int Check() const;  // returns 0 if OK
};
```

```
// Plasma definition data
struct PlasmaDefinitionStruct
{
  int NComp;  // number of plasma components
  int PDGridSize;  // size of integration matrix

  double Density;  // total density [m-3]
  double ElGyroW;  // electron cyklotronic frequency
  double ElPlasmaW;  // electron plasma frequency

  // electron gyro angular frequency [rad/s] of each plasma component
  double GyroW[MaxPlasmaComp];
  // squared electron plasma angular frequency [(rad/s)^2] of each plasma component
  double PlasmaW[MaxPlasmaComp];

  // kind (cold, hot) of each plasma component
  int CompKind[MaxPlasmaComp];
  // parameters of each plasma component
  // content defined by corresp. CompKind
  double CompPars[MaxPlasmaComp][MaxPars];

  // length of definition area of dist. function
  double CompDistLength[MaxPlasmaComp];
  // normalization coeficient of momentum
  double CompPNorm[MaxPlasmaComp];

  // distribution function of each hot plasma component
  double** CompDistFunc[MaxPlasmaComp];

  // derivation of distr. function by p_parallel
  double** CompDistFuncDeriv1[MaxPlasmaComp];
  // derivation of distr. function by p_perpendicular
  double** CompDistFuncDeriv2[MaxPlasmaComp];
  // order of interpolation polynom
  int N_polynom;

  PlasmaDefinitionStruct(ControlDataStruct& CD);
  ~PlasmaDefinitionStruct();

  // checking parameters and their consistency
  int Check() const;  // returns 0 if OK

  // returns derivation of component comp's distr. functions
  // by p_parallel at point (i,j)
  double Derivation1(int comp, int i, int j);
  // returns derivation of component comp's distr. functions
  // by p_perpendicular at point (i,j)
  double Derivation2(int comp, int i, int j);
  // returns derivation of component comp's distr. functions
  // by p_parallel at p_par
  double Derivation1(int comp, int i, double p_par);
  // returns derivation of component comp's distr. functions
  // by p_perpendicular at p_par
  double Derivation2(int comp, int i, double p_par);

  // returns thermal velocity of i-th component
  double V(int i = 0) const;
  // returns velocity of i-th component from distr. function
  int DistV(int i, double* v) const;
  // returns the size of velocity of i-th component
  // from distr. function
  double DistV(int i) const;
  // returns the size of velocity
  // from total distr. function
  double DistV() const;
};

void PrintHead();  // program header
void PrintError(const char* text);  // error
void PrintWarning(const char* text);  // warning
void PrintMessage(const char* text);  // message

// separates line to values
```

```
int ReadLine(char* line, char** values);

// reads program control data from control_data.csv
int ReadControlData(ControlDataStruct& cd);

// reads plasma definition from plasma_data.csv
int ReadPlasmaDefs(PlasmaDefinitionStruct& pdd);

// reads or computes plasma distribution grid
int ReadPlasmaDistribution(PlasmaDefinitionStruct& pdd,
                           char** filenames);

// data output
// writes headline to output file
void OutputHeadline(ControlDataStruct& CD);
// writes data line to output file
void Output(int res, cmplx Omega, r_vector& K,
            double k, double theta,
            PlasmaDefinitionStruct& PD,
            ControlDataStruct& CD);
```

io.cpp

```
#include <io.h>

void PrintHead()
{
  printf("The Plasma Dispersion Relation Solver\n");
}

void PrintError(const char* text)
{
  printf("ERROR! %s\n", text);
}

void PrintWarning(const char* text)
{
  printf("WARNING! %s\n", text);
}

void PrintMessage(const char* text)
{
  printf("%s\n", text);
}

int ControlDataStruct::Check() const
{
  if (GridSize < 7) {
    PrintError("Grid size is too small.");
    return -1;
  }
  if (Accuracy <= 0.0) {
    PrintError("Accuracy is not positive.");
    return -1;
  }
  if (DifferenceAccuracy <= 0.0) {
    PrintError("Difference accuracy is not positive.");
    return -1;
  }
  if (Method < 0 || Method > 3) {
    PrintError("Undefined root searching method.");
    return -1;
  }
  if (DerivationMethod < 0 || DerivationMethod > 1) {
    PrintError("Undefined derivation method.");
    return -1;
  }
  if (IntegrationMethod < 0 || IntegrationMethod > 2) {
    PrintError("Undefined integration method.");
    return -1;
  }
```

```
  if (N_polynom < 1) {
    PrintError("Bad order of interpolation polynom.");
    return -1;
  }
  if (MaxIterLoops < 0) {
    PrintError("Negative max iter. loops.");
    return -1;
  }
  if (MaxBesselOrder < 0) {
    PrintError("Negative max bessel order.");
    return -1;
  }
  if (BesselAccuracy <= 0.0) {
    PrintError("Bessel sum accuracy is not positive.");
    return -1;
  }
  if (BesselAccuracy >= 1.0) {
    PrintError("Bessel sum accuracy is not less then 1.");
    return -1;
  }

  if (UnknownIsOmega) {
    if (InitGuess[0] <= 0) {
      PrintError("Incorrect init guess for omega.");
      return -1;
    }
  } else {
    if (Polar) {
      if (InitGuess[0] <= 0) {
        PrintError("Non-positive init guess for k.");
        return -1;
      }
      if (InitGuess[1] < 0) {
        PrintError("Negative init guess for theta.");
        return -1;
      }
    } else {
      if (InitGuess[1] < 0) {
        PrintError("Negative init guess "
                   "for k_perpendicular.");
        return -1;
      }
    }
  }

  return 0;
}

// separates line to values
// line is expected to be in format value0:value1
int ReadLine(char* line, char** values)
{
  char* eol = strchr(line, '\n'); if (eol) *eol = 0;
  eol = strchr(line, '\r'); if (eol) *eol = 0;
  values[0] = line;
  char* colon = strchr(line, ':');
  if (!colon) return -1;
  *colon = 0;  // end of name;
  values[1] = colon + 1;
  return 0;
}

// reads program control data from control_data.csv
int ReadControlData(ControlDataStruct& cd)
{
  FILE* f = fopen("control_data.csv", "r");
  if (!f) {
    PrintError("Can't read file control_data.csv!");
    return -1;
  }
  int result = 0;

  int var1 = 0;
```

```
int var2 = 0;
double init[2];
double final[2];
double step[2];

int size = 256;
char line[size];
int line_number = 0;
while (fgets(line, size, f) > 0) {
  line_number++;
  if (line[0] == '#') continue;  // comment
  char* values[2];
  if (ReadLine(line, values) < 0) {
    printf("ERROR! Can't read line %d!\n", line_number);
    result = -1;
    break;
  }
  char name[size];
  char value[size];
  strcpy(name, values[0]);
  strcpy(value, values[1]);

  int iv = 0;
  double dv = 0.0;
  if (strcmp(name, "grid size") == 0) {
    if (sscanf(value, "%d", &iv) != 1) {
      PrintError("Cannot read grid size.");
      result = -1;
      break;
    }
    cd.GridSize = iv;
    // should be odd (number of intervals is even)
    if (cd.GridSize % 2 == 0) cd.GridSize++;
  }
  else if (strcmp(name, "accuracy") == 0) {
    if (sscanf(value, "%lf", &dv) != 1) {
      PrintError("Cannot read accuracy.");
      result = -1;
      break;
    }
    cd.Accuracy = dv;
  }
  else if (strcmp(name, "difference accuracy") == 0) {
    if (sscanf(value, "%lf", &dv) != 1) {
      PrintError("Cannot read accuracy.");
      result = -1;
      break;
    }
    cd.DifferenceAccuracy = dv;
  }
  else if (strcmp(name, "max iter. loops") == 0) {
    if (sscanf(value, "%d", &iv) != 1) {
      PrintError("Cannot read max iter. loops.");
      result = -1;
      break;
    }
    cd.MaxIterLoops = iv;
  }
  else if (strcmp(name, "max bessel order") == 0) {
    if (sscanf(value, "%d", &iv) != 1) {
      PrintError("Cannot read max bessel order.");
      result = -1;
      break;
    }
    cd.MaxBesselOrder = iv;
  }
  else if (strcmp(name, "bessel sum accuracy") == 0) {
    if (sscanf(value, "%lf", &dv) != 1) {
      PrintError("Cannot read bessel sum accuracy.");
      result = -1;
      break;
    }
    cd.BesselAccuracy = dv;
```

```
}
else if (strncmp(name, "distr. func. file", 17) == 0) {
  int comp = 0;  // default 1st component
  char comp_name[28];
  for (int i = 0; i < MaxPlasmaComp; i++) {
    sprintf(comp_name, "distr. func. file for comp %d", i + 1);
    if (strcmp(name, comp_name) == 0) comp = i;  // an other comp
  }
  int size = strlen(value);
  cd.DistrFFile[comp] = new char[size + 1];
  strcpy(cd.DistrFFile[comp], value);
}
else if (strcmp(name, "root searching method") == 0) {
  if (strcmp(value, "bisection") == 0) cd.Method = 0;
  else if (strcmp(value, "newton") == 0) cd.Method = 1;
  else if (strcmp(value, "global newton") == 0) cd.Method;
  else {
    PrintError("Cannot read root searching method.");
    result = -1;
    break;
  }
}
else if (strcmp(name, "derivation method") == 0) {
  if (strcmp(value, "2 point") == 0) cd.DerivationMethod = 0;
  else if (strcmp(value, "5 point") == 0) cd.DerivationMethod = 1;
  else {
    PrintError("Cannot read derivation method.");
    result = -1;
    break;
  }
}
else if (strcmp(name, "integration method") == 0) {
  if (strcmp(value, "simpson") == 0) cd.IntegrationMethod = 0;
  else if (strcmp(value, "open") == 0) cd.IntegrationMethod = 1;
  else {
    PrintError("Cannot read integration method.");
    result = -1;
    break;
  }
}
else if (strcmp(name, "interpolation polynom order") == 0) {
  if (sscanf(value, "%d", &iv) != 1) {
    PrintError("Cannot read order of interpolation polynom.");
    result = -1;
    break;
  }
  cd.N_polynom = iv;
}
else if (strcmp(name, "variable 1") == 0) {
  if (strcmp(value, "k") == 0)
    var1 = 1;
  else if (strcmp(value, "theta") == 0)
    var1 = 2;
  else if (strcmp(value, "k_par") == 0)
    var1 = 3;
  else if (strcmp(value, "k_per") == 0)
    var1 = 4;
  else if (strcmp(value, "frequency") == 0)
    var1 = 5;
  else if (strcmp(value, "frequency dumping") == 0)
    var1 = 6;
  else {
    PrintError("Cannot read variable 1 from control file.");
    result = -1;
    break;
  }
}
else if (strcmp(name, "variable 2") == 0) {
  if (strcmp(value, "k") == 0)
    var2 = 1;
  else if (strcmp(value, "theta") == 0)
    var2 = 2;
  else if (strcmp(value, "k_par") == 0)
```

```
    var2 = 3;
  else if (strcmp(value, "k_per") == 0)
    var2 = 4;
  else if (strcmp(value, "frequency") == 0)
    var2 = 5;
  else if (strcmp(value, "frequency dumping") == 0)
    var2 = 6;
  else {
    PrintError("Cannot read variable 2 from control file.");
    result = -1;
    break;
  }
}
else if (strcmp(name, "init value 1") == 0) {
  if (sscanf(value, "%lf", &dv) != 1) {
    PrintError("Cannot read init value 1.");
    result = -1;
    break;
  }
  init[0] = dv;
}
else if (strcmp(name, "init value 2") == 0) {
  if (sscanf(value, "%lf", &dv) != 1) {
    PrintError("Cannot read init value 2.");
    result = -1;
    break;
  }
  init[1] = dv;
}
else if (strcmp(name, "final value 1") == 0) {
  if (sscanf(value, "%lf", &dv) != 1) {
    PrintError("Cannot read final value 1.");
    result = -1;
    break;
  }
  final[0] = dv;
}
else if (strcmp(name, "final value 2") == 0) {
  if (sscanf(value, "%lf", &dv) != 1) {
    PrintError("Cannot read final value 2.");
    result = -1;
    break;
  }
  final[1] = dv;
}
else if (strcmp(name, "step 1") == 0) {
  if (sscanf(value, "%lf", &dv) != 1) {
    PrintError("Cannot read step 1.");
    result = -1;
    break;
  }
  step[0] = dv;
}
else if (strcmp(name, "step 2") == 0) {
  if (sscanf(value, "%lf", &dv) != 1) {
    PrintError("Cannot read step 2.");
    result = -1;
    break;
  }
  step[1] = dv;
}
else if (strcmp(name, "follow mode") == 0) {
  if (strcmp(value, "yes") != 0)
    cd.FollowMode = false;
}
else if (strcmp(name, "polar coordinates") == 0) {
  if (strcmp(value, "yes") == 0)
    cd.Polar = true;
}
else if (strcmp(name, "log scale") == 0) {
  if (strcmp(value, "yes") == 0)
    cd.LogScale = true;
}
```

```
  else if (strcmp(name, "init guess 1") == 0) {
    if (sscanf(value, "%lf", &dv) != 1) {
      PrintError("Cannot read init guess 1.");
      result = -1;
      break;
    }
    cd.InitGuess[0] = dv;
    cd.InitGuessDefined = true;
  }
  else if (strcmp(name, "init guess 2") == 0) {
    if (sscanf(value, "%lf", &dv) != 1) {
      PrintError("Cannot read init guess 2.");
      result = -1;
      break;
    }
    cd.InitGuess[1] = dv;
    cd.InitGuessDefined = true;
  }
  else if (strcmp(name, "output") == 0) {
    int pos = 0;
    for (int i = 0; i < MaxOutputColumn; i++)
      if (cd.Output[i] == 0) {
        pos = i;
        break;
      }
    if (pos < MaxOutputColumn) {
      int size = strlen(value);
      cd.Output[pos] = new char[size + 1];
      strcpy(cd.Output[pos], value);
    }
  }
  else {
    printf("ERROR! Can't read line %d!\n", line_number);
    result = -1;
    break;
  }

}  // end of reading
fclose(f);

if (result != 0) return result;

bool inconsistent_variables = true;
bool var_change = false;
if (var1 == 1 && var2 == 2) {
  inconsistent_variables = false;
  cd.Polar = true;
}
if (var1 == 2 && var2 == 1) {
  inconsistent_variables = false;
  var_change = true;
  cd.Polar = true;
}
if (var1 == 3 && var2 == 4)
  inconsistent_variables = false;
if (var1 == 4 && var2 == 3) {
  inconsistent_variables = false;
  var_change = true;
}
if (var1 == 5 && var2 == 6) {
  inconsistent_variables = false;
  cd.UnknownIsOmega = false;
}
if (var1 == 6 && var2 == 5) {
  inconsistent_variables = false;
  var_change = true;
  cd.UnknownIsOmega = false;
}

if (inconsistent_variables) {
  PrintError("Inconsistent variables.");
  return -1;
}
```

```
  if (var_change) {
    cd.InitValue[0] = init[1];
    cd.InitValue[1] = init[0];
    cd.FinalValue[0] = final[1];
    cd.FinalValue[1] = final[0];
    cd.Step[0] = step[1];
    cd.Step[1] = step[0];
  } else {
    cd.InitValue[0] = init[0];
    cd.InitValue[1] = init[1];
    cd.FinalValue[0] = final[0];
    cd.FinalValue[1] = final[1];
    cd.Step[0] = step[0];
    cd.Step[1] = step[1];
  }
  if (var1 == 5) {  // frequency
    cd.InitValue[0] *= 2*M_PI;
    cd.FinalValue[0] *= 2*M_PI;
    if (!cd.LogScale) cd.Step[0] *= 2*M_PI;
  }
  if (var2 == 5) {  // frequency
    cd.InitValue[1] *= 2*M_PI;
    cd.FinalValue[1] *= 2*M_PI;
    if (!cd.LogScale) cd.Step[1] *= 2*M_PI;
  }

  return cd.Check();
}

PlasmaDefinitionStruct::PlasmaDefinitionStruct(ControlDataStruct& CD) :
  NComp(0), PDGridSize(CD.GridSize), Density(0),
  ElGyroW(0), ElPlasmaW(0), N_polynom(CD.N_polynom+1)
{
  for (int i = 0; i < MaxPlasmaComp; i++) {
    GyroW[i] = 0.0;
    PlasmaW[i] = 0.0;
    CompKind[i] = 0;
    CompPars[i][0] = 0.0;
    CompPars[i][1] = 1.0;
    CompPars[i][2] = 1.0;
    CompPars[i][3] = 0.0;
    CompPars[i][4] = 0.0;
    CompPars[i][5] = 1.0;
    CompPars[i][6] = 1.0;
    CompPars[i][7] = 1.0;
    for (int j = 8; j < MaxPars; j++)
      CompPars[i][j] = 0.0;
    CompDistLength[i] = 0.0;
    CompPNorm[i] = 1.0;
    CompDistFunc[i] = 0;
    CompDistFuncDeriv1[i] = 0;
    CompDistFuncDeriv2[i] = 0;
  }
}

PlasmaDefinitionStruct::~PlasmaDefinitionStruct()
{
  for (int i = 0; i < MaxPlasmaComp; i++) {
    if (CompDistFunc[i]) {
      for (int j = 0; j < PDGridSize; j++)
        delete[] CompDistFunc[i][j];
      delete[] CompDistFunc[i];
    }
    if (CompDistFuncDeriv1[i]) {
      for (int j = 0; j < PDGridSize; j++)
        delete[] CompDistFuncDeriv1[i][j];
      delete[] CompDistFuncDeriv1[i];
    }
    if (CompDistFuncDeriv2[i]) {
      for (int j = 0; j < PDGridSize; j++)
        delete[] CompDistFuncDeriv2[i][j];
      delete[] CompDistFuncDeriv2[i];
```

```
    }
  }
}

int PlasmaDefinitionStruct::Check() const
{
  if (NComp > MaxPlasmaComp) {
    PrintError("Maximum number of components exceeded.");
    return -1;
  }
  for (int i = 0; i < NComp; i++) {
    if (GyroW[i] <= 0.0) {
      // unknown gyro frequency and magnetic field
      PrintError("Undetermined gyro frequency.");
      return -1;
    }
    if (PlasmaW[i] <= 0.0) {
      // unknown electron plasma frequency and density
      PrintError("Electron plasma frequency not determined!");
      return -1;
    }
    if (CompPars[i][0] < 0.0) {
      printf("ERROR! Wrong density for component %d.", i);
      return -1;
    }
    if (CompPars[i][2] < 0.0) {
      printf("ERROR! Mass of component %d is not positive.", i);
      return -1;
    }
    if (CompPars[i][3] < 0.0) {
      printf("ERROR! Temperature of component %d is not positive.", i);
      return -1;
    }
  }

  return 0;
}

// returns derivation of component comp's distr. functions
// by p_parallel at point (i,j)
double PlasmaDefinitionStruct::Derivation1(int comp, int i, int j)
{
  if (!CompDistFuncDeriv1[comp]) {
    PrintError("Derivations of dispersion function undefined!");
    return 0;
  }
  return CompDistFuncDeriv1[comp][i][j];
}

// returns derivation of component comp's distr. functions
// by p_perpendicular at point (i,j)
double PlasmaDefinitionStruct::Derivation2(int comp, int i, int j)
{
  if (!CompDistFuncDeriv2[comp]) {
    PrintError("Derivations of dispersion function undefined!");
    return 0;
  }
  return CompDistFuncDeriv2[comp][i][j];
}

// returns derivation of component comp's distr. functions
// by p_parallel at p_par
double PlasmaDefinitionStruct::Derivation1(int comp, int i, double p_par)
{
  const double h = CompDistLength[comp] / (PDGridSize - 1);
  const double p_min = - 0.5 * PDGridSize * h;
  if (p_par < p_min) return 0.0;  // by definition

  int index = int((p_par - p_min) / h);
  const int N = N_polynom;
  double x[N], y[N];
  for (int k = 0; k < N; k++) {
    int j = index - N/2 + k;
```

```
    x[k] = p_min + j * h;
    if (j < 0 || j >= PDGridSize) y[k] = 0;  // by definition
    else y[k] = Derivation1(comp, i, j);
  }
  double err;
  return polynom_interpolation(x, y, N, p_par, err);
}

// returns derivation of component comp's distr. functions
// by p_perpendicular at p_par
double PlasmaDefinitionStruct::Derivation2(int comp, int i, double p_par)
{
  const double h = CompDistLength[comp] / (PDGridSize - 1);
  const double p_min = - 0.5 * PDGridSize * h;
  if (p_par < p_min) return 0.0;  // by definition

  int index = int((p_par - p_min) / h);
  const int N = N_polynom;
  double x[N], y[N];
  for (int k = 0; k < N; k++) {
    int j = index - N/2 + k;
    x[k] = p_min + j * h;
    if (j < 0 || j >= PDGridSize) y[k] = 0;  // by definition
    else y[k] = Derivation2(comp, i, j);
  }
  double err;
  return polynom_interpolation(x, y, N, p_par, err);
}

// returns thermal velocity of i-th component
double PlasmaDefinitionStruct::V(int i) const
{
  double kT = CompPars[i][3];
  kT *= fabs(CompPars[i][1]);  // * charge
  // kT = mv^2/2
  return sqrt(2 * kT / CompPars[i][2]);
}

// returns velocity of i-th component from distr. function
int PlasmaDefinitionStruct::DistV(int comp, double* V) const
{
  for (int i = 0; i < 3; i++) V[i] = 0.0;

  if (CompDistFunc[comp] == 0) return -1;

  int grid_size = PDGridSize;
  double h = CompDistLength[comp] / (grid_size - 1);

  // integration over p_parallel (-oo ... +oo)
  double v_par[grid_size];
  double v_per[grid_size];
  for (int i = 0; i < grid_size; i++) {
    v_par[i] = 0.0;
    v_per[i] = 0.0;
    for (int j = 0; j < grid_size; j++) {
      int x = j - grid_size / 2 - 1;
      int y = i;
      v_par[i] += CompDistFunc[comp][i][j] * x*x;
      v_per[i] += CompDistFunc[comp][i][j] * y*y;
    }
  }
  // integration over p_perpendicular (0 ... +oo)
  for (int i = 0; i < grid_size; i++) {
    V[0] += v_par[i] * i;
    V[1] += v_per[i] * i;
  }
  for (int i = 0; i < 3; i++) {
    V[i] *= h * h * h;
    // integration over theta
    V[i] *= 2 * M_PI;
    V[i] *= CompPNorm[comp] * CompPNorm[comp] * CompPNorm[comp];
    V[i] = sqrt(V[i]);
    V[i] *= h / CompPars[comp][2];
```

```
  }

  return 0;
}

// returns the size of velocity of i-th component
// from distr. function
double PlasmaDefinitionStruct::DistV(int comp) const
{
  double sqr_v = 0.0;
  double V[3];
  if (DistV(comp, V) == 0)
    for (int i = 0; i < 3; i++) sqr_v += V[i] * V[i];
  else
    PrintError("Cannot calculate velocity. "
               "Maybe distribution function is not defined.");

  return sqrt(sqr_v / 3.0);
}

// returns the size of total velocity
double PlasmaDefinitionStruct::DistV() const
{
  double V = 0.0;
  for (int i = 0; i < NComp; i++)
    if (CompKind[i] == 1) {  // just for hot plasma
      double v = DistV(i);
      V += v*v * CompPars[i][0] / Density;
    }
  return sqrt(V);
}

// reads plasma definition from plasma_data.csv
int ReadPlasmaDefs(PlasmaDefinitionStruct& pdd)
{
  FILE* f = fopen("plasma_data.csv", "r");
  if (!f) {
    PrintError("Can't read file plasma_data.csv!");
    return -1;
  }

  int size = 256;
  char line[size];
  int line_number = 0;
  while (fgets(line, size, f) > 0) {
    line_number++;
    if (line[0] == '#') continue;  // comment
    char* values[2];
    if (ReadLine(line, values) < 0) {
      printf("ERROR! Can't read line %d!\n", line_number);
      return -1;
    }
    char name[size];
    char value[size];
    strcpy(name, values[0]);
    strcpy(value, values[1]);

    bool found = true;
    double dv = 0.0;
    int iv = 0;
    int comp = 0;
    if (strcmp(name, "Num. density [1/cm^3]") == 0) {
      if (sscanf(value, "%lf", &dv) != 1) {
        printf("ERROR! Cannot read value in line %d.",
               line_number);
        return -1;
      }
      pdd.Density = dv * 1.0e6;  // cm^-3 -> m^-3
      double koef = e_elect * e_elect / (m_elect * eps_0);
      pdd.ElPlasmaW = pdd.Density * koef;
    }
    else if (strcmp(name, "DC mag. field [nT]") == 0) {
      if (sscanf(value, "%lf", &dv) != 1) {
```

```
    printf("ERROR! Cannot read value in line %d.",
           line_number);
    return -1;
  }
  double koef = e_elect / (m_elect * c_light);
  pdd.ElGyroW = dv * koef;
}
else if (strcmp(name, "Electron gyro freq. [Hz]") == 0) {
  if (sscanf(value, "%lf", &dv) != 1) {
    printf("ERROR! Cannot read value in line %d.",
           line_number);
    return -1;
  }
  pdd.ElGyroW = 2 * M_PI * dv;
}
else if (strcmp(name, "Electron gyro freq. [rad/s]") == 0) {
  if (sscanf(value, "%lf", &dv) != 1) {
    printf("ERROR! Cannot read value in line %d.",
           line_number);
    return -1;
  }
  pdd.ElGyroW = dv;
}
else if (strcmp(name, "Electron plasma freq. [Hz]") == 0) {
  if (sscanf(value, "%lf", &dv) != 1) {
    printf("ERROR! Cannot read value in line %d.",
           line_number);
    return -1;
  }
  pdd.ElPlasmaW = 4 * M_PI * M_PI * dv * dv;
}
else if (strcmp(name, "Electron plasma freq. [rad/s]") == 0) {
  if (sscanf(value, "%lf", &dv) != 1) {
    printf("ERROR! Cannot read value in line %d.",
           line_number);
    return -1;
  }
  pdd.ElPlasmaW = dv * dv;
}
else if (strcmp(name, "No. of plasma components") == 0) {
  if (sscanf(value, "%d", &iv) != 1) {
    printf("ERROR! Cannot read value in line %d.",
           line_number);
    return -1;
  }
  pdd.NComp = iv;
}
else {
  for (int i = 0; i < MaxPlasmaComp; i++) {
    char comp_name[33];
    sprintf(comp_name, "Comp %d type of dielectric tensor",
            i + 1);
    if (strcmp(name, comp_name) == 0) {
      if (sscanf(value, "%d", &iv) != 1) {
        printf("ERROR! Cannot read value in line %d.",
               line_number);
        return -1;
      }
      pdd.CompKind[i] = iv;
      found = true;
      break;
    }
    int param_idx = -1;
    sprintf(comp_name, "Comp %d num. density %%", i + 1);
    if (strcmp(name, comp_name) == 0) param_idx = 0;
    sprintf(comp_name, "Comp %d signed charge / charge of el.",
            i + 1);
    if (strcmp(name, comp_name) == 0) param_idx = 1;
    sprintf(comp_name, "Comp %d mass / mass of electron", i + 1);
    if (strcmp(name, comp_name) == 0) param_idx = 2;
    sprintf(comp_name, "Comp %d temperature [eV]", i + 1);
    if (strcmp(name, comp_name) == 0) param_idx = 3;
    sprintf(comp_name, "Comp %d norm. parallel drift", i + 1);
```

```
        if (strcmp(name, comp_name) == 0) param_idx = 4;
        sprintf(comp_name, "Comp %d loss cone depth (delta)", i + 1);
        if (strcmp(name, comp_name) == 0) param_idx = 5;
        sprintf(comp_name, "Comp %d temperature anisotropy", i + 1);
        if (strcmp(name, comp_name) == 0) param_idx = 6;
        sprintf(comp_name, "Comp %d loss cone anisotropy", i + 1);
        if (strcmp(name, comp_name) == 0) param_idx = 7;
        if (param_idx >= 0) {
          if (sscanf(value, "%lf", &dv) != 1) {
            printf("ERROR! Cannot read value in line %d.",
                   line_number);
            return -1;
          }
          pdd.CompPars[i][param_idx] = dv;
          found = true;
          break;
        }
        found = false;
      }
    }
    if (!found) {
      printf("ERROR! Can't read line %d!\n", line_number);
      return -1;
    }
  }
  fclose(f);

  for (int i = 0; i < pdd.NComp; i++) {
    // gyro frequency qB/mc
    pdd.GyroW[i] = pdd.ElGyroW;
    pdd.GyroW[i] *= pdd.CompPars[i][1];  // charge
    pdd.GyroW[i] /= pdd.CompPars[i][2];  // mass
    pdd.GyroW[i] = fabs(pdd.GyroW[i]);

    // plasma frequency wp^2 = n*q^2/(m*eps_0)
    pdd.PlasmaW[i] = pdd.ElPlasmaW;
    pdd.PlasmaW[i] *= pdd.CompPars[i][0] / 100.0;  // density
    pdd.PlasmaW[i] *= pdd.CompPars[i][1] * pdd.CompPars[i][1];  // charge
    pdd.PlasmaW[i] /= pdd.CompPars[i][2];  // mass
    pdd.PlasmaW[i] = fabs(pdd.PlasmaW[i]);

    pdd.CompPars[i][0] *= pdd.Density / 100.0;
    pdd.CompPars[i][1] *= e_elect;
    pdd.CompPars[i][2] *= m_elect;
  }

  return pdd.Check();
}

void CalcDistrFunction(double** F, double& L, double& p_norm,
                       const double m, const double kT,
                       const double v_dr, const double delta,
                       const double alpha1, const double alpha2,
                       int grid_size, double A)
{
  // kT means thermal velocity kT = m/2v^2 in J
  double maxwell_coef = 5;
  double p_real = sqrt(2 * m * kT);
  p_norm = 1.0 / p_real;
  L = p_real * maxwell_coef;
  double c = maxwell_coef / grid_size;
  for (int i = 0; i < grid_size; i++)
    for (int j = 0; j < grid_size; j++) {
      double x = c * i;  // v_per / v_th
      double y = c * (j - 0.5 * grid_size);  // v_par / v_th

      double result = 0.0;
      if (alpha1 != 0) result += exp(-x*x / alpha1);
      if (alpha2 != 0) result -= exp(-x*x / alpha2);
      result *= 1 - delta;
      if (delta != 1) result /= alpha1 - alpha2;
      result += (delta / alpha1) * exp(-x*x / alpha1);
      result *= exp(-y*y - v_dr*v_dr);
```

```
      // normalization
      result /= M_PI * sqrt(M_PI);
      F[i][j] = A * result;
    }
}

// reads or computes plasma distribution grid
int ReadPlasmaDistribution(PlasmaDefinitionStruct& pdd,
                           char** filenames)
{
  int result = 0;

  for (int i = 0; i < pdd.NComp; i++)
    if (pdd.CompKind[i] == 1) {  // just for hot plasma
      pdd.CompDistFunc[i] = new double*[pdd.PDGridSize];
      for (int j = 0; j < pdd.PDGridSize; j++)
        pdd.CompDistFunc[i][j] = new double[pdd.PDGridSize];

      if (filenames[i]) {
        printf("... of component %d "
               "from file %s ...\n", i + 1, filenames[i]);
        FILE* f = fopen(filenames[i], "r");
        if (!f) {
          printf("ERROR: Cannot open file with distr. function"
                 " (%s).\n", filenames[i]);
          return -1;
        }
        for (int j = 0; j < pdd.PDGridSize; j++)
          for (int k = 0; k < pdd.PDGridSize; k++)
            pdd.CompDistFunc[i][j][k] = 0.0;

        int size = 32 * pdd.PDGridSize;
        char line[size];
        int j = 0;
        while (fgets(line, size, f) > 0) {
          char* eol = strchr(line, '\n'); if (eol) *eol = 0;
          eol = strchr(line, '\r'); if (eol) *eol = 0;
          char* value = line;
          int k = 0;
          while (value && strlen(value) &&
                 j < pdd.PDGridSize) {
            char* semicolon = strchr(value, ';');
            if (semicolon)
              *semicolon = 0;

            double dv = 0.0;
            if  (sscanf(value, "%lf", &dv) != 1) {
              printf("Cannot read record %d at line %d.", k, j);
              result = -1;
              break;
            }
            pdd.CompDistFunc[i][j][k] = dv;

            if (semicolon)
              value = semicolon + 1;
            else
              value = 0;
            k++;
          }
          j++;
        }
        fclose(f);
      } else {
        // pdd.CompPars[i][3] means temperature in eV
        double T = pdd.CompPars[i][3];
        if (T <= 0.0) {
          PrintError("Temperature is not positive.");
          return -1;
        }

        T *= fabs(pdd.CompPars[i][1]);  // * charge
        double m = pdd.CompPars[i][2];  // mass
        //double A = pdd.CompPars[i][0] / pdd.Density;  // rel. density
```

```
        double A = 1;
        CalcDistrFunction(pdd.CompDistFunc[i], pdd.CompDistLength[i],
                          pdd.CompPNorm[i], m, T,
                          pdd.CompPars[i][4], pdd.CompPars[i][5],
                          pdd.CompPars[i][6], pdd.CompPars[i][7],
                          pdd.PDGridSize, A);
      }
    }

  // computing and writing to file of total distr. function
  double** distr = new double*[pdd.PDGridSize];
  for (int i = 0; i < pdd.PDGridSize; i++)
    distr[i] = new double[pdd.PDGridSize];
  for (int i = 0; i < pdd.PDGridSize; i++)
    for (int k = 0; k < pdd.PDGridSize; k++)
      distr[i][k] = 0;

  for (int i = 0; i < pdd.NComp; i++)
    if (pdd.CompKind[i] == 1) {  // just for hot plasma
      for (int j = 0; j < pdd.PDGridSize; j++)
        for (int k = 0; k < pdd.PDGridSize; k++)
          distr[j][k] += pdd.CompPars[i][0] /*density*/ *
            pdd.CompDistFunc[i][j][k] / pdd.Density;
    }

  FILE* f = fopen("distr_f.csv", "w");
  for (int j = 0; j < pdd.PDGridSize; j++) {
    for (int k = 0; k < pdd.PDGridSize; k++) {
      if (k > 0) fprintf(f, ";");
      fprintf(f, "%.16g", distr[j][k]);
    }
    fprintf(f, "\n");
  }
  fclose(f);

  for (int j = 0; j < pdd.PDGridSize; j++)
    delete[] distr[j];
  delete[] distr;

  return result;
}

void OutputHeadline(ControlDataStruct& CD)
{
  FILE* output = fopen("output.csv", "w");
  if (!output) {
    PrintError("Cannot open output file.");
    return;
  }

  bool first_column = true;
  for (int i = 0; i < MaxOutputColumn; i++) {
    if (CD.Output[i] == 0) continue;

    if (first_column)
      first_column = false;
    else
      fprintf(output, ";");

    fprintf(output, "%s", CD.Output[i]);
  }
  fprintf(output, "\n");

  fclose(output);
}

void Output(int res, cmplx Omega, r_vector& K,
            double k, double theta,
            PlasmaDefinitionStruct& PD,
            ControlDataStruct& CD)
{
  FILE* output = fopen("output.csv", "a");
  if (!output) {
```

```
  PrintError("Cannot open output file.");
  return;
}

cmplx N = c_light * k / Omega;
// normalization to the 1st plasma component
double v = PD.V();
double norm_coef = v / PD.GyroW[0];

bool first_column = true;
for (int i = 0; i < MaxOutputColumn; i++) {
  if (CD.Output[i] == 0) continue;

  if (first_column)
    first_column = false;
  else {
    printf(";");
    fprintf(output, ";");
  }

  if (strcmp(CD.Output[i], "k") == 0 ||
      strcmp(CD.Output[i], "size of wave vector") == 0) {
    printf("%s=%.16g", CD.Output[i], k);
    fprintf(output, "%.16g", k);
  }
  else if (strcmp(CD.Output[i], "norm_k") == 0 ||
      strcmp(CD.Output[i], "normalized k") == 0) {
    printf("%s=%.16g", CD.Output[i], k * norm_coef);
    fprintf(output, "%.16g", k * norm_coef);
  }
  else if (strcmp(CD.Output[i], "theta") == 0 ||
           strcmp(CD.Output[i], "th") == 0) {
    printf("%s=%.16g", CD.Output[i], theta);
    fprintf(output, "%.16g", theta);
  }
  else if (strcmp(CD.Output[i], "k_par") == 0 ||
           strcmp(CD.Output[i], "parallel k") == 0) {
    printf("%s=%.16g", CD.Output[i], K[0]);
    fprintf(output, "%.16g", K[0]);
  }
  else if (strcmp(CD.Output[i], "norm_k_par") == 0 ||
           strcmp(CD.Output[i], "normalized parallel k") == 0) {
   printf("%s=%.16g", CD.Output[i], K[0] * norm_coef);
    fprintf(output, "%.16g", K[0] * norm_coef);
  }
  else if (strcmp(CD.Output[i], "k_per") == 0 ||
           strcmp(CD.Output[i], "perpendicular k") == 0) {
    printf("%s=%.16g", CD.Output[i], K[1]);
    fprintf(output, "%.16g", K[1]);
  }
  else if (strcmp(CD.Output[i], "norm_k_per") == 0 ||
           strcmp(CD.Output[i], "normalized perpendicular k") == 0) {
    printf("%s=%.16g", CD.Output[i], K[1] * norm_coef);
    fprintf(output, "%.16g", K[1] * norm_coef);
  }
  else if (strcmp(CD.Output[i], "omega") == 0 ||
           strcmp(CD.Output[i], "w") == 0 ||
           strcmp(CD.Output[i], "frequency") == 0) {
    double f = real(Omega);  // in rad/s
    if (strcmp(CD.Output[i], "frequency") == 0)
      f /= 2.0 * M_PI;  // in Hz
    printf("%s=%.16g", CD.Output[i], f);
    fprintf(output, "%.16g", f);
  }
  else if (strcmp(CD.Output[i], "imaginary omega") == 0 ||
           strcmp(CD.Output[i], "im_w") == 0 ||
           strcmp(CD.Output[i], "imaginary part of frequency") == 0) {
    double f = imag(Omega);  // in rad/s
    if (strcmp(CD.Output[i], "imaginary part of frequency") == 0)
      f /= 2.0 * M_PI;  // in Hz
    printf("%s=%.16g", CD.Output[i], f);
    fprintf(output, "%.16g", f);
  }
```

```
    else if (strcmp(CD.Output[i], "normalized omega") == 0 ||
             strcmp(CD.Output[i], "norm_w") == 0 ||
             strcmp(CD.Output[i], "normalized frequency") == 0) {
      // normalization to the electron cyklotron frequency
      // normalized frequency and normalized omega are the same
      double gw = PD.GyroW[0];
      gw *= PD.CompPars[0][2] / m_elect;
      gw /= PD.CompPars[0][1] / e_elect;
      gw = fabs(gw);
      printf("%s=%.16g", CD.Output[i], real(Omega) / gw);
      fprintf(output, "%.16g", real(Omega) / gw);
    }
    else if (strcmp(CD.Output[i], "N") == 0) {
      printf("%s=%.16g", CD.Output[i], real(N));
      fprintf(output, "%.16g", real(N));
    }
    else if (strcmp(CD.Output[i], "square N") == 0) {
      printf("%s=%.16g", CD.Output[i], norm(N));
      fprintf(output, "%.16g", norm(N));
    }
    else if (strcmp(CD.Output[i], "v") == 0) {
      printf("%s=%.16g", CD.Output[i], PD.DistV());
      fprintf(output, "%.16g", v);
    }
    else if (strcmp(CD.Output[i], "v/c") == 0) {
      printf("%s=%.16g", CD.Output[i], v / c_light);
      fprintf(output, "%.16g", v / c_light);
    }
    else if (strcmp(CD.Output[i], "result") == 0) {
      printf("%s=%d", CD.Output[i], res);
      fprintf(output, "%d", res);
    }

  }
  printf("\n");
  fprintf(output, "\n");

  fclose(output);
}
```