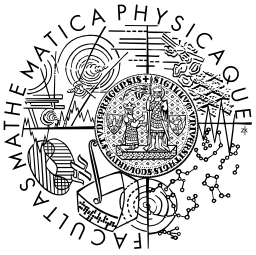


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Jáchym Bártík

Refaktoring systému pro sledování kvality dat ve vrcholovém detektoru Belle II

Ústav částicové a jaderné fyziky

Vedoucí bakalářské práce: RNDr. Peter Kvasnička

Studijní program: Fyzika

Studijní obor: Obecná fyzika

Praha 2020

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

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 má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Za odborné vedení práce děkuji RNDr. Peterovi Kvasničkovi. Chtěl bych vyzdvihnout zejména trpělivost, s jakou mě zásoboval podnětnými poznámkami a komentáři, stejně jako dobrými radami. S pochopením fungování kódu a experimentu Belle II obecně mi dále pomáhali Mgr. Tadeáš Bilka a doc. RNDr. Peter Kodyš, CSc., za což jim patří moje poděkování. V neposlední řadě děkuji prof. RNDr. Zdeňkovi Doležalovi, Dr., za to, že mě zahrnul do pravidelných konferencí pražské skupiny Belle II, které pro mě byly velmi přínosné.

Název práce: Refaktoring systému pro sledování kvality dat ve vrcholovém detektoru Belle II

Autor: Jáchym Bártík

Ústav: Ústav částicové a jaderné fyziky

Vedoucí bakalářské práce: RNDr. Peter Kvasnička, Ústav částicové a jaderné fyziky

Abstrakt: Tato bakalářská práce se zabývá refaktoringem části kódu pro sledování kvality dat z vrcholového detektoru Belle II. Druhou částí práce je implementace sledování pohybů half-shellů pixelového a stripového vrcholového detektoru Belle II v čase.

Po shrnutí základních informací o experimentu Belle II, jeho vrcholovém detektoru a softwarovém systému basf2 popíšeme funkční chování modulů pro sledování kvality dat. Analyzujeme kód modulů `TrackDQM` a `AlignDQM` a opravíme chyby, které jsme v nich našli. V rámci refaktoringu rozdělíme výpočty do více jednodušších částí a výrazně zredukujeme duplicitní kód. Nakonec přidáme histogramy reziduálů z half-shellů, které po vyhodnocení na reálných datech fitujeme a zobrazíme v závislosti na čase.

Přínosem této práce je lépe udržovatelný a snáze rozšiřitelný kód, stejně jako opravy mnoha chyb. Dalším výsledkem je objevení pravidelných pohybů half-shellů, jejichž pochopení může vést ke zlepšení alignmentu detektoru.

Klíčová slova: Belle II, basf2, data quality monitoring, refaktoring, half-shell

Title: Refactoring of the data quality monitoring system in the Belle II vertex detector

Author: Jáchym Bártík

Institute: Institute of Particle and Nuclear Physics

Supervisor: RNDr. Peter Kvasnička, Institute of Particle and Nuclear Physics

Abstract: This bachelor thesis deals with refactoring of a part of the Data Quality Monitoring (DQM) code for the Belle II vertex detector. In the second part of the thesis, monitoring of vertex detector half-shells movement over time is implemented.

After summarizing the basic information about the Belle II experiment, its vertex detector and the basf2 software system, we will describe the functional behaviour of the data quality monitoring modules. We analyze the code of the `TrackDQM` and `AlignDQM` modules and fix several errors. As a part of refactoring, we divide the calculations into several simpler parts and significantly reduce the code duplication. Finally, we add histograms of residuals from the half-shells, which we evaluate on real data, fit and display as a function of time.

The contribution of this work is a code that is easier to maintain and extend, as well as free of some obvious bugs. Another result is a discovery of periodical movements of vertex detector half-shells, the understanding of which can lead to improvement of detector alignment.

Keywords: Belle II, basf2, data quality monitoring, refactoring, half-shell

Obsah

Úvod

Fyzika je věda, jejímž cílem je popsat realitu na té nejzákladnější úrovni. Nástroji, které k tomu používá, jsou fyzikální zákony – odpozorovaná pravidla, o kterých očekáváme, že se jimi svět kolem nás bude řídit. Tyto zákony mají některé společné vlastnosti, které považujeme za samozřejmé, protože se s nimi setkáváme neustále. Příkladem toho jsou symetrie, neboli transformace, které nijak nemění chování fyzikálního systému.

Jazykem, ve kterém fyzikální zákony vyjadřujeme, je matematika a její rovnice. Transformace daná symetrií má potom význam konkrétní matematické operace, kterou můžeme aplikovat na tyto rovnice tak, že je tím vůbec nezměníme. Jednou takovou transformací je například posun počátku souřadnicové soustavy do libovolného bodu prostoru. Přepíšeme-li polohy všech částic do nových souřadnic, rovnice představující fyzikální zákony budou v nových souřadnicích vypadat stejně jako v těch starých. Tato skutečnost je známá jako translační symetrie a protože se jedná o spojitou symetrii, přísluší jí nějaká zachovávající se fyzikální veličina, v tomto případě hybnost. Kromě spojitých symetrií existují ještě diskrétní, které představují nespojité transformace systému.

Zásadní význam symetrií spočívá v tom, že nám dávají konkrétní matematické podmínky, které lze klást na kandidáty na fyzikální zákony. Vzhledem k tomu, že translační symetrie se zdá být univerzálně platná, můžeme se při hledání nových teorií omezit pouze na ty, které ji automaticky splňují. Každá pozorovaná symetrie musí být reflektována ve fyzikálních rovnicích, a naopak každé transformaci, která nemění výsledné rovnice, musí odpovídat symetrie světa kolem nás. O to zajímavější jsou případy, kdy se ukáže, že se u některých fyzikálních procesů rovnice změní při transformaci, která se jinak ve všech ostatních případech chová jako symetrie. Tomu se říká narušení symetrie.

Standardní model je dnes velmi úspěšnou fyzikální teorií, jejíž závěry již potvrdilo mnoho experimentů. Jednou z mála skutečností, které nedokáže uspokojivě vysvětlit, je zřejmá nerovnováha mezi počty částic a antičástic v našem vesmíru. Tato asymetrie může mít i jiná vysvětlení, například počáteční nerovnováhu. Pokud by ale byla způsobena tím, že se částice a antičástice chovají odlišně, měla by být vidět v rovnicích standardního modelu.

Hledání základních symetrií má v částicové fyzice dlouhou historii. Podle dosavadních poznatků se zdá, že se zachovává CPT symetrie. Ta kombinuje tři operace – sdružení náboje (*charge conjugation*), transformaci parity (*parity transformation*) a obrácení času (*time reversal*). První z nich znamená, že každou částici nahradíme její antičásticí. Druhá představuje zrcadlení vzhledem k libovolné rovině (neboli obrácení prostorové souřadnice kolmé na tuto rovinu), zatímco třetí obrací časovou souřadnici.

Původně se předpokládala platnost všech tří symetrií samostatně, protože k jejich narušení dochází pouze u některých procesů zprostředkovaných slabou interakcí. Po prokázání narušení C a P symetrie se uvažovalo o platnosti kombinované CP symetrie, ale později se ukázalo, že ani ta se nezachovává. To však nebylo v souladu s tehdejším standardním modelem, obsahujícím pouze čtyři kvarky. Vyřešení tohoto problému si vyžádalo předpověď třetí rodiny kvarků, která byla

později experimentálně potvrzena.

Samotné narušení CP symetrie se však projevuje velmi slabě, takže nalézt jevy, ve kterých k němu dochází, není vůbec snadné. S tím souvisí i to, že ačkoli ze standardního modelu vyplývá jistá asymetrie mezi částicemi a antičásticemi, je příliš malá na to, aby vysvětlila nerovnováhu v našem vesmíru. Objevování nových procesů narušujících CP symetrii se věnuje velké úsilí, protože jejich pochopení by mohlo vést k vyřešení jednoho z největších problémů standardního modelu a současné fyziky obecně.

Slibnou oblastí výzkumu narušení CP symetrie je měření rozpadů B mezonů, což jsou částice složené z jednoho b antikvarku ($\overline{\mathrm{b}}$) a jednoho lehčího kvarku (d, u, s, c). Příslušná antičástice, $\overline{\mathrm{B}}$, se skládá z b kvarku a lehčího antikvarku. Pravděpodobnosti a další parametry rozpadů B mezonů byly podrobně měřeny na urychlovačích částic v rámci experimentů BaBar a Belle. V těchto urychlovačích se srážely elektrony s pozitrony při těžišťové energii blízké $\Upsilon(4\mathrm{S})$ rezonanci, která se následně rozpadala na pár $\mathrm{B}\overline{\mathrm{B}}$. Tento proces umožňuje vytvářet B mezony ve velkých množstvích, díky čemuž je možné měřit parametry jejich rozpadů s malou nejistotou. Zařízení tohoto typu se proto označují jako B-továrny.

Po ukončení experimentu Belle a následné modernizaci urychlovače byl spuštěn experiment Belle II. Tím se dostáváme k podstatě této práce, která se zabývá refaktoringem kódu na kontrolu kvality dat z experimentu Belle II.

V současné době si fyzikální výzkum bez pomoci počítačů již nedokážeme představit. Objemy dat, které jsou nutné pro získání rozumně přesných výsledků z částicových experimentů, není možné zpracovat ručně. Stejně tak není v lidských silách ovládat nesmírně složité stroje, kterými jsou urychlovače částic a částicové detektory. Vývoj experimentální fyziky je dnes podmíněn jednak technologickou úrovní detektorů a dalších zařízení, která dokážeme sestrojit, a za druhé softwarem, pomocí kterého tyto přístroje ovládáme a zpracováváme naměřená data.

Vzrůstající nároky na software vedou k větší složitosti kódu, která výrazně zpomaluje opravy existujících chyb stejně jako další vývoj. Na jednotlivých projektech zároveň spolupracuje stále více lidí, což dále zvyšuje požadavky na přehlednější a snadněji pochopitelný kód. To je přesně to, čeho se budeme v rámci této práce snažit dosáhnout.

V první kapitole se budeme zabývat experimentem Belle II. Seznámíme se s jeho detektorem a dále s některými subdetektory, které budou významné pro další části práce. Mimo to si ukážeme základní postupy, které se při zpracování dat používají.

V druhé kapitole si popíšeme softwarový systém, který slouží k vyhodnocování dat z detektoru Belle II. Zaměříme se na to, co je pro tuto práci nejzásadnější, a sice na moduly, generující histogramy pro monitorování kvality dat, a z nich zejména na moduly `TrackDQM` a `AlignDQM`. V závěru k teoretické části práce si představíme cíle, kterých bychom chtěli v praktické části dosáhnout.

Následující kapitola se bude zabývat refaktoringem obou modulů. Nejdřív vytvoříme specifikaci, určující, co přesně by měly dělat. Potom se budeme snažit přepsat jejich kód tak, aby se choval stejně, jako předtím, ale současně byl přehlednější a snáze se udržoval a dále vyvíjel. Přitom si projdeme jak konkrétní

změny, které v kódu uděláme, tak důvody a rozhodnutí, které k nim povedou.

Ve čtvrté kapitole si ukážeme výhody refaktorovaného kódu, když jej rozšíříme o některé nové funkce. Konkrétně se pokusíme sledovat dlouhodobé posuny jednotlivých částí detektoru tím, že přidáme a následně statisticky zpracujeme histogramy, zobrazující reziduály ze zásahů senzorů z vybraných částí detektoru.

V poslední kapitole budeme diskutovat výsledky, ke kterým jsme v předchozích částech práce došli.

1. Experiment Belle II

1.1 Experiment Belle

Nepřímé narušení CP symetrie bylo poprvé pozorováno při oscilacích neutrálních kaonů v druhé polovině minulého století [6]. Později bylo objeveno přímé narušení při rozpadech kaonů. Cílem experimentu Belle bylo prokázání narušení CP symetrie při rozpadech B mezonů.

Detektor Belle byl umístěn na urychlovači KEKB, který se nachází v Japonsku nedaleko od Tsukuby. Experiment probíhal mezi roky 1999 a 2010, během této doby bylo dosaženo integrované luminozity při sběru dat $1\,040\,\mathrm{fb}^{-1}$. Z toho celkem $711\,\mathrm{fb}^{-1}$ připadá na produkci $\Upsilon(4S)$ rezonance, což představuje přibližně $772 \cdot 10^6$ párů $B\overline{B}$. V roce 2009 byl překonán světový rekord v hodnotě okamžité luminozity [2], maximální hodnota této veličiny za celou dobu provozu urychlovače byla $2{,}11 \cdot 10^{34}\,\mathrm{cm}^{-2}\cdot\mathrm{s}^{-1}$. V urychlovači se srážely elektrony s pozitrony, a to převážně při energiích 8 GeV resp. 3,5 GeV, aby těžišťová energie srážky 10,58 GeV dosáhla hodnoty nutné pro vznik $\Upsilon(4S)$ rezonance. [4]

Na tomto projektu má zásadní podíl japonská organizace KEK (*High Energy Accelerator Research Organization*), ale podíleli se na něm vědci z celého světa. Kromě pozorování narušení CP symetrie byly v rámci experimentu změřeny i parametry několika dalších rozpadů. Výsledky z Belle také vedly k udělení Nobelovy ceny za fyziku v roce 2008 pro Makoto Kobajašiho a Tošihide Masukawu. Na Belle následně navázal experiment Belle II s cílem objevit novou fyziku za standardním modelem v oblasti B mezonů.

1.2 Urychlovač SuperKEKB

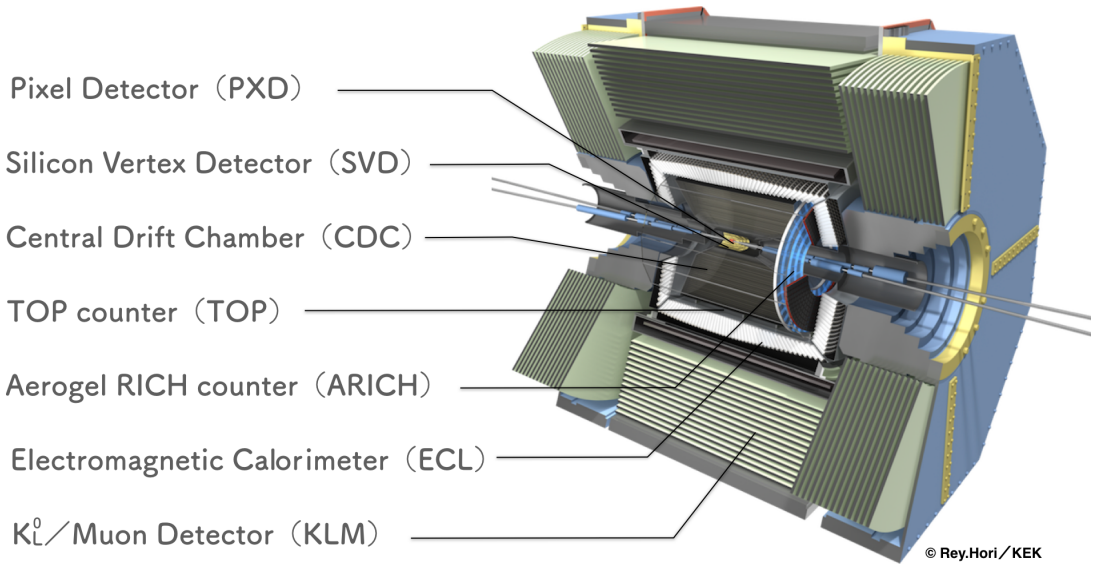
V roce 2010 započala přestavba na urychlovač SuperKEKB, který se nyní nachází ve stejném tunelu, jako se nacházel jeho předchůdce. Zásadním zlepšením bylo zvýšení okamžité luminozity na hodnotu $8 \cdot 10^{35}\,\mathrm{cm}^{-2}\cdot\mathrm{s}^{-1}$, což je téměř čtyřicetkrát více než u urychlovače KEKB. Mírně se zmenšila asymetrie v energiích jednotlivých částic, a to na hodnoty 7 GeV resp. 4 GeV, čemuž odpovídá těžišťová energie 10,57 GeV. [7]

Od roku 2016 probíhala první testovací měření částic kosmického záření, v roce 2018 se začaly měřit srážky částic bez vrcholového detektoru a od dubna roku 2019 nabírá detektor Belle II plnohodnotná fyzikální data.

1.3 Detektor Belle II

Na urychlovači SuperKEKB se nachází jeden detektor, který se skládá z několika subdetektorů (viz obrázek 1.1). V této práci se budeme zajímat především o jeho centrální část, vrcholový detektor, skládající se ze dvou pixelových (PXD) a čtyř stripových (SVD) vrstev. Tyto subdetektory budou podrobněji popsány v následujících odstavcích. Mimo nich se na měření podílí

- CDC (*central drift chamber*), zajišťující (stejně jako PXD a SVD) určení drah (*tracking*) částic,
- detektory Čerenkovova záření TOP (*time-of-propagation*) a ARICH (*aerogel ring-imaging Cherenkov detector*), které slouží k identifikaci částic,
- ECL (*electromagnetic calorimeter*), detekující fotony,
- KLM (K^0_L *and muon detector*), který, jak již název napovídá, identifikuje kaony a miony. [7]



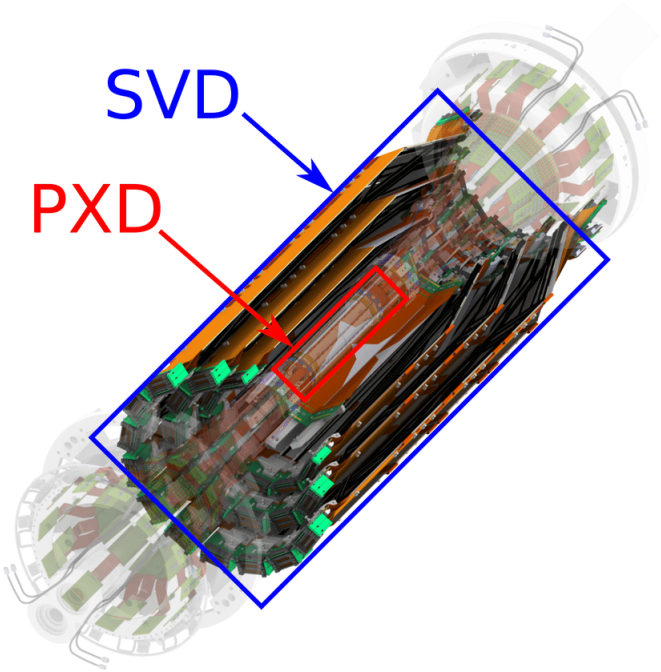
Obrázek 1.1: Struktura detektoru Belle II.

1.3.1 Vrcholový detektor

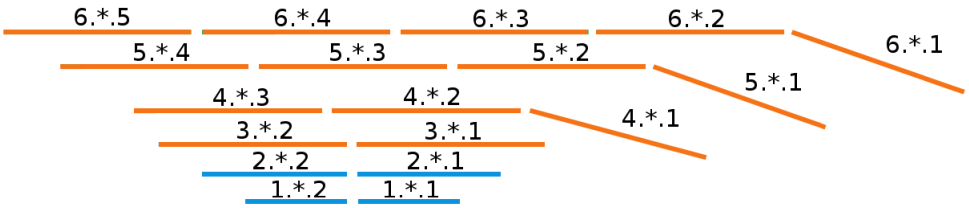
Detektory PXD (*pixel detector*) a SVD (*silicon vertex detector*) se souhrnně nazývají VXD (*vertex detector*). Oba druhy detektorů se skládají z obdélníkových křemíkových destiček, kterým se říká senzory (*sensory*). Opěrná struktura, na které je umístěno několik senzorů v řadě za sebou, se nazývá *ladder*. Laddery jsou uspořádány do válcových vrstev (*layerů*). Těch je ve vrcholovém detektoru Belle II celkem šest. Pohled z boku na část VXD s vyznačenými vrstvami a senzory je na obrázku 1.3. Všechny senzory jsou znázorněny na obrázku 1.4, kde je dále vidět pořadí ladderů ve vrstvách a počty senzorů v ladderech. Nákres struktury VXD můžeme vidět na obrázku 1.2. [7]

V dalším textu používáme veličiny u a v. Jedná se o lokální souřadnice senzoru, kde v je vždy rovnoběžná se směrem pohybujících se svazků částic a u je na ni kolmá. Podrobnější popis je možné najít v sekci Parametrizace.

Aby bylo možné celý křemíkový detektor sestavit, skládá se ze dvou částí, z nichž každá obsahuje přibližně polovinu ladderů z každé vrstvy. Každá z polovin se označuje jako *half-shell*. Tím je pixelový detektor rozdělen na části Yin a Yang, zatímco stripový detektor je rozdělen na části Pat a Mat, což je znázorněno na obrázku 1.4. [12]



Obrázek 1.2: Nákres PXD a SVD detektorů.



Obrázek 1.3: Pohled z boku na šest ladderů z VXD. Modrá barva označuje PXD, hnědá je pro SVD. První číslo z každé trojice označuje vrstvu, třetí senzor.

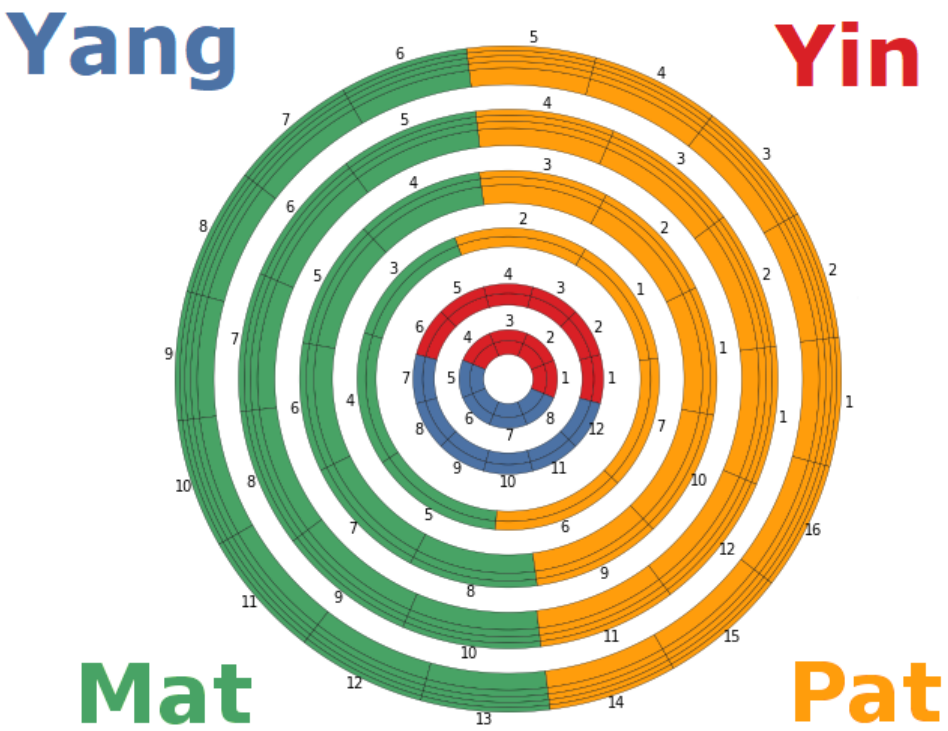
1.3.2 Pixelový křemíkový detektor

Nejblíže k interakčnímu bodu se nachází dvě vrstvy PXD typu DEPFET (*depleted field effect transistor*). Tyto detektory měří souřadnice (u, v) procházejících částic. PXD vrstvy nebyly součástí detektoru v experimentu Belle, byly přidány až u detektoru Belle II. Jejich parametry můžeme vidět v tabulce 1.1. Rozměry pixelů se v různých místech senzorů liší, což je v tabulce naznačeno rozmezím hodnot. [8]

Druhá vrstva není v době psaní této práce kompletní, zatím v ní jsou nainstalované pouze dva laddery.

1.3.3 Stripový křemíkový detektor

Za dvěma vrstvami PXD následují čtyři vrstvy SVD. Ty se označují také jako stripový detektor, protože měřící plochy nejsou na rozdíl od PXD pixely, ale proužky (*stripy*). Ty jsou na jedné straně orientovány ve směru u a na druhé ve směru v. V rámci jednoho senzoru se tak nachází proužky obou orientací, čímž je možné změřit obě souřadnice prolétávající částice. Pro každý SVD senzor



Obrázek 1.4: Schéma VXD se zvýrazněnými half-shelly. Čísla představují pořadí ladderu ve vrstvě.

Tabulka 1.1: Současné parametry PXD typu DEPFET.

vrstva	1	2
počet ladderů ve vrstvě	8	12
počet senzorů v ladderu	2	2
počet pixelů v senzoru $(u \times v)$	250×768	250×768
rozměry pixelu $(u \times v)$	$50 \times (55 - 60)$ μm^2	$50 \times (70 - 85)$ μm^2

vznikají dva různé zásahy (*hity*) ve dvou soustavách stripů, zatímco pro PXD vzniká jen jeden. Rozdíl ve zpracování dat se projeví v případě, že senzor zasáhne více částic – potom jsou výstupem z SVD dvě množiny zásahů, jedna pro u a druhá pro v, a je na nás, abychom z dalších dostupných dat zjistili, kterému zásahu v u odpovídá který ve v. Naopak v případě PXD tento problém odpadá, protože měřením rovnou získáme jednu množinu dvojic (u, v).

Přesné parametry jsou zobrazeny v tabulce 1.2. [8]

1.4 Zpracování dat

Výstupem VXD jsou polohy zásahů v lokálních souřadnicích. Ty je potřeba převést do globálních souřadnic, viz sekce Parametrizace. Tím získáme body v prostoru, které se pokusíme seskupit do drah (*tracků*). Jelikož se zajímáme o dráhy nabitých částic v magnetickém poli, ze vzorce pro Lorentzovu sílu vyplývá, že se jedná o šroubovice. Ty jsou jednoznačně určeny pěti parametry, jejichž hodnoty zjistíme fitem. Přitom ještě musíme vzít v úvahu to, že částice re-

Tabulka 1.2: Parametry SVD.

vrstva	3	4	5	6
počet ladderů ve vrstvě	7	10	12	16
počet senzorů v ladderu	2	3	4	5
počet pruhů měřících u/v	768/768	768/512		
šířka pruhů měřících u/v	50 µm/160 µm	(50 − 75) µm/240 µm		

agují s materiálem detektoru, v důsledku čehož mění směr svého pohybu a ztrácí energii.

Z parametrů fitu a z informací z dalších částí detektoru zjistíme, o jaké částice se jedná. Posledním krokem je fitování průsečíků jejich drah, čímž najdeme interakční body a můžeme zrekonstruovat dráhy i těch částic, které nejsou detektorem přímo pozorovatelné. [7]

1.4.1 *Tracking*

Tímto termínem se označuj proces, kterým z bodů v prostoru, představující jednotlivé zásahy, získáme parametry drah reálných fyzikálních částic. Samotné přiřazení zásahů do drah je náročnou kombinatorickou úlohou, která se řeší pomocí neuronových sítí. Konkrétně se používá Hopfieldova síť, trénovaná na Monte Carlo simulacích. Ty věrně simulují skutečné srážky s tím rozdílem, že u nich známe všechny parametry částic, čili i to, které zásahy patří do které dráhy. K fitování se používají různé algoritmy z knihovny GenFit2, založené na Kálmánově filtru. [14]

Parametrizace

Pro fitování drah používáme globální souřadnice x, y a z. Rovnoběžně se směrem pohybu srážejících se částic se nachází osa z, na ni jsou kolmé osy x a y. Jak již jsme zmínili, VXD senzory jsou velmi tenké obdélníkové destičky, proto na nich zavádíme lokální souřadnice u, v a w s počátkem ve středu senzoru. Osa v míří ve směru delšího rozměru senzoru, zatímco osa u míří ve směru kratšího rozměru senzoru. Na ně je kolmá osa w, směřující ven z detektoru. Souřadnicová soustava (v, u, w) je pro každý senzor pravotočivá. Většina senzorů je umístěna ve válcových vrstvách okolo svazku, proto je u nich osa v rovnoběžná s osou z. Ostatní senzory jsou mírně skloněné, viz obrázek 1.3, čili tam tomu tak není.

Označíme-li globální resp. lokální souřadnice vektoru jako $\mathbf{r} = (x, y, z)$ resp. $\mathbf{q} = (u, v, w)$, bude v ideálním případě platit

$$\mathbf{r} = \mathbf{R}^\top \mathbf{q} + \mathbf{r}_0 \,, \tag{1.1}$$

kde R je matice rotace a $\mathbf{r}_0$ je pozice středu senzoru v globálních souřadnicích. Kvůli výše zmíněným vlivům však zavádíme opravu na rotaci ΔR a na polohu $\Delta\mathbf{q}$, čímž dostaneme vztah

$$\mathbf{r} = \mathbf{R}^\top \Delta\mathbf{R} \, (\mathbf{q} + \Delta\mathbf{q}) + \mathbf{r}_0 \,. \tag{1.2}$$

1.4.2 *Alignment*

Pro určení invariantních hmot a dalších parametrů produktů srážek potřebujeme znát jejich dráhy s co největší přesností. Měření bodů zásahů VXD senzorů v lokálních souřadnicích je relativně přesné, chyba se pohybuje v řádu jednotek až nižších desítek mikrometrů [7]. Převod do globálních souřadnic už je o něco problematičtější, protože k tomu musíme určit polohy senzorů v prostoru, což je zatíženo mnohem větší chybou, než samotné měření zásahů v senzorech. Primárním důvodem je to, že detektor nedokážeme sestavit s požadovanou přesností. Konstrukce detektoru i jednotlivé senzory se navíc vlivem proměnlivých podmínek různě deformují.

Výchozí pozice senzorů tak musíme opravit pomocí samotných naměřených dat při procesu zvaném *alignment*. Ten spočívá v tom, že kvalita fitu rekonstruovaných dat se při špatně určených polohách senzorů snižuje. Normalizovaný dráhový reziduál spočítáme jako

$$\mathbf{z}_{ij}(\boldsymbol{\tau}_j, \mathbf{a}) = \frac{\mathbf{u}_{ij}^{\mathrm{m}} - \mathbf{u}_{ij}^{\mathrm{p}}(\boldsymbol{\tau}_j, \mathbf{a})}{\sigma_{ij}} = \frac{\mathbf{r}_{ij}(\boldsymbol{\tau}_j, \mathbf{a})}{\sigma_{ij}}\,, \tag{1.3}$$

kde $\mathbf{u}_{ij}^{\mathrm{m}}$ je naměřená poloha zásahu i v dráze j, $\mathbf{u}_{ij}^{\mathrm{p}}(\boldsymbol{\tau}_j, \mathbf{a})$ je předpovězená poloha tohoto zásahu a σ_{ij} je nejistota měření. Dále $\boldsymbol{\tau}_j$ jsou parametry j-té dráhy a $\mathbf{a}$ jsou hledané parametry alignmentu. Ty popisují posun, rotaci a deformaci každého senzoru. Veličina $\mathbf{r}_{ij}$ se pak označuje jako reziduál. Při alignmentu se snažíme zvolit takové parametry $\mathbf{a}$, abychom minimalizovali součet druhých mocnin normalizovaných dráhových reziduálů

$$\chi^2(\boldsymbol{\tau}, \mathbf{a}) = \sum_{j}^{N_{\mathrm{d}}} \sum_{i}^{N_{\mathrm{z}}^j} \mathbf{z}_{ij}^2(\boldsymbol{\tau}_j, \mathbf{a})\,, \tag{1.4}$$

kde N_{d} je počet rekonstruovaných drah a N_{z}^j je počet zásahů v j-té dráze.

K tomu je potřeba velké množství rekonstruovaných drah, neboť počet parametrů je v řádu tisíců[1]. Používají se data nejen z měření částic vzniklých při srážkách v urychlovači, ale například i z kosmického záření. Základem samotného výpočtu je algoritmus Millipede II. [9]

Reziduály mají význam i při trackingu, protože minimalizací χ^2 podle $\boldsymbol{\tau}$ fitujeme dráhy.

Half-shelly

Aby byla nejistota měření co nejmenší, je třeba alignment senzorů provádět pravidelně. Pokaždé přitom fitujeme hodnoty tisíců parametrů, což sice je zvládnutelné, když k tomu máme dostatek dat a vhodné nástroje, ale zase nemůžeme sledovat dlouhodobé chování jednotlivých senzorů. Řešením může být sledování pohybů větších částí detektoru najednou. V této práci se zaměříme na half-shelly, konkrétně se budeme zajímat o reziduály senzorů z daných half-shellů v globálních souřadnicích.

[1] Detektor tvoří 40 PXD senzorů a 172 SVD senzorů, což je celkem 212. Pro každý senzor potřebujeme 6 parametrů pro posun a rotaci a dalších 7 parametrů popisujících deformace. Ve výsledku tak hledáme hodnoty 2 756 parametrů.

2. Software v experimetu Belle II

Tato kapitola se zabývá technickými detaily software k měření, zpracování a uchovávání dat z experimentu Belle II. Po krátkém přehledu podíváme na jeho jednotlivé části. Jako první zmíníme ROOT [5], který je zásadní v tom, že definuje objekty, s kterými pracují všechny ostatní části frameworku. Dále se seznámíme s kódem, který byl specificky napsán pouze pro tento experiment. V poslední sekci si projdeme cíle této práce a podrobněji se zaměříme na kód, který budeme později upravovat.

2.1 Přehled frameworku basf2

Téměř veškerý software, který je v experimentu Belle II potřeba k analýze a simulacím, je obsažen ve specializovaném frameworku basf2 (Belle II *analysis software framework* 2), který se skládá ze tří částí. Veřejně dostupné programy a knihovny se souhrnně nazývají *externals* a patří mezi ně například SCons, Git, Python 3, ROOT, GenFit2 a Geant4. Nástroje pro správu a vývoj kódu se nazývají *tools*. Obsahují skripty na instalaci ostatních částí frameworku, nebo třeba na administraci kódu. Poslední a pro nás nejdůležitější částí je samotný kód na zpracování dat, který je z naprosté většiny napsán v jazyku `C++`. Mimo to obsahuje rozhraní umožňující spouštět `C++` kód ze skriptů napsaných v Pythonu. Python tak slouží jako skriptovací jazyk pro celý framework. Tento přístup kombinuje výpočetní výkon, přehlednost a výhody silného typování `C++` se snadnou použitelností Pythonu. [10]

2.1.1 ROOT

Nezbytným požadavkem na systém basf2 je spolehlivý a jednotný způsob uchovávání velkého množství dat. Systém je proto postaven na ROOTu, což je soubor nástrojů pro zpracování, vizualizaci a uchovávání dat. Původně byl navržen pro potřeby částicové fyziky, neboť jej vyvíjí organizace CERN, ale je možné jej použít i v jiných vědních oborech a při zpracování dat obecně.

Základem ROOTu je knihovna napsaná v `C++`, která obsahuje například třídy pro různé druhy histogramů (`TH1F`[1]), funkcí (`TF1`[2]), ale i prvků souborového systému (`TFile`). Na objektový přístup je kladen velký důraz, všechny třídy v ROOTu dědí ze základní třídy `TObject`. Díky tomu sdílejí spousty společných funkcí, například metody `GetName`, `Copy` nebo `Draw`.

Zásadní výhodou ROOTu je, že každý `TObject` je možné snadno převést do binární formy, komprimovat a uložit, a to nezávisle na souborovém systému. K tomu slouží *streamery*, což jsou funkce pro ukládání konkrétních typů objektů, které jsou vytvořeny automaticky s využitím introspekce. Z tohoto důvodu jsou datonosné třídy v basf2 potomkem třídy `TObject` a naměřená data jsou už na počátku

[1] Ačkoli se to na první pohled nezdá, název této třídy dává smysl. První písmeno, `T`, mají společné všechny datové třídy v ROOTu. Následuje `H` jako histogram, `1` značí jednu nezávislou osu x a `F` znamená datový typ `float`. Například `TH2I` by byl 2D histogram s osami x, y a s proměnnými typu `int`.

[2] Tentokrát `F` značí, že se jedná o funkci, a `1` opět znamená jednu nezávislou proměnnou.

zpracování převedena do ROOTovských souborů a v této formě jsou dále uchovávána. Podstatným prvkem těchto souborů je sloupcová datová struktura `TTree`, která umožňuje velmi rychlý přístup k datům.

Dále, součástí ROOTu je i program, fungující jako interpret jazyka `C++`. Dokáže provádět jednotlivé příkazy zvlášť, čím fakticky převádí `C++` na skriptovací jazyk. Rozhraní PyROOT umožňuje přístup k ROOTovským objektům z prostředí Pythonu, což je zásadní pro funkci Pythonu v rámci celého frameworku, jak bylo zmíněno výše.

Nakonec, ROOT obsahuje uživatelské rozhraní, přes které lze procházet ROOTovské soubory, zobrazovat jednotlivé grafy a histogramy apod., zprostředkované například třídou `TBrowser`. [3]

2.1.2 Vývoj softwaru

Basf2 rozhodně není malý projekt, jenom samotný kód v nezkompilované podobě zabírá jednotky gigabajtů a podílí se na něm stovky lidí. Aby se všichni, kdo se na vývoji frameworku podílí, navzájem domluvili, používá se několik různých systémů. Základem je verzovací systém Git, který je dnes standardem u většiny softwarových projektů. Nad ním pracují programy od společnosti Atlassian, například

- Jira, *issue tracker*, neboli systém udržující přehled o chybách a obecně problémech v kódu, ale také o nových funkcích nebo návrzích na vylepšení těch stávajících,
- Bitbucket, on-line Git repozitář,
- Confluence, wiki obsahující například dokumentaci, návody a používané konvence.

Tyto systémy běží na serverech německé výzkumné organizace DESY (*Deutsches Elektronen-Synchrotron*), která se na experimentu Belle II také podílí. [10]

2.2 Moduly a knihovny

Každý větší softwarový projekt musí řešit problémy spojené se vzrůstající složitostí kódu, která jednak výrazně zpomaluje další vývoj a za druhé je často zdrojem těžko odhalitelných chyb. Kód v basf2 je proto rozdělen do knihoven a modulů. Knihovna vždy obsahuje funkce, které se týkají nějaké části zpracování dat. Přitom může využívat nástroje z dalších knihoven. Oproti tomu moduly na sobě navzájem nezávisí, ke své činnosti používají pouze funkce z knihoven.

Celé toto dělení má ale hlubší význam. Modul je nejmenší částí kódu, která může běžet samostatně, a představuje tak základní stavební kámen zpracování dat. Pro každou úlohu si uživatel může vybrat, které moduly a v jakém pořadí se mají použít. Uspořádaná sekvence modulů se pak nazývá cesta (*path*). Cest můžeme vytvořit i více, společně s podmínkami, podle kterých se mezi cestami přechází.

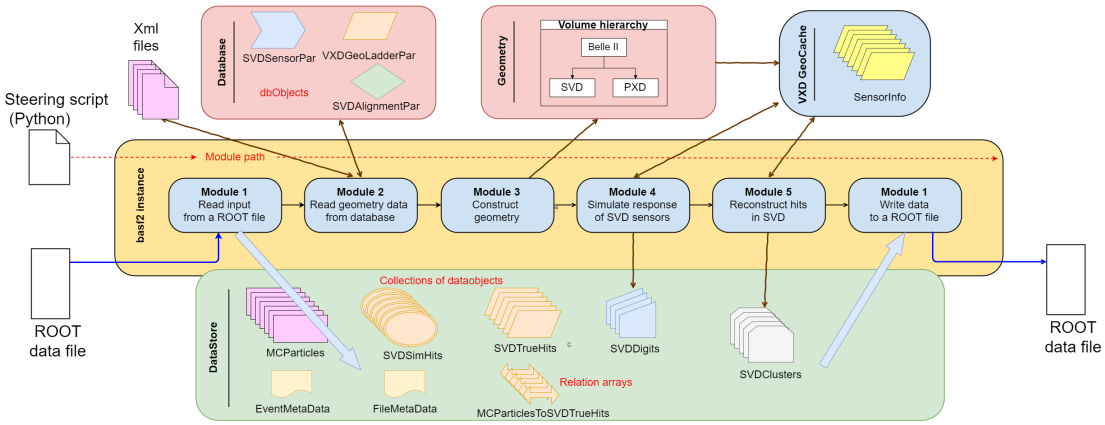
Dále, všechny moduly pracují nad stejným úložištěm dat. Jeden z modulů tak slouží jen k tomu, že ze souboru načte dosud nezpracovaná data. Další z nich

vytvoří objekty, reprezentující konkrétní zásahy v detektoru. Následuje rozpoznávání drah, fitování a analýza, popsané v předchozí kapitole. Nakonec přijde na řadu modul, který data uloží do nového souboru. Celá cesta se vykonává zvlášť pro každou událost (*event*), dokud nejsou všechna data vyhodnocena. Tento proces je ilustrován na obrázku 2.1.

2.2.1 Data

Společné úložiště zpracovaných dat se nazývá *DataStore* a přistupuje se k němu pomocí generických tříd `StoreObjPtr<T>` pro jednotlivé objekty a `StoreArray<T>` pro pole. Jako parametr `T` volíme typ objektů, které chceme získat. Mezi objekty z DataStore existují různé vztahy, například že jedné dráze (`Track`) odpovídá rekonstruovaná dráha (`RecoTrack`). Tyto vztahy jsou realizovány pomocí relací, což není nic jiného než oboustranné indexy. Relace je možné vytvořit mezi objekty, jejichž předkem je třída `RelationsObject`.

Dalším sdíleným zdrojem je databáze, obsahující například konfiguraci nebo popis geometrie detektoru. Databáze je umístěna na serveru, ze kterého se na začátku každého *runu* stáhnou data podle požadavků použitých modulů. Lokální úložiště se nazývá *DBStore*, příslušné třídy jsou `DBObjPtr<T>` a `DBArray<T>`.



Obrázek 2.1: Ukázka role jednotlivých modulů při zpracování dat. Dále vidíme, jak moduly přistupují do DataStore a do DBStore.

2.2.2 Společné vlastnosti modulů

Každý modul má podobnou základní strukturu, kterou dědí z třídy `Module`. Obecně funguje tak, že z existujících kolekcí dat v DataStore vytváří nové kolekce. Tuto třídu dále rozšiřuje třída `HistoModule`, ze které vychází všechny moduly pracující s histogramy. Ty z dat z DataStore vytváří histogramy, které se za běhu programu ukládají na stejné místo, odkud je *HistoManager* po skončení *runu* uloží na disk. Jelikož se v této práci zabýváme právě potomky `HistoModule`, ukážeme si některé funkce této třídy.

- Konstruktor – Zpravidla obsahuje funkce `setDescription` (nastaví popis modulu, který se zobrazí v nápovědě), `setPropertyFlags` (nastaví indikátory některých vlastností modulu (*flag*), například zda je schopný pracovat paralelně) a `addParam` (umožní nastavit danou privátní proměnnou modulu z prostředí Pythonu).
- `void initialize()` – Je zavolána právě jednou, a to před začátkem samotného zpracování dat. Slouží k iniciaizaci proměnných hodnotami ze skriptu a k zaregistrování přístupů k potřebným objektům v DataStore a DBStore. Mimo to by měla obsahovat makro `REG_HISTOGRAM`, které registruje modul u HistoManageru.
- `void defineHisto()` – Funguje podobně jako `initialize`, ale místo nastavení proměnných má vytvořit objekty histogramů.
- `void beginRun()` – Volá se před každým *runem*, což je soubor všech událostí z jednoho souvislého měření.
- `void event()` – Hlavní část modulu, slouží ke zpracování dat jedné události.
- `void endRun()` – Je zavolána po konci runu.
- `void terminate()` – Volá se při ukončování modulu, aby dealokovala používanou paměť a další systémové zdroje.

Specifickým rysem modulů, které jsou potomkem třídy `HistoModule` (a obsahují makro `REG_HISTOGRAM`), je, že za ně HistoManager spravuje paměť alokovanou pro jednotlivé histogramy. Není ji tedy třeba dealokovat ve funkci `terminate()` (naopak by to vedlo k chybám). Jak jsme již zmínili, obsah všech histogramů se také automaticky ukládá do souborů.

2.2.3 Moduly `TrackDQM` a `AlignDQM`

Výraz DQM je zkratkou pro *data quality monitoring*, neboli proces sledování kvality dat. DQM moduly obecně vytváří histogramy, které zobrazují různé aspekty kvality naměřených dat. Tyto histogramy sleduje obsluha experimentu, díky čemuž může posuzovat kvalitu dat už v průběhu runu nebo krátce po něm.

V této práci se zabýváme moduly `TrackDQM` a `AlignDQM`. Nejdříve se seznámíme s tím, jak fungují. Z hlediska kódu se liší pouze tím, že `AlignDQM` vytváří všechny histogramy, co `TrackDQM`, a k tomu ještě nějaké navíc. Následující popis je proto platný pro oba moduly.

Inicializace

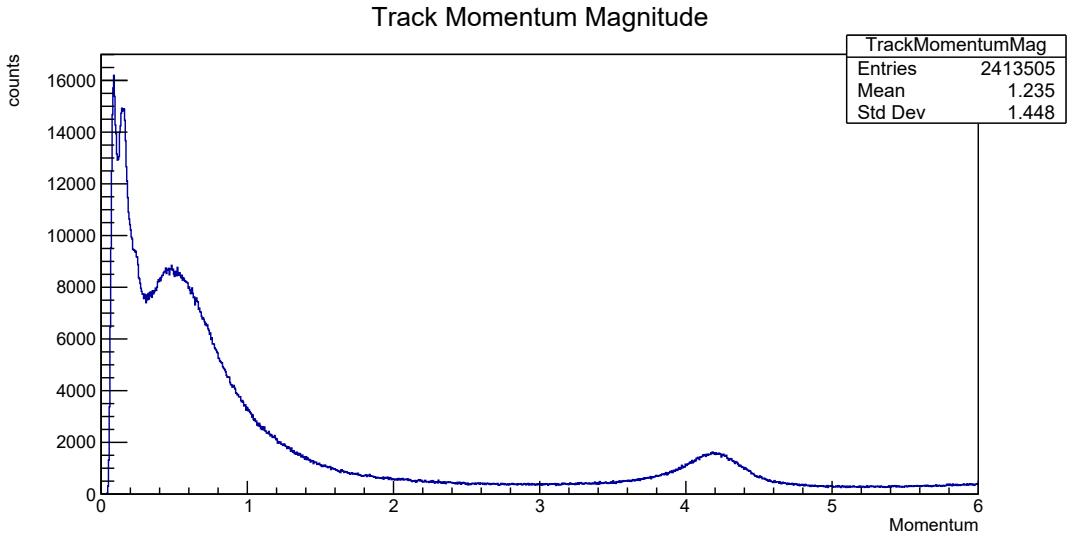
Parametry obou modulů jsou názvy kolekcí drah a rekonstruovaných drah, pod kterými je najdeme v DataStore. Při inicializaci na těchto kolekcích zavoláme metodu `isOptional`, která říká, že příslušná data jsou volitelným vstupem tohoto modulu (a tedy DataStore nemá vyvolat chybu, pokud je nenajde).

Následuje funkce `defineHisto`, ve které jsou definované všechny histogramy. Dále se tu vytvoří objekty třídy `TDirectory`, což jsou jakési vnitřní složky v souboru `TFile`, které jej logicky dělí na více částí. V `AlignDQM` jsou tři hlavní složky – první pro celková data, parametry drah a další veličiny, druhá pro data z každé vrstvy zvlášť a třetí pro každý senzor zvlášť. V `TrackDQM` je pouze první a část třetí.

Rovnou zmíníme i funkci `beginRun`, ve které je potřeba na každý histogram zavolat funkci `Reset`. Důvodem je to, že po skončení runu se histogram uloží a zůstanou v něm data, která je potřeba před dalším runem vynulovat.

Zpracování dat

Nyní se podrobněji podíváme na funkci `event`, ve které probíhá samotné zpracování dat. Ze všeho nejdříve je potřeba z DataStore získat validní vstupní data, tedy kolekce fitovaných a rekonstruovaných drah. Pokud nejsou k dispozici, modul oznámí chybu a dále nepokračuje. Následně procházíme dráhy jednu po druhé a získáváme z nich data, která zapisujeme do histogramů. Některé veličiny, které nás zajímají, se týkají přímo drah, například hybnost v jednotlivých souřadnicích nebo χ^2 a další parametry fitu. Pro ilustraci uvádíme histogram velikosti hybnosti na obrázku 2.2, vygenerovaný modulem `TrackDQM`.

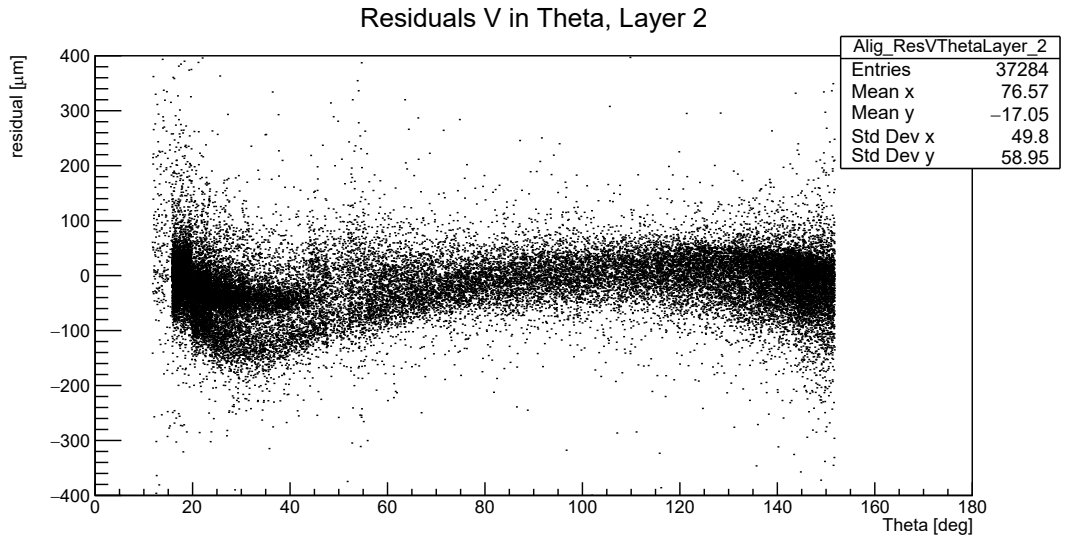


Obrázek 2.2: Histogram velikosti hybnosti ve standardních jednotkách. Data pochází z runu číslo 1 742.

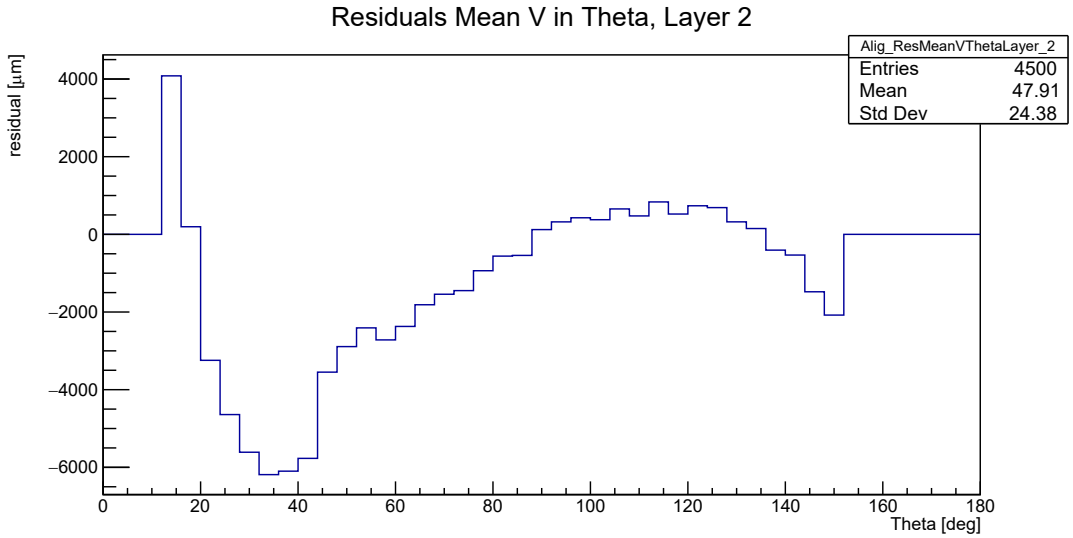
Pro každou dráhu ještě projdeme všechny její zásahy a spočítáme veličiny, které vyjadřují vlastnosti konkrétních zásahů, především se jedná o reziduály a souřadnice. Samotný výpočet je relativně přímočarý, na všechno používáme funkce definované v jiných částech frameworku.

Některé histogramy se plní až ve funkci `endRun`, protože potřebují kompletní data z celého runu. Vytváří se tak, že už hotový 2D histogram rozdělíme na proužky podél jedné osy, spočítáme průměrnou hodnotu v každém proužku a tu uložíme do příslušného sloupce v novém 1D histogramu. Ukázku 2D histogramu

můžeme vidět na obrázku 2.3, jemu odpovídající 1D histogram je na obrázku 2.4. Oba byly vytvořené modulem `AlignDQM`.



Obrázek 2.3: Histogram zásahů senzorů z druhé vrstvy podle reziduálů a globální sférické souřadnice θ. Data pochází z runu číslo 1 742.



Obrázek 2.4: Histogram, vzniklý rozdělením grafu na obrázku 2.3 na proužky rovnoběžné se svislou osou a následným spočítáním jejich průměru podle možných hodnot reziduálů.

2.3 Cíle této práce

Zatím jsme se věnovali popisu experimentu Belle II a frameworku basf2. Většina dosavadního textu pouze shrnuje současný stav věcí a vysvětluje některé technické podrobnosti kódu. V příštích kapitolách přejdeme od teorie k praxi a pokusíme se přispět něčím novým. Nyní si ještě uvedeme konkrétní cíle, kterých se v praktické části práce budeme snažit dosáhnout.

2.3.1 Refaktoring

Při vývoji softwaru se často stává, že zpočátku neznáme nejvhodnější řešení problému a nevíme, jak přesně naprogramovat to, co chceme. Přitom zkoušíme různé přístupy a teprve během psaní kódu se seznamujeme s funkcemi, které chceme použít. Občas se v průběhu času mění i samotné zadání práce a požadavky na výsledný program.

To vede k nutnosti pružně upravovat velké části kódu, což se většinou řeší kopírováním. Pozůstatkem rychlého vývoje jsou i matoucí názvy funkcí a proměnných, které v původním kontextu dávaly smysl, ale nyní už nikomu nic neříkají. Aniž si to uvědomujeme, ve výsledku píšeme složitý a nepřehledný kód, který obsahuje spousty duplicit. Nebezpečí vzniku špatného kódu je všudypřítomné a pokud nemáme od začátku jasnou představu, jak konkrétně danou funkcionalitu naprogramovat, dokážeme tomu jen stěží zabránit.

Smyslem refaktoringu je přepsat kód tak, aby se zvenku choval stejně, ale uvnitř byl přehlednější a snáze se udržoval a dále rozšiřoval. Přesně toho se budeme snažit dosáhnout u modulů `TrackDQM` a `AlignDQM`. Tento požadavek vychází z toho, že čas od času je potřeba například přidat nové histogramy nebo změnit ty stávající. V současném kódu je kvůli tomu potřeba provést úpravy na několika různých místech, což celý proces výrazně zpomaluje a zároveň to zvyšuje riziko vzniku chyb. Mimo to se pokusíme kód zpřehlednit a více zdokumentovat, co se v něm odehrává, což výrazně zkrátí čas potřebný k budoucím změnám.

Dále máme v plánu přidat některá drobná rozšíření, například možnost upravovat vlastnosti konkrétních histogramů z prostředí Pythonu pomocí parametrů modulů.

2.3.2 Sledování posunů half-shellů

Druhou částí zadání této práce bylo přidat do obou modulů histogramy zobrazující posuny half-shellů. Konkrétně se budeme zajímat o nevychýlené reziduály v globálních souřadnicích, protože změna jejich průměrné hodnoty vypovídá o změně alignmentu a tedy o pohybu v detektoru. Přitom rozdělíme zásahy do čtyř skupin podle toho, z jakého half-shellu (Yin, Yang, Pat a Mat) pochází. Pro každou z nich vytvoříme tři histogramy, ve kterých zobrazíme průměty reziduálů do směrů x, y a z.

Histogramy následně vyhodnotíme pro všechna experimentální data naměřená v nějakém období a výsledky statisticky zpracujeme. Díky tomu budeme moct vidět pohyby half-shellů v závislosti na čase.

3. Refaktoring

Vlastní práci jsme začali důkladným studiem kódu, který tvoří moduly `TrackDQM` a `AlignDQM`. Snažili jsme se porozumět nejen jednotlivým funkcionalitám, ale i tomu, jak jsou implementovány. Shrnutí takto získaných poznatků bude obsahem první části této kapitoly.

Dále jsme zkoumali kvalitu a organizaci kódu. Ve druhé sekci se seznámíme s nejvýraznějšími nedostatky, které jsme přitom objevili. Následovat bude část, kde se budeme zabývat jejich odstraněním, stejně jako dalšími způsoby, jak zlepšit současný kód. Na konci kapitoly pak zmíníme nové funkce, o které jsme moduly rozšířili.

3.1 Specifikace modulů

Nyní si popíšeme, jak přesně byly oba moduly původně implementovány. Při refaktoringu, kterým se zabýváme v dalších sekcích, jsme z tohoto popisu vycházeli, ale zároveň jsme mírně změnili chování modulů. Důvodem jsou především opravy chyb, které jsme v kódu našli. O kódu před refaktoringem budeme mluvit jako o současné verzi, o kódu po něm budeme mluvit jako o nové verzi.

V dalším textu primárně uvádíme kód z modulu `TrackDQM`, ale ten z `AlignDQM` je téměř stejný. Nicméně, v průběhu předchozího vývoje kódu byly objevovány chyby, z nichž některé byly následně opraveny jen v jednom z modulů. V takovém případě vyjdeme z toho modulu, kde jsou opraveny správně. Stejně tak vždy zvolíme modul, kde je nějaká funkcionalita navíc oproti tomu druhému.

Kód v ukázkách neodpovídá skutečnému kódu přesně znak po znaku, v zájmu zpřehlednění výkladu uvádíme zjednodušenou verzi, místy doplněnou o komentáře. Většina funkcí v modulu je velmi přímočarých, proto si je nebudeme podrobně popisovat a budeme se zabývat pouze funkcí `event`.

3.1.1 Funkce `event`

K ověření vstupních dat opět použijeme metodu `isOptional`, která mimo jiné vrací informaci o tom, jestli existují. V případě záporné odpovědi se ještě nemusí jednat o chybu, je klidně možné, že v této události žádné dráhy nejsou. Nicméně v tu chvíli nemáme co zpracovávat, takže skončíme. Následující kód ukazuje načtení a ověření kolekce drah, pro rekonstruované dráhy je situace obdobná, akorát je uložíme do proměnné `recoTracks`.

```
StoreArray<Track> tracks(m_TracksStoreArrayName);
if (!tracks.isOptional() || !tracks.getEntries())
      return;
```

Rovnou si můžeme všimnout, že ještě probíhá kontrola, zda kolekce nejsou prázdné. K tomu slouží metoda `getEntries`, která vrací jejich délku. V tomto případě to ale není zcela nezbytné, protože by se na to přišlo hned v další části, až bychom se přes ně pokusili iterovat.

Následuje `try` blok, kterému se budeme věnovat později. Libovolná výjimka je odchycena pomocí `catch (...)`. Toto není ideální postup, protože výjimky

většinou upozorňují na chybu v programu, kterou je potřeba odstranit. Nicméně u DQM modulů je zásadní, aby nikdy neskončily chybou, proto je tu použita tato konstrukce. V nové verzi se alespoň pokusíme vypsat, o jakou výjimku se jedná. Po `catch` bloku už se jenom v `TrackDQM` plní histogram s počtem chyb, ke kterým při zpracování dat došlo.

V `try` bloku se dějí tři věci – nejprve se vynulují proměnné, které zaznamenávají pořadí dráhy v kolekci a další podobné údaje. Následuje `for` cyklus, kterým projdeme všechny dráhy. Na konci `try` bloku jsou informace o počtu drah atd. zaznamenány do příslušných histogramů.

```
int iTrack = 0;
... // Deklarace dalších proměnných

for (const Track& track : tracks) {
        ... // Zpracování jedné dráhy
}

if (m_Tracks != nullptr)
        m_Tracks->Fill(iTrack);
... // Plnění dalších histogramů
```

Na tomto místě je potřeba zmínit, že před samotným plněním histogramů pomocí funkce `Fill` se kontroluje, zda příslušné objekty existují. V nové verzi to nahradíme tím, že na začátku funkce `event` zkontrolujeme, zda proběhla funkce `defineHisto`, čímž bude zaručeno, že všechny histogramy jsou definovány.

3.1.2 Zpracování jedné dráhy

Vraťme se k cyklu procházejícímu objekty typu `Track`. Nejdříve je potřeba najít příslušnou rekonstruovanou dráhu, k čemuž slouží funkce `getRelationsTo`. Ve skutečnosti získáme kolekci objektů typu `RecoTrack`, z nichž vybereme ten první. Podobným způsobem dostaneme kolekce objektů odpovídajících jednotlivým PXD, SVD a CDC zásahům a tím zjistíme jejich počty v dané dráze, které budeme potřebovat dále.

```
RelationVector<RecoTrack> recoTrack = track.getRelationsTo<RecoTrack>
        (m_RecoTracksStoreArrayName);
if (!recoTrack.size())
        continue;

RelationVector<PXDCluster> pxdClustersTrack = DataStore::getRelationsWithObj
        <PXDCluster>(recoTrack[0]);
int nPXD = (int)pxdClustersTrack.size();
... // Ostatní proměnné spočítáme podobně
```

Pro další zpracování je zásadní získat objekt třídy `TrackFitResult`, který obsahuje parametry fitu. K tomu použijeme metodu, která ze všech dostupných fitů vybere ten, který je hmotností částice nejblíže pionu.

```
TrackFitResult* tfr = track.getTrackFitResultWithClosestMass(Const::pion);
if (tfr == nullptr)
        continue;
```

V současné verzi se kontroluje, zda se vrátil nějaký objekt, ale to není potřeba, protože metoda sama zaručuje vrácení validního fitu. Z dosud získaných parame-

trů zkonstruujeme ladící hlášku, kterou následně vypíšeme.

Dále se naplní tři histogramy (konkrétně φ, θ a $\cos\theta$ momentu hybnosti dráhy)

```
float Phi = atan2(tfr->getMomentum().Py(), tfr->getMomentum().Px()) *
        TMath::RadToDeg();
m_MomPhi->Fill(Phi);
```

a pokračuje se ve zpracování fitu, uvozeném konstrukcí

```
if (recoTrack->wasFitSuccessful()) {
        if (!recoTrack->getTrackFitStatus())
                continue;
        ... // Vyhodnocení rekonstruované dráhy
}
```

za kterou se naplní zbylé histogramy. Všechny přitom čerpají data z objektu `TrackFitResult`, nebo z již předtím spočítaných parametrů dráhy. Toto je výsledkem vývoje kódu – původně se všechny histogramy, zmíněné v této sekci, plnily uvnitř podmínky testující úspěšnost fitu. Později přišel požadavek, aby se tři výše jmenované histogramy plnily vždy, a proto byly vyjmuty před podmínku. Následně se objevila potřeba plnit i ostatní histogramy vždy, a proto byly také vyjmuty z podmínky, ale až za ní.

Tato konstrukce je problematická z toho důvodu, že příkaz `continue` přejde rovnou k vyhodnocení další dráhy, čímž nejen ukončí tuto podmínku, ale přeskočí i všechno za ní. To by znamenalo, že by se histogramy za podmínkou plnily jindy, než ty před ní. Proto jsme v nové verzi přesunuli plnění všech histogramů před podmínku.

Nicméně funkce `wasFitSuccessful` sama testuje, jestli náhodou metoda `getTrackFitStatus` nevrací `nullptr`, čili to není nutné kontrolovat znovu. Proto jsme v nové verzi druhou podmínku odstranili.

V případě úspěšného fitu pokračujeme dále. Nejdřív naplníme histogramy, závislé na parametrech fitu (například χ^2), které získáme z objektu typu `TrackFitStatus`. Poté definujeme proměnné, které budeme potřebovat ve výpočtech dále. Nakonec pomocí `for` cyklu projdeme a vyhodnotíme zásahy z dráhy, které jsou reprezentovány třídou `RecoHitInformation`.

```
iHit = 0;

for (auto recoHitInfo : recoTrack[0]->getRecoHitInformations()) {
        ... // Vyhodnocení zásahu
        iHit++;
}
```

Význam proměnné `iHit` vysvětlíme později. Funkce `getRecoHitInformations` má parametr určující, jestli má zásahy seřadit. Jeho defaultní hodnota je `false`. V nové verzi je budeme chtít seřazené, proto použijeme následující kód.

```
for (auto recoHitInfo : recoTrack[0]->getRecoHitInformations(true)) {
        ... // Vyhodnocení zásahu
}
```

3.1.3 Zpracování jednoho zásahu

Nejdřív zkontrolujeme několik podmínek – zda objekt s informacemi o zásahu existuje, zda byl daný zásah použit ve fitu a zda se jedná o PXD nebo SVD senzor.

```
if (!recoHitInfo)
        continue;
if (!recoHitInfo->useInFit())
        continue;
if (!((recoHitInfo->getTrackingDetector() == RecoHitInformation::c_PXD) ||
        (recoHitInfo->getTrackingDetector() == RecoHitInformation::c_SVD)))
        continue;
```

Reziduál

Nyní je potřeba získat nevychýlený reziduál, což je vektor s jednou nebo dvěma souřadnicemi podle toho, jestli se jedná o SVD nebo PXD zásah. V původní verzi se postupovalo tak, že se vzal odkaz na `genfit::Track`[1], na kterém se zavolala metoda `getPointWithMeasurement` s argumentem udávajícím, kolikátý zásah nás zajímá. Tím jsme dostali `TrackPoint`, na kterém jsme následně zavolali funkci `getFitterInfo`, která vrátila informace o fitu. Pokud vše proběhlo správně, pomocí dalších metod jsme získali hledaný reziduál.

```
auto& genfitTrack = RecoTrackGenfitAccess::getGenfitTrack(*recoTrack[0]);

if (!genfitTrack.getPointWithMeasurement(iHit)->getFitterInfo())
        continue;

bool biased = false;
TVectorD resUnBias = genfitTrack.getPointWithMeasurement(iHit)
        ->getFitterInfo()->getResidual(0, biased).getState();
```

Tento přístup je problematický hned z několika důvodů. Předně, objekty typu `genfit::Track` bychom neměli vůbec používat, místo nich máme třídu `RecoTrack`. Dále, právě se nacházíme v cyklu, který iteruje přes všechny zásahy. Proč bychom měli k získání fitu používat veličinu `iHit`, která tak funguje jako druhý iterátor? Pokud už máme odkaz na informace o zásahu uložený v proměnné `recoHitInfo`, dávalo by větší smysl použít přímo jej.

Výsledkem byla velmi těžko odhalitelná chyba. V některých případech totiž nebyly oba seznamy stejně seřazené, což vedlo k tomu, že jsme dostali reziduál z jiného zásahu, než jaký jsme právě zpracovávali. Jak uvidíme dále, pokud jsme pomocí příkazu

```
if (recoHitInfo->getTrackingDetector() == RecoHitInformation::c_PXD)
```

zjistili, že se jedná o PXD zásah, přečetli jsme z něj obě souřadnice (u i v). Ale reziduál z SVD zásahu obsahoval pouze jednu souřadnici. Jelikož jsou data

[1] Tato třída pochází z knihovny Genfit2, která se v basf2 používá k fitování. Plný název včetně jmenného prostoru jsme použili proto, aby se nepletla se třídou `Track`, o které mluvíme v sekci Funkce `event`. Třída `genfit::track` není pro potřeby basf2 dostatečná, například neumožňuje vytvářet relace (viz sekce Data). Naopak `RecoTrack` už pochází čistě z basf2 a je jakousi nadstavbou na `genfit::Track`.

uskladněná v poli příslušné délky, druhá souřadnice se vzala z místa v paměti, ke kterému bychom vůbec neměli mít přístup. To shodou okolností nevyvolalo výjimku, místo toho se přečetla hodnota, která tam zůstala od posledního PXD reziduálu a proto ještě nebyla přepsána. Situaci navíc komplikovaly CDC zásahy, které sice normálně nevyhodnocujeme, ale tímto způsobem se jejich reziduály přimíchaly k těm z VXD.

Při simulacích srážek částic jsme zjistili, že tato chyba nastává pro přibližně 5% zásahů. U částic z kosmického záření pak byly chybné téměř všechny. V nové verzi kódu jednak nepoužijeme `genfit::Track`, a za druhé informace o fitu získáme přímo z objektu `recoHitInfo`.

```
if (!recoTrack[0]->getCreatedTrackPoint(recoHitInfo)->getFitterInfo())
        continue;

bool biased = false;
TVectorD resUnBias = recoTrack[0]->getCreatedTrackPoint(recoHitInfo)
        ->getFitterInfo()->getResidual(0, biased).getState();
```

Výpočty dalších veličin

Vyhodnocení SVD zásahu je složitější, proto jej uvádíme nejdříve. V sekci Stripový křemíkový detektor jsme zmínili, že jeden senzor vytvoří pro jeden zásah dva objekty, z nichž první má pouze souřadnici u a druhý pouze v. Proto postupujeme tak, že hledáme dva po sobě jdoucí zásahy, které pochází ze stejného senzoru, což zjistíme pomocí identifikačního čísla senzoru (ID). Parametry zásahu v příslušné souřadnici (pozici v lokálních souřadicích a reziduál) stejně jako ID si průběžně ukládáme. Jakmile zjistíme, že zpracováváme SVD zásah se stejným ID senzoru, jako měl ten předešlý, víme, že máme k dispozici všechny potřebné informace a můžeme pokračovat. Jinak se další vyhodnocování přeskočí a přejdeme rovnou na následující zásah.

```
IsSVDU = recoHitInfo->getRelatedTo<SVDCluster>()->isUCluster();
VxdID sensorID = recoHitInfo->getRelatedTo<SVDCluster>()->getSensorID();

if (IsSVDU) {
        ... // Výpočet pozice a reziduálu ve směru u

} else {
        ... // Výpočet pozice a reziduálu ve směru v

        if (sensorIDPrew == sensorID) {
                ... // Výpočty dalších veličin, plnění histogramů daty
        }
}
sensorIDPrew = sensorID;
```

V současném kódu to je tak, že se shodnost senzorů zjišťuje pouze u zásahů se souřadnicí v. To je z toho důvodu, že senzory jsou vždy v pořadí u, v směrem od místa, kde částice vznikají, čili se nepředpokládalo, že bude potřeba řešit dráhy, ve kterých by byly zásahy obráceně. Nicméně například částice z kosmického záření mohou cestovat i opačným směrem, takže má opodstatnění zabývat se zásahy v obou možných pořadích. V nové verzi proto porovnáváme senzory

v obou případech.

Výpočet pro SVD senzor ve směru u probíhá následovně. Z objektu třídy `SVDCluster` získáme pozici zásahu, ze které vytvoříme bod v lokálních souřadnicích. Ten převedeme do globálních pomocí funkce `pointToGlobal`. Dále spočítáme φ ve stupních a zjistíme reziduál, který převedeme na mikrometry.

```
posU = recoHitInfo->getRelatedTo<SVDCluster>()->getPosition();
TVector3 rLocal(posU, 0 , 0);
TVector3 ral = info.pointToGlobal(rLocal, true);
fPosSPU = ral.Phi() / TMath::Pi() * 180;
ResidUPlaneRHUnBias = resUnBias.GetMatrixArray()[0] *
       Unit::convertValueToUnit(1.0, "um");
```

Pro směr v postupujeme obdobně, akorát nás místo φ zajímá θ a příslušné lokální souřadnice jsou `rLocal(0, posV, 0)`.

U PXD senzorů se jak pozice v u, tak ve v spočítají zároveň, čili se rovnou počítá s oběma souřadnicemi `rLocal(posU, posV, 0)`. Tyto vektory se používají pouze pro výpočet parametrů φ a θ, což jsou sférické globální souřadnice. Pokud by ve všech senzorech byla osa v rovnoběžná s osou z a osa u by ležela v rovině xy, byly by oba přístupy ekvivalentní. Nicméně osy v a z nejsou u některých senzorů rovnoběžné, navíc senzory mohou být různě deformované a otočené. Proto současný výpočet φ a θ není pro SVD zásahy zcela přesný a v nové verzi k němu použijeme souřadnice u i v zároveň, tak jak je tomu u PXD.

Dále si všimněme dvou převodů ze standardních[2] jednotek. K tomu bychom měli používat třídu `Unit`, kde jsou všechny jednotky definované. Prvním argumentem metody `convertValueToUnit` by měla být samotná veličina, kterou chceme převést. Postup v ukázce výše sice v tomto případě funguje, ale například u logaritmických jednotek by tomu tak nebylo.

Nevýhodou této metody je ale to, že při převodu hledá v seznamu příslušnou jednotku, což není nejefektivnější. Proto v nové verzi použijeme přímo konstanty, definované třídou `Unit`. Příslušné řádky potom budou vypadat následovně.

```
fPosSPU = ral.Phi() / Unit::deg;
ResidUPlaneRHUnBias = resUnBias.GetMatrixArray()[0] / Unit::um;
```

3.2 Co bylo potřeba změnit

V předchozí sekci jsme sepsali jakousi specifikaci toho, jak mají oba moduly fungovat. Přitom jsme zmínili chyby, které jsme v kódu našli a které jsme ještě před refaktoringem opravili. Jinak jsme se ale přesně drželi této specifikace. Nyní se budeme zabývat nedostatky, které souvisí se samotnou strukturou kódu a které jsme odstranili v rámci refaktoringu.

[2]Programovací konvence frameworku basf2 definuje standardní jednotky centimetr, nanosekunda, radián, GeV, Kelvin, Tesla a elementární náboj [1]. Veličiny v jiných jednotkách by měly být zřetelně označeny.

3.2.1 Duplicitní kód

Při programování jakéhokoli alespoň trochu složitého projektu se nevyhneme tomu, že je nutné některé části kódu vykonávat opakovaně, ale pokaždé s malou změnou. Občas je zase potřeba zvolit jednu z několika cest, kudy se program za běhu vydá, které jsou si velmi podobné, ale přece se trochu liší. Pokud oba tyto případy řešíme kopírováním kódu z jedné části programu do druhé, vzniká duplicitní kód.

Jednou ze zásad psaní přehledného a srozumitelného kódu je, že by v něm každá funkcionalita měla být implementována na nejvýše jednom místě. Důvodem k tomuto přístupu je mnoho negativních důsledků, které duplicitní kód přináší. [13]

Zásadním problémem opakování je špatná rozšiřitelnost kódu. Pokud totiž chceme vylepšit funkcionalitu, která je nakopírovaná v několika částech programu, musíme tu stejnou změnu provést ve všech kopiích. To je nejen pracné a zdlouhavé, ale přináší to i velké riziko vzniku chyb. V rozsáhlém kódu snadno ztratíme přehled o tom, kolik kopií dané funkcionality existuje, a tak se lehce stane, že nezměníme všechny. V budoucnu můžeme chtít přidat další vylepšení, které bude závislé na existenci toho předešlého. Výsledné chyby se velmi špatně odhalují, protože i když nepochází z poslední změny v kódu, projeví se až po ní.

Tento problém je obzvláště nepříjemný v projektech, na kterých spolupracuje mnoho různých lidí. Často totiž vylepšení implementuje někdo jiný než autor původního kódu, což zvyšuje pravděpodobnost, že nebude nakopírováno všude. O to závažnější je, že duplicitní kód z přesně stejných důvodů brání snadnému opravování chyb.

Typickým způsobem odstranění duplicit je vytvoření funkce, která požadovanou sekvenci vykonává, a následné volání této funkce všude, kde je daná sekvence potřeba. Případné drobné rozdíly mezi jednotlivými případy je možné vyřešit pomocí podmínek ve funkci či pomocí dědičnosti objektů, se kterými se pracuje. Nevýhodou tohoto přístupu je, že každé volání funkce či vyhodnocování podmínky stojí nějaký výpočetní výkon. Pokud zpracováváme rozsáhlá fyzikální data, hraje každé zpomalení roli. Proto je použití duplicitního kódu na místě v případě, kdyby jeho odstranění vedlo k nezanedbatelnému prodloužení výpočetní doby celého algoritmu.

Překladače jazyka `C++` však podporují tzv. *inlinování* funkcí. To znamená, že při kompilování se celé tělo metody vloží na místo jejího volání. Program se tak přeloží stejně, jako by se přeložil kód, ve kterém bychom obsah funkce prostě zkopírovali a vložili sami. Zda k inlinování dojde závisí na rozhodnutí překladače, ale obecně je to tím pravděpodobnější, čím je funkce kratší a jednodušší.

Zdlouhavé definice histogramů

Běžně se pod pojmem funkce chápe algoritmus, který na základě několika vstupních argumentů spočítá výstupní hodnotu. Duplicitní kód tohoto typu je snadné najít a následně nahradit právě voláním příslušné funkce. V některých případech ale není na první pohled zřejmé, že se část kódu zbytečně opakuje. Například při definici histogramů, viz následující ukázka.

```
name = str(format("Alig_TrackMomentumX"));
title = str(format("Track Momentum X"));
m_MomX = new TH1F(name.c_str(), title.c_str(), 2 * iMomRange, -fMomRange,
       fMomRange);
m_MomX->GetXaxis()->SetTitle("Momentum");
m_MomX->GetYaxis()->SetTitle("counts");
```

Každý histogram typu `TH1F` obsahuje jméno, titulek, rozlišení osy x, minimální a maximální hodnotu na ose x a popisky os x a y. Pro typ `TH2F` pak potřebujeme ještě parametry osy y a popisek osy z. Nicméně popisky os nejde zadat do konstruktoru, je potřeba je nastavit zvlášť. Když se nad tím zamyslíme, u každého histogramu je potřeba speciálně nastavit popisky os, čili je to vlastně také opakování kódu.

Můžeme si to představit jako funkci, která vezme výše uvedené parametry, vytvoří histogram a rovnou mu nastaví popisky os. Pokud implementujeme metodu, která vystihne tento algoritmus a vykoná jej obecně pro všechny histogramy, můžeme se snadno zbavit duplicitního kódu. Jedná se sice jen o pár řádků, které za každý histogram ušetříme, ale těch se v programu nacházejí desítky. Nemluvě o tom, že jakékoli změny teď stačí provést jen jednou v definici konkrétní funkce.

```
m_MomX = Create("Alig_TrackMomentumX", "Track Momentum X", 2 * iMomRange,
       -fMomRange, fMomRange, "Momentum", "counts");
```

Dále bychom chtěli zdůraznit, že tím jsme se zdaleka nezbavili všech duplicit. Povšimněme si, že u různých histogramů se teď opakuje stále stejná sekvence argumentů. Například část histogramů s reziduály má stejné rozlišení i rozsah osy x. To je také ve své podstatě opakování – akorát se místo algoritmu opakují argumenty funkce. Řešení tohoto problému se věnujeme později.

Nejasné závislosti

Při přidávání nového histogramu je potřeba jak nastavit jeho parametry, tak určit, kdy a jaká data se do něj mají přiřadit. Na tom není nic překvapivého, k prvnímu použijeme konstruktor `TH1F` a ke druhému funkci `Fill`. Co už ale zní trochu překvapivě, ve funkci `beginRun` musíme na každý histogram zavolat funkci `Reset`, viz sekce Inicializace. To jednak přináší potřebu opakovat jeden konkrétní proces pro všechny histogramy, a za druhé to rozšiřuje množinu akcí, které je třeba provést při přidání nového histogramu.

První zmiňovaný problém jsme už rozebrali výše, ale druhý představuje něco nového. V tomto případě hrozí, že si programátor neuvědomí, co všechno je nutné zařídit, aby nově přidaný histogram správně fungoval. Obecně bychom se proto měli snažit, aby se funkce, jejíž existence je implikována jinou části programu, vykonala automaticky. V tomto konkrétním případě se nabízí jednoduché řešení – ve funkci `beginRun` projdeme pomocí smyčky všechny histogramy a na každém zavoláme funkci `Reset`.

Samozřejmě je potřeba pomocí programovacího jazyka specifikovat, co znamenají všechny histogramy. K tomu využijeme to, že jsme v předešlé sekci implementovali funkci vytvářející jednotlivé objekty typu `TH1F*`. Nyní ji jen stačí rozšířit o řádek, ve kterém daný ukazatel na histogram před vrácením zařadíme do seznamu (v `C++` případě se nabízí `vector`). Projít všechny histogramy nyní znamená projít tento seznam.

3.2.2 Komplikovaný kód

Pojem programování běžně chápeme jako synonymum pro psaní kódu. Ve skutečnosti se však z naprosté většiny jedná o čtení [11]. Čteme v dokumentaci, čteme zdrojové kódy od ostatních programátorů a především čteme kód, který jsme napsali my sami. Z tohoto důvodu je zásadní udržovat kód přehledný a snadno čitelný. Přitom postupujeme od jednotlivých metod směrem k celému programu. Při pohledu na každou funkci by mělo být hned jasné k čemu slouží a co se v ní dělá. Pokud tomu tak není, nejspíš bychom měli danou metodu rozdělit na více jednodušších metod. Tento problém má obecně více různých příčin.

Délka metod

Jednoduchým důvodem složitosti nějakého úseku kódu může být to, že má příliš mnoho řádků. Názory na to, jaký počet řádků znamená, že už je funkce příliš dlouhá, se různí. Lepším měřítkem je to, že funkce by měla dělat právě jednu věc. Jinak řečeno, veškerou činnost funkce bychom měli být schopni vyjádřit jednou větou. Obecně se však můžeme shodnout na tom, že když už je metoda delší než obrazovka našeho počítače, máme problém.

Další užitečnou vlastností kratších a jednodušších funkcí je kromě větší přehlednosti také to, že snižují pravděpodobnost vzniku duplicitního kódu. Pokud už máme kód logicky rozdělený do funkcí, snadněji najdeme způsob, jak bychom mohli některou metodu zobecnit a znovu použít místo toho, abychom pouze zkopírovali její obsah.

V neposlední řadě, správně a jasně pojmenované funkce s vhodnými názvy argumentů často slouží jako nejlepší dokumentace toho, co daný blok kódu dělá. Nemusíme se tak soustředit na detaily, stačí si jen přečíst názvy funkcí a rovnou vidíme jakýsi stručný obsah celku. Tento přístup je ale dvojsečný, protože zavádějící název metody je spolehlivým zdrojem zmatení a těžko odhalitelných chyb. Proto je zásadní dodržovat obecně používané konvence a volit vhodná a jednoznačná jména funkcí.

Dobrou analogií k této problematice jsou fyzikální vzorce. Při řešení konkrétní úlohy nás většinou nezajímá odvození vztahu pro výpočet každé jedné veličiny. Místo toho se snažíme použít rovnice, o kterých víme, že platí (neboť jsme to už dříve dokázali), a zároveň na první pohled vidíme, co vyjadřují (protože používáme konzistentní a jednoznačné značení). Dále zavádíme různé pomocné konstanty a funkce, abychom nemuseli neustále opisovat to samé. Přitom ale dobře víme, jaké nepříjemnosti může způsobit nejasně definovaná veličina či nekonzistentní značení.

Přílišné zanořování

Vedle příliš dlouhého kódu je dalším zdrojem složitosti také přílišná hloubka zanoření. Po dvou `for` cyklech, třech `if` podmínkách a několika `try` blocích už nedokážeme bez delšího přemýšlení říct, kde se právě teď v kódu nacházíme. Názory na maximální hloubku zanoření v jedné funkci se stejně jako u délky kódu různí, ale použití tří a více různých úrovní už je ve většině případů moc. Opět nejde tolik o „šířku“ kódu, ale spíše o princip, že funkce má dělat jednu věc. U tří do

sebe zanořených podmínek či cyklů už tomu tak s největší pravděpodobností není.

Problémem příliš dlouhých metod je to, že si při jejich čtení musíme pamatovat, co všechno se v nich stalo. Naopak u příliš zanořeného kódu musíme najednou řešit všechny cesty, kterými se program může vydat. Jejich počet s každým dalším větvením narůstá, stejně jako složitost funkce, kterou se právě snažíme pochopit.

Nakonec si všimněme, že jednotlivé větve `if` podmínky jsou často relativně podobné, pouze s drobnými rozdíly. Například vyhodnocování měření bodu průletu senzoru částicí se liší podle toho, zda se jednalo o PXD nebo SVD senzor. Nicméně oba případy mají několik společných rysů – musíme spočítat souřadnice a následně je zanést do histogramů. V případě PXD detektoru je to poněkud přímočaré, u SVD je ale potřeba nejdřív určit jednu souřadnici a až potom druhou, a to postupně ze dvou různých zásahů. Další zpracování dat je už ale stejné, čili je možné pro něj vytvořit jednu společnou funkci. Na tomto příkladu vidíme, že i přílišná hloubka zanoření může souviset s duplicitním kódem.

Dokumentace

Říká se, že dobrý kód se dokumentuje sám. I když to není zcela pravda, měli bychom se alespoň snažit psát co nejsrozumitelnější kód. Výše jsme zmínili, jak je důležité volit výstižné a jednoznačné názvy funkcí. Stejně bychom měli postupovat i při pojmenovávání proměnných.

Při programování je nám samozřejmě jasné, co která proměnná znamená, ale upravujeme-li kód, který jsme napsali před nějakou dobou (popřípadě, který napsal někdo jiný), budou nám špatně zvolené názvy výrazně ztěžovat práci.

Typickým příkladem je slovo *index* ve významu pořadí v nějakém seznamu. Při vyhodnocení jednoho zásahu se používají následující řádky z `AlignDQM` (ačkoli ne hned takto za sebou).

```
int indexLayer = gTools->getLayerIndex(sensorID.getLayerNumber());
int index = gTools->getSensorIndex(sensorID);
int index = gTools->getLayerIndex(sensorID.getLayerNumber())
       - gTools->getFirstLayer();
```

V basf2 má každá vrstva své číslo od jedné do šesti. U senzorů to není tak snadné, proto jsou popsány pomocí svého ID. Občas ale potřebujeme projít všechny vrstvy nebo senzory, k čemuž slouží indexy. Zároveň existuje bijekce mezi číslem a indexem vrstvy, resp. mezi ID a indexem senzoru. Veličinu, představující pořadí vrstvy, dává smysl pojmenovat `indexLayer` (ačkoli trochu lépe by znělo `layerIndex`). Na druhou stranu, samotný název `index` pro pořadí senzoru je značně zavádějící – podle názvu by se mohlo jednat o pořadí čehokoli. Vhodným jménem by bylo například `sensorIndex`.

Navíc, o kousek dál se už veličina `index` používá pro něco ještě jiného, tentokrát pro relativní pořadí vrstvy vzhledem k té první při plnění histogramu korelací mezi vrstvami. Zde by se hodil název `correlationIndex`.

V různých částech stejného modulu můžeme najít i tyto řádky.

```
int iLay = gTools->getLayerIndex(layer.getLayerNumber());
int iLayer = id.getLayerNumber();
int iSensor = id.getSensorNumber();
```

Tyto názvy pochází z programovací konvence, kde se přímo doporučuje proměnné typu `int`, které identifikují nějaký objekt, pojmenovat jako tento objekt s příponou `i`. Problém je samozřejmě v tom, že v tomto případě není jasné, jestli chceme k identifikaci použít index nebo číslo vrstvy.

Dalším příkladem matoucích názvů jsou zkratky. Vzhledem k tomu, že dnešní vývojová prostředí nabízí řadu nástrojů, které za nás automaticky doplňují slova, je psaní celých názvů téměř stejně tak rychlé, jako používání zkratek. Podívejme se na následující sekvenci.

```
posU = recoHitInfo->getRelatedTo<PXDCluster>()->getU();
posV = recoHitInfo->getRelatedTo<PXDCluster>()->getV();
TVector3 rLocal(posU, posV, 0);
TVector3 ral = info.pointToGlobal(rLocal);
fPosSPU = ral.Phi() / TMath::Pi() * 180;
fPosSPV = ral.Theta() / TMath::Pi() * 180;
```

Ačkoli tento úsek nedělá nic složitého, nedokážeme na první pohled říct, co. Pokud dále v kódu najdeme samostatně proměnnou `fPosSPV`, nebude nám už vůbec jasné, co představuje.

Na prvních třech řádcích získáme `posU` a `posV`, což jsou u a v souřadnice zásahu, a vytvoříme z nich bod, pojmenovaný `rLocal`. Ten převedeme do globálních souřadnic, čímž dostaneme `ral`. Následně spočítáme globální sférické souřadnice φ a θ, které zároveň převedeme na stupně.

V nové verzi to zapíšeme takto.

```
m_position.SetX(recoHitInfo->getRelatedTo<PXDCluster>()->getU());
m_position.SetY(recoHitInfo->getRelatedTo<PXDCluster>()->getV());
TVector3 globalPosition = sensorInfo->pointToGlobal(m_position, true);
m_phi_deg = globalPosition.Phi() / Unit::deg;
m_theta_deg = globalPosition.Theta() / Unit::deg;
```

Některé věci se změnily, například to, že pozici už od začátku ukládáme ve formě vektoru. Zaměřme se ale pouze na názvy proměnných. Předpona `m_` vychází z programovací konvence a říká, že se jedná o instanční proměnnou. Přípona `_deg` (opět podle konvence) označuje veličinu v nestandardních jednotkách, konkrétně ve stupních.

Druhým nejlepším způsobem dokumentace, hned po srozumitelném kódu a vhodných názvech proměnných a funkcí, jsou komentáře. V basf2 se navíc používá program Doxygen, který automaticky vytváří dokumentaci na základě specifických značek v komentářích. Například, funkce `pointToGlobal` je definována takto.

```
/** Convert a point from local to global coordinates
 * @param local point in local coordinates
 * @param reco Use sensor position in reconstruction (true) or in nominal
      geometry (false)
 * @return point in global coordinates
 */
TVector3 pointToGlobal(const TVector3& local, bool reco = false) const;
```

Text na prvním řádku za `/**` se v dokumentaci zobrazí jako krátký popis u funkce. Značka `@param` označuje argument funkce, `@return` zase návratovou hodnotu.

Podle konvence musí být v basf2 komentář před každou třídou, funkcí i instanční proměnnou. Mimo to se doporučuje používat klasické komentáře především tam, kde je sice jasné co daný úsek kódu dělá, ale už není jasné proč.

Nevýhodou komentářů je totiž to, že nijak nekontrolujeme, zda jsou správně. Je-li kód špatně, skončí to v lepším případě překladovou chybou, v horším případě chybou za běhu programu. Je-li špatně komentář, nemáme na to jak přijít, pokud si toho náhodou někdo nevšimne. Přitom je snadné změnit část kódu a zapomenout upravit příslušný komentář.

Proto bychom se při dokumentaci neměli spoléhat na to, že pomocí komentářů zpřehledníme jinak nesrozumitelný kód. Lepší je psát jasný a dobře čitelný kód a komentovat to, co jinak vyjádřit nedokážeme – například proč jsme zvolili takový přístup, jaký jsme zvolili.

3.3 Jak jsme postupovali

Nyní si rozebereme řešení konkrétních problémů, se kterými jsme se při refaktoringu setkali. Některými jsme se již zabývali v podkapitole Co bylo potřeba změnit, další zde uvádíme poprvé. Smyslem následující odstavců ale není jen shrnout, k čemu jsme přitom došli a jak výsledný kód vypadá. Podíváme se totiž i na samotný proces, který k tomu vedl, a projdeme si rozhodnutí, která jsme museli učinit.

3.3.1 Rozdělení do více kratších metod

První, na co jsme se zaměřili, byla funkce `event`, která se spouští pro každou událost a zodpovídá za většinu výpočtů při plnění histogramů. Přidání téměř jakéhokoli nového histogramu vyžaduje změnit funkci `event`, a proto je pro budoucí rozšiřitelnost kódu zásadní, aby tato funkce byla přehledná a srozumitelná. V tuto chvíli však měla přes 200 řádků na délku a až 9 vrstev zanoření. Základní myšlenka byla jednoduchá – rozdělíme funkci na několik kratších a přehlednějších. Ideálním místem na rozdělení byly právě jednotlivé podmínky, které většinou kontrolovaly, zda byla předchozí operace úspěšná a zda je možné pokračovat dál. Potom se ještě nabízely cykly, které procházeli všechny dráhy a následně všechny zásahy.

Výhodou kratších funkcí by kromě přehlednosti bylo i to, že v případě potřeby změnit nějakou část výpočtu nám bude stačit upravit pouze jednu funkci. Můžeme pak vytvořit nový modul, který všechno zdědí z původního modulu, ale tuto jednu konkrétní funkci přepíše. Tomuto aspektu se věnujeme v další sekci.

Rozdělení na metody má však jednu zásadní nevýhodu – v průběhu algoritmu se alokuje spousta lokálních proměnných, které jsou potřeba během celého výpočtu. Jejich životnost končí spolu s funkcí, ve které byly alokovány. Toto chování je žádoucí v případě, že máme pouze jednu funkci na celý algoritmus, ale jakmile jej rozdělíme do více metod, máme problém. V zásadě existují dva způsoby, jak tyto proměnné předat dál – buď jako argumenty funkce, nebo jako globální (resp. instanční) proměnné.

Při zpracování fyzikálních dat se může objevit spousta různých veličin a dalších objektů. Předávání hodnot pomocí argumentů by však bylo nejen zdlouhavé

a pracné, ale přímo by to odporovalo principu nahraditelnosti jednotlivých metod, který jsme zmínili dříve. V případě, že bychom například na začátek algoritmu přidali novou proměnnou, bylo by ji nutné následně přidat do argumentů všech následujících funkcí. Nestačilo by tak změnit jen jednu metodu, museli bychom změnit všechny. Proto jsme se rozhodli předávat hodnoty pomocí instančních proměnných.

Třída `EventProcessor`

Instanční proměnné obecně vyjadřují stav daného objektu. Když se nad tím zamyslíme, zjistíme, že nedává smysl, abychom proměnné potřebné pro výpočty několika funkcí zvolili za instanční proměnné celého modulu, protože nereprezentují stav dané instance, ale pouze jednoho jejího výpočtu. Řešením tohoto problému bylo vytvoření třídy pouze pro tento konkrétní výpočet, kterou jsme nazvali `EventProcessor`. Ve funkci `event` potom pomocí konstruktoru, kterému předáme vstupní data, vytvoříme objekt třídy `EventProcessor`, na kterém následně zavoláme metodu `Run`.

Všechny další funkce z třídy už jsou neveřejné a tak je celý algoritmus odstíněn od původního modulu. Samozřejmě, modul stále obsahuje ukazatele na všechny histogramy, do kterých je nutné přiřadit data. Možným řešením by bylo například mezi argumenty konstruktoru předat buď tyto odkazy, nebo celý seznam, který jsme už vytvořili dříve. Histogramů jsou ale desítky a vyhledávání v seznamu by bylo výrazně pomalejší, než celý zbytek výpočtu. Histogramy jsme proto rozdělili do skupin podle toho, které konkrétní proměnné potřebovaly, a pro každou skupinu jsme na modulu vytvořili specifickou veřejnou funkci, které jsme v argumentech předali hodnoty daných proměnných. Nyní už jen stačí, aby si `EventProcessor` pamatoval ukazatel na modul, na kterém zavolá příslušné metody.

Příkladem takové funkce je `FillMomentumCoordinates`, která plní histogramy s hybností částic v různých směrech.

```
void FillMomentumCoordinates(const TrackFitResult* tfr)
{
  m_MomX->Fill(tfr->getMomentum().Px());
  m_MomY->Fill(tfr->getMomentum().Py());
  m_MomZ->Fill(tfr->getMomentum().Pz());
  m_Mom->Fill(tfr->getMomentum().Mag());
}
```

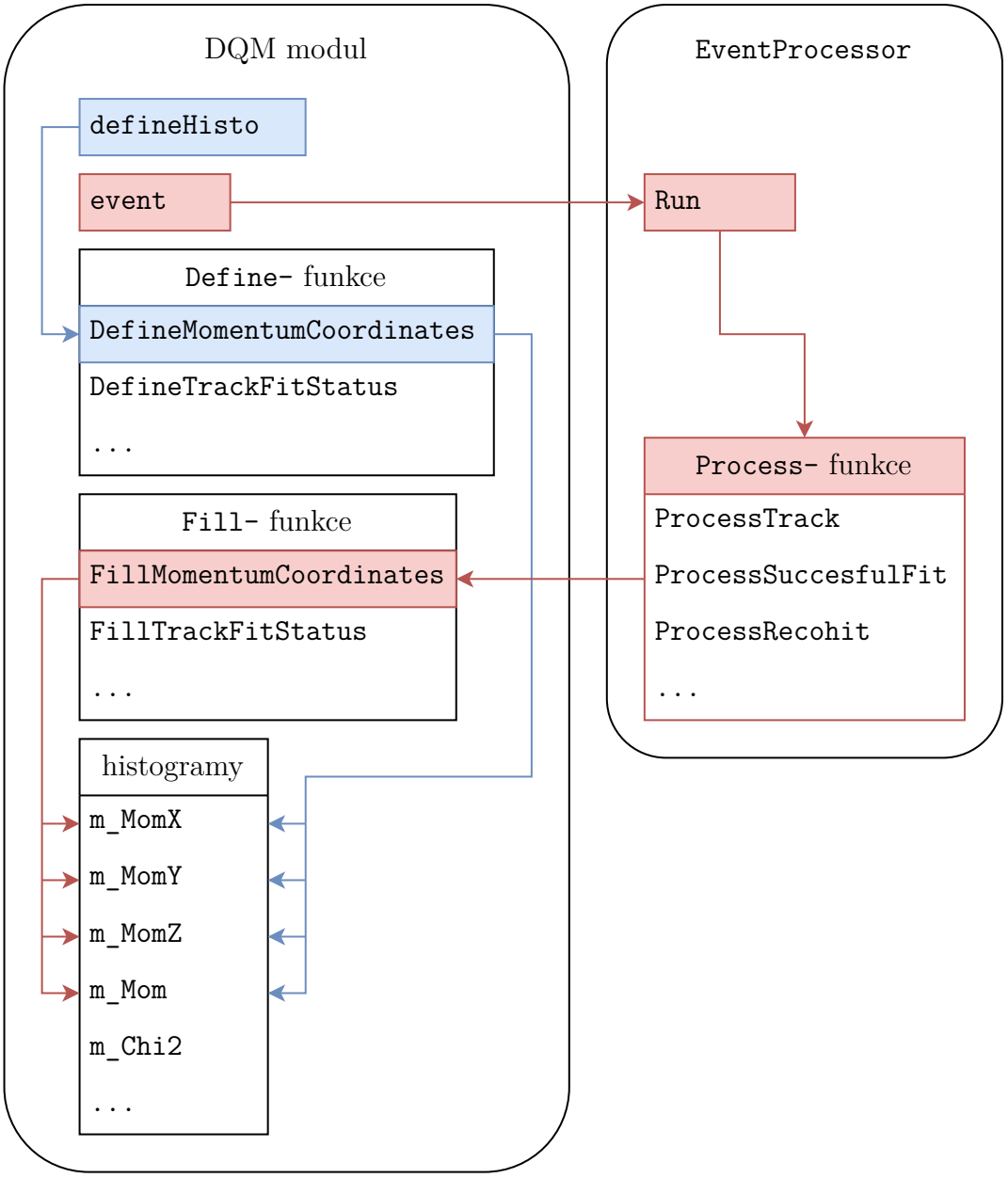
Tuto funkci následně zavolá `EventProcessor` pomocí ukazatele na modul.

```
m_histoModule->FillMomentumCoordinates(trackFitResult);
```

Celý proces, který jsme právě popsali, shrnuje schéma na obrázku 3.1.

3.3.2 Sjednocení modulů

Jak jsme již zmínili dříve, moduly byly ve skutečnosti dva, a to velmi podobné. Až na drobné detaily byl `TrackDQM`, co se rozsahu histogramů týká, podmnožinou `AlignDQM`. Algoritmus výpočtu funkce `event` byl u obou téměř stejný, samozřejmě s tím rozdílem, že v `AlignDQM` se plnilo více histogramů. I v ostatních aspektech



Obrázek 3.1: Základní struktura DQM modulu a pomocné třídy `EventProcessor`. Červené šipky ukazují průběh zpracování jedné události, od zavolání funkce `event` na modulu až po plnění histogramů. Modré šipky zobrazují, jak jsou objekty histogramů vytvořeny.

se ale oba moduly velmi podobaly, takže nás napadlo použít dědičnosti. Původně oba moduly dědily z třídy `HistoModule`, která zase dědila z `Module` (viz schéma na obrázku 3.2). Nyní jsme pro ně vytvořili společného předka `DQMHistoModuleBase`, který teprve dědil z `HistoModule`. Obdobně jsme `EventProcessor` nahradili třídami `TrackDQMEventProcessor` a `AlignDQMEventProcessor`, které obě dědí z `DQMEventProcessorBase`. Výsledné schéma můžeme vidět na obrázku 3.3.

Při tomto postupu jsme naplno využili výhod virtuálních funkcí. Jak jednotlivé kroky výpočtu, tak metody pro plnění a definice histogramů jsou virtuální, čili je snadné vytvořit nový modul, který se bude v něčem odlišovat od těch současných. Jednoduše zdědí vše od toho, kterému se nejvíce podobá, a přepíše jen ty

funkce, u kterých to bude potřeba. Tím jsme se zároveň zbavili naprosté většiny duplicitního kódu, nyní je všechno napsáno pouze na jednom místě, ze kterého se to dědí dál.

Díky virtualitě ale často ani nemusíme měnit přímo výpočet. Jak `TrackDQMEventProcessor`, tak `AlignDQMEventProcessor` totiž dědí ukazatel na modul, který je sice uložen v proměnné typu `DQMHistoModuleBase`, ale uvnitř si pamatuje svoji skutečnou třídu. Pokud chceme přidat histogram, který nepotřebuje žádné nové proměnné, stačí jen změnit funkci na modulu, která histogramy plní. Protože bude virtuální, v příslušném procesoru se zavolá nová verze dané funkce, kterou jsme právě implementovali. Tento příklad mimo jiné ukazuje výhody oddělení výpočetní části modulu, neboli algoritmu, od úložiště výsledných dat, tedy histogramů.

Oba původní moduly si byly tak podobné, že nynější `AlignDQMEventProcessor` se od svého předka liší v pouze jedné funkci, zatímco `TrackDQMEventProcessor` se neliší dokonce v žádné. Největší rozdíly mezi samotnými moduly jsou při definicích histogramů, na které se podíváme nyní.

3.3.3 Zjednodušení definic histogramů

Duplicitní kód v definicích histogramů jsme už zmínili v sekci Zdlouhavé definice histogramů. K tomuto tématu ještě poznamenejme, že některé histogramy jsou ve formě polí, například podle počtu vrstev nebo senzorů. Ačkoli jsou generovány najednou pomocí cyklů, není snadné vytvořit pro to obecnou funkci, protože některé jejich parametry (například jména a popisky) závisí na tom, o jakou vrstvu nebo jaký senzor se konkrétně jedná. Možným řešením by bylo, kdybychom dané funkci jako argument předali delegáta na jinou funkci, která by z identifikátoru vrstvy či senzoru určila název resp. popis histogramu.

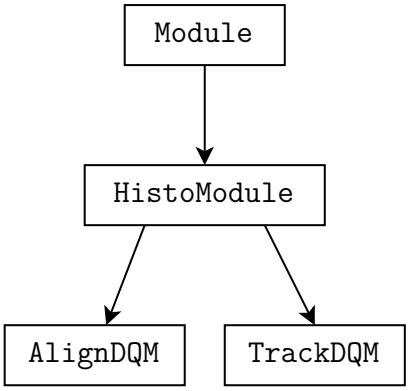
Ukázalo se však, že jednodušší bude použít knihovnu Boost, která obsahuje objekty typu `format`, což jsou vlastně jakési šablony pro textové řetězce. Místo delegátů tak stačí předat tyto šablony. To sice není zcela obecné, ale funguje to pro naprostou většinu histogramů. Jedinou výjimkou jsou histogramy korelací mezi vrstvami, ale těch je velmi málo. Vytvořili jsme funkce `CreateLayers` a `CreateSensors`, které mají podobné argumenty jako `Create` s tím rozdílem, že jméno a popis jsou typu `format` místo `string`.

```
m_TRClusterHitmap = CreateLayers(format("TRClusterHitmapLayer%1%"),
       format("Cluster Hitmap for layer %1%"), ... );
```

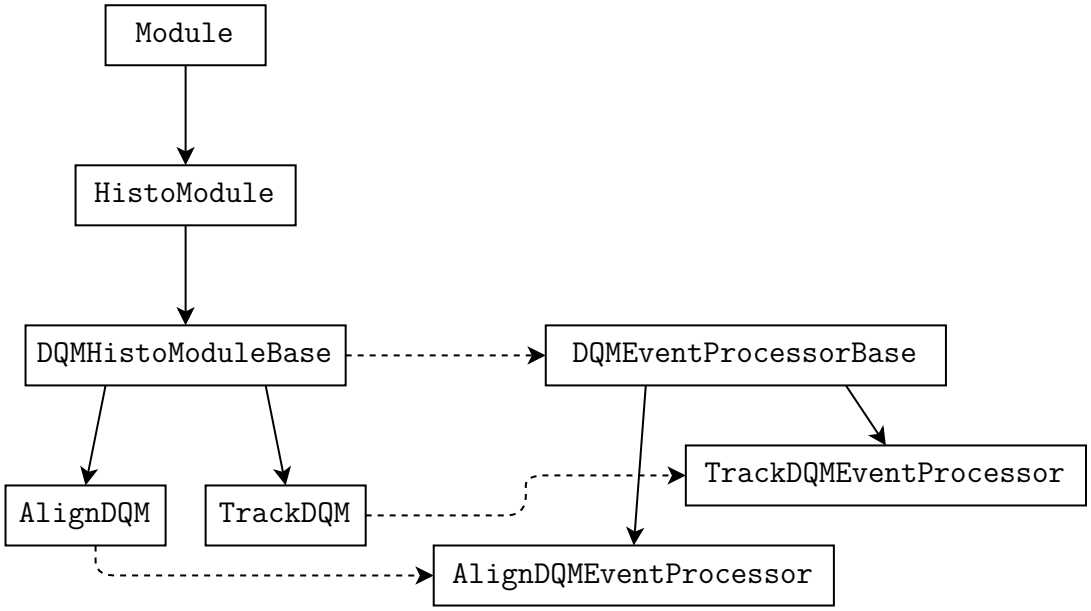
Ve výsledném histogramu se sekvence `%1%` nahradí číslem vrstvy. U funkce `CreateSensors` by se nahradila například výrazem `1-2-3`, což znamená 3. senzor ve 2. ladderu z 1. vrstvy.

Třída `THFFactory`

I když se nám podařilo zredukovat značné množství opakujícího se kódu, pořád bylo co zlepšovat. Především bylo stále nutné zadávat všechny parametry každého histogramu, ačkoli se často opakovaly. Proto jsme se snažili vytvořit konstrukci, která by nám umožnila jedním příkazem vyjádřit například: „rozsah osy x ve všech následujících histogramech bude $\langle 0,3 \rangle$.“ Opět jsme k tomu mohli použít instanční proměnné, ale stejně jako v případě proměnných při výpočtu funkce



Obrázek 3.2: Původní struktura dědičnosti tříd.



Obrázek 3.3: Struktura dědičnosti tříd po refaktoringu. Přerušované čáry naznačují, který DQM modul používá který `EventProcessor`.

`event` by ani nyní nedávalo smysl, aby byly součástí modulu jako takového. Mohli jsme k tomu vytvořit speciální třídu, něco jako `EventProcessor`, ale tím bychom si moc nepomohli, protože bychom stále potřebovali způsob, jak nově vytvořené histogramy předat původnímu modulu. To by se sice dalo zařídit pomocí `friend` tříd, ale to by zase bylo (vzhledem k vzájemné dědičnosti tříd a modulů) velmi nepřehledné.

Proto jsme se rozhodli pro vytvoření jakési továrny na histogramy. Základní myšlenka je taková, že vytvoříme objekt třídy `THFFactory`, kterému bude možné pomocí metod přiřadit hodnoty jednotlivých parametrů. Potom na něm zavoláme funkci, které už bude stačit předat pouze ty parametry, které jsme dosud nenastavili. Tato funkce už jen vytvoří požadovaný histogram. Velmi by se hodilo, pokud by jazyk `C++` podporoval pojmenované argumenty, ale tak tomu bohužel není. Proto jsme museli použít takzvaný *named parameter idiom*.

To stručně řečeno funguje tak, že pro každý parametr zavedeme funkci, jejíž vstupem bude hodnota parametru a výstupem bude objekt `THFFactory`. Díky tomu můžeme zřetězit více těchto funkcí za sebou a až na konci zavoláme vlastní

metodu na vytvoření histogramu, což se bude efektivně chovat stejně jako pojmenované parametry. Továrna tak pro každý potřebný parametr obsahuje dvě funkce – jedna nese jeho název (například `nbinsx`), druhá ještě obsahuje slovo `Default` (tedy `nbinsxDefault`). První jmenovaná nastaví hodnotu parametru pouze jednorázově, do prvního zavolání funkce na vytvoření histogramu. Druhá pak nastaví danou hodnotu na neomezenou dobu. Slouží k nastavení nějaké vlastnosti společné pro více následujících histogramů, zatímco jednorázový způsob má při vyhodnocení přednost a umožňuje tak učinit výjimku (nebo doplnit hodnotu, kterou jsme dosud nenastavili).

V obou modulech se používají dva typy histogramů, a to `TH1F` a `TH2F`. Jelikož 2D histogramy jsou co se parametrů týká pouze rozšířením těch 1D, mohli bychom mít pro každý typ samostatnou továrnu, přičemž jedna by dědila od druhé. To by ale bylo nepraktické z toho důvodu, že je často výhodné nastavit hodnotu určitého parametru pro všechny následující histogramy bez ohledu na to, jakého typu jsou. Proto jsme implementovali jen jednu továrnu se dvěma funkcemi – `CreateTH1F` a `CreateTH2F`.

Zároveň jsme chtěli mít možnost použít `THFFactory` i pro definování polí histogramů pro vrstvy a senzory. K tomu bychom ale potřebovali umět nastavit parametry `name` a `title` jako objekty typu `format`. To by samozřejmě výrazně zkomplikovalo dosud používané nastavení pomocí typu `string`. Proto jsme se rozhodli, že `name` a `title` budou přímo argumenty funkcí `CreateTH1F` a `CreateTH2F` a nepůjde je nastavit jako ostatní parametry. To ale dává smysl, protože jméno a titulek každého histogramu jsou unikátní, čili bychom je stejně museli pokaždé zadávat znovu. Nyní jsme mohli přidat funkce `CreateLayersTH1F` a `CreateSensorsTH1F` (obdobně pro `TH2F`), které mají také argumenty `name` a `title`, ale typu `format`.

Továrna nevytváří histogramy přímo, místo toho volá k tomu určené veřejné funkce modulu, které mimoto ještě zařídí například přidání nově vytvořeného histogramu na seznam, zmíněný v sekci Nejasné závislosti. Z tohoto důvodu je v konstruktoru továrny nutné předat ukazatel na modul.

Použití `THFFactory` v praxi může vypadat například takto.

```
auto factory = THFFactory(this).nbinsxDefault(2 * iMomRange)
       .xlowDefault(-fMomRange).xupDefault(fMomRange)
       .xTitleDefault("Momentum").yTitleDefault("counts");
m_MomX = factory.CreateTH1F("TrackMomentumX", "Track Momentum X");
m_MomY = factory.CreateTH1F("TrackMomentumY", "Track Momentum Y");
m_MomZ = factory.CreateTH1F("TrackMomentumZ", "Track Momentum Z");
m_Mom = factory.xlow(.0).CreateTH1F("TrackMomentumMag",
       "Track Momentum Magnitude");
```

Původní způsob definice histogramu `m_MomX` jsme popsali v sekci Zdlouhavé definice histogramů.

Třída `THFAxis`

Dále jsme si všimli, že některé parametry se často mění společně. Konkrétně rozlišení, rozsah a popisek osy závisí především na tom, jako veličinu chceme v grafu zobrazit. Proto jsme vytvořili třídu `THFAxis`, která obsahuje hodnoty

`nbins`, `low`, `up` a `title`. Do `THFFactory` jsme přidali funkce `xAxis` a `xAxisDefault`, které umožňují jednorázově nebo trvale nastavit tyto čtyři parametry osy x podle hodnot, které jsme uložili do daného objektu `THFAxis`. U osy y jsme postupovali obdobně.

Také jsme přidali konstruktor, který jako argument přijme jinou `THFAxis` a zkopíruje z ní všechny parametry, které je pak možné dále přenastavit. Důvodem bylo to, že mnohé osy se lišily v pouze jednom parametru (například v popisku, kde v jednom případě byla veličina u a ve druhém v).

Předchozí ukázku definice histogramů s hybností můžeme snadno upravit, aby pracovala přímo s `THFAxis`, stačí jen přepsat první řádek.

```
auto momentum = THFAxis(2 * iMomRange, -fMomRange, fMomRange, "Momentum");
auto factory = THFFactory(this).xAxisDefault(momentum)
       .yTitleDefault("counts");
```

Hlavní výhoda `THFAxis` se projeví tehdy, když máme několik různých veličin a chceme je vykreslit proti sobě. Příkladem jsou parametry dráhy, jako třeba φ, d_0, z_0, ω a další, mezi kterými děláme kartézský součin.

Nejdřív si definujeme jednotlivé osy a továrnu.

```
auto phi = THFAxis(iPhiRange, -fPhiRange, fPhiRange, "#phi [deg]");
auto D0 = THFAxis(iD0Range, -fD0Range, fD0Range, "d0 [cm]");
auto Z0 = THFAxis(iZ0Range, -fZ0Range, fZ0Range, "z0 [cm]");
auto omega = THFAxis(iOmegaRange, -fOmegaRange, fOmegaRange, "Omega");
... // Osy pro další veličiny
auto factory = THFFactory(this);
factory.zTitleDefault("Arb. Units");
```

Nyní už snadno vytvoříme příslušné histogramy.

```
m_PhiD0 = factory.xAxis(phi).yAxis(D0).CreateTH2F("PhiD0", "...");
m_PhiZ0 = factory.xAxis(phi).yAxis(Z0).CreateTH2F("PhiZ0", "...");
m_PhiOmega = factory.xAxis(phi).yAxis(omega).CreateTH2F("PhiOmega", "...");
m_D0Z0 = factory.xAxis(D0).yAxis(Z0).CreateTH2F("D0Z0", "...");
m_D0Omega = factory.xAxis(D0).yAxis(omega).CreateTH2F("D0Omega", "...");
... // Histogramy pro další kombinace veličin
```

3.4 Rozšíření o nové funkce

Jedním z nedostatků obou modulů je to, že i kvůli velmi malým změnám je nutné upravit a následně znovu zkompilovat `C++` kód. Zařídit, aby bylo možné například přidávat nové histogramy nebo měnit funkce na plnění těch stávajících bez úprav `C++` kódu, by bylo téměř nemožné a zároveň neefektivní. Na druhou stranu, některé parametry, například rozsahy, rozlišení nebo popisky os histogramů, nemají na průběh výpočtu žádný vliv. Funkcionalita, umožňující snadno měnit tyto veličiny, by se rozhodně hodila, minimálně k testovacím účelům.

Proto jsme přidali nový parametr modulu, nazvaný `histogramParameterChanges`. Jeho typem je `vector<tuple<string, string, string>>`, neboli seznam trojic textových řetězců. Prvním z nich je název histogramu, který chceme upravit. Druhý pak označuje veličinu a třetí hodnotu, kterou do ní chceme přiřadit. Například výraz `["TrackMomentumY", "xlow", "-60"]` znamená, že dolní hranice osy x v histogramu se jménem `TrackMomentumY` má být -60.

Dále jsme vytvořili funkci, která na základě těchto tří řetězců provede potřebnou změnu, popřípadě ohlásí, že není možná (například v případě neexistujícího názvu histogramu či veličiny). Nyní už jen stačí po definování všech histogramů projít všechny trojice a na každou zavolat tuto funkci.

Nalezení histogramu není nijak složité, vzhledem k tomu, že máme k dispozici seznam, zmíněný v sekci Nejasné závislosti. Stejně tak při hledání veličiny postupně vyzkoušíme všechny možnosti. To se může zdát neefektivní, ale vzhledem k tomu, že tento proces provádíme pouze jednou během běhu modulu, jeho časová náročnost je zanedbatelná. Nakonec, v některých případech ještě musíme převést `string` na hodnotu typu `int` nebo `double`, k čemuž použijeme předdefinované funkce jazyka `C++`.

V závěru této kapitoly si shrneme, co všechno jsme v rámci refaktoringu dělali. Prvním krokem bylo porozumění kódu z modulů `AlignDQM` a `TrackDQM` a vytvoření specifikace toho, jak mají fungovat. Přitom jsme si všimli několika chyb, které jsme rovnou opravili. Následně jsme přepsali kód obou modulů tak, aby se z vnějšku choval stejně, ale přitom byl přehlednější, srozumitelnější a snadněji rozšiřitelný. Snažili jsme se především o dvě věci – o odstranění duplicit v kódu (jak v rámci modulů, tak mezi nimi navzájem) a o snížení složitosti kódu rozdělením do více jednodušších celků. Nakonec jsme do modulů přidali parametr, umožňující upravovat některé vlastnosti histogramů z prostředí Pythonu.

V příští kapitole se budeme zabývat druhým praktickým úkolem této práce, a sice vytvořením a vyhodnocením histogramů, zobrazujících vzájemné pohyby větších částí detektoru.

4. Sledování pohybů half-shellů

4.1 Implementace

4.1.1 Úpravy DQM modulů

Prvním krokem bylo rozšířit moduly `TrackDQM` a `AlignDQM` o příslušné histogramy. Díky refaktoringu stačilo téměř všechen potřebný kód napsat pouze jednou v `-Base` třídách.

Do třídy `DQMHistoModuleBase` jsme přidali metodu `DefineHalfShellsVXD`, kterou následně voláme z funkce `defineHisto` na obou modulech. Dále jsme vytvořili metody `FillHalfShellsPXD` a `FillHalfShellsSVD`, kterým v argumentech předáme reziduál v globálních souřadnicích a informaci, o který half-shell se jedná.

```
void DQMHistoModuleBase::FillHalfShellsPXD(TVector3 globalResidual_um,
        bool isNotYang)
{
        if (isNotYang) {
                m_UBResidualsPXDX_Yin->Fill(globalResidual_um.x());
                m_UBResidualsPXDY_Yin->Fill(globalResidual_um.y());
                m_UBResidualsPXDZ_Yin->Fill(globalResidual_um.z());
        } else {
                m_UBResidualsPXDX_Yang->Fill(globalResidual_um.x());
                m_UBResidualsPXDY_Yang->Fill(globalResidual_um.y());
                m_UBResidualsPXDZ_Yang->Fill(globalResidual_um.z());
        }
}
```

Reziduál se používá i v jiných histogramech, čili jsme jej ve třídě `DQMEventProcessorBase` spočítali už dříve. Akorát ho nyní musíme převést do globálních souřadnic, k čemuž slouží funkce `vectorToGlobal`. Do třídy `DQMEventProcessorBase` jsme připsali následující řádky – převod reziduálu do metody, která počítá společné proměnné pro PXD a SVD,

```
m_globalResidual_um = sensorInfo->vectorToGlobal(m_residual_um, true);
```

a zavolání funkce na plnění histogramů do metody vyhodnocující PXD (resp. obdobně pro SVD) zásah.

```
m_histoModule->FillHalfShellsPXD(m_globalResidual_um,
        IsNotYang(m_sensorID.getLadderNumber(), m_layerNumber));
```

K rozlišení jednotlivých half-shellů jsme vytvořili funkce `IsNotYang` a `IsNotMat`, které podle čísla vrstvy a ladderu vrací příslušnou logickou hodnotu, přičemž k rozhodování slouží schéma na obrázku 1.4.

```
bool DQMEventProcessorBase::IsNotYang(int ladderNumber, int layerNumber)
{
        switch (layerNumber) {
                case 1:
                        return ladderNumber < 5 || ladderNumber > 8;
                case 2:
                        return ladderNumber < 7 || ladderNumber > 12;
                default:
```

```
                return true;
        }
}
```

Důvod, proč funkce odpovídají na otázku zda senzor není v daném half-shellu, je ten, že se to implementuje jednodušeji než opačná otázka. Zároveň ale na úrovni samotné funkce nelze říct, že „senzor není v Yang“ znamená „senzor je v Yin“, protože by ještě mohl být v Pat nebo v Mat. V místě, kde funkci voláme, už ale víme, zda se jedná o PXD nebo SVD senzor, čili v tu chvíli můžeme obě tvrzení považovat za ekvivalentní.

4.1.2 Zpracování dat

Ke skutečným fyzikálním datům z experimentu jsem bohužel neměl přístup. Za pomoc s výběrem vhodných runů a jejich následným zpracováním modulem `TrackDQM` děkuji Tadeášovi Bilkovi a Peterovi Kodyšovi. Vzhledem k potřebám této práce se jednalo o surová data z detektoru bez kalibrace, alignmentu a možnosti stanovit kritéria pro výběr vhodných drah.

Dále jsme napsali jednoduchý skript, který fitoval histogramy z jednotlivých runů dvojitou Gaussovou funkcí, danou rovnicí

$$f(x) = A_1 \mathrm{e}^{-\frac{(x-\mu_1)^2}{2\sigma_1^2}} + A_2 \mathrm{e}^{-\frac{(x-\mu_2)^2}{2\sigma_2^2}} . \tag{4.1}$$

Počáteční parametry byly zvoleny tak, aby první člen fitoval vyšší a užší část peaku, zatímco druhý fitoval nižší a širší složku, což jsme kontrolovali podmínkou $A_1 > A_2$. Mimo to jsme ještě zaznamenávali medián a střední hodnotu každého histogramu, kterou budeme v dalším textu značit $\overline{\mu}$.

Ukázky fitů můžeme vidět na obrázcích 4.1 a 4.2. Úzký centrální peak na prvním z nich se nacházel v histogramech pro Yin a Yang v souřadnicích x a y, přičemž ve všech byl stále přímo ve středu. Zároveň jsme provedli simulace se záměrně špatným alignmentem, na kterých se centrální peak také jinak nepohnul, ačkoli zbytek histogramu se viditelně měnil. Proto jsme jej záměrně nefitovali.

Z databáze runů experimentu Belle II jsme zjistili čas začátku a dobu trvání každého runu, z čehož jsme spočítali čas v polovině runu. Nakonec jsme vynesli závislosti parametrů fitu na čase.

4.2 Testování na reálných datech

Sledovali jsme hodnoty čtyř výše zmíněných parametrů, popisujících posuny čtyř částí detektoru ve třech souřadnicích, čili jsme měli k dispozici spoustu různých grafů. Z nich jsme vybrali ty, které se nám zdály nejzajímavější a zároveň měli nejmenší nejistoty měření. Některé uvádíme níže, další pak v příloze A.

Do grafu na obrázku 4.3 jsme vynesli parametry μ_1, $\overline{\mu}$ a medián pro reziduály v části Yang ve směru x. Na pár místech můžeme vidět jakési oscilace, které ale nejsou v takto velké časové ose zcela zřetelné. Proto jsme data z prvních několika dní zobrazili v obrázku 4.4, kde už jsou denní výkyvy mnohem jasnější. U parametru μ_1 jsme uvedli chyby fitu, dále jsme přidali svislé čáry označující začátky příslušných dnů.

Podobné oscilace jsou vidět i v dalších časových úsecích, konkrétně na obrázcích A.1 a A.2 (tentokrát už pouze pro μ_1, ale ostatní parametry se chovají podobně). Našli jsme je i v jiné části detektoru (obrázek A.3 s mediánem v Pat ve směru x), stejně jako v další souřadnici (obrázek A.4 s mediánem v Yang ve směru y).

4.2.1 Korelace

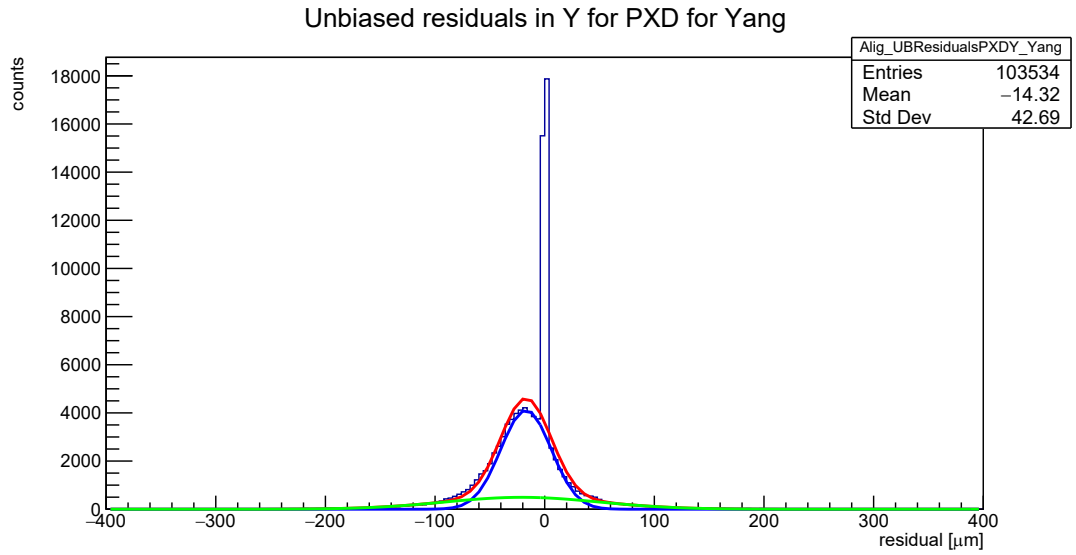
Výsledné grafy si byly relativně podobné, proto jsme vykreslili jednotlivé veličiny proti sobě. Příklad můžeme vidět na obrázcích A.5, A.6 a A.7, které porovnávají medián v x pro Yin, Yang a Pat. Na prvních dvou jsou zřetelně vidět dvě rovnoběžné linie, které jsou časově odlišené – jedna pochází z období od 5. do 12. března, které už bylo v grafu na obrázku 4.4, druhá je pak tvořena všemi ostatními daty. Podobné uspořádání obsahuje obrázek A.8, kde akorát linie nejsou rovnoběžné.

Také jsme spočítali korelace mezi všemi half-shelly v každém směru, výsledky jsou v tabulce B.1 v příloze B. První část představuje korelace mezi všemi daty, druhá obsahuje jen data od 5. do 12. března a třetí je z období od 13. března. Hodnoty v první části tabulky jsou relativně velké, přičemž v některých případech se rozdělením na dvě různá období ještě výrazně zvětší. Také si všimněme, že zdaleka nejmenší korelace jsou v ose z.

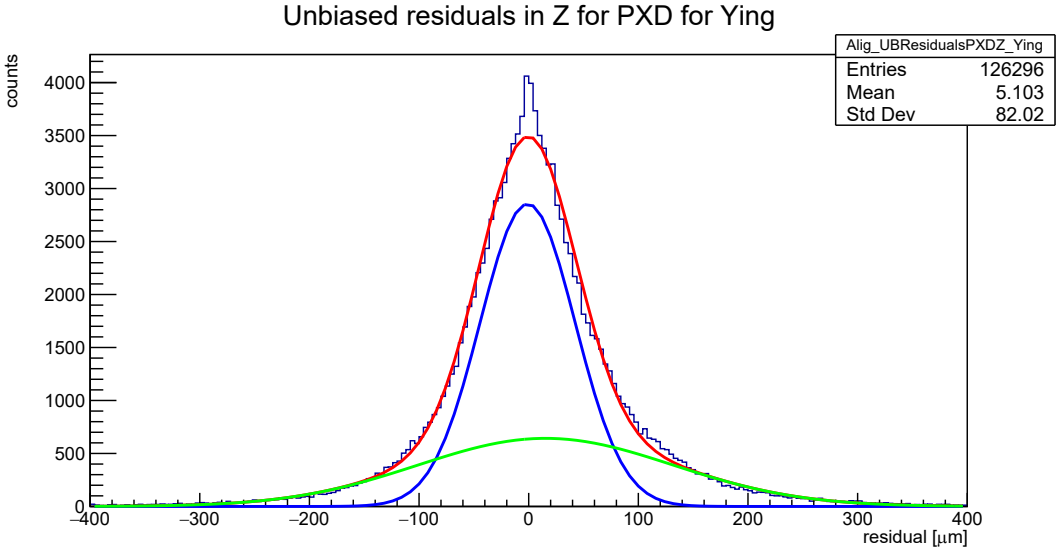
Velikostí korelací myslíme jejich absolutní hodnotu. Kladné korelace odpovídají synchronizovanému pohybu half-shellů ve stejném směru, záporné korelace naopak znamenají pohyb od sebe.

Pro parametr μ_1 byly korelace mezi veličinami obecně menší, viz tabulka B.2. Nejzajímavější nám přišly grafy na obrázcích A.9 a A.10, které zobrazují Yang vs. Pat pro x resp. y.

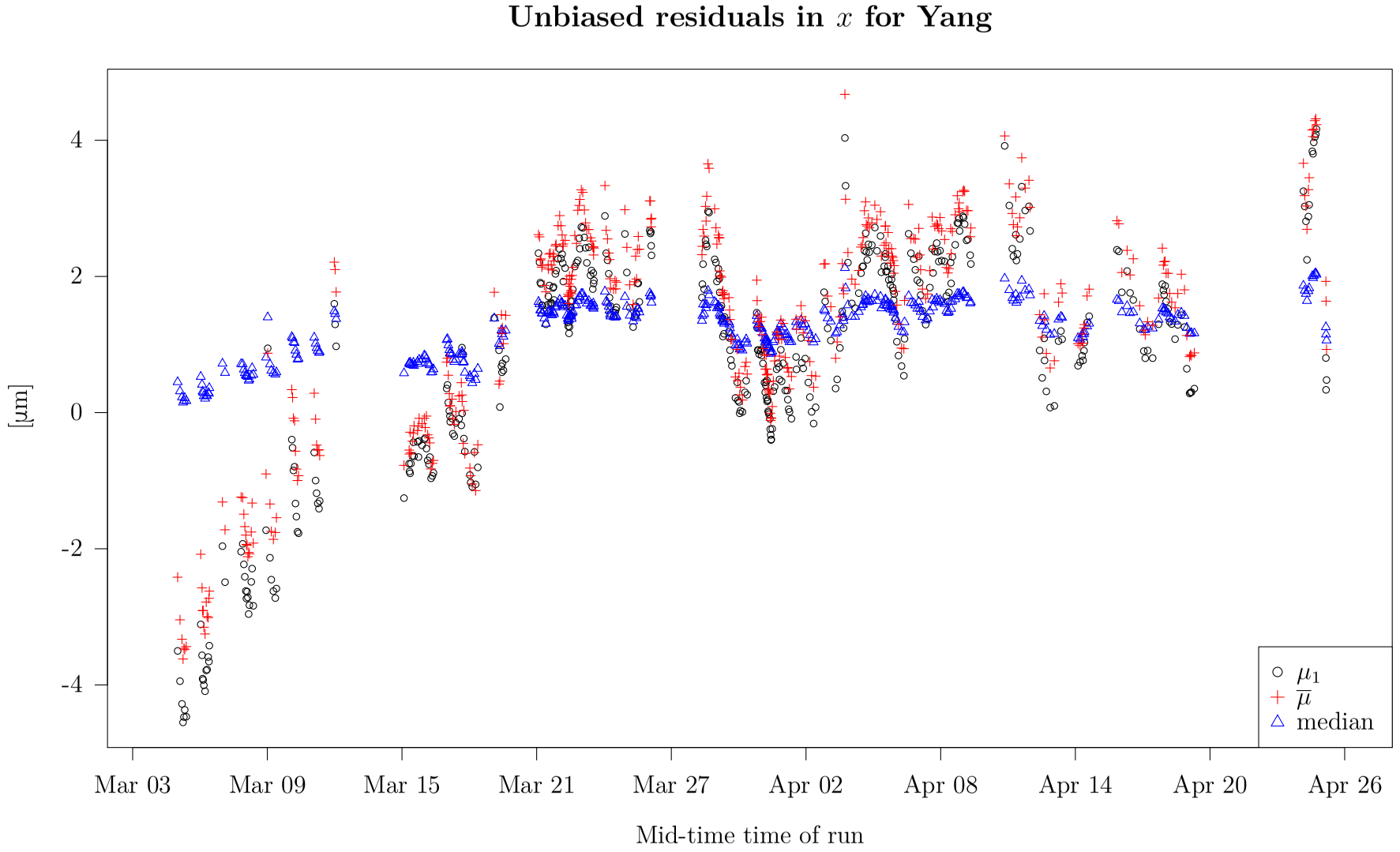
Z grafů korelací se zdá, že data z období před 12. březnem jsou kvalitativně odlišná od těch, která byla naměřena potom. To, že hodnoty leží na dvou rovnoběžných přímkách, znamená, že příslušné grafy reziduálů mají pro oba half-shelly podobný časový průběh, ale v jednom z nich došlo mezi prvním a druhým obdobím ke konstantnímu posunu hodnot nahoru nebo dolů. To odpovídá situaci, ve které se oba half-shelly delší dobu pohybují synchronizovaně, poté jsou v jednu chvíli navzájem posunuty a následně se opět pohybují stejně. V některých grafech se mění i úhel mezi korelačními přímkami, který má význam poměru velikostí vzájemných pohybů half-shellů.



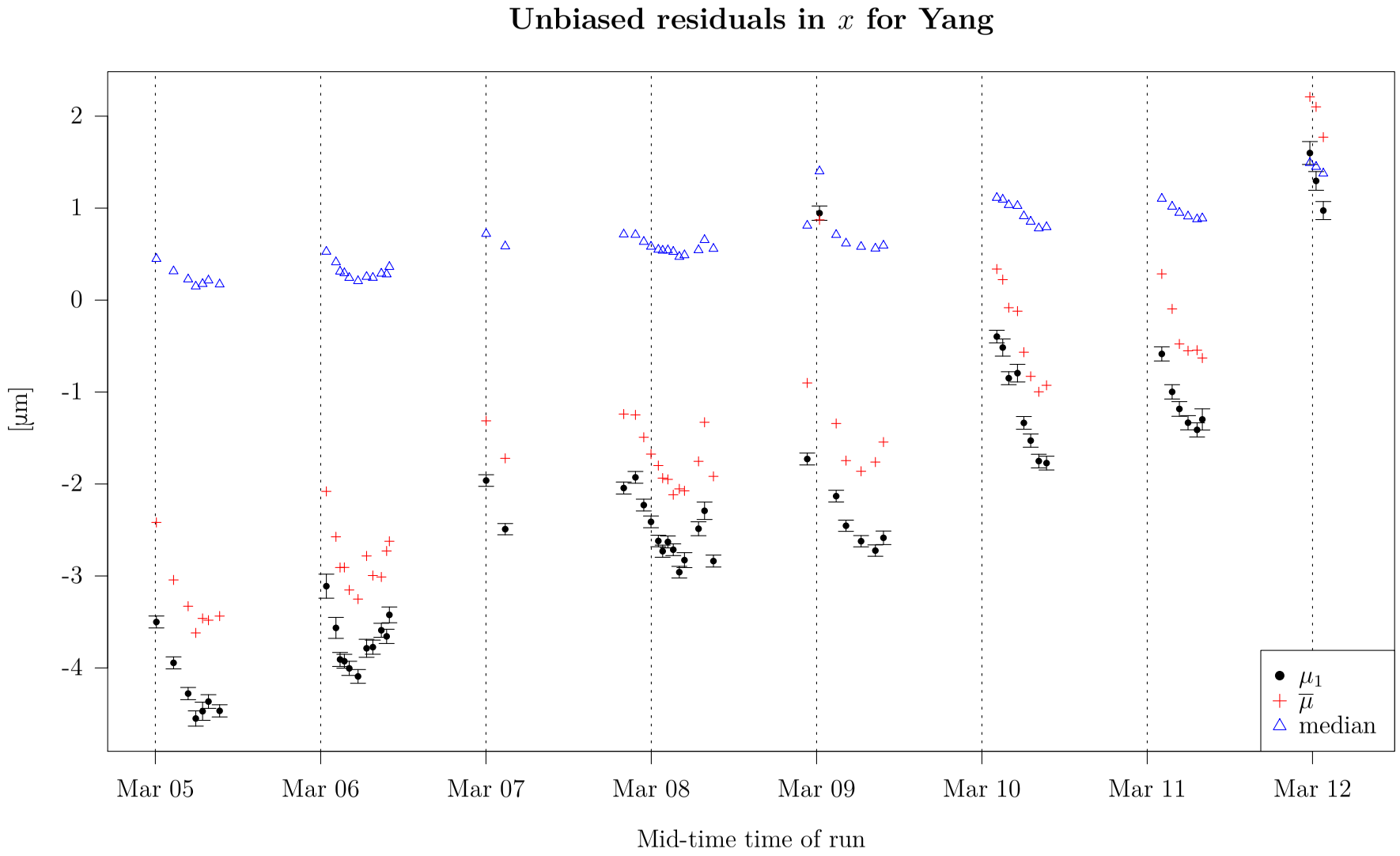
Obrázek 4.1: Ukázka fitu histogramu reziduálů v half-shellu Yang ve směru y. Modrá křivka je první složkou Gaussovy funkce (parametr μ_1), druhá složka (μ_2) je zeleně. Výsledná funkce (4.1) je červeně. Data pochází z runu číslo 1 188.



Obrázek 4.2: Ukázka fitu histogramu reziduálů v half-shellu Yin ve směru x. Modrá křivka je první složkou Gaussovy funkce (parametr μ_1), druhá složka (μ_2) je zeleně. Výsledná funkce (4.1) je červeně. Data pochází z runu číslo 1 188.



Obrázek 4.3: Parametry μ_1, střední hodnota a medián z histogramů reziduálů v senzorech z části Yang ve směru x.



Obrázek 4.4: Parametry μ_1, střední hodnota a medián z histogramů reziduálů v senzorech z části Yang ve směru x.

5. Diskuze

5.1 Refaktoring

5.1.1 Validace

Na praktickou část práce jsme měli tři základní požadavky. Výsledný kód musí být funkční, dostatečně rychlý a především musí dělat to, co po něm chceme.

K ověření první podmínky jsou v basf2 testy, které simulují spousty různých scénářů včetně těch chybových. Poté co se náš kód úspěšně zkompiloval a prošel všemi testy, jsme jej vyzkoušeli na reálných datech, kde opět fungoval bez problémů. To sice není nezvratný důkaz, ale je to alespoň dobrý indikátor toho, že náš kód pravděpodobně neskončí chybou.

Basf2 dále umí na konci výpočtu zobrazit časy, po které běžely jednotlivé moduly. Tím jsme ověřili, že nové moduly pracují v průměru rychleji, než ty původní. Použili jsme k tomu vzorky s řádově tisíci událostmi, kde oba moduly spotřebovaly jednotky sekund. To ještě nezaručuje, že náš kód bude za všech okolností rychlejší, než ten původní. Na druhou stranu můžeme říct, že nebude nijak výrazně pomalejší.

Porovnáním výsledných histogramů jsme zjistili, že výstupy nových a původních modulů se shodují, samozřejmě s výjimkou těch histogramů, které jsme schválně změnili. Kompletní seznam úprav v kódu je možné najít v příloze C.

Změny probíhaly ze dvou důvodů – buď se jednalo o opravy chyb, nebo o snahu sjednotit chování modulů. Opravení několika chyb je jakýmsi vedlejším produktem této práce, během které jsme se museli důkladně seznámit s kódem. Druhý důvod pak vychází přímo z refaktoringu, přesněji ze snahy o odstranění duplicitního kódu. Přitom jsme si všimli rozdílů mezi oběma moduly v místech, kde by žádné být neměly. Díky tomu, že všechen stejný kód byl přesunut do třídy, která je společným předkem obou modulů, jsou další rozdíly mnohem lépe vidět.

5.1.2 Kód

Při programování zřídkakdy existuje správné řešení, které je nejlepší ve všech možných situacích. Každý kód je výsledkem mnoha různých kompromisů, a jako takový má nutně své nevýhody. Příkladem je efektivita kódu na jedné straně a jeho srozumitelnost na straně druhé.

Tím nechceme říct, že přehledný a dobře čitelný kód musí nutně být pomalý, a už vůbec ne to, že komplikovaný kód bude rychlý. Spíš jde o to, že každé zlepšení srozumitelnosti kódu může vést k menší efektivitě. V mnoha případech se to vyplatí, například když vytvoříme metodu, která nahradí duplicitní kód. Zavolání této funkce sice zabere nějaký malý čas, ale ten bude vzhledem k celkové době běhu program zanedbatelný. Na druhou stranu, výsledný kód bude mnohem přehlednější. Každopádně je nutné si uvědomit, že k něčemu takovému může dojít, a zvážit, zda to za to stojí. Oba moduly sice zabírají jen relativně malý čas z celkové doby výpočtu, ale několikanásobné zpomalení by se již mohlo projevit.

Další kompromis, který jsme museli učinit, byl mezi optimalizací a rozšiřitel-

ností. Čím více specializujeme kód k řešení konkrétní úlohy, tím těžší jej bude v budoucnosti rozšířit o nové funkcionality. Příkladem jsou funkce na definice histogramů, které jsme přidali primárně proto, abychom omezili opakování stále stejných sekvencí příkazů. V tomto případě se ukázalo, že je lepší vytvořit relativně jednoduché funkce, které fungují pro naprostou většinu histogramů, než se snažit o zcela obecné metody, jejichž použití by ale bylo výrazně složitější.

Na podobný problém jsme narazili při návrhu hierarchie modulů. Vzhledem k tomu, jak moc si byly podobné, bylo jasné, že bude potřeba většinu kódu sjednotit na jedno místo. V úvahu přicházelo několik možností – mohli jsme například vytvořit jeden modul, který by uměl vše, a `AlignDQM` a `TrackDQM` by z něj zdědily to, co by potřebovaly. Také jsme mohli zařídit, aby se to, které histogramy se vykreslí, řídilo z konfiguračního souboru či z prostředí Pythonu, ze kterého se moduly typicky spouští.

Výhodou obou těchto přístupů je, že umožňují snadno a rychle měnit chování modulů, druhý zmíněný k tomu ani nepotřebuje zasahovat do `C++` kódu. Nevýhodou je nutnost velmi obecného kódu, který je sice univerzální, ale zároveň není v mnoha ohledech optimální. V případě, že by se software během experimentu rychle vyvíjel a by bylo potřeba každou chvíli změnit zobrazované histogramy, zvolili bychom nejspíše jednu z těchto možností.

Řešení, pro které jsme se rozhodli, využívá virtuální funkce, které umožňují, aby moduly zdědily ze společného předka jen to, co potřebují, a zbytek přepsaly. Díky tomu se dají libovolně přizpůsobit a zároveň neobsahují téměř žádný duplicitní kód. I tento přístup má samozřejmě své nevýhody, například může vést k složité hierarchii (vícenásobných) dědičností, pokud bychom chtěli vytvořit spoustu podobných, ale mírně odlišných modulů.

Nový modul, který bychom mohli chtít přidat, bude buď podmnožinou stávajících funkcionalit, nebo bude obsahovat některé zcela nové. Příkladem první možnosti by mohl být modul `PXDOnlyDQM`, který by zobrazoval pouze histogramy z PXD zásahů. V takovém případě nebude nutné téměř nic přepisovat, maximálně vytvoříme mezistupeň mezi třídou `DQMHistoModuleBase` a dvěma současnými moduly, do kterého přesuneme ty funkcionality, které `PXDOnlyDQM` nebude potřebovat. Ten pak bude dědit přímo z `DQMHistoModuleBase`, která bude obsahovat jen metody pro vykreslování PXD histogramů, zatímco současné moduly budou dědit z nového mezistupně.

Když už jsme přidali modul s PXD zásahy, budeme pravděpodobně chtít přidat i nějaký jen pro SVD zásahy. Řešení bude podobné jako u PXD, akorát paralelně přidáme třídu umožňující zobrazovat SVD histogramy a u modulů s oběma typy zásahů použijeme vícenásobnou dědičnost. K získání modulu se zcela novými funkcionalitami by stačilo vytvořit potomka některého ze stávajících modulů následně jej rozšířit.

V obou případech bychom se vyhnuli složitým změnám v hierarchii modulů, stejně jako rozsáhlým úpravám jednotlivých tříd. Přidávání stále dalších modulů, které by kombinovaly nové a současné funkcionality, by si nejspíš už vyžádalo zásadní změny dosavadní struktury, ale jak jsme již zmínili výše, k tomuto účelu nebyla navržena.

5.2 Pohyby half-shellů

V grafech z předchozí kapitoly je vidět, že jednotlivé half-shelly se vůči sobě pohybují, občas dokonce pravidelně s periodou o délce jednoho dne. Zatím nevíme, čím jsou tyto pohyby způsobené, ani je nedokážeme lépe popsat. Podstatné však je zjištění, že má smysl half-shelly podrobně sledovat.

Alignment detektoru je poměrně náročný proces, při kterém fitujeme řádově tisíce parametrů. Každá informace, kterou o detektoru získáme, může vést k lepšímu pochopení vztahů mezi jednotlivými parametry, což nám umožní jejich počet snížit. V konečném důsledku budeme schopni provádět alignment častěji, protože nám bude trvat kratší dobu, než pro něj nasbíráme dostatek dat. Častější alignment se pak projeví snížením nejistoty měření.

Každý z parametrů, které jsme z histogramů dostali, má trochu odlišný význam. Například medián je stabilní vůči výrazným odchylkám, čili je vhodný pro sledování dlouhodobého vývoje. Při kontrole kvality dat nás ale často zajímají právě okamžité odchylky, které jsou dobře vidět u průměru a μ_1.

Střední hodnota části histogramů v half-shellu Yin v souřadnicích x a y byla příliš blízko nule a proto se v nich první komponenta funkce (4.1) fitovala na úzký centrální peak. Toto bychom mohli zlepšit lepším modelem s více parametry, nebo stanovením podmínek na výběr jen těch dat, která nás zajímají. Každopádně výše popsané výsledky jsou vidět i v ostatních parametrech, stejně jako v half-shellech, kde všechny fity probíhaly podle očekávání.

V předchozí kapitole jsme naznačili, že data naměřená před a po 12.3. se liší. Důvodem pozorovaných rozdílů je nejspíše údržba, při které se vypne detektor a některé části urychlovače a proběhnou různé testy. Údržba se provádí přibližně každé dva týdny, přičemž jeden z termínů byl právě 12.3.

Při údržbě dojde k vypnutí magnetického pole, což výrazně změní síly, působící na detektor. Jednotlivé senzory i podpůrná konstrukce se začnou postupně deformovat, aby se dostaly do nového rovnovážného stavu. Toto dopružování nemusí mít deterministický průběh.

Závěr

V rámci této práce jsme zdokumentovali funkce modulů `TrackDQM` a `AlignDQM`. Přitom jsme našli několik chyb, které jsme rovnou opravili. Tím jsme získali podklady pro následný refaktoring obou modulů.

Výpočetní logiku z funkce `event` jsme přesunuli do samostatné třídy, kde jsme ji rozdělili do kratších a přehlednějších metod. Většina kódu na vyhodnocení zásahu byla stejná pro PXD i SVD, proto jsme ji mohli zapsat pouze jednou funkcí. Mimo to jsme přejmenovali většinu proměnných a obecně jsme zlepšili dokumentaci.

Následně jsme vytvořili nového společného předka pro oba moduly, třídu `DQMHistoModuleBase`, do které jsme sloučili všechen kód, který měly společný. Tím jsme se zbavili spousty duplicit. Nejvíce opakujícího se kódu zbylo u definic histogramů, většinu jsme odstranili díky k tomuto účelu vytvořeným funkcím a třídě `THFFactory`.

Celkově jsme zlepšili kvalitu a rozšiřitelnost kódu. Refaktorované moduly jsme otestovali na skutečných datech z experimentu. Ověřili jsme, že fungují správně a zároveň o něco rychleji, než jejich předchozí verze.

Implementovali jsme histogramy reziduálů z half-shellů v jednotlivých souřadnicích, pomocí kterých byla zpracována experimentální data z období od 5.3. do 26.4. 2020. Z těchto histogramů jsme získali parametry, popisující posuny v half-shellech, které jsme zobrazili v závislosti na čase. Také jsme spočítali na korelace mezi parametry.

Zjistili jsme, že half-shelly se pohybují v rozsahu až několika mikrometrů a že má smysl tyto pohyby dále sledovat. V některých časových obdobích jsme našli pravidelné denní oscilace. Také jsme zaznamenali relativně velký posun, ke kterému došlo během údržby detektoru.

Seznam použité literatury

[1] Software CodingConventions [online]. Dostupné z: `https://confluence.desy.de/display/BI/Software+CodingConventions`. Citováno: 7. 5. 2020.

[2] KEKB breaks luminosity record. Dostupné z: `https://cerncourier.com/a/kekb-breaks-luminosity-record/`. Citováno: 18.5. 2020.

[3] ROOT Reference Documentation [online]. Dostupné z: `https://root.cern.ch/doc/master/index.html`. Citováno: 15. 4. 2020.

[4] Brodzicka, J., Browder, T., Chang, P., Eidelman, S., Golob, B., Hayasaka, K., Hayashii, H., Iijima, T., Inami, K., Kinoshita, K., Kwon, Y., Miyabayashi, K., Mohanty, G., Nakao, M., Nakazawa, H., Olsen, S., Sakai, Y., Schwanda, C., Schwartz, A., Trabelsi, K., Uehara, S., Uno, S., Watanabe, Y., Zupanc, A. a for the Belle Collaboration (2012). Physics achievements from the Belle experiment. *Progress of Theoretical and Experimental Physics*, **2012**(1). ISSN 2050-3911.

[5] Brun, R. a Rademakers, F. (1997). ROOT — An object oriented data analysis framework. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, **389**(1), 81 – 86. ISSN 0168-9002. New Computing Techniques in Physics Research V.

[6] Christenson, J.H. and Cronin, J.W. and Fitch, V.L. and Turlay, R. (1964). Evidence for the 2π Decay of the K_2^0 Meson. *Physical Review Letters*, **13**, 138.

[7] Doležal, Z. a Uno, S. (2010). Belle II Technical Design Report.

[8] Group, B. I. T. (2020). Track Finding at Belle II.

[9] Kandra, J. a Bilka, T. (2017). Alignment and physics performance of the Belle II vertex detector. *PoS*, **FPCP2017**, 053.

[10] Kuhr, T., Pulvermacher, C., Ritter, M., Hauth, T. a Braun, N. (2019). The Belle II Core Software. *Computing and Software for Big Science*, **3**(1).

[11] Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR. ISBN 978-0-13-235088-4.

[12] Martin Kaplan (2020). Use of kinematic-constrained decays in the alignment of the Belle II detector. Bachelor's thesis, Charles University.

[13] Pierce, B. C. (2002). *Types and Programming Languages*. The MIT Press. ISBN 0-262-16209-1.

[14] Schlüter, T. (2014). Vertexing and Tracking Software at Belle II.

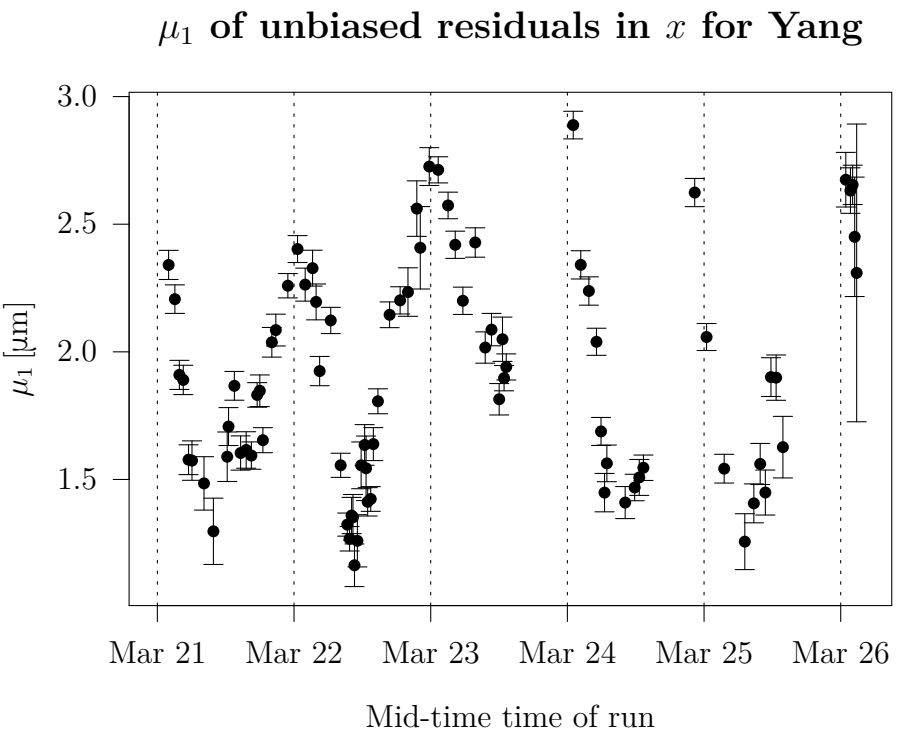
Seznam obrázků

Seznam tabulek

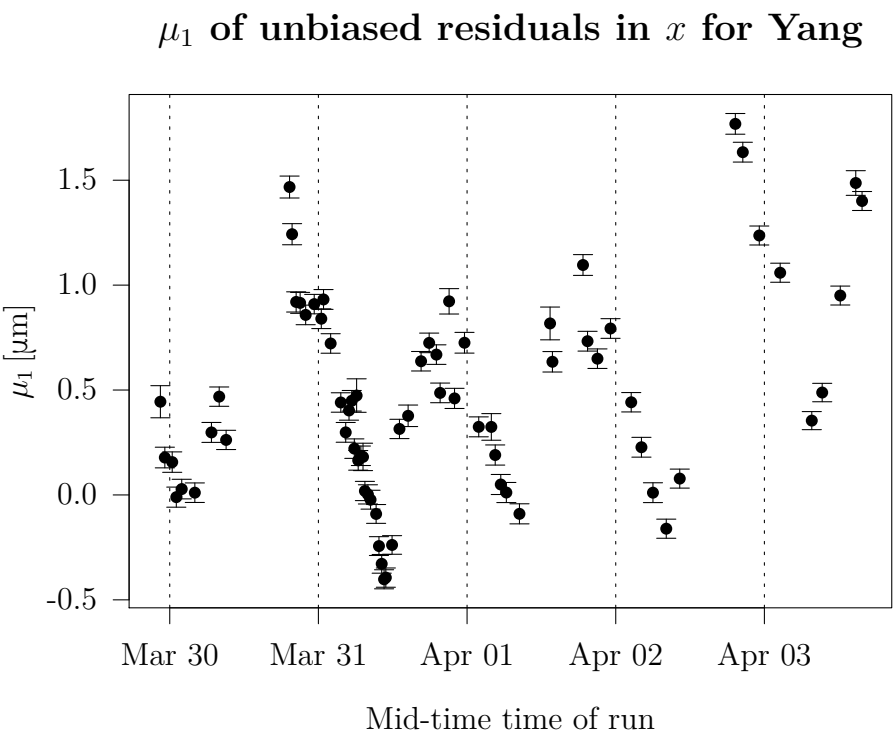
Seznam použitých zkratek

- CPT (*charge conjugation, parity transformation, time reversal*) – sdružení náboje, transformace parity a obrácení času
- DQM (*data quality monitoring*) – monitorování kvality dat
- PXD (*pixel detector*) – pixelový křemíkový detektor
- SVD (*silicon vertex detector*) – stripový křemíkový detektor
- VXD (*vertex detector*) – vrcholový detektor
- CDC (*central drift chamber*) – centrální driftová komora
- TOP (*time-of-propagation*) – systém rozlišující částice na základě doby jejich průchodu detektorem
- ARICH (*aerogel ring-imaging Cherenkov detector*) – detektor Čerenkovova záření, sloužící k rozpoznání jednotlivých částic
- ECL (*electromagnetic calorimeter*) – elektromagnetický kalorimetr
- KLM (K_L^0 *and muon detector*) – detektor kaonů a mionů
- DEPFET (*depleted field effect transistor*) – vyčerpaný polem řízený tranzistor
- basf2 (Belle II *analysis software framework* 2) – framework pro analýzu dat z experimentu Belle II
- KEK (*High Energy Accelerator Research Organization*) – japonská výzkumná organizace zastřešující experiment Belle II
- DESY (*Deutsches Elektronen-Synchrotron*) – výzkumná organizace sdružená kolem elektronového synchrotronu v německém Hamburku

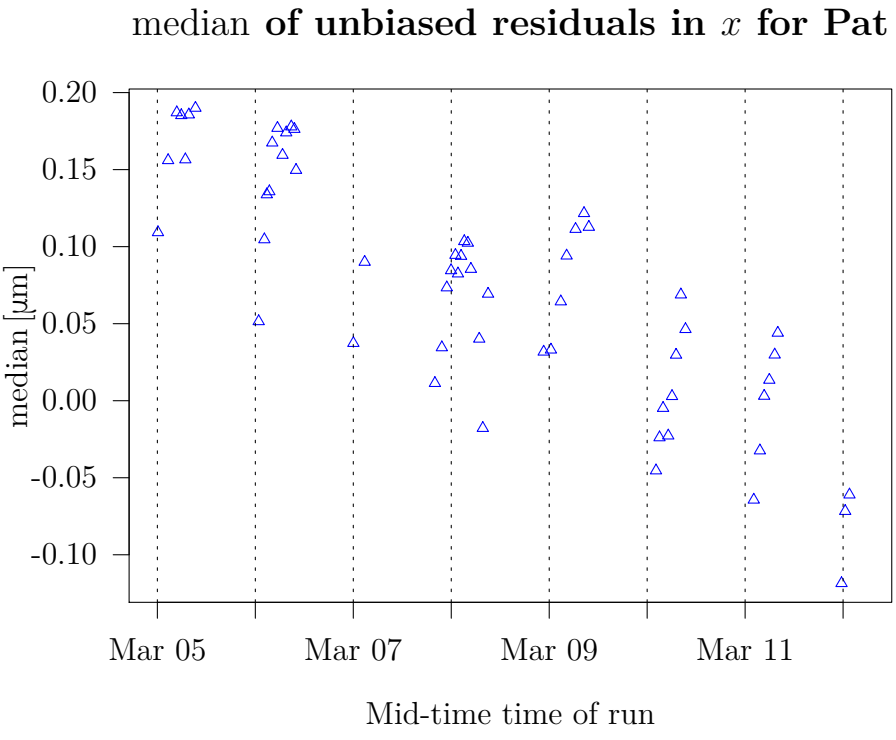
A. Grafy posunů v half-shellech



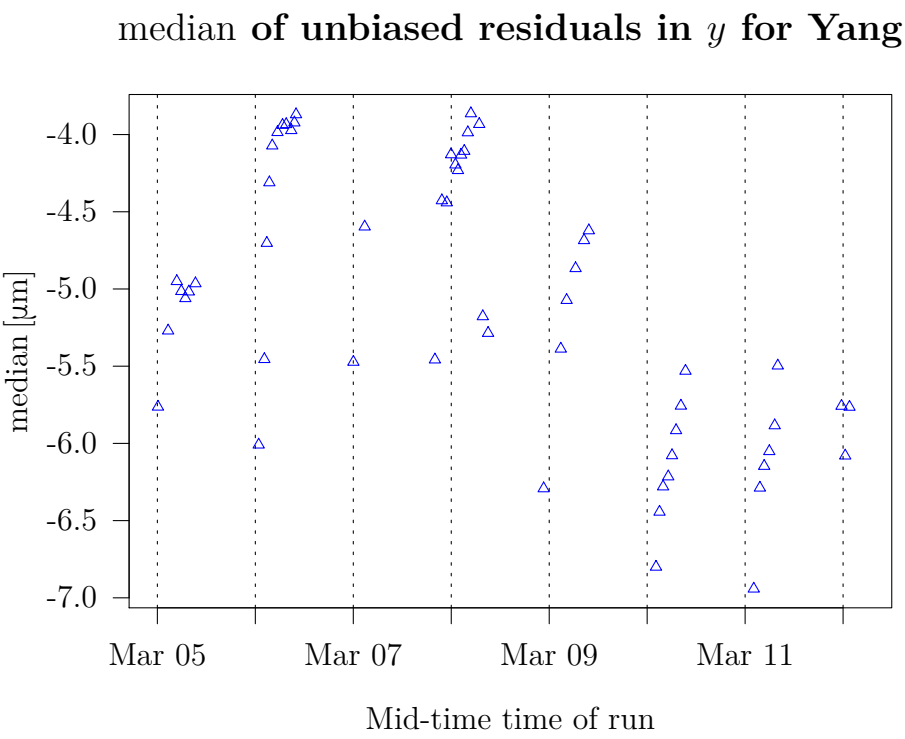
Obrázek A.1: Střední hodnota první komponenty z histogramů reziduálů v senzorech z části Yang ve směru x.



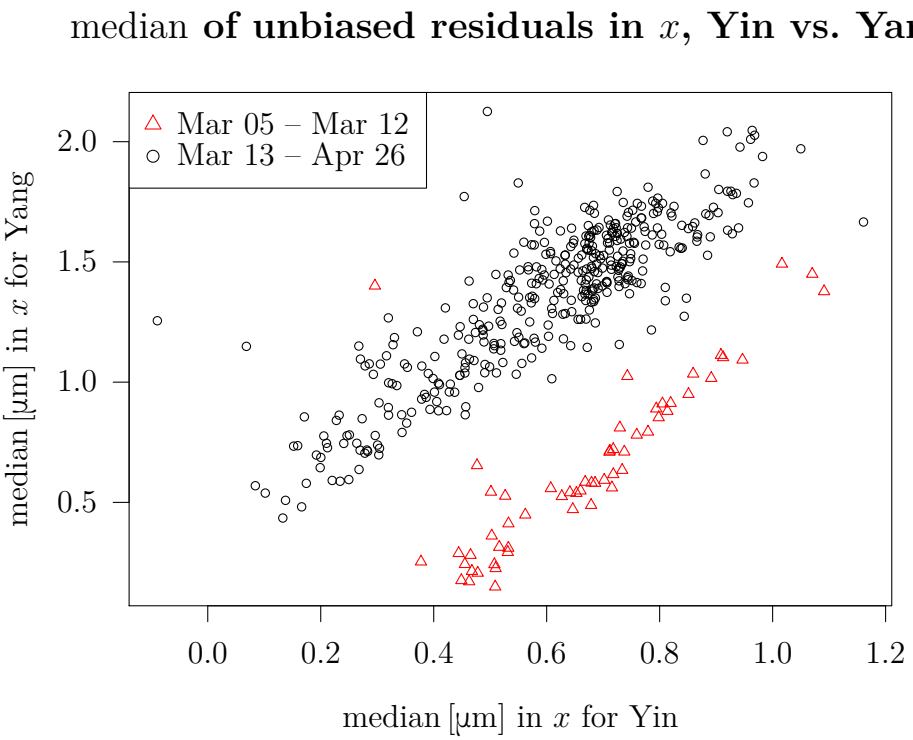
Obrázek A.2: Střední hodnota první komponenty z histogramů reziduálů v senzorech z části Yang ve směru x.



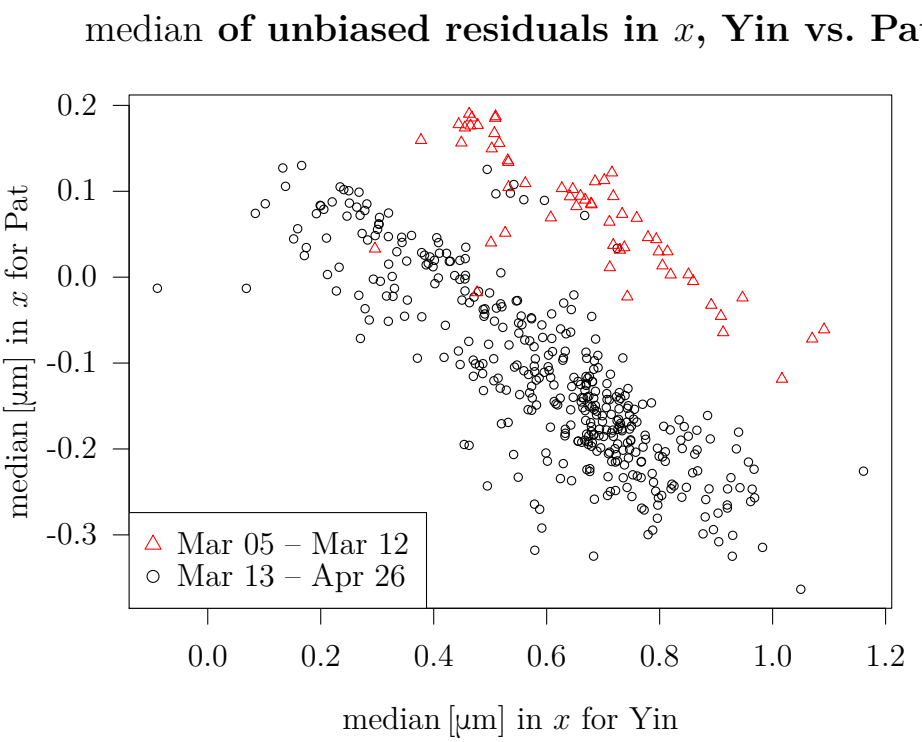
Obrázek A.3: Medián z histogramů reziduálů v senzorech z části Pat ve směru x.



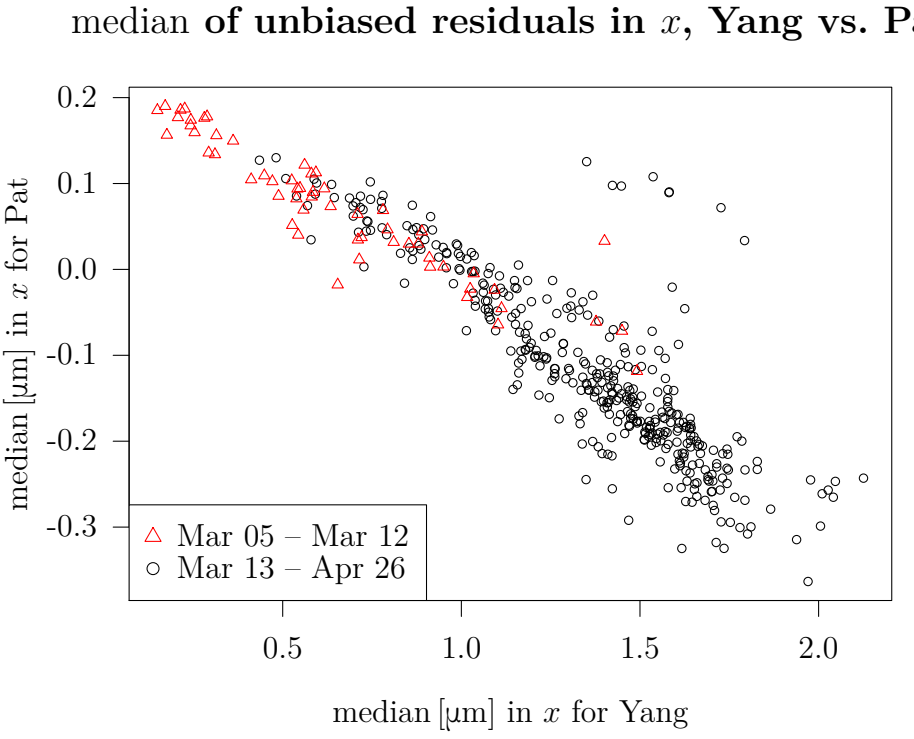
Obrázek A.4: Medián z histogramů reziduálů v senzorech z části Yang ve směru y.



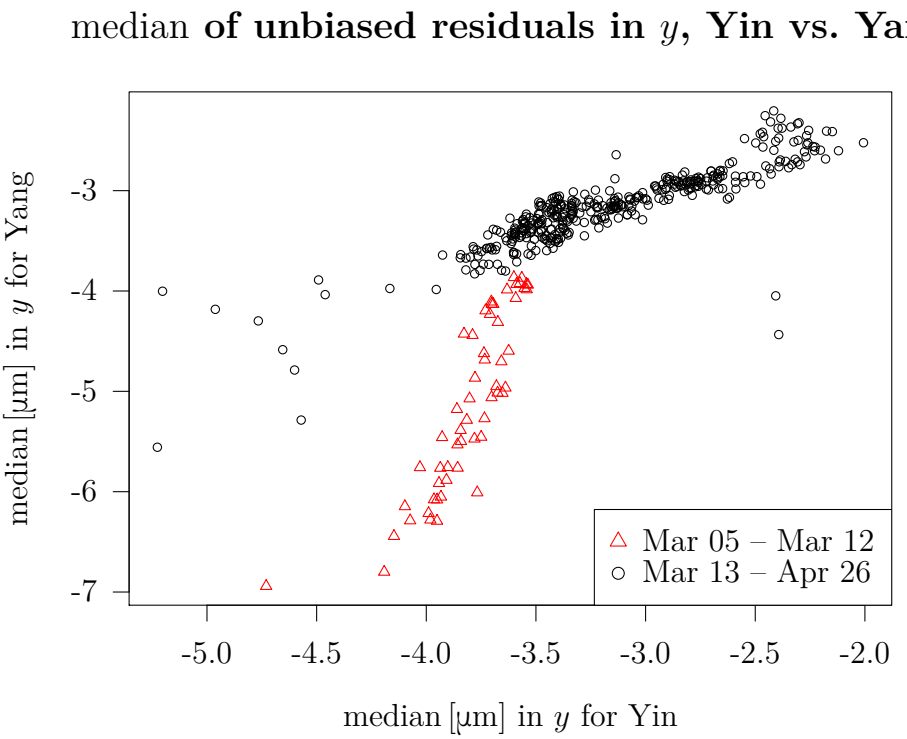
Obrázek A.5: Korelace mediánu z histogramů reziduálů v senzorech z částí Yin a Yang ve směru x.



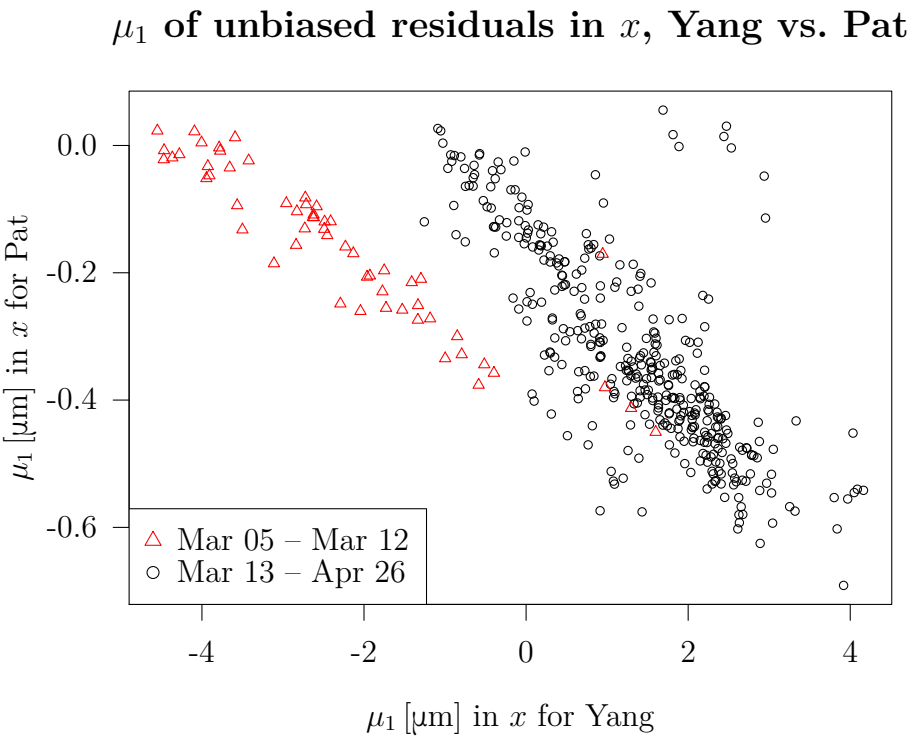
Obrázek A.6: Korelace mediánu z histogramů reziduálů v senzorech z částí Yin a Pat ve směru x.



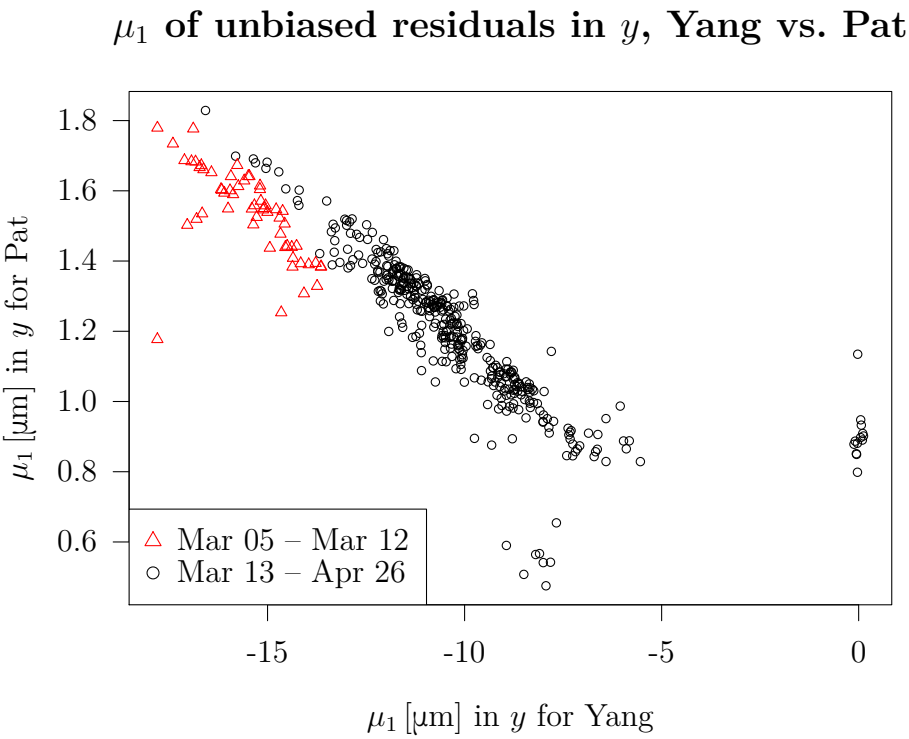
Obrázek A.7: Korelace mediánu z histogramů reziduálů v senzorech z částí Yang a Pat ve směru x.



Obrázek A.8: Korelace mediánu z histogramů reziduálů v senzorech z částí Yin a Yang ve směru y.



Obrázek A.9: Korelace střední hodnoty první komponenty z histogramů reziduálů v senzorech z částí Yang a Pat ve směru x.



Obrázek A.10: Korelace střední hodnoty první komponenty z histogramů reziduálů v senzorech z částí Yang a Pat ve směru y.

B. Tabulky korelací posunů v half-shellech

Tabulka B.1: Číselné hodnoty korelace mediánu histogramů reziduálů pro všechny možné kombinace half-shellů v různých směrech. První část tabulky byla spočítána ze všech dat, druhá z dat z období mezi 5. a 12. březnem a třetí z dat naměřených po 13. březnu.

				x	y	z
	Yin	vs	Yang	0,615	0,686	0,166
	Yin	vs	Pat	−0,643	−0,876	−0,498
	Yin	vs	Mat	−0,740	−0,910	0,086
	Yang	vs	Pat	−0,900	−0,760	−0,744
	Yang	vs	Mat	−0,822	−0,678	0,398
	Pat	vs	Mat	0,899	0,895	−0,669
Mar 05 – Mar 12	Yin	vs	Yang	0,787	0,481	0,417
	Yin	vs	Pat	−0,803	−0,584	−0,534
	Yin	vs	Mat	−0,857	−0,490	−0,121
	Yang	vs	Pat	−0,930	−0,267	−0,662
	Yang	vs	Mat	−0,863	−0,554	−0,255
	Pat	vs	Mat	0,796	0,655	0,166
Mar 13 – Apr 26	Yin	vs	Yang	0,858	0,874	0,347
	Yin	vs	Pat	−0,829	−0,877	−0,334
	Yin	vs	Mat	−0,871	−0,911	−0,333
	Yang	vs	Pat	−0,858	−0,843	−0,863
	Yang	vs	Mat	−0,776	−0,806	0,522
	Pat	vs	Mat	0,888	0,901	−0,645

Tabulka B.2: Číselné hodnoty korelace parametru μ_1 z histogramů reziduálů pro všechny možné kombinace half-shellů v různých směrech. První část tabulky byla spočítána ze všech dat, druhá z dat z období mezi 5. a 12. březnem a třetí z dat naměřených po 13. březnu.

				x	y	z
	Yin	vs	Yang	−0,890	0,532	0,419
	Yin	vs	Pat	0,540	−0,631	−0,560
	Yin	vs	Mat	0,757	−0,640	0,029
	Yang	vs	Pat	−0,752	−0,842	−0,718
	Yang	vs	Mat	−0,897	−0,695	0,509
	Pat	vs	Mat	0,825	0,739	−0,548
Mar 05 – Mar 12	Yin	vs	Yang	−0,882	−0,514	0,370
	Yin	vs	Pat	0,921	0,368	−0,134
	Yin	vs	Mat	0,766	−0,460	−0,080
	Yang	vs	Pat	−0,912	−0,622	−0,147
	Yang	vs	Mat	−0,868	0,137	0,193
	Pat	vs	Mat	0,814	0,180	0,275
Mar 13 – Apr 26	Yin	vs	Yang	−0,732	0,491	0,569
	Yin	vs	Pat	0,391	−0,633	−0,701
	Yin	vs	Mat	0,600	−0,680	0,019
	Yang	vs	Pat	−0,774	−0,789	−0,615
	Yang	vs	Mat	−0,874	−0,497	0,229
	Pat	vs	Mat	0,812	0,683	−0,417

C. Podrobný seznam změn v kódu

C.1 Funkční změny

C.1.1 Chyby, které jsme opravili

- Konstrukce ověřující úspěšnost fitu dráhy mohla vést k rozdílným podmínkám pro plnění histogramů. (Zpracování jedné dráhy)
- Při zpracování dráhy se používaly různě seřazené seznamy zásahů a reziduálů, což vedlo k chybě v případě, že zásah z jednoho typu detektoru vyhodnocoval reziduál z jiného. (Reziduál)
- Zásahy z SVD se vyhodnocovaly pouze tehdy, když první zásah pocházel z části měřící u a druhý z v. (Výpočty dalších veličin)
- Při převodu polohy SVD zásahu do globálních souřadnic se používala pouze informace z u nebo z v, místo z obou zároveň. (Výpočty dalších veličin)
- Jednotky různých veličin byly převáděny neoptimálně. (Výpočty dalších veličin)
- V modulu `AlignDQM` byl histogram zobrazující $\cos\theta$ počítán nepřesně.

C.1.2 Další úpravy

- Změnili jsme názvy a složky některých histogramů, aby byly stejné pro oba moduly a tím pádem jsme je mohli generovat stejnými funkcemi.
- Ze stejného důvodu jsme do `TrackDQM` přidali histogram veličiny θ.
- Sjednotili jsme rozsahy a rozlišení několika histogramů, které měly být stejné v obou modulech, ale nebyly.
- Zpřesnili jsme většinu ladících a chybových hlášek.

C.2 Refaktoring

- Výpočty z funkce `event` jsme rozdělili do spousty kratších funkcí, které jsme přesunuli do samostatné třídy. Pro plnění histogramů jsme vytvořili `Fill-` metody. (Rozdělení do více kratších metod)
- Funkci `defineHisto` jsme rozdělili do kratších `Define-` funkcí. Dále jsme přidali několik funkcí pro automatizaci definic histogramů. (Zdlouhavé definice histogramů)
- To nám umožnilo snadno sestavit seznam všech histogramů, pomocí kterého jsme například zjednodušili resetování histogramů ve funkci `beginRun`. (Nejasné závislosti)

- Vytvořili jsme třídy `THFFactory` a `THFAxis`, které dále usnadňují definice histogramů. (Zjednodušení definic histogramů)
- Veškerý kód, který byl u obou modulů stejný, jsme přesunuli do jejich společného předka. (Sjednocení modulů)
- Změnili jsme názvy proměnných, aby bylo jasnější, co znamenají. Dále jsme veličiny, které nebyly ve standardních jednotkách, označili podle konvence. (Dokumentace)

C.3 Nové funkcionality

- Do modulů jsme přidali parametr, který umožňuje měnit některé vlastnosti histogramů z prostředí Pythonu. (Rozšíření o nové funkce)
- Implementovali jsme sledování posunů v half-shellech pomocí histogramů, zobrazujících reziduály v daných částech detektoru. (Úpravy DQM modulů)