OPA documentation version 4.2.0

Andreas Streun, September 3, 2025

1 How to use this document

This document is intended to support program developers, who want to maintain the existing OPA code, or to rewrite it in another environment, or to extract useful parts for integration into other codes.

When reading this document, the source code may be examined in parallel, preferably in the Lazarus IDE. In order to see how the code works, OPA may be executed or the tutorial [1] may be consulted. Further information on using OPA is found in the user guide [2] and the underlying physics is outlined in [3].

In this document following styles are used:
Pieces of code are shown in `typewriter style`. `GUI` is a graphical user interface, usually based on class `TForm`. `Frame` if a user-defined GUI-component, usually based on class `TFrame`. `Unit` is a standard Pascal unit.

A GUI or a frame has a list of `actions` corresponding to the event handlers of its components, for example a button to be pressed. The pure Pascal units are passive, they contain a set of public `procedures` which are called from the GUIs. Frames too may contain public `procedures`. Thus in this documentation, action lists are given for the GUIs and frames, and lists of public procedures for the frames and units. All modules may offer public variables, here coloured `x`, `y`, `z` for GUIs, frames, units.

GUI components accept user input and/or launch events. Some have captions, like **Buttons**, others not, like for example a (plot area) responding to mouse clicks. The general scheme is **component** → `event handler`. Event handlers may call private procedures of the GUI or public procedures from units as listed in the `uses` section. For the units `unit`>`procedure` refers to a procedure in a unit and `unit`:`variable` to a public variable..

Red text gives informations about bugs and other problems and/or suggestions for future improvements. Green text mentions particular features which distinguish OPA from other beam dynamics codes and should be preserved.

Questions may be adressed to the author at `mailto:psi@andreas-streun.de`.

2 OPA status

Following semantic versioning [4] version 4.063 of OPA was renamed 4.1.0. However, since up to now only the author worked on the code, the distinctions MAJOR, MINOR, PATCH rather refer to the changes as visible to the users than to the API.

Current version is OPA 4.2.0. It includes 44 Pascal units `.pas`, 28 of them are `GUIs` which are accompanied by a Lazarus form `.lfm` (created by the Lazarus IDE), and 16 of them are Pascal `units`. Five of the GUIs are `frames`, i.e. self-defined components embedded in other GUIs. Four units are located in a parallel folder, here called `../com/` since they are not OPA specific but used with other programs too. Table 1 lists the units. The column "Size" lists the lines of code as a rough estimate for the complexity of the unit. Furthermore, OPA includes a couple of image files `.ico, .bmp` to draw symbols on buttons.

A GUI defines a class, given in table 1 in the column "class". The actual GUI is the one and only instance of this class, i.e. a public variable, usually with the name as the class but without the leading "T". A frame also defines a class but no instance, instead the parent GUI will define one or more instances of this class as variables.

In general GUIs should handle the user interactions only while the Pascal units do all the calculations. This separation was largely realized in OPA, however not strictly. Some minor calculations are done in the GUI in some places.

In version 4.2.0 all files were renamed following this convention for the file name: `opa*` is a first level GUIs, i.e. one of the main design programs launched from the main menu. `*lib` is a plain Pascal units which contains a library of physics procedures. `o*` is a second level GUI, which is launched by a first level GUIs, for example a start menu. `*frame` is a frame. Files in the `../com` folder were not renamed.

Table 1 also serves as a **table of contents** with the first column listing the section numbers: Section 3 will explain the "backbone" of OPA, which are the global datatypes and variables, the main menu and units for mathematics and graphics which are used by many GUIs. Section 4 contains lattice file handling and the two editors, text-based and "LEGO" block like.

Beam physics starts with section 5 on the interactive linear optics design panel and related GUIs and units and continues with section 6 on off-momentum optics and bucket viewer. The non-linear optimization module is explained in section 7, and section 8 covers injection and orbit correction.

The three tracking modules with their library are explained in section 9. Section 10 contains an editor for longitudinal gradient bends. Finally, machine layout and magnet currents are in section 11.

Table 1: OPA units overview

Sec	Name	class	Purpose	Size
3.2	globlib		global variables and procedures	2986
3.3	opamenu	TMenuForm	the main menu	901
3.4	mathlib		mathematical library	2075
3.5	../com/vgraph	Vplot	real number graphics library	1276
3.6	../com/asfigure	TFigure	general plot procedure	314
3.7	../com/conrect		contour plot calculation	332
3.8	../com/asaux		little helpers	349
4.1	latfilelib		file reading and writing	1839
4.2	opatexteditor	TFormTxtEdt	text editor for lattice file	190
4.3	opaeditor	TFormEdit	interactive lattice editor	346
4.4	oelecreate	TEditElemCreate	creation of an element	144
4.5	oeleedit	TEditElemSet	edit element parameters	861
4.6	osegedit	TEditSegSet	edit segment parameters	444
5.1	opalinop	Toptic	linear optics design	1143
5.2	knobframe	TKnob	slider to set element property	421
5.3	ostartmenu	Tstartsel	setting for start parameters	615
5.4	obetenvmag	TsetEnvel	setting for optical functions plot	372
5.5	otunematrix	TtuneMatrix	adjustment of lattice tune	251
5.6	owriteomrk	TWOMK	write optics markers	125
5.7	linoplib		linear beam dynamics calculations	2840
5.8	elemlib		element propagation calculations	2387
5.9	omatching	TMatch	linear optics matching	1457
5.10	omatchscan	TsetMatchScan	parameter scan	119
5.11	opatunediag	TTunePlot	show the tune diagram	634
6.1	opamomentum	Tmomentum	momentum dependence and optimization	949
6.2	momentumlib		momentum dependent calculations	480
6.3	opabucket	TopabucketView	plot RF bucket	640
7.1	opachroma	TChroma	nonlinear optimization	1413
7.2	csexframe	TCSex	controller for nonlinear element	454
7.3	chamframe	TCHam	bar indicator for Hamiltonian mode	416
7.4	chromreslib		little helper to pass results handles	79
7.5	chromelelib		little helper to pass element handles	93
7.6	ochromsvector	TSVectorPlot	show Hamiltonian as complex vector	145
7.7	chromlib		nonlinear dynamics calculations	1813
8.1	opaorbit	TOrbit	orbit correction and injection	2353
9.1	opatrackps		phase space tracking	1422
9.2	opatrackda		dynamic aperture tracking	1666
9.3	opatrackt		Touschek tracking	1726
9.4	tracklib		particle tracking calculations	1502
10.1	opalgbedit		longitudinal gradient bend editor	702
10.2	lgbeditlib		longitudinal gradient bend calculations	471
11.1	opageometry		geometric machine layout and matching	2010
11.2	opacurrents		export magnet currents to machine control	303
11.3	testcode		test of new features	472

3 Main programs and general use

3.1 Main program

opa.lpr

The main program file which allocates the units as described below and creates some of the forms. It is created more or less automatically by the Lazarus IDE.

3.2 Global data

globlib

This unit defines global constants, types and variables and thus is OPA's "database". It is also initialized first when OPA starts and sets environment variables and default values. Public variables defined here are used by the other modules, since almost all of them use `globlib`. These "global" variables contain the basic lattice data, status flags, handles to access GUI components and temporary data to be exchanged between modules. This unit also provides many public procedures for various non-physics tasks. Only an overview can be given here, and the most important or complex things will be explained. Everything else should become clear from the code, hopefully.

The `globlib` unit grew large large and heterogenous over the years and should be split into several units for the different functions to be performed, or procedures should be moved to other units if applicable.

Like other beam dynamics programs OPA works on a lattice file. It contains *elements*, mainly magnets of different type, and *segments*, which are line-ups of elements and segments. One of the segments is selected and expanded to become *the lattice* to work with. Element parameters, e.g. quadrupole strength in the lattice file can be numbers or arithmetic expressions using *variables*, which are part of the lattice file too.

Uses: `../com/mathlib`, `../com/asaux`

Public constants, types and variables

Constants include general settings, array sizes, color definitions, name strings and values of flags later to be referenced by name.
Following type definitions may be considered as non-trivial:

`ElementType` is a case-record for defining element parameters such as length, quad focusing strength, dipole bending angle etc. Several (but not all) parameters may be arithmetic expressions instead of numbers. For more details see the user guide [2] or the source code.

variable_type is for variables defined by name and value or arithmetic expressions. Variables are referenced in arithmetic expressions of element parameters or in other variables.

AbstractEleType represents either an element or a segment and contains pointers to previous and next abstract elements. The abstract element may be inverted or repeated.

SegmentType is for segments, which are series of abstract elements, defined by pointers to its initial and final abstract element.

LatticeType is a lattice entry. The lattice is built by expanding a segment, so it contains only elements, and additional information on direction (if the element is inverted), and data relevant for orbit correction (misaligmnets and if the element sits on a girder).
Unlike several other codes OPA handles correctly nested inversions of segments to obtain the correct orientation of the element in the lattice. This is relevant for bending magnets with unequal edge properties.

OmarkType is a set of optics data as property of the element type "optics marker".

OpValType is an extended set of optics data used for propagation in elemlib.

GirderType describes a girder by first and last lattice entry of elements sitting on the girder, and by the type of connection to adjacent girders.

CurveType is a point in an optical functions curve including a pointer to the next point.

CurvePlotType defines a curve by a pointer to its initial point.

DefaultType defines a user setting by name and value. Default values are defined by globlib>Initialization but can be modified by the user and saved.

Other types are mainly shortcut definitions to gather interrelated data.
Here are the most important global variables. If not mentioned otherwise, the type name is the variable name with suffix type. Usually only one instance exists:

Elem and Ella are arrays of Elementtype, where the first one is read from file and manipulated in the editors, while the second one are a subset of elements used in the lattice and to be modified in the various calculations. After terminating a calculation the user is asked to save or cancel the changes, which causes the Ella-data to be copied back to the Elem array or not.

Segm, Lattice and Girder are arrays of corresponding types. Maximum array length is defined in the const section of the unit.

Glob saves some global parameters like beam energy, default aperture size etc.

Status is a group of status flags to be set by the outcome of calculations in order to enable or disable buttons and to re-use data in other calculations.

MainButtonHandles provides global control of the various options in opamenu to enable or disable them based on the status flags.

GlobDef and Def of DefaultType are arrays of user-defined settings, which are read at start and when switching the working folder.

Beam saves beam parameters like tune, emittance etc.

SnapSave stores intermediate results in order to re-use them in other modules, for example

analytical results from the nonlinear optimization to compare them later with tracking. (Furthermore, some data are saved which are relevant for machine control and to be exported by `opacurrents` to an EPICS `.snap` file.) Data are not invalidated yet, if something was modified in the lattice.

There are more variables of basic types, which better should not be public to avoid errors and confusion. Wherever possible they may become local, or they should be hidden behind set- and get-procedures.

Public procedures

This listing of procedures is more or less inverse to the (historical and illogical) arrangement in the source file:

`Initialization` sets some global variables to reasonable initial values, sets all status flags to false, and sets default values for ∼250 user settings. Then it sets the OPA directory and tries to read the files `opa4_path.ini` which contains a list of last used files. Then it reads `opa4_glob_ini` containing global user settings (at the moment this is only the amount of output). Finally, it takes the folder from the first entry in the list of last used files to set the working folder and reads the file `opa4_set.ini` in this folder to restore the ∼250 user settings. If these files are missing, the default values are used.

`GlobDefWriteFile` and `DefWriteFile` write the user settings back to these files if the session is regularly terminated by **opamenu** → `ExitOPA`.

`MakeLattice` builds the lattice from elements and segments. Since a segment istelf contains elements and segments, the internal procedure `SegLat` is called recursively to unpack the data. The elements, which are used in the lattice are copied from the `Elem` to the `Ella` array, and correctors and monitors with generic names `CH,CV,MON` are expanded into separate instances named `CH001,CH002...` to adress them individually in orbit correction in unit `opaorbit`. Finally status flags are set.

`GirderSetup` evaluates the girder structure if contained in the lattice and allocates the lattice elements to the corresponding girders.

`IniElem` sets default values for new elements.

`AppendAE, ClearSeg, NAESeg` are procedures to handle segments, which are series of pointers to elements or other segments.

`AppendCurve, ClearCurve` prepare data sets for plotting curves of optical functions.

`AppendChar` builds a linked list of characters for `opatexteditor` (wrong place).

`putkval, getkval, putSexkval, getSexkval` functions are shortcuts to set or get the parameter which is considered the strength of a magnet.

`FillComboSeg` fills the list of last used files in `opamenu`.

`OPALog` writes a message to the log window in `opamenu`.

`MainButtonEnable` enables or disables the options in `opamenu` depending on the status flags. For example, tracking is only enabled if a periodic solution exists.

PassMainButtonHandles and passErrLogHandle take handles to the GUI components of opamenu to enable other units using globlib to enable/disable them.

EllaSave and Elcompare check if elements in lattice (Ella) have been changed with regard to the original elements (Elem) and ask the user if the changes should be saved.

The procedures and functions listed under "Calculator" are an arithmetic evaluator adapted from [16]. Other procedures not listed here include variable, element and segment to string conversion and other utilities which may be self-explanbatory.

3.3 Menu

opamenu

The main GUI for reading and writing files and to launch the other GUIs. It includes message (log) and status windows. It is always open in the background. Closing it exits OPA.

Uses: globlib, opaeditor, opatexteditor, latfilelib, opalinop, opatunediag, opamomentum, opachroma, opatrackps, opatrackda, opatrackttt, opaorbit, opageometry, opalgbedit, opabucket, opacurrents, testcode

Actions

FormCreate called at start creates the GUI and fills the menu items with data: the list of last used files is established and the status labels are set. Then handles to most components are passed to globlib to make them available to all GUIs, in order to send messages to the log window, update the status in the status window and enable or disable options depending on the result of calculations. Note, that the initialization of globlib is executed first to provide the required information.

File Menu

New → fi_new delete all data to start new lattice from scratch.

Open... → fi_open read OPA lattice file or try to read a lattice file from Tracy2, Tracy3, MAD-X, elegant or BMAD. The file name is displayed as "active file".

Last Used ► → fi_last select file from list of last used files

Save, Save as... → fi_[save,svas] save OPA lattice file with old or new name.

Export to ► → ex_[tracy2,tracy,madx,elegant,bmad,opanovar] save as file for other programs. Last option expands all arithmetic expressions in lattice and saves as OPA file.

Exit → fi_exit save all settings from session and exit OPA (just closing the GUI will exit OPA without saving settings).

Edit Menu

Text Editor → ed_text launch lattice file text editor opatexteditor.

OPA Editor → ed_oped launch interactive "LEGO block" editor opaeditor.

Design Menu

Linear Optics → ds_opti launch interactive linear design opalinop.

Off-momentum Optics → ds_dppo launch momentum dependent optics opamomentum

Non-linear Dynamics → ds_sext launch non-linear optimizer opachroma

Orbit Correction → ds_orbc launch orbit & injection panel opaorbit

Injection Bumps → ds_injc launch orbit & injection panel opaorbit

RF opabucket Viewer → ds_rfbu launch RF bucket visualization opabucket

Geometry Layout → ds_geo launch geometry layout and matching opageometry

LGB Optimizer → ds_lgbo launch editor for longitudinal gradient bends opalgbedit

Tracking Menu

Phase Sapce → tr_phsp launch phase space tracking opatrackps

Dynamic Aperture → tr_dyna launch dynamic aperture tracking opatrackda

Touschek Lifetime → tr_ttau launch Touschek lifetime tracking opatracktt

Extra Menu

Output ► → tm_di select the amount of output provided in the log window.
The implementation is incomplete and inconsistent: some output goes to the log window, some to a terminal console (only visible if it is open) and some to a file diagopa.txt, which is created (but never closed...).

Magnet Currents → tm_cur launch panel to calculate magnet currents opacurrents

The other menu items in this group are temporary tests calling procedures from unit testcode and not relevant to be documented here.
(Segment name) → ComboSeg is a drop-down list of the available segments. The one selected is expanded to become the lattice, calling globlib>MakeLattice.

Show lattice expansion → butlatsh displays the expanded lattice in the log window.

Print → ButLogPrt prints the content of the log file to LogPrint.txt in the working directory.

Clear → ButLogClr clears the log window.

3.4 Mathematics library

mathlib

A library of mathemical functions: definition of types and operations with vectors, matrices and complex numbers. It further contains some special functions, among them the Touschek integral function, and implementations of Powell's minimizer, LU decomposition and Singular Value Decomposition taken from [14], but extended to dynamic array size. OPA does not link external libraries but all algorithms needed are included in the code.

Uses: none

Public constants, types and variables

Types define vectors and matrices for beam dynamics, geometry and for the Powell minimization procedure. The only public variables are the parameters for Powell.

Public procedures

Most procedures are elementary operations on vectors, matrices and complex numbers, or for special functions, and don't need to be explained.

`CTouschek` and `CTouschek_pol` solve the Touschek lifetime integral, see appendix C.1 in [3] for details. The first procedure solves the integral based on Simpson's rule, the second procedure uses a polynomial approximation for speed-up.

`EulerAng` calculates rotation angles from a rotation matrix as explained in [5].

`LUDCMP` and `LUBKSB` peform LU-decomposition and backsubstitution to solve linear systems of equations as described in [14]. This is used in particuar for matrix inversion, `MatInv` and `MatDet`. The procedures are set fix to 5 dimensions.

`svdcmp` and `svbksb` perform singular value decomposition and backsubstitution to solver lineaer systems of equations as described in [14]. SVD is superiour to LUD for non-square and degenerate systems. An packing/unpacking algorithms was added to use the procedures with arbitrary dimension. The procedure is used in orbit correction in `opaorbit` and for setting octupole families in `opachroma`.

`Powell` is an implementation of Powell's minimizer for steepest gradient search in N dimensions taken from [14]. It is used for sextupole optimization in `opachroma`, for longitudinal gradient optimization in `opalgbedit`, and (test mode only) for non-linear optimization in `opamomentum`.

3.5 Graphics library

`../com/vgraph`

Class `Vplot` based on `TObject` contains a Lazarus `TCanvas` object. Plot commands for `TCanvas` use integer screen pixel coordinates. `../com/vgraph` wraps these standard plotting commands with procedures of same name but accepting physics coordinates as real numbers. Data are either scaled and forwarded to `TCanvas` or written to an Ecapsulated postscript file `*.eps`.

Furthermore procedures for crating nice axes, for drawing circles and ellipses and for grabbing the screen image have been added.

Uses: none

Public constants, types and variables

There are no public variables. A GUI will define one or more variables of class `Vplot`

Public procedures

`Create` accepts a handle to a `TCanvas` object, which is needed to construct a `Vplot` object. Further some initialization is done.

`PS_start` opens an `*.eps` file of name `psfile` supplied by the calling procedure for output and sets the private flag `PS` to `true` to direct the output to the file and not to the screen. A file header is written, defining the BoundingBox as the canvas screen size. Then macros for text alignment are defined in postscript language. If opening the file fails, an error message is returned.

`PS_stop` closes the `*.eps` file and sets the flag `PS` to `false`.

`SetRange*`, `SetMargin*` etc set plot ranges and scaling as displayed in Fig.1. The vertical screen coordinate counts downwards whereas the physical coordinates as well as the Postscript coordinates count upwards.

`getpx,getpy` and `PS_getpxr,PS_getpyr` translate physical coordinates x, y into screen, resp. Postscript coordinates (upper/lower line):

$$\texttt{px} = \texttt{px0} + \frac{\texttt{pxrange}}{x_{\text{range}}}(x - x_{\min}) \qquad \texttt{py} = \left\{ \begin{array}{r} \texttt{py0+} \\ \texttt{pytotal} - \texttt{py0-} \end{array} \right\} \frac{(-\texttt{pyrange})}{y_{\text{range}}}(y - y_{\min})$$

`AdjustAspectRatio` adjusts the plot ranges such, that the unit is the same horizontal and vertical, i.e. a circle appears as circle not as an ellipse.

`Axis` plots an axis with even numbers as annotations and ranges, also extracting an exponent if needed.

`GetAxisSpace` returns the space required for the axis annotations without drawing the axis in order to adjust the plot range accordingely. It is called first by the calling GUI, in particular by `../com/asfigure`.

`Circle` and `Ellipse` plot a circle or a (sheared) ellipse in beam dynamics notation.

`GrabImage` copies the screen canvas to the system clipboard, useful to catch plots for draft notes.

All the other procedures are mainly wrappers for the standard plot commands or shortcuts to draw arrows, symbols etc.

`Stroke` does nothing on the screen but terminates a series of plot commands in Postscript. Curve plotting procedures need to terminate a `LineTo...` loop by calling `stroke`.

Direct EPS export is a convenient function to immediately create high resolution graphics ready for publication in a LaTeXdocument.

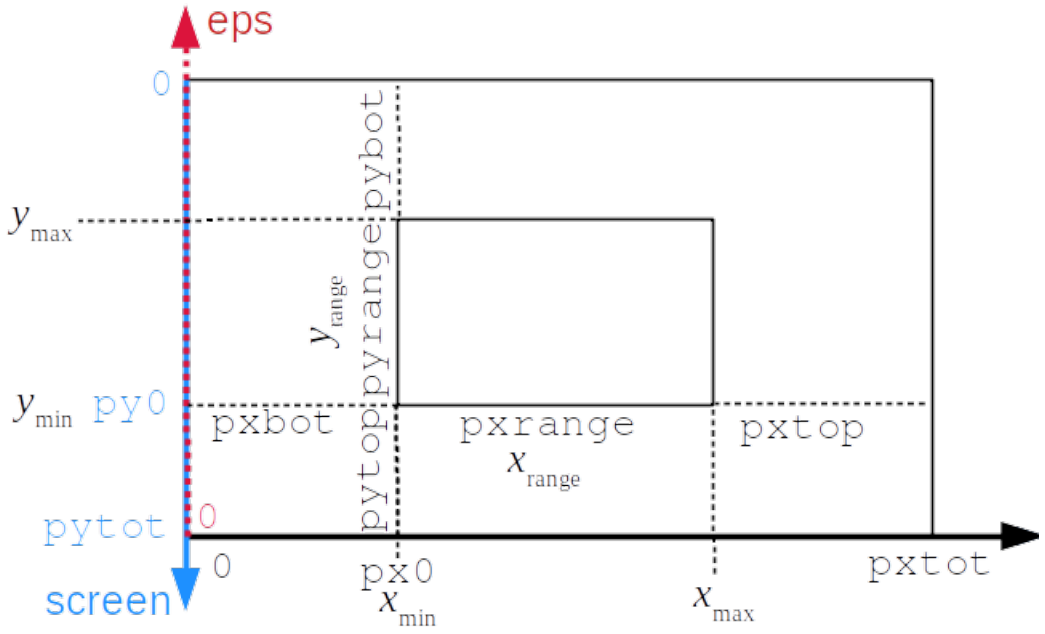

Figure 1: ../com/vgraph coordinates

3.6 Plot area

../com/asfigure

This frame contains a paintbox (i.e. a plot area) to be embedded in a GUI and a Vplot object using the paintbox canvas. Added functionality includes handles to edit fields in order to translate mouse actions to numbers.

Uses: ../com/vgraph, ../com/asaux

Public constants, types and variables

plot is a variable of class Vplot in ../com/vgraph. It is accessible to the parent GUI for direct plotting.

Public procedures

assignScreen creates Vplot and passes the paintbox canvas in order to plot on the screen.

forceMargin* set fixed margins for the plot and supresses auto-scaling by subsequent calls to Init or vgraph>Axis.

Init starts the plot. Inputs are the horizontal and vertical ranges, flags where to put axes and the descriptions of the axes, a flag to mark the origin and another flag to adjust the aspect ratio (i.e. increasing the smaller of the two ranges until the units are of same size, see vgraph>AdjustAspectRatio).

`SetSize` to be called from the parent GUI on resizing is used to change the size of this frame.

`PassEditHandle*` accepts the handle to a `TEdit` field in order to link it to the mouse position inside this frame.

`PassFormHandle` accepts a handle to the parent GUI. (?)

`GetRange` returns the range of a rectangular region in physical coordinates, which has been marked by dragging the mouse inside the plot.

`UnfreezeEdit` frees the edit fields again, when it had been frozen by an Mouse/up event (see below).

Actions

(plot area) → `pMouseDown,pMouseMove,pMouseUp` handles mouse events inside the plot area. When edit fields are linked, the physical coordinates of the plot are displayed in the fields while moving the mouse. On releasing the mouse, the edit field freezes, i.e. saves the last data and changes its color. This is used in `opatrackps` to see the particle coordinates corresponding to the cursor position.
When the mouse is dragged, the region between mouse down and up is used to rescale the plot area. This is used to zoom in in `opageometry` – This works in Windows but not in Linux, probably due to different event handling – not yet understood.

(plot area) → `pDblClick` grabs the plot image to the Clipboard on double-clicking the mouse.

3.7 Contour plot calculation

`../com/conrect`

This contour plot calculation is taken from [15] with minor adaptations. Given a 2D array of rectangular gridded data it returns an array of contour lines.

Uses: none

Public constants, types and variables

`Con...` are a set of types to store the data. `ConLinesType` contains points of a straight line belonging to an contour of height index `ih`, and `ConLinesArray` is an array of these.

Public procedures

`Conrec` is the only procedure, calculating the contours.

3.8 Utilities

../com/asaux

This unit is a collection of "little helpers" for formatting numbers and strings, for defining colors etc. Some of this functions may be obsolete meanwhile, since newer versions of Lazarus and Pascal may include corresponding functions. Procedures are short and probably self-explaining. Not all of them are used in OPA (the `../com` folder units are used by several programs).

Uses: none

4 Lattice I/O and Edit

4.1 Lattice files

latfilelib

Lattice data are read from and saved to files called `*.opa`. Reading a file copies its content into a linked list of characters (basically a long string) in `opamenu`>`ReadFile`.

There is an option to read additional files of magnet calibrations and allocations (i.e. real magnet types and names associated with the element), in order to prepare data for the machine control system.

Files from other beam dynamics programs can be read and written, however, this includes only the basic parameters and may require some manual editing to make these files work. This includes Tracy-2 or -3 files `*.lat`, Elegant files `*.ele,*.lte`, MAD files `*.mad` and MAD sequence files `*.seq`.

Uses: `globlib`, `mathlib`, `../com/asaux`

Public procedures

`Latread` evaluates the character list of type `TextBuffer`: comments, enclosed by { } brackets, are skipped. An input line is terminated by a semicolon (;), its length is unlimited and it may break over several lines in the lattice file. Input lines are evaluated by searching first for a colon (:) which identfies an element or segment, or, if not found, a variable.

A variable input line contains a name, an equal sign (=) and a value or an expression. Reserved variables are found searching for the names TITLE, ALLOCATION, CALIBRATION (containing a title of the lattice and the names of optional allocation and calibration files), and by searching in the list of keywords `globkeyw` containing ENERGY and other global parameters. Other names not matching these keywords are considered as user-defined variables.

An element or segment input line contains a name, the colon (:) and a comma-separated list of tokens.

An element is recognized if the colon is followed by a valid element type name in list elemkeyw (which is based on ElemName in globlib). Then the tokens are expected to contain each a parameter identifier, an equal sign (=) and a value or expression. There are many parameters for the different element types. The element is created and the values are assigned.

For a segment the tokens are names of elements or segments, both are assigned to a linked list of type AbstractEleType as defined in globlib.

If names of allocation or calibration files were given, these files are read now.

Finally, the lattice is built from the last segment calling globlib>MakeLattice.
LatReadCom is called after LatRead and searches the lattice string for a pair of tokens {com and com} to save the text between. All other comments contained in the lattice and enclosed in { } brackets only are not saved.
ReadAllocation and ReadCalibration read additional optional files containing lists of magnet types and real names and the magnet calibrations in order to calculate the magnet currents in opacurrents. An example for SLS is found at
https://ados.web.psi.ch/slsdesc/optic/magnets.html.

The ElementType in globlib contains a pointer to a linked list of NameListType, which contains real names of elements, name of power supply, i.e. control channel, polarity etc.
lteconvert, madconvert and madseqconvert try to make Elegant *.lte and MAD *.mad and *.seq files understandable for OPA. They work on the TextBuffer string by exchanging keywords and inserting conversion factors without analyzing the lattice itself. This is then done by a subsequent call to LatRead. Tracy *.lat and Elegant *.ele files can be digested directly by LatRead.
WriteLattice accepts a variable mode of value 0..5 to write an OPA, Tracy-2, Elegant, MAD-X, BMAD or Tracy-3 file. mode 6 writes an OPA-file with all arithmetic expressions expanded into numbers.

Different beam dynamics codes use different units (e.g. degrees or radians) and different definitions of magnet order and strength. Only OPA, Elegant and Tracy-3 handle correctly nested inversions of segments to implement an asymmetric element (e.g. dipole with different entry and exit edge angles) correctly in the lattice. Export to the other programs requires to create and insert inverted segments (prefix I_) for all segments containing asymmetric elements.

All programs mentioned above allow arithmetic expressions instead of plain values for element parameters. However, unlike the others, Elegant uses inverse Polish notation, which is not included here, therefore expressions are expanded, i.e. resolved to values, for Elegant.

Further export of data is straightforward: first variables, then elements, then segments. The complete lattice file is returned as one long string.

Type TextBuffer is a linked list of characters and has historical origin. It could be replaced by a long string, which would be much simpler.

4.2 Text Editor

opatexteditor

A simple Notepad-like text editor to edit the lattice file. The content of the edit window is one variable of type `textbuffer`, i.e. the editor "does not know" what the content means. At exit or when a test is requested, `latfilelib`>`LatRead` is used to analyze the lattice.

Uses: `globlib`, `latfilelib`

Public procedures

`Init` called from `opamenu` accepts a handle to the drop down list of segments in order to change its content when segments are created or deleted.

`LoadLattice` called from `opamenu` calls `latfilelib`>`WriteLattice` to write the lattice file into the text window `EdtWin`. Then the text window is copied into a text buffer to save its content.

Actions

(text window) → `disableOKBut` disables the **OK** button when text is changed. No other action is done on text input.

Test → `filetest` performs a test of the input by calling `latfilelib`>`LatRead`. Before, the input is copied into a buffer, then, by calling `globlib`>`PassErrLogHandle`, the output from `LatRead` is redirected from the `opamenu` main log window to the local error log window `myerrlog`, because the main GUI may be covered by the text editor GUI. If the test is successful, the **OK** button is enabled. Then the handle for output is set back to the main GUI.

OK → `fileget` reads the lattice in the same way, now knowing that the input is a valid lattice, and exits. At Exit, the segment drop down list in the mnain GUI is set to the last segment, from which the lattice was built by `latfilelib`>`LatRead`.

Cancel → `fileRestore` ignores the text window, reads the lattice from the buffer as saved at `Init` and exits.

Nowadays better text editors may be available, which include colored mark-up, auto-complete options and real time checks.

4.3 Interactive Editor

opaeditor

The "OPA Editor" allows to create elements and compose a lattice without knowing element type names and lattice file syntax. It is more suitable for beginners, while the text editor may be more convenient for experienced users.

The GUI displays two lists, one for variables and elements, the other one for segments. In addition some global parameters like beam energy and default magnet aperture may be set.

Uses: `globlib`, `oelecreate`, `oeleedit`, `osegedit`, `../com/asaux`

Public procedures

`Init` called from `opamenu` accepts a handle to the drop down list of segments in order to change its content when segments are created or deleted. Init is also called internally to update the list contents.

Actions

(Element list) → `ListBoxEClick`: if the first line of the list is licked, the object `EditElemCreate` of class `oelecreate` is initialized by passing handles to this list and to the `osegedit` object for editing segments. If another line of the list is clicked then object `EditElemSet` of class `oeleedit` is intitalized by calling two different procedures depending on if the clicked item represents a variable or an element, and a handle to this list is passed.

(Segment List) → `ListBoxSClick` initializes object `EditSegSet` of class `osegedit` by passing a handle to this list.

Beam energy → `EditGloEKeyPress`, (aperture fields) → `EditA*KeyPress`, **Magnet pole radius** → `EdrrefKeyPress` are edit fields to accept values for beam energy, for element default apertures, and for default magnet pole inscribed radius.

(Comment window) (no event) text window `MemCom` accepts a comment, which will be saved in the lattice file between the {`com...com`} tokens.

Invert all dipole polarities → `ButDipInv` inverts all dipole polarities.

Expand undulators → `ButExpUndClick` expands an undulator into a series of dipoles and drift spaces creating the elements and re-defining the undulator as segment. Usually undulator is a basic element, and the series of dipoles and drifts is executed internally and not visible to the user.

Set all apertures → `ButAllAperClick` sets all apertures to the values given in the `EditAx`, `EditAy` fields thus overwriting individual element settings.

Set all (not only if larger) (no event) if checkbox `ChkAll` is checked, apertures of all elements are set, otherwise only those which have a larger aperture than the values in the fields.

Invert rotations in inverted segments (no event): If checkbox `ChkGloRi` is ticked or not defines if a beam rotation is inverted (like an asymmetric elements) or not when a segment containing the rotations is inverted. A corresponding flag `glob.rot_inv` is set and saved in the lattice file.

Exit → `ButExit` saves data, updates the segment drop down list in `opamenu` and closes the GUI.

4.4 Element Creator

oelecreate

This small GUI pops up, when `new entry` is clicked in the `opaeditor` elements and variables list. A new element or variable is created after selecting its type and giving it a name. Then the GUI is closed and the `oeleedit` GUI pops up.

Uses: `globlib`, `oeleedit`, `osegedit`

Public procedures

`Init` accepts handles to the `opaeditor` elements list and to the segment editor `osegedit`, because if an element is created or subsequently modified by `oeleedit`, the changes should appear in these two GUIs. Then the list of element types is filled with the types defined in `globlib`.

Actions

(Kind) → `ComETypeChange` selects the type of element from the list.

(Name) → `EdENameChange` accepts the name of the element.

Create → `ButCreClick` tests the input, if a type was defined, if the new entry is a variable or an element, and adds it to the corresponding arrays. Then the GUI closes itself and launches `oeleedit` using one of the two initialization procedures for variables and elements. If it is an element, in case the segment editor `osegedit` is open, it is updated.

Cancel → `ButCanClick` closes the GUI doing nothing.

Bug: there is no check for duplicate entries, should be added.

4.5 Element Editor

oeleedit

Main component of the GUI is a table to edit all parameters of an element or a variable. Some fields may contain algebraic expressions to calculate parameters from variables. This procedure is called in four different ways, for elements and for variables, and from editor or from optics design. For (iron dominated) magnets, a possible pole-profile is plotted.

Uses: `globlib`, `linoplib`, `../com/asfigure`, `../com/asaux`

Public procedures

`InitE`, `InitEVar`, `InitO`, `InitOVar` are four different initialization procedures, for elements and variables, and for calling from the interactive editor `opaeditor` or from the linear optics

design opalinop. In edit mode, a handle to the element list in opamenu is passed, the element index is taken from the list, and the element to be edited is taken from the original element array Elem. In optics mode, the index is given explicity and adresses the Ella subset of elements in the lattice, and a handle to the plot window of opalinop is passed to enable an update of the plot when the element is changed. For variables there is only one array Variable (The arrays are defined in globlib).
Then two private procedures are called, which are common for edit and optics mode, to fill the table with data of the element or variable. For elements several lines are filled depending on the type of element, for variables it is only one line. Some element properties and variables may have a value *and* an arithmetic expression (like "A+B"), which is a string for variables and a pointer for elements. If the string is empty or the pointer is `nil`, the value is taken in edit mode. In optics mode, the calculated value from the expression is shown too ("2.5 = A+B") for elements. For variables the expression is shown in both modes.
If the element is a quadrupole or a bending magnet, the GUI is extended by a plot area of type ../com/asfigure to show its pole profile.

getJelem just returns the index of the selected element in order to enable opalinop to kill this GUI if the user assigns the element to a knob.

Actions

Done → butOKClick and **Apply** → butApplyClick both read the table and update the element. The **Apply** button is only enabled in optics mode and does not close the GUI in order to view how the change affects the optics.
The private procedure SetElem for updating the element performs several steps: In edit mode, it reads the name of the element, checks if it already exists, and, if not, if this element is used in segments, and asks, if all these entries should be renamed too. Then, in both modes, the table is read and the values are assigned to the element depending on its type. This includes conversion from convenient input units to internal SI-units. The validity of data is checked, i.e. if values are valid numbers, and if expressions can be executed without error. If everything is ok, then, in edit mode, data are written to the selected element in the Elem array and updated in the elements list of the calling opamenu GUI. In optics mode, the element in the Ella array is updated, the optics is recalculated calling linoplib>OpticReCalc, and the plot in the calling opalinop GUI is updated by sending it a `repaint` event.
For variables, the private procedure SetVar performs similar actions: In edit mode it reads the name, checks if it already exists, and, if not, it this variable is used in any expression, and asks if all these entries should be renamed too. Then in both modes, its expression is read from the table. (If the expression can be converted into a valid number, this value is assigned and the expression set to an empty string to identify a primary variable, which is a pure number.) Then, depending on the mode, the opamenu list is updated or the opalinop plot is updated.

Cancel → butCancelClick does nothing but closing the GUI.

(Parameter table) → TabDrawCell checks if input is a valid number and marks it red in case of error. This is not done in edit mode, since input may be an expression. In optics mode,

expressions as input are not editable.

ppaint reacts on repaint events for the form and plots the magnet profile, it is activated at initialization.

4.6 Segment Editor

osegedit

The GUI displays an editable string grid with names of elements and segments, which are marked in color depending on its kind and type.

Uses: globlib, ../com/asaux

Public procedures

Init passes a handle to the segment list in opaeditor and loads the abstract element sequence of the segment, which was clicked in the list, into the string grid. If entries contain an inversion flag or a multiplication factors, this is written to the grid cell too. The segment name is written to the (Modify segment) name field at top, and its periodicity (replication factor) to the (periodicity) field at bottom. If the first line of the segment list was clicked, a new segment is to be created and grid and name edit field stay empty.

Actions

(Grid) → drawCell analyzes the content of the cell and assigns a color code for element type, segment or unknown using private procedure ElemCol_C.

(Grid) → gridKeyDown reacts to pressing the CTRL+INSERT or CTRL+DELETE keys on the keyboard by inserting or deleting a grid cell, and shifting up or down the higher cells using private procedure ShiftCells.

(Modify segment) → EditSegNameChange: If the name of the segment was changed, the button to create a new segment is enabled.

update → butokClick and **create** → butcreClick call private procedure SaveSeg to read the grid cells and edit fields, and to perform several checks before creating or updating the segment:
If the grid is empty and the segment is not used by any other segment, then the segment is deleted, otherwise an error message is sent.
If the name is invalid or if the name already exists, or if the segment contains unknown names, error messages are sent.
If tests were successful, in case of (create) a new segment is appended to the list, in case of (update) the existing segment is deleted first by disposing all its abstract element pointers in globlib>ClearSeg.
If an existing segment was renamed, the entry is updated in all others segments. Then, the new segment sequence is saved as linked list of abstract elements by globlib>AppendAE. If

the segment was deleted, it is removed from the global Segm array and from the opaeditor segment list, else the list is updated. Finally the exit procedure destroys the GUI.

delete → butdelClick sets all grid cells to empty strings and performs the delete procedures for an empty segment as described above, and exits.

cancel → butcanClick does nothing but exit.

5 Linear Optics

5.1 Linear optics design

opalinop

This is the main GUI for linear optics development. It contains a plot window to show optical functions along the lattice, a table to display beam parameter results, a couple of knobs to be assigned to elements and several buttons for various options and for changing the display. Unlike all other GUIs, the plot window here is not based on ../com/asfigure, because it is too complex. All the beam dynamics calculations are done by linoplib.

Uses: knobframe, ostartmenu, otunematrix, owriteomrk, obetenvmag, omatching, linoplib, opatunediag, oeleedit, globlib, mathlib, ../com/vgraph, ../com/asaux

Public procedures

Init is called from opamenu passing a handle to a tune diagram as defined in opatunediag, which is further passed to linoplib. A Vplot object as defined in ../com/vgraph is created, the corresponding variable for the plot window named vp, however, is defined in linoplib, where the calculations take place. A handle to the parameter table tab is also passed to linoplib. Then some inital values are set for various data.
If the lattice contains variables, copies of their values are saved, and all elements containing expressions are evaluated to check if the variables are used or not in the lattice. For each variable in use, a label is created underneath the plot window. If it is a primary variable (i.e. a number), which can be edited, the label is yellow, if it is a dependant variable (i.e. an expression) the label is pale yellow.
Finally the GUI is resized based on saved user settings. This causes a repaint event of the plot area:

Actions

(GUI resize) → FormResize reacts to a resize of the GUI by the user or by the Init procedure. The GUI is resized by reserving space for buttons etc. and adjusting plot area, parameter table and labels for the variables to left over space. Then as many knobs as fit into the GUI are created. these knobs of class TKnob are defined in Since a knob may affect the

values of other knobs (in case it will be assigned to a variable), each knob receives handles to all other knob calling public procedure BrotherHandles of knobframe. If the GUI was shrinked, knobs which don't fit in anymore are freed (i.e. destroyed). Finally, the sizes of all components of the GUI are calculated and set.

(plot area) → pwpaint is not activated by the user but by the system if the paintbox of the plot area receives a generic repaint event. This is also done by opamenu after calling Init. The event calls the private procedure MakePlot, which performs these actions:
The plot is initialized, i.e. plotting axes etc. The abscissa is always the longitudinal coordinate along the lattice. Regions representing magnets in the lattice are calculated, plotted and saved to later enable reaction on mouse operations.
If MakePlot is called for the first time, the start GUI ostartmenu is launched in front of this GUI and will start the first calculation calling linoplib>OpticCalc. Finally the resulting optical functions are plotted.
This way to launch the start menu is a bit awkward, there may be a better solution.

(plot area) → pwMouseMove shows the name of an element when hovering over.

(plot area) → pwMouseDown, pwMouseUp drag an element to a knob, if the left mouse button was pressed inside an element region and released inside a knob region. If an element editor oeleedit is open for this element, it is killed. If the element is already connected to another knob, an error message is displayed.
If the element is double-clicked, the element editor oeleedit is opened in optics mode, and if there was a connection to a knob, it is released.
If the right mouse button is pressed, the optical function values at this location are shown in the lower part of the table tab.

(variable labels) → varbutMouseDown, varbutMouseUp perform the corresponding actions to a variable, if the mouse is pressed inside the region of a yellow variable label and released inside a knob region.

Start → butStarClick launches the start GUI ostartmenu again, to change initial parameters of the calculation.

PlotMode → butenvlClick launches the GUI obetenvmag to select plot of beta functions, envelopes or magnetic fields, and to set some plot parameters.

TuneMatrix → buttuneClick launches the GUI otunematrix to smoothly adjust the lattice tune within a small range by changing all quadrupoles.

Matching → butmatClick launches the GUI omatching for automatic matching of optical functions to target values by changing selected knobs (i.e. magnet strengths). Since some magnets may be connected to knobs, a handle to all knobs is passed that the matching program may set them to new values.

Write OMK → butwomkClick launches a small GUI owriteomrk to select the optics markers to overwrite with the current optics data. (An optics marker is an element to store local optics data, which may be used as a starting point for forward/backward optics calculations.)

linear → butnlinClick toggles between linear and non-linear calculation by setting the flag

UseSext, and performs the calculation. The caption of the button is changed to **nonlinear** in non-linear mode. This option has only an effect on off-axis beams.

Kicker OFF → butkickClick toggles between kickers on and off by setting the flag UsePulsed, and performs the calculation. The caption of the button is changed to **Kicker ON** if kickers are on.

->txt → butdataClick writes four text files to the working folder, which are named (opa file name)_data, _beta, _mag, _rad.txt. They contain the equilibrium beam parameters, the beta functions along the lattice, the magnet field data and another output of beta functions for radiation calculations.

->EPS → butepsClick exports the plot to an encapsulated postscript file in the workig folder, which is named (opa file name)_betas, _envel, _magfd.eps depending if betas, envelopes or magnetic fields are shown. The procedure PS_start and PS_stop are called to switch on and off the postscript mode in ../com/vgraph.

Exit → butexitClick calls globlib>EllaSave to check for changes of element parameters and to ask the user to save the changes. Then user settings are saved and the GUI is closed.

(GUI close) → FormClose closes the GUI without saving anything.

<–> → buzoominClick zooms into the plot. The six other similar buttons perform similar actions like zoom out, shift left etc.

Internally, elements are subdivided in slices of a maximum length corresponding to one pixel on the screen, so a mini-beta-focus or the dispersion oscillation inside an undular may be resolved!

∧**B(x)** → buyupClick magnifies vertically the plot of betafunctions or the horizontal envelope, if in envelope mode. In magnetic field plot mode, it has no effect. The four other similar buttons are to shrink the beta plot and to magnify/shrink the dispersion plot, and to return to automatic scaling.

save → bucsaveClick saves the plot data to a second set of curves, which then is plotted in a darker color then the current curves in order to have a visual comparison of the changes. These actions take place in linoplib.

clear → bucclearClick removes the saved curves.

5.2 Parameter knob

knobframe

This component provides a knob to control an element parameter considered as strength as defined by OPAgloal>putkval/getkval. Values may be set by slider or edit field, including range limits and a reset function. As many knobs as fit into the GUIs are dynamically embedded in optics design opalinop and orbit correction opaorbit. Knob actions trigger calculations and plots. Knobs may stay passive and only display the value, if the parameter is an expression controlled by a variable (which may be connected to another knob). Therefore a knob has to know his fellow knobs and to trigger their updates.

Uses: `linoplib`, `globlib`, `mathlib`, `../com/asaux`

Public procedures

`Init` gives a name using an index provided by the calling GUI (because it numbers the knobs) and initializes the knob as being not yet connected.

`SetSize` adjusts the knob to the size as given by the calling GUI. There is a minimum and maximum size, within this range the knob component is "elastic". If the range is exceeded, the number of knobs is adjusted by the parent GUI.

`Brotherhandles` accepts handles to the other knobs in the GUI.

`Load` and `LoadVar` connect the knob to an element or variable, accepting as input the index of the element or variable and a handle to the plot window of the parent GUI. If the parameter is an expression, the knob stays in passive mode, if it is a number, the knob is active. Element/variable name is written to the knob and range limits are set to values of same magnitude like the current value. The current value is saved. The element/variable is tagged in order to not connect it twice.

`UnLoad` removes the element/variable tags and sets back all captions to the initial status of being not connected.

`KUpdate` sets the knob to a new value by adjusting edit field and slider, and, if exceeded, also the range limits. If the knob was updated manually, private procedure `Action` is called to trigger calculations and plots and, if needed, to update the other knobs. If the knob is updated in this way from another plot, no further action takes place.
If the knbos are components of `opalinop` the *optics* is re-calculated and the plot is updated depending which mode (betas, envelopes, magnetic field) is set. If the knob is component of `opaorbit` the *orbit* is re-calculated and the plot is updated.
If the knob is connected to a variable, changing it may affect other knobs connected to elements or variables using *this* variable in an expression. So all elements' expressions are evaluated to update their values, and, if they are connected to another knob, the `KUpdate` procedure is called for the *other* knob (therefore it's not a circular reference). Of course, this affects only knobs which are in passive mode. This feature is yet only implemented for optics mode, since in orbit/injection mode no expressions are allowed for corrector magnets and kickers, which are the only elements to be connected. Thus it wasn't needed, however this is an uneccessary restriction.

`getella` and `getvar` return the index of the connected element/variable.

Actions

(value field) → `editKKeyPress` calls `KUpdate` to perform actions and adjust the range if the input value exceeds it.

(slider) → `sliderScroll` only performs actions, since the slider cannot exceed the given range.

(min/max fields) → editmin/maxKeyPress as well as >< → butwidClick and <> → butnarClick adjust the range limits by setting minimum/maximum or widening/narrowing the range, all calling private procedure SetKrange, which sets the small range fields and the slider parameters.

Reset → butresClick resets the knob to the value it had when connecting it and performs the actions.

X → butfreeClick disconnects the knob calling Unload.

5.3 Start parameters

ostartmenu

This GUI pops up on initialization of the optics module opalinop or the orbit/injection module opaorbit (see sec.8), or when pressing the **Start** buttons in these modules. The starting conditions for the optics calculation are to be selected, either periodic/symmetric, or forward/backward from start/end of lattice or from one of the optics markers. In case of forward/backward calculation the initial beam parameters may be entered manually, in orbit/injecition mode this is only the orbit. In optics mode it includes also the normal mode beta functions, the dispersions and the elements of the coupling matrix.

Uses: linoplib, globlib, mathlib, ../com/asaux

Public procedures

Load accepts a number for the mode of operation, which is 0,1,2 for optics, orbit, injection, and a handle to the parent form, which is either opalinop or opaorbit. Depending on the mode radio buttons to select periodic/symmetric or forward/backward options are enabled or disabled (per/symm makes no sense in injection mode). Then the lattice is searched for optics marker and corresponding radio buttons are added to the GUI. In optics mode a check box to perform calculations with or without coupling becomes visible and is checked at start if the lattice contains coupling elements. Finally the panels to enter/edit initial beam parameters are enabled/disabled and filled with data calling the private procedure set_pan_ini.

Exit is called from the parent GUI (opalinop or opaorbit) on exit to close this GUI too if it is still open.

Actions

(radio buttons) → rbutClick enables the tables to edit initial parameters depending on the selection of periodic/symmetric or forward/backward calculation, calling the private procedure set_pan_ini, and enables the buttons to start the calculations.

(periodic solution) → rbutperChange, this radio button has an additional event to hide/unhide the **flip** check box to start with flipped solution in case of coupling. (Further a button **pp** appears, which is a temporary test only.)

coupling → chk_coupChange hides/unhides the **flip** check box.

dp/p[%]= → but_dppClick unhides an edit field to enter a value for momentum offset, and a check box to select if one calculation for $Deltap/p$ or three calculations for $0, \pm\Delta p/p$ are to be done.

Apply → butapplyClick (or **Close** → butcloClick) starts the calculation calling private procedure Go (and closes this GUI). Before starting the calculation several flages are set depending on the mode (optics with or without coupling, orbit, injection) and the initial conditions. Afterwards, the plot procedures linoplib>OpticCalc or linoplib>OrbitCalc are called.

Exit → butexitClick closes this GUI and the parent GUI (opalinop or opaorbit).

(Coupling panel) → ed_couKeyPress,ed_couExit check the input into the edit fields of the coupling matrix (rarely used).

5.4 Optics plot mode selection

obetenvmag

This GUI is launched by opalinop to select what to show: beta functions (normal mode and/or projected) and dispersions, envelopes with orbit and apertures or magnetic fields. Additional parameters may be set.

Uses: globlib, mathlib, ../com/asaux

Public procedures

Load sets GUI components (check boxes, fields etc.) for the current plot settings (explained below) and saves initial settings in order to restore them later.

Actions

(Beta, Envelope, Mag. field radio buttons) → rmodClick selects one of the three plot modes (Betas, Envelopes, Magnetic fields), unhides the corresponding panel and hides the two other, and enables options depending on flags: one flag is coupling, another flag is for using equilibrium emittances or input emittances. Further the data table on the right side of opalinop is configured, and table and plot are updated calling procedures linoplib>InitBetaTab,FillBetaTab. Then the private procedure MakePlot>sets plot ranges etc. and calls the corresponding plot procedures linoplib>PlotBeta,EnvPlot,MagPLot.

Apply → butgoClick also calls for a new plot and updates the data table.

Close → butcanClick closes the GUI.

Beta and Dispersion panel:

Fix plot range → cbxbetamaxClick enables the fields for max. beta and dispersion if checked. Then, **Apply** will read the edit fields **max. betafunction** and **max.dispersion** and fix the plot range. Otherwise the plot wll auto-scale.

normalmode → chk_betabChange and **projected** → chk_betxyChange select to show normal mode and/or projected beta functions. Without coupling this is the same. (The third option **n.mode one-turn** is only a test.)

Dispersion → rbu_dspChange, **det(C)** → rbu_cdetChange, **Orbit** → rbu_orbiChange radio buttons select to show the dispersion functions, the determinant of the coupling matrix or the orbit.

Envelopes panel:

use equilibrium / use input values → rbuClick radio buttons select to use equilibrium emittances and energy spread or manual input values for envelope calculation. In the equilibrium case, if the lattice is uncoupled, the field **emittance coupling** is enabled, otherwise both emittances from the coupled calculation are used. In the input case, three fields **horizontal emittance**, **vertical emittance** and **rms energy spread** are enabled.

Magnet field panel:

Fix plot range → chk_bfieldfixClick enables the fields for min. and max. field if checked. Otherwise the plot wll auto-scale. **Apply** will read the **Reference radius** field for pole-tip field calculation, and the two edit fields **min. B[T]** and **max.B[T]** to set the plot range.

5.5 Tune matching

otunematrix

This small GUI launched by opalinop calculates a 2×2 sensitivity matrix how the lattice tunes depend on a scaling factor applied to the two groups of all horizontally and all vertically focusing quads. A new tune may be entered manually, and the inverse of the sensitivity matrix is then used to adjust all quads correspondingly. This procedure converges for small tune changes and was also used in SLS control by exporting the sensitivity matrix to EPICS channels from module opacurrents. The code is simple and may not need further explanation.

Uses: linoplib, globlib, mathlib, ../com/asaux

5.6 Optics marker update

owriteomrk

Optics markers are markers in the lattice which can store beam data (betas etc.) and may be used as starting points for forward/backward calculations by ostartmenu. This GUI establishes a list of all optics markers in the lattice with tick marks and asks if their data should be overwritten with the current optics solution. This may be ambiguos since an optics

marker may appear several times in a lattice, so the data refer to the first instance, counting from begin of lattice. The code is simple and may not need further explanation.

Uses: globlib, ../com/asaux

5.7 Lattice calculations

linoplib

Linear beam optics includes closed orbit finder, periodic solution for coupled and uncoupled lattices, and radiation integrals. This unit has handles to the plot areas and tables of the optics and orbit/injection GUIs, opalinop and opaorbit. Plot routines show beta functions and dispersions, or orbit, envelopes and apertures, or magnetic fields. Lattice and local beam parameters are filled into the tables, and several output files can be printed.

Uses: elemlib, opatunediag, globlib, mathlib, ../com/vgraph, ../com/asaux

Public constants, types and variables

Several variables used by opalinop, ostartmenu etc. are defined in the interface section, i.e. as public, for convenience (i.e. lazyness). Most of them are flags to control the flow. Few of them may require an explanation:

vp defines a vgraph object for the optics GUI opalinop, which creates it at start. Later, linoplib itself will plot to vp. The orbit/injection GUI opaorbit, however, defines its own plot areas of class asfigure and only gets the data from linoplib. The reason for this different solutions was partly historical, partly due to the higher complexity of the opalinop plot.

Opval is a dynamic 2-dimensional array of OpvalType to save optics parameters after each lattice element and for up to three calculations, e.g. for on- and ±off-momentum.

Obeam is a 3-element array to save beam parameters like tunes, emittances etc. for up to three calculations.

Public procedures

allocOpval allocates the array of optics values.

shiftOpval shifts up the optics values for one solution the next row in the lattice; this is used to save a solution for comparison with a newer one.

setTabHandle receives a handle to the table in opalinop in order to fill in optics results

setTunePlotHandle receives a handle to the tune diagram GUI opatunediag in order to plot working points.

setOrbitHandle receives a handle to the opaorbit GUI. Actually it is only needed to send it a repaint event, it will then get and plot the data from linoplib.

Beam dynamics procedures:

OptInit initializes all variables (beta functions, transfer matrix etc.) to be used for the optics calculations.

Lattel is the basic procedure for all calculations. It propagates the optical functions through one lattice element. Input is the lattice index and a mode flag telling it what to include in the calculation (integrals, misaligmnents etc).

ClosedOrbit tries to find the fixpoint of the one-turn map, i.e. the closed orbit, by a Newton-Raphson root finder with the *local* transfermatrix as Jacobian. It returns a flag to tell if it failed or not. For on-momentum calculation without misalignments, it is trivial and the orbit is on-axis. Else down-feeds from non-linearities affect the local transfermatrix and require some iterations, usually only a few since the method converges quadratically.

Periodic tries to find the periodic optics solution, if the closed orbit was found. Depending on coupling it switches to two procedures:

Flatperiodic calculates the periodic solution for uncoupled, i.e. flat lattices without coupling elements, where horizontal and vertical dimension can be calculated independently, and vertical dispersion is always zero. The algorithm is described in Klaus Wille's book [7].

NormalMode calculates the periodic solution for coupled lattices using the Edward-Teng formalism in the version of Sagan and Rubin and implemented as described in [6]. Section 2 of the "inside OPA" guide [3] gives a summary. If successful the procedure returns the normal mode beta functions, which have no physical meaning and exist in an uncoupled system (coordinates a, b) and the coupling matrix, which translates the normal mode system to the real coordinate system (coordinates x, y). It further returns the 4-vector of dispersions and the lattice tunes.

Symmetric calculates the symmetric solution with $\alpha_x = \alpha_y = D'_x = 0$ and works only for uncoupled lattices. It is rarely used and not well tested. The algorithm is also described in [7].

LinOp is a *private* procedure to propagate the beta functions through the lattice and save the values after each element to the Opval array by calling procedure StoVar. Usually it would start at the begin of the lattice with initial conditions from the periodic solution or entered manually. But it may also start at an optics marker and propagate forward and backward to end and start of lattice. Care is taken to handle inverted elements and derivatives of optical parameter and to sum up integrals correctly. Depending on the input flag `latmode` only the beta functions and dispersions are calculated or chromaticities and radiation integrals are calculated too by passing the flag to the Lattel routine, which passes it further to the element propagation routines.

OpticCalc performs the full linear optics calculation by calling ClosedOrbit, Periodic and LinOp. Afterwards, the working point is plotted to the tune diagram (in case of periodic solution only) and the table in `opalinop` is filled calling procedures FillBeamTab and FillBetaTab.
The code is a bit messy due to the different cases it has to handle – periodic/symmetric solution or forward/backward from an optics marker, how many calculations to do, on- or off-momentum.

OrbitCalc is a simplified version of OpticCalc used by opaorbit for calculation of orbit only. An additional feature is the ability to do several turns which is useful in injection simulation.

MatchValues is a function returning an array of type globlib:MatchFuncType containing optics values at the begin and end of the lattice, and optionally, at an intermediate point. It is used by omatching for matching optics parameters to target values.

Plot procedures:

SliceCalcN calculates optical functions *inside* an element by subdividing it into slices of a length corresponding to one pixel on the screen. Calculations start at each lattice element, where optical functions have been before calculated by Linop and saved in the Opval array. Elements outside the plot window are skipped, if the user zoomed in to a part of the lattice in opalinop. The variable globlib:CurvePlot stores the start and end point and the number of slices and their length, and the optical functions at the slices are saved in the variable globlib:Curve, which is a linked list of pointers for storing optical data. Procedure elemlib:SlicingN performs these calculation and generates the curve.
It's a unique feature of OPA to display the optical functions at pixel resolution. This allows to export nice plots ready for publication.

PlotBeta displays the beta functions and dispersions. The procedure is lengthy due to the various plot modes as selected in obetenvmag, but in principle it is relatively simple: After setting the ranges, the internal procedure BetaCurveN is called to generate the curves calling SliceCalcN and to plot them. More than one curve for each function is plotted if a previous curve was saved for comparison, or if a $0, \pm\Delta p/p$ calculations was done.

PlotEnv displays the envelope functions. Before, they need to be calculated. Therefore several arrays are created to hold all the data for the orbit and the different contributions to the beam size (from dispersion, from coupling). They are calculated either using input or equilibrium emittances and energy spread as selected in obetenvmag. Also the apertures are plotted calling procedure PlotApertures.

PlotMag displays the pole-tip field of the magnets. No slicing is done since the magnets have no longitudinal variation of the fields (longitudinal gradient bends are approximated by a stack of dipoles). In case of the solenoid, the longitudinal field is shown. For combined function magnets only dipole and quadrupole field is used for pole-tip field calculation. The reference radius as set in obetenvmag is assumed to be the same for all magnets.

PlotOrbit is used by the orbit & injection GUI opaorbit only, whereas the other plot procedures are only used by the optics GUI opalinop. opaorbit does the plotting by itself, so this plot procedure just sends a repaint event to its formhandle.

Output

FillBeamTab and FillBetaTab write equilibrium beam parameters and beta functions at the location which was right-clicked (see above: opalinop action (plot area) → pwMouseDown) to the table on the right side of opalinop. In envelope mode, beam sizes, correlations and beam tilt angles are calculated and displayed.

Print... These procedures print various data to files: lattice data in text or html format, beta functoins curves, magnet and radiation data.

5.8 Element calculations

elemlib

This unit contains the procedures to propagate optical parameters through the different elements in the lattice. They all receive the variable mode telling them what to include in the calculation, by evaluating globlib>switch. Most elements receive the varialbe idir to set forward or reverse orientation of the element (for example important for a dipole with different edge angles). All other parameters are explained in the user guide [2]. The flags UseSext,UsePulsed in globlib switch on/off non-linearities and kickers. Coupled calculation is selected by the element procedures. Another set of procedures performs the sliced calculations for thick elements to provide data in pixel-resolution for a nice display.

Uses: globlib, mathlib

Procedures

This section tries to list the procedures in a logical order (not as they apppear in the file) for best understanding. Some are public, to be called by linoplib, others are private and commented out in the INTERFACE section of the file.

Beam propagation

Propagate receives the transfermatrix of an element and a flag on coupling to propagate the optical functions from start to end of the element using two procedures:

MCC_prop proceeds *with* coupling to propagate the normal mode beta functions and the coupling matrix. In the beginning, on- and off-diagonal 2×2 submatrices are extracted from the 5×5 transfer matrix, and a test is done, if a mode flip will happen. Then follows a longer section to handle a possible mode flip. This section is partly experimental and needs further elaboration. It does not work correctly with regard to tunes and integrals in all cases. At the end follows the normal case which works reliably with weak coupling. Phase advances are calculated by procedure PhaseAdvance and added to the ring tunes, and the dispersion vector is propagated, which always works.

MBD_prop proceeds *without* coupling where the transfer matrix is known to be block-diagonal. Nevertheless, possible coupling in the beam has to be propagated too by transforming the coupling matrix, which may have non-zero elements in this case.

PropForward just copies optics parameters after propagation (suffix 2) to parameters before propagation (suffix 1) in order to continue with the next element.

PhaseAdvance calculates the block-diagonal normalized normal mode matrix from the local beta functions using the transformation matrix from function CirMat, and extracts the phase advances from the traces of the sub-matrices. getdTune is a simplified version for a transfer matrix which is known to be block-diagonal.

GetBeta12 calculates the projections of the normal mode beta functions to the physical beta functions.

Pathlength calculates the path length by estimating the second order matrix elements as defined in TRANSPORT (see comment in code for details).

The elements

The ..._Matrix functions return 5×5 transfer matrices of the basic linear elements. Since OPA does not include longitudinal dynamics, there are no rows and columns for phase/time and the 55-element is always 1. The fifth column of the matrix contains the dispersion production vector. Drift space, quadrupole, sector magnet and edge kick have block-diagonal matrices, i.e. they don't couple. Coupling elements are the explicit rotation and the solenoid. The Bending magnet matrix is a sector matrix sandwiched between edge kicks. The matrix functions are called by the element procedures. OPA knows 17 different elements as listed at the very beginning of `globlib`:

There are no procedures for markers, optics markers and photon beam markers, because they don't act on beam dynamics. There is a placeholder procedure Monitor for beam position monitors but it does nothing. The septum element –as seen from the stored beam – is handled as a drift space in beam dynamics.

DriftSpace, Quadrupole and Bending are the standard linear lattice elements. Actually Bending switches to Quadrupole if the bend angle is zero, and Quadrupole switches to Driftspace if the gradient is zero. So Bending is the most complex procedure and should be explained in some detail. It performs the following steps: at first the magnet is misaliged, if misalignments were requested, then a rotation is applied if the magnet is tilted. Depending on the direction of the magnet, the edge angles are interchanged and the edge kick matrices are calculated. Then the beam (i.e. the optical functions) is propagated through the entry edge because the values at the beginning of the sector are needed to estimate the phase advance inside the sector (procedure getdtune) in order to set the right number of points for numerical evaluation of the radiation integrals. Before doing so, the complete bending magnet matrix including entry edge, sector and exit edge is calculated twice, first on-axis (TM0) as needed for the closed orbit finder, then off-axis (TM) to include dispersion downfeeds for the integrals. If radiation integrals or chromaticities are requested, the sector is split in a number of slices, the beam is propagated step-wise, and the integrals are obtained by Simpson's rule (see section 3 in [3] for details, also for the other elements). Finally, the beam is propagated through the complete bending magnet, and rotation and misalignments are undone at exit. Quadrupole contains a simplified version of this algorithm, but including radiation integrals too, since there may be off-axis contributions.

Combined is a bending magnet which also has a sextupole component. It is modelled as a series of bending magnets with thin sextupole kicks between. Having both Combined and Bending has historical reasons, and maintaining both is for backward compatibility, but in principole they could be united.

Undulator contains a loop over rectangular bends and drift spaces, where the parameters of these sub-elements are set by the procedure UnduFacs. For the radiation integrals a simplified algorithm using averages is applied since the bends inside the array are small.

Solenoid is a simplified treatment of the solenoid providing beam rotation and focusing in both planes. Note, that the transverse vector potential components of the solenoid are not

taken into account, also contributions to chromaticity and radiation are not considered.

Rotation is an explicit beam rotation. It's rarely used, because rotated elements like skew quadrupoles are rather set up by passing a rotation angle to the corresponding element procedure. Then the element matrix internally is sandwiched between the rotation and its inverse.

ThinSextupole and Multipole are thin multipole kicks – basically, a sextupole is just a multipole of order 3, but the dedicated procedure executes faster. The procedures apply a non-linear kick to the orbit and a transfer matrix to the optics for off-axis gradient downfeed. (For order 2, quadrupole, there is a gradient on-axis too.) Contributions to radiation integrals are usually negligible and therefore not included. The Multipole procedure uses explicit expressions up to octupole (order $n = 4$). For higher orders the general complex expression is evaluated (see comment in code). A vertical offset of the beam requires coupled treatment due to skew quad down-feed.

Sextupole is a thick sextupole, internally modelled as a series of drifts and thin sextupole kicks as the most simple second order symplectic integrator. A corresponding procedure for a thick general multipole does not exist yet.

Kicker is a time-dependent thick multipole. Parameters delay and period (of half sine bump) and the time of flight from the start of the lattice define the kick strength applied. Like the thick sextupole, the kicker is subdivided in a series of drifts and thin kicks. The transverse shape of the kick can be selected to be of multipole or sine shape (BESSY type NLK). Note that for the NLK, the kicks are only correct in the midplane ($y = 0$).

HCorr and VCorr are correctors, i.e. weak dipoles which just give a kick to the orbit and to the dispersion function without further effect on the beam optics.

Slicing

The SliceSet function calculates the number of slices for the optical functions inside an element to be displayed in one-pixel resolution on the screen. The record globlib:CurvePlot contains a field slice for the length of the slice in physical coordinates, i.e. in meters, calculated by opalinop>Zoom. The SliceSet function checks if the element is outside, partially inside or completey inside the current plot window, or if the plot window is completely inside the element. It writes start and end position inside the element and the number and length of slices to the CurvePlot record and returns true if at least a part of the element will be visible in the plot area.

SlicingN receives transfer matrices msl for a slice, mof for the invisible part of the element left of the plot window, and min and mex for edge focusing at entry and exit of the element (applies to bends only). Additional matrices mof0,msl0 without dispersion down-feeds are supplied to show both orbit and dispersion functions correctly. The procedure further receives the optics values at the entry to the element, because these are known from the previous run of linoplib>Lattel. Then optical parameters at the slices are calculateed using a simplified propagation procedure and the results are appended to the globlib:Curve variable for later plotting by linoplib. Finally the optics value at the end of the calculation (which may be still inside the element) overwrite the optics values from input.

SkippingN is only needed for undulators, which are modelled as a loop over drifts and bends, and where several of these basic elements may be left of the plot window. The optics values are propagated through these invisible basic elements to get the correct initial values at the first partially visible undulator pole. In this wasy it is possible to zoom into an undulator and see the dispersion oscillation!

The Slice(Drift,Quad,Sol,Bend)N procedures call SliceSet and calculate the required transfermatrices for SlicingN and SkippingN.

SliceUnduN loops over bends and drifts of the magnet array if at least a part of the undulator is inside the plot window.

SextKickN is a simplified version of ThinSextupole to propagate optics values through a sextupole kick.

SliceSextN and SliceCombN make use of SextKickN because they model thick sextupole and combined function magnet as loop over drifts or bends with sextupole kicks between.

5.9 Linar matching

omatching

Matching implements a linear equation solver based on LU decomposition for the linear optics. For historical reasons, it does not use the solver in mathlib>LUDCMP,LUBKSB but the algorithm was explicitly coded in the 1980's by Klaus Wille (procedure Step) , and is documented in his book [7]. Matching is done from a start point to an end point in the lattice. The lattice tunes and optics functions at an additional intermediate point may be included too. After selecting these options, a list of available knobs is established. Then the user selects the constraints, i.e. the optics parameters at end (and intermediate) point, sets the target values and selects at least the same number of knobs (e.g. quadrupole strength, variable etc.) from the list. The solver proceeds using the square matrix of most sensitive knobs, which is recalculated after each step to take into account non-linearities. If successful, the method converges quadratically, if not it returns messages, what went wrong (procedure Term). Reduced step size can be set for slower but more robust convergence. It is possible to run a scan over some range of target values.
There are three panels at the right side of the GUI, one for the knobs, one for the matching process and one for parameter scans. Only one of them is visible at a time.
The algorithm itself with catches for errors as implemented over the years, is quite robust and could be used further in principle. However, nowadays better and well proven solvers may be available, which also include limit ranges for the knobs. Further, the present procedure is restricted to uncoupled lattices and to only one intermediate point besides start and end of matching, which sometimes is a nuisance. The present GUI is well conistent (how options are enabled etc.) but its rigid structure makes extensions cumbersome. It should be re-organized for example using tabs.
Expecting a rewrite of this module in future, only a brief summary of the most important actions will be given here.

Uses: linoplib, knobframe, omatchscan, globlib, ../com/asfigure, ../com/asaux

Public procedures

Load initializes the GUI. It receives handles to the PaintBox plot window and to the array of knobframe knobs in opalinop in order to be able to set the knobs to new values and to show the new optical functions after matching. Then it searches for optis markers, which may be used as start, end or intermediate point for matching, and adds them to the ini/mat/midpoint pulldown lists. Several user settings are restored from a previous session, since often the same matching scenario is used again.

Actions

starting,matching,intermediate → (ini,mat,mid)pointChange call the private procedure knoblist to establishes the list of available knobs after selecting the start, end (and optionally intermediate) points for matching. For each knob a named check box is created.

Optics parameter check boxes → cselClick and **Knob check boxes** → knobSelect build the lists of constraints and knobs and set the **Go** button if there are at least as many knobs as constraints.

incl.bends → cbxIncBendsClick includes bending magnets in the knob list, because often they are not used, so the panel is less crowded if they are not shown.

Go → gobutClick hides the knob panelpanknobs and shows the matching panel panmat with a grid of knob values. Then the iteration starts until the procedure Step returns true because it terminated with success or failure, or the user stopped the iteration pressing **Abort** → bustopClick. The last option requires the Pascal procedure Application.ProcessMessages, which reacts to GUI events, to be called inside the loop. After termination buttons become active to show the result in opalinop, to accept the new values, to reset to initial values or to retry the calculation (for example with reduced step size to be entered in the **Fraction** [%] field, in case of bad convergence.)

Show → butShowClick actually only hides the omatching GUI (which is modal!) to uncover the opalinop GUI which is behind and shows the current optical functions anyway.

Accept → butaccClick saves the settings and closes omatching.

Reset → butaccresClick restores the old knob values, recalculates the optics, saves the scenario and closes the GUI.

Retry → butrtyClick restores the old knob values and sets up for a new iteration.

Scan → ScanClick prepares for a parameter scan, i.e. a series of matching iterations for a range of target values. For this purpose, the scan panel panfig becomes visible, the modal GUI omatchscan is launched to enter range parameters, and then the matching series are started, the results are saved and plotted eventually. The <, > buttons under the plot window allow to toggle between displays for the knobs and functions.

5.10 Parameter scan

omatchscan

This very simple modal GUI is launched by **omatching** → ScanClick that the user may enter the range for a parameter scan. At exit it writes the data to the variable linoplib:ScanPar, from where omatching takes them to perform the scan.

Uses: linoplib, globlib, ../com/asaux

5.11 Tune diagram

opatunediag

The tune diagram GUI is created at start of OPA already and provides the variable tuneplot, which is passed in the initialization procedures of the modules using it: opalinop shows the working point, i.e. the tunes of a periodic solution. opamomentum shows the variation of tunes with momentum. opachroma shows an analytical estimate of the tune footprint. And opatrackps shows the amplitude dependent tune variation. In the background always shown is the web of resonance lines.

Uses: globlib, ../com/asfigure, ../com/asaux

Public procedures

Diagram makes the tune diagram visible if a periodic solution exists. Input is the point in tune space where to center the diagram. The resonance lines inside the visible tune region up to the selected order are calculated by private procedure getLines and characterized as regular, skew, non-systematic. The periodicity of the lattice is stored in globlib:NPer and determines the selection of systematic resonances.

AddTunePoint adds a tune point to the diagram. If the input flag connect is `true` the point is connected to the previous point. The inputs dpp, dppmax are used to show off-momentum points in a color range. Used by opalinop and opamomentum.

AddChromLine adds a line to show the chromaticity, i.e. the variation of tune with momentum, with chroomaticities up to third order and the momentum range as input. Used by opachroma.

AddTushPoint adds points from amplitude dependant tune shift tracking. Input xmy controls the color of the points: horizontal, coupled, vertical in blue, purple, red; outside aperture in grey. Used by opatrackps.

Refresh clears the diagram from points which had been added.

Actions

(On create) → FormCreate sets defaults and restores user settings.

(On resize) → FormResize adjusts the GUI if the user changes its size.
All other actions control the plot: The buttons (Order) + − increase/decrease the maximum resonance order. The **nsys** and **skew** check boxes switch on/off non-systematic and skew resonances. The buttons under the plot window are to zoom in/out, to shift the region right/left, up/down and to re-center. **EPS** exports the diagram.

6 Longitudinal optics

OPA does not contain true longitudinal dynamics, i.e. all calculations are for fixed momentum offset, and tracking proceeds only in 4-D, not in 6-D. There are no cavities and no acceleration.

6.1 Momentum dependence

opamomentum

This GUI shows optical functions at lattice start and lattice integrals like tunes, emittance etc. for some momentum range. This functionality is the same like performed in opalinop/linoplib: for periodic lattices closed orbit and periodic solution are calculated, then for any lattice, the initial parameters are propagated and the integrals are calculated. The plotted curves of momentum dependent parameters are then fitted by polynomials. In case of pathlength, coefficients are saved internally to be used later in the bucket viewer opabucket.
For the particular problem to design a non-linear bunch-compressor for an FEL [10], the module once was extended to set target values for polynomial coefficients and allocate non-linear elements to align the polynomial to the target by means of a Powell minimzer. This feature is never used for synchrotrons and storage rings, where the side-effects of this brute force alignment are devastating with regard to non-linear dynamics. Since this extension is irrelevant, improvised and fragile, it should be removed – however, this requires some disentangling.
Therefore only the first part of the GUI is briefly explained here.

Uses: momentumlib, linoplib, opatunediag, globlib, mathlib, ../com/asfigure, ../com/asaux

Public procedures

Start called from opamenu receives a handle to the tune diagram. Then it assigns handles defined in momentumlib to some of its GUI components to give momentumlib access. Then user settings are restored, and some initialization is done, and the GUI is resized.

Actions

(On create) → FormCreate builds the GUI by dynamically creating its components. These include radio button for the 20 different availabe plots as defined in the header of momentumlib. More components, which have the panmin panel as parent, are for the minimizer and to be removed.

(On resize) → FormResize performs a dynamic resize if the user changes the size of the GUI, distributing evenly the available space.

(On paint) → FormPaint calls momentumlib>MakePlot

periodic → chkPerClick sets the optics mode to periodic. Of course, a periodic solution may not exist for all values in the momentum range. These failed points will be omitted in the plot.

Go → butGoClick performs the calculation calling momentumlib>FullCalc. It also plots the result for the currently selected plotmode, including calculation and display of the fitted polygon. Polynomial coefficients are listed in table gfit.

Fit → butFitClick repeats fit and plot with the selected fit parameters.

Units for table → butfunitClick toogles between showing the polynomial coefficients in table gfit in SI-units or in convenient units as used for the plot annotations.

(Radio buttons) → rbClick sets the plot mode according to the clicked radiobutton and repeats fit and plot.

Events not mentioned are either trivial (e.g. check for valid number in edit fields), or they belong to the (working but irrelevant) Powell minimizer, which is started by the **optimize** button.

6.2 Calculations

momentumlib

This unit contains the physics and plotting for the opamomentum GUI.

Uses: linoplib, opatunediag, globlib, mathlib, ../com/asfigure, ../com/asaux

Public constants, types and variables

The header contains parameters for the 20 plot modes: color of plot, number format for table, annotation of radio buttons in opamomentum etc. Some plots show two functions, for example horizontal and vertical tune shift. The arrays ires1,2 control, what to plot. ires2=-1 suppresses the second curve.

result is an array of a 20-element array to store the results over the momentum range. The size of the array depends on the number of momentum values to be calculated.

The ..._HANDLE variables refer to components of the opamomentum GUI.

res_funit contains factors to switch between SI-units and plot units in the polynomial coefficients table.

All the variables containing "op_" or "kn_" or "Pen" in the name belong to the obsolete minimizer.

Public procedures

Makeplot shows the selected results. Valid points are plotted as symbols. The internal procedure fit_and_plot performms a polynomial fit of the requested order over the valid points, shows the fit result as (an) additional line(s) in the plot, and writes the fit coefficients to the table. The internal procedure plot_qclines adds the predictions for the chromaticities from the non-linear optimizer opachroma if available, and if tunes were selected for plotting.

FullCalc contains the complete calculation:
PreCalc prepares for the calculation by setting up the momentum values. In case for periodic solution requested, it first checks the on-momentum solution. If successful, the tune diagram opatunediag is launched and the working point is plotted. If chromaticity data are available from a previous run of opachroma, they are also shown in the tune diagram as a line. BUG: diagram appears but no plot, area stays grey.
Calculate loops over the momentum range, calculates the optics parameters and integrals for each momentum value, and stores the results in the result array if valid.
Afterwards MakePLot is called.
PostCalc finally plots results to the tune diagram if available to show the chromatic tune shift.

Procedures CalcPenalty, ShowPenalty, CalcTarg are obsolete.

6.3 RF bucket viewer

opabucket

The RF bucket is calculated based on up to five orders of momentum compaction and RF harmonics as described in [3]. A contour plot of equipotentials and the separatrix are displayed, and momentum acceptance and the bunch length are estimated.

The coefficients of momentum compaction may be taken over from a previous calculation of momentum dependent optics, opamomentum. The estimates for momentum acceptance and bunch length are saved to be used later in Touschek tracking opatracktt.

For "playing" with the bucket, the viewer may also be used with manual input for cavity voltages and momentum compaction, even without any lattice. Therefore the option is always enabled in opamenu.

Uses: globlib, mathlib, ../com/conrect, ../com/asfigure, ../com/asaux

Actions

(On show) → FormShow performs all initializations when the GUI becomes visible: private procedure Layout is called to arrange the components (although this GUI does not react to resize by user), user settings are loaded and filled into the edit fields. With this settings, the bucket is calculated immediately and shown by calling private CalcBucket and MakePlot:

CalcBucket performs all the calculations as described in [3]. This also includes the call to ../com/concrect>conrec to get the equipotential contours, because from these the energy acceptance and bunch length are estimated. However, this requires the central fixpoint to be elliptic (i.e. no overstretched bunches). And bunch length can be estimated only if there is a value for energy spread from a previous optics run.

MakePlot just plots the results and writes them to labels in the GUI.

(On paint) → FormPaint just calls MakePlot again.

Show bucket → butconClick calls CalcBucket and MakePlot again.

EPS → butepsClick sets mode to postscript export and calls MakePlot to write file (lattice name)_bucket.eps.

mode → butmodClick toggles between "lattice" and "play" mode, i.e using momentum compaction coefficients from the lattice or from manual input.

use Lattice → butcopyClick copies the lattice coefficients to the edit fields for manual modification in "play" mode. This knob and the **mode** knob are only visible if an optics calculation had been done before.

Exit → butexClick saves the user settings and closes the GUI.

7 Non-linear Optimization

The units of this section are linked in a rather complex way as indicated by the uses declarations, since the main GUI opachroma contains many frames for knobs, csexframe, and for results, chamframe, which affect each other. This required the two small units chromreslib, chromreslib, to pass the handles. csexframe declares use of chromreslib, which declares use of chamframe. Vice versa, chamframe declares use of csexframe in the implementation section, i.e. at runtime, not in the interface section, to avoid a circular reference.
There may be a better way to declare these dependencies.

OPA's non-linear optimizer is a rather useful tool to get a quick optimization of dynamic aperture – or to help in understanding why it can't work. Manually playing with the multipoles may give a "feeling" by visualizing the effect on the Hamiltonian modes and the tune footprint, before the automatic minimizer is started. The code executes fast, since all coefficients to sextupole family strengths and products of two families, are calculated only once at start. Multiple periods are realized by a complex factor derived for this purpose [3].

7.1 Non-linear optimization panel

opachroma

The GUI contains a fixed set of first and second order sextupole Hamiltonian modes and a variable set of knobs connected to non-linear multipole families. These are the sextupoles in first place, which have to maintain corrected chromaticities during optimzation, and optional octupoles and even decapoles. Also sextupole moments in combined function magnets may be included.

The Hamiltonian modes are scaled by reasonable betatron amplitudes and momentum range to make them comparable. In addition weight factors are set manually. Optimization may be done by hand "to get a feeling", but usually it is done automatically with frequent user intervention to adjust weights, exclude certain knobs etc. For minimization, the scaled and weighted Hamiltonian modes are summarized to provide a scalar penalty function.

A Powell minimizer is employed to supress first and second order sextupole terms. Second order terms may also be suppressed by an SVD procedure using the octupole families. Both algorithms are based on [14] and coded in unit `mathlib`. Decapoles may only be used manually for third order chromaticity, which is the only third-order effect included.

Basically, SVD could optimize sextupoles too using the linear system of first order sextupole terms and the linearized (i.e. local tangential) system of second order terms as suggested by Johan Bengtsson and previously implemented in Tracy-2. The mixed approach using Powell and SVD has historical origins but also turned out to perform quite robust.

Chromaticities up to third order are calculated by numeric differentiation, whereas the other Hamiltonian modes are calculated analytically. A real/complex factor is applied to non-resonant/resonant modes to change the periodicity of the structure [3].

Besides frames for Hamiltonian modes and multipoles, the GUI contains several edit fields for amplitudes, ranges, periodicity and minimizer settings, many check boxes to select knobs and Hamiltonian modes to be in- or excluded and several buttons to change settings and to control program flow.

BUG: in the linux version, the frames in the GUI don't update. After touching the GUI (i.e. resize by tiny amount) it works. Not yet understood.

Uses: `chamframe`, `csexframe`, `chromreslib`, `chromelelib`, `chromlib`, `ochromsvector`, `opatunediag`, `globlib`, `mathlib`, `../com/asaux`

Public procedures

`Start`, called by `opamenu` receives a handle to the tune diagram `opatunediag` (to show the prediction of the tune footprint), and then performs many actions:

A fixed number (26) of `chamframe` frames is installed to show the Hamiltonian modes and change their weights. This requires passing to each one handles to the other ones, because changing a weight may result in rescaling all. Handles are passed to `chromreslib` to perform

an update of all, if called by a knob csexframe. Further handles to GUI components, e.g. labels to show pathlength, are passed to chromlib, which contains the physics part. Check boxes are added to select groups of modes to be included in the optimization.

Then the cursor is set to "hourglass" while chromlib>ChromInit is called, which may need some time to set-up the matrices which connect the vectors of knobs (first order) and knob products (seond order) to the Hamiltonian modes. It also reads user settings, so the next step is to update the GUI with these settings and to use them for a first calculation of the penalty function. The Hamiltonian modes are set to show the results.

As next step, the knobs are installed, one instance of csexframe for each family of sextupoles, octupoles, decapoles and combined function magnets. Each sextupole needs handles to the other ones, because if a family is assigned to automatically maintain constant chromaticity, a change of another family will change its value. Octupole family handles are passed to chromelelib, which controls the SVD actions.

Check boxes are added to sextupole families to select them for chromaticity corrections, or to exclude them from optimizaton. If there are octupoles, check boxes are added to the second order terms to select them for the octupole-SVD, and controls for the octupole-SVD are made visible. Further, range edit fields for octupoles and decapoles are made visible if elements of these kinds are present.

Finally a tune diagram is drawn to show the tune footprint estimate based on the initial calculation.

Actions

("inc row) → Check(2Q,Qxx,Cr2,Oct)Click include in or exclude from the calculations the half integer chromatic resonances, amplitude dependent tune shifts (ADTS), nonlinear chromaticities and 2nd order resonant terms, and perform a new calculation.

("O" row) → CheckOctClick selects the Hamiltonian mode beside the checkbox for the octupole SVD (only visible if the lattice contains octupoles), sets up the SVD matrices and initializes the filter for weight factors to no cut (threshold 0).

Periods → edPer(KeyPress,Exit) sets the number or periods, calls chromlib>S_Period to recalculate the complex periodicity factors to be multiplied with the Hamiltonian modes and updates calculation and GUI.

2Jx, 2Jy, dpp, F → EdScPar(KeyPress,Exit) reads values for the relevant betatron amplitudes and momentum spread and updates the Hamiltonian modes. The last one is a factor how to weight the quadratic sum of sextupoles strengths.

("ξ" row) → CSelectClick reacts to the dynamically created row of checkboxes and selects the corresponding sextupole family for automated correction of linear chromaticity. Exactly two families have to be selected to activate the AutoChrom flag. Then the linear chromaticities and the corresponding csexframe instances are highlighted in yellow and calculations are repeated.

incl.Comb. → ChkCombClick in/excludes combined function bends from the sextupole

families and en/disables the corresponding csexframe instances.

max B(3[,4[,5]])L, step B(3[,4[,5]])L → edSOD(KeyPress,Exit) are fields to set ranges and step size for manual change for the sextupoles [and octupoles [and decapoles]]. The corresponding csexframe instances are updated, because the number they show turns red if the range is exceeded.

Smatrix → butSmatClick calls chromlib>S_TestOut to write sextupole data to a file. For testing, not used anymore.

select → butSelectClick launches ochromsvector to show the first order sextupole kicks in the complex plane. At first, all the resulting vectors of the first order resonance terms are shown and the eight corresponding chamframe instances are highlighted in same color like the vectors. Pressing the button repeatedly shows the single sextupole kicks for one of them, and the corresponding chamframe instance is highlighted in blue.

Minimizer initial step → EdMinAmp(KeyPress,Exit) sets the start amplitude of the Powell optimizer. Seems to have little impact, since Powell anyway scales the step size.

Dpp num.diff. → EdNumDiff(KeyPress,Exit) sets the dpp intervall for the numeric differentiation of tune to get the chromaticities. In particular third order chromaticity is sensitive to this number, and the value has to be test to get a stable result.

Start → ButMinClick prepares the arrays for the Powell minimizer, starts mathlib>Powell and turns the caption of the button to **Break**. The function PenaltyFunction is passed as a parameter to Powell, which will send it a vector of internal knob values. PenaltyFunction contains a hyperbolic scaling of these knobs to the sextupole strengths in order to not exceed the range (however, this may cause a sextupole to get stuck at its maximum value without further progress). If automatic chromaticity or sextupole SVD are enabled, these steps are performed first before calculating all the drive terms and add them quadratically with weights as set in the chamframe instances to compose a single scalar penalty function and write it to a label in the GUI. If the penalty improved the color of the label turns green, if not it is red. Calling Application.ProcessMessages allows the GUI to handle events while the minimizer is running in order to interrupt, if the user pressed **Break**.

Exit → ButExClick saves data for chromaticity and ADTS internally for later comparison to tracking results, and closes the GUI.
If there are octupoles in the lattice, controls for the SVD appear in the GUI. If some of the second order modes have been selected (see above, ("O" row) SVD is set up calling chromlib>Oct_SVDCMP and the line **Con=..., Nw=...** shows the condition, i.e. the ratio of minimum to maximum weight factor and the number of weight factors.

−,+ → ButOsvdN(m,p)Click decreases or increases the weight factor spectrum (setting the smallest ones to zero) to find the best solution. (Small N_w gives a robust but incomplete solution.)

SVD → ButOsvdDoClick calls chromlib>Oct_SVBKSB to set the octupoles and updates the Hamiltonian modes and penalty function.

undo → ButOsvdUndoClick restores the octupoles to the values they had before pressing **SVD**.

auto → chkOsvdAutoClick sets the SVD to automatic mode. Then it will run in a master-slave mode while the Powell minimizer is working, for example to maintain constant ADTS.

all off, res → ButOct(Off, Res)Click sets all octupoles to zero and restores the values again.

7.2 Element controller

csexframe

At start, one instance of this frame is embedded in the GUI for each family of sextupole, octupole, decapole and combined function bend. It provides knobs to increase or decrease the magnet strength. Changing the value triggers a calculation of the Hamiltonian modes. If the minimizer is running, the strength field follows.

Uses: chromreslib, chromelelib, chromlib, ochromsvector, globlib, ../com/asaux

Public procedures

This frame contains several public procedures, because the parent GUI opachroma and other units in this section have to control it. Further, the instances in the parent GUI have to control each other.

Init called by opachroma when creating the instance receives a type information if it is sextupole, octupole or decapole and an index for identification. It then gets the corresponding element family and its value, sets the name label and strength field (calling SetVal), and sets the appearance depending on the type.

Brothers, also called by opachroma receives handles to all instances of csexframe including the instance itself.

SetSize places the components inside the frame depending on the available space, which depends on the size of the parent GUI and on the number of knobs.

getTabs returns the position of label, value field and lock checkbox for opachroma to properly align the headline above the knobs.

SetRange sets the minimum and maximum value from the range fields in opachroma and calls private procedure LimitVal to cut the value to the limit if it exceeds the range and set the font color to red.

SetVal sets the value and also calls LimitVal.

UpdateValbyMin is used by the minimizer function opachroma>PenaltyFunction to only update the value but not trigger any calculation.

mySetColor changes the appearance, called by opachroma if the knob is reserved (e.g. for chromaticity correction).

Lock,UnLock locks or unlocks a knob. If locked the frame components are disabled. In case of an octupole family already included in the SVD, the SVD is set-up again without this family.

Chrom(Lock,Unlock) locks/unlocks a sextupole knob reserved for automatic chromaticity correction and disables/enables the check box to lock/unlock manually.

Actions

The event handlers are trivial: The edit field reacts to manual input and calls private procedure UpdateVal. The << <, >, >> buttons decrease or increase the value by a step as given in the corresponding fields in the opachroma GUI, or by a 20 times larger step. The **off** and **res** buttons set the value to zero or restore the initial value. And clicking the check box locks/unlocks the element.

Private procedure UpdateVal sets the value and recalculates the drive terms and the penalty function. In case of a sextupole knob, however, if automated chromaticity correction or automated octupole SVD are active, these are executed before. For chromaticity chromlib>ChromCorrect is called. For octupole SVD the private procedure OsvdStep recalculates the second order terms which may have been affected by the change in sextupole strength, and then peforms another iteration of the SVD.

7.3 Result indicator

chamframe

26 instances of this frame are created in the opachroma GUI at start. This frame displays the result for one of the Hamiltonian modes and allows a weight factor to be set for the minimizer, and, in case of non-resonant terms, a target value, which is subtracted for penalty calculation. The result is shown as number and as bar. The number is the true value in SI units, whereas the bar size corresponds to the value multiplied with relevant amplitudes (set in **2Jx** etc. fields in opachroma) and the relative weight factor. Changing the weight may force a resize of the bar scale and thus affects all other chamframe instances too. Changing a target may affect sextupoles or octupoles if automated chromaticity correction or octupole SVD is active, therefore access to csexframe instances is needed. Since csexframe already uses chamframe (via chromreslib), the use is not declared in the interface section to avoid a circular reference but in the implementation section.

Uses: chromelelib, chromlib, globlib, ../com/asaux – and csexframe at runtime.

Public procedures

This frame contains several public procedures, because the parent GUI opachroma and other units in this section have to control it. Further, the instances in the parent GUI have to control each other.
Init called by opachroma when creating the instance sets default values and the appearance to distinguish resonant from non-resonant modes.

SetLab writes the name of the mode to the name label.

Brothers, also called by opachroma receives handles to all instances of chamframe including the instance itself.

getChromSexHandles receives handles to the two sextupole families assigned to automated chomaticity correction.

passOctupoleHandle receives a handle to an octupole family and adds it to an internal array of octupole handles. These are needed to update the octupole families if automated octupole SVD is active, and weight or target for one of the modes assigned is changed.

SetSize places the components inside the frame depending on the available space, which depends on the size of the parent GUI and on the number of knobs.

getTabs returns the position of target, value and weight fields for opachroma to properly align the headline above the knobs.

SetValmax sets the maximum value to scale the bar.

SetWeight sets the weight factor field and its font color.

UpdateWeight accepts a new weight factor and adjusts the scale of *all* chamframe instances to maintain comparability of the bars. If automated octupole SVD is active, a change of weight will affect its result, therefore the SVD is repeated. This will change the second order modes, so the scale of all instances is adjusted again.

SetTarget sets the target field (non-resonant modes only).

UpdateTarget accepts a new target value and peforms the same steps like UpdateWeight (see above). In addition, if automated chromaticity correction is active and the target value actually is one of the desired linear chromaticities, then chromlib>ChromCorrect is called, and the two sextupole families assigned to chromaticity correction are updated.

UpdateHam writes the value field and calls private procedure PlotBar to plot the bar.

mySetColor changes the appearance, called by opachroma if the mode is to be highlighted for various reasons.

MinInclude sets the color to grey and disables all components, if this mode is not included in penalty calculation.

Actions

The event handlers are trivial: The target and weight edit fields react to manual input and call UpdateTarget or UpdateWeight (see above). The − +, buttons decrease or increase the weight.

(bar panel) → PanBarPaint reacts to a repaint event. It first erases the bar by filling the panel with black color, than it plots the bar like PlotBar. (Should be united with PlotBar).

7.4 Result handles

chromreslib

This small unit organizes handles, which are accessed by other units. It was introduced to avoid too may passes between these units.

Uses: chamframe, chromlib

Public procedures

Pass_CH_Handles and Pass_LabPath_Handles receive handles to the 26 chamframe instances and to the three pathlength labels in the opachroma GUI.

UpdateHamilton calls chamframe>UpdateHam for all handles.

UpdatePath writes path length data to the label handles.

UpdatePenalty writes the current penalty to the label in the opachroma GUI.

7.5 Element handles

chromelelib

This small unit organizes handles related to the octupole SVD.

Uses: chromlib

Public procedures

Pass_Osvd_Handles receives handles to all the components of the opachroma GUI related to octupole SVD.

Osvd_Status contains some logic to enable or disable the components of the octupole SVD depending on the status. It further displays condition (ratio of min. to max. SVD weight factor) and number of weight factors in use.

Osvd_Wfilter filters the SVD weights, i.e. increases or decreases the null space, depending on how many weight factors are to be cut off.

7.6 Vector diagram

ochromsvector

This little form contains only a plot window to show the sextupole kicks in the complex plane in order to visualize phase cancellations. All control is done by the **Select** button in the opachroma GUI.

Uses: chromlib, globlib, mathlib, ../com/asfigure

Public procedures

Init is always called by **opachroma** → butSelectClick (see above) and steps up the variable chromlib:VSelect to select one of the eight first order resonant sextupole Hamiltonians for display or to compare the sum vectors of all. The caption of the form is set to tell which mode it is, and Star is called to do the plot.

Star plots a star diagram of complex vectors for all sextupole kicks in the lattice. The vectors are calcualted by chromlib>getSVector. The colors for the sextupole families as given by the chromlib:SexFam record are the same like the sextupole knob csexframe color in the opachroma GUI. The sum vector is shown as a white line, enclosed by a white circle to better see it. In comparison mode, the sum vectors for the eight first order resonant terms are shown in one diagram. The colors are the same like the colors of the corresponding chamframe instances showing the first order modes.

Actions

(on create) → FormCreate and (on close) → FormClose get and set the the user defined size of the window and save the setting on close.

7.7 Calculations

chromlib

This unit is the base for all other units and GUIs of this section. It contains constants, type declarations and variables, and all the physics procedures.

Uses: linoplib, elemlib, opatunediag, globlib, mathlib, ../com/vgraph, ../com/asaux

Public constants, types and variables

Several array constants contain indices for calculating the Hamiltonian modes and annotations, colors etc.

Type declarations define SextupoleType for a single sextupole kick, and (Sex, Octu, Deca) FamType for the (sextu, octu, deca)-pole families.

SM1_row defines one row of a matrix to get the contribution to the 10 first order modes from one of the sextuple families. SM2_row is for the 11 second order modes (ADTS and resonant) and for one *product* of two sextupole families. The corresponding matrix rows for octupoles are contained in OctuFamType as fields omat, om1.

In the public variables section the sextupole matrices SM1, SM2 are defined as arrays of SM(1,2)_row of variable length, since the size will depend on the number of families. The group of variables named osvd_... are 1-dimensional arrays to pack the matrices for octupole

SVD, they will be unpacked to 2-dimensional arrays in mathlib>SVDCMP. Other variables of simpler types are for keeping intermediate results and control flags.

Public procedures

ChromInit restores user settings, sets up all the arrays and performs calculations which are only needed once at start like the SM1, SM2 matrices.

In order to set up the arrays the element list (Ella) is searched for sextupole families and the lattice line-up is searched for sextupoles installed. Also combined function magnets are included since they may contain a sextupole component. Before, the periodic solution is calculated, which can not fail, since the non-linear optimizaiton GUI is only enabled if a periodic solution exists. With the initial values from the periodic solution, the optical funcitons are calculated element by element. Beta functions and phases at each sextupole *kick* are obtained and stored in the Sextupole array.

Optical functions are also calculated at entry, midpoint and exit of quadrupoles and bending magnets, since they contribute to chromaticity and chromatic half-integer modes (this is an approximation, therefore there may be small differences between chromaticity obtained this way and from numeric differentiation).

Octupoles are handled a bit differently than sextupoles by building the OctuFam family array "on the fly" and not in advance, and calculating the constant part of the octupole matrix omatpre, om1pre already here. For decapoles only a small matrix dmat for their first order effect on third order chromaticity is calculated.

When finished with collecting the elements in the lattice, a linked list of sextupole kicks belonging to the sextupole families is established and the start pointer is saved in SexFam.kid.

Finally the sextupole matrices SM1, SM2 are calculated calling procedure S_Matrix, which may take a while, the periodicity factors are calculated calling S_Period, and the tune diagram opatunediag is launched.

When ChromInit is finished, all data have been established for opachroma to compose the GUI, i.e. install csexframe knobs etc.

These procedures are called by ChromInit:

S_Matrix calculates the first and second order extupole matrices SM1, SM2: a lattice may contain some 100 sextupoles, each one modelled by several kicks, so a total of about 1000 kicks. But since they belong to a small number ($\sim$ 10) families, the phase and amplitude terms to be multiplied with the sextupole strength are calculated in advance and stored in the matrices. Then the optimizer will only vary the vector of family strengths and multiply it with the matrix to get the Hamiltonian modes. For the second order this procedure is mandatory to get results in acceptable time, since it contains double sums over the cross-terms from all kicks, which may result in a million terms to be added. The formulae coded are found in [11] and [12].

S_Period calculates a complex factor to be multiplied with the Hamiltonian mode if more than one period is assumed. Calculating one period and apply this factor is much faster than

calculating the whole lattice made from many identical periods. Entering zero to the procedure gives the extrapolated result for an infinite number of periods. The formula coded are found in [11] and [3].

S_TestOut writes the matrices and optics data at multipoles to a file named (lattice)_smatrix.txt for testing or cross-check with other codes.

set_cd_matrix sets up a matrix for numeric differentiation of tunes to get chromaticity up to third order from a polynomial fit.

Following procedures are performed on user input or during optimization:

UpdateChromMatrix sets up the 2×2 matrix how two selected sextupole families affect the linear chromaticity. Then the matrix is inverted and saved to be used for automated chromaticity correction.

getLinChroma calculates the linear chromaticity by adding the sextupole contribution to the initially calculated, constant contribution from quadrupoles and bending magnets.

ChromCorrect sets two selected sextupole families to adjust the chromaticity to the set values, contained in HamTarg([0],[1]) as target values.

Update(SexFam, SexFamInt, OcFamInt) set the multipoles to the given value in two ways: in the arrays initially compiled for fast calculation of the Hamiltonian modes, and in the element list Ella for chromaticity calculation from numeric differentiation by variation of momentum offset of the periodic solution.

The DriveTerms... procedure calculate different groups of Hamiltonian modes as included by the check boxes in the ("inc" row) in the opachroma GUI. These are the first order geometric terms, the first order chromatic terms, the ADTS, the second (and third) order chromaticity, the octupole resonant modes, and the square sum of all sextupole strenghts.

The Penalty functions subtracts target values (for non-resonant modes), gets the absolut value of the (complex) mode, applies relevant amplitude factors and manually set additional weights (in chamframe), and finally sums up all these results quadratically to get a simple number as objective for the minimizer.

HamScaling calculates the weight factor corresponding to relevant betatron amplitudes and momentum range to be applied by Penalty.

Oct_sv(dcmp,bskb) contain the octupole SVD. The first procedure sets the size of the linear system to be solved by SVD depending on the number of selected Hamiltonian modes and octupole families. It builds the coefficient matrix of the system, where resonant modes get two rows, because they are complex, whereas the real-valued non-resonant modes get one row. And in packs the matrix in the 1-dimensional array osvd_aupack in order to run the SVD procedure from [14] as coded in mathlib>SVDCMP with variable array size. The scond procedure gets the vector of target values for the selected modes and performs the backsubstituion calling mathlib>SVBKSB to set the octupole families. The information on null space, i.e. cut off weighting factors, is transmitted in the array osvd_wuse.

getSVector calculates the complex vectors of sextupole kicks for ochromsvector (see above).

TDiagPlot shows a prediction of the tune footprint from ADTS and chromaticity in the tune

diagram. Procedure TDiagPlotFull refreshes the tune diagram before doing so.

ChromSaveDef and ChromDispose, finally called when closing opachroma, save user settings and release the dynamic arrays and linked lists, and CloseTuneDiagram closes the tune diagram.

8 Orbit and Injection

8.1 Orbit and injection

opaorbit

Dpending on a flag set at start, this GUI is used for orbit or injection studies.

In orbit mode, correlated misalignments are applied, the response matrix is calculated and pseudo-inverted using an SVD procedure in order to correct the orbit. A loop function may run several error seeds to obtain some statistics. Note, that correctors and BPMs defined with reserved names CH,CV,MON in the lattice file are internally expanded to a set of individual elements, e.g. CH001... when unpacking the lattice by globlib>MakeLattice.

In injection mode, kickers may be synchronized for correct timing, and the injected (and/or stored) beam trajectory is calculated for a few turns.

In both modes, when leaving the unit, results for misalignments and corrector settings, or for kicker settings, are saved internally and will be used in subsequent linear optics calculations or tracking. There are problems with misalignments in tracking, not yet solved.

The GUI contains two plot windows for horizontal and vertical orbit and a line-up of elements, which may be dragged to knobs. These are correctors and beam position monitors (BPM) in orbit mode, and kicker magnets and BPMs in injection mode. In orbit mode a panel becomes visible which contains several controls to set misalignments, correct the orbit, get orbit statistics and switch the plot mode. In injection mode a simpler panel appears in the same place which contains controls to set kicker timing and select turn number.

ostartmenu is called in both modes to set the initial conditions, where the periodic option is disabled in injection mode, because it makes no sense.

opaorbit is similar to opalinop for linear optics design, because both show the orbit and provide clickable elements to be dragged to knobs. Furthermore, data are transferred from one to the other to calculate the optics on a corrected orbit. Thus it would make sense to combine both GUIs into one, perhaps using tabs for the different modes. However, opaorbit uses two ../com/asfigure instances for plotting and the clickable elements are realized as TShape objects on the form, underneath the plot, whereas opalinop has its own, much more complex plot window with bars inside the plot for the elements.

Uses: knobframe, linoplib, ostartmenu, globlib, mathlib, ../com/asfigure, ../com/asaux

Public procedures

Start receives a flag to set orbit or injection mode. Then the form passes a handle to itself to linoplib to enable a repaint. The girders are set up calling globlib>GirderSetup. Then the lattice is searched for correctors, resp. kickers and monitors, arrays are declared to manage them, and symbols are created on the form. After restoring user settings and filling edit fields, ostartmenu is launched to get started.

Actions

Actions available in both modes

Some of these actions(marked by $*$) don't make sense in injection mode and should be hidden or disabled. The historical reason is, that the orbit mode was first and the injection mode was "squeezed" in later.

(repaint) → FormPaint plots the orbit. The procedure is called by linoplib>PlotOrbit via the handle to the form passed at initialization. Orbit calculation and plotting is triggered by ostartmenu and also by some components in the GUI. Plotting is done by private procedure Makeplot which calls several other private procedures: Get (Orbit, Cor, Mis) Max get the maxima of data for plot scaling, plot(X,Y) plot the orbit, WeightPlot shows a histogram of orbit correction SVD weighting factors, BPMshow fills the BPM readout panel, LabelShow updates the correction status and CodStat fills a panel with orbit statistics data. In injection mode some of these procedures do nothing.

(symbols) → BoxMouse(Down, Up) drag an element shown as symbol (TShape) to either a knob (class knobframe) if it is an active element, i.e. corrector or kicker, or, if it is a BPM to the BPM panel panbpm to show its readout.

<–> , >–< , –––> , <––– → butzooClick are zoom functions, similar but simpler than in opalinop.

"BPM" panel panbpm:

$*$ **Orbit/Reference** → butBPMmpodeClick toogles between "Orbit" mode, where the BPM shows the readout (also x', y' which a real BPM can't read), and "Reference" mode, where values for x, y can be entered to be used as a reference in orbit correction.

$*$ **Set** → butBPMrefClick saves the reference.

"Start and statistics" panel panctr:

Start → butStaClick launches the ostartmenu start panel again to change initial conditions etc. BUG: plot error when starting from an optics marker.

include nonlinear elements → chkSextClick switches on/off nonlinear elements for the calculations.

$*$ **BPM/all/Cor** → butCODstatClick selects which statistical data to show in the table underneath the button: BPM only, all elements or corrector strength.

"Show" panel panplo:

* **Orbit+BPMs, Correctors, Misalignments** → butplotOCClick selects the plot mode: orbit, corrector strengths or misalignments.

* **keep max.** → chkKeepMaxClick freezes the maximum value for the plot, may be used to avoid rescaling after orbit correction.

–> EPS → butEPSClick exports the current plot to an `.eps` file.

Actions available in orbit mode only

"Misalign and correct" panel panmis:

set → butMisalClick reads misalignment data from the edit fields and calls private procedure SetMisalignments to set correlated Gaussian distributed misalignment with the rms values from the edit fields and cut at a number of sigma also given there. First, the girders are set, then the secondary girders supported by other girders, then compound elements, i.e. series of lattice elements with no space between which are assumed to be one rigid body, and finally individual single elements. Finally the orbit is calculated and shown. The edit fields contain data for the girder ends, for the elements relative to the girder and for the joint play of a virtual link assumed to connect girders ("train link"). Misalignments are sway and heave, and for girders and elements also roll. The **El/Gird** field contains a percentage how to reduce the misalignment of elements on girder relative to single elements, assuming that they may be aligned better. The **seed** field takes any number to generate different random sees for testing.

include BPM → butMonabsClick includes or excludes the BPMs from misalignments, i.e. if their readings are absolute or relative to their own misalignments, which is the realistic case. The caption of the button is changed to "exclude" if BPMs are absolute.

zero → butZeroClick and **re-set** → butSetAgainClick sets all misalignments to zero or restores them again.

Correct → butCorClick performs the orbit correction: first (procs PrepOrbCorr, GetResponseMatrix), if not yet done, the arrays for the SVD are dimensioned, the response matrix is set up based on the analytical, linear lattice model, and the SV decomposition is done calling `mathlib`>SVDCMP.
Then (OrbitCorrection) the orbit is corrected by passing the target vector of BPM references to `mathlib`>SVBKSB. The correction result is evaluated (CodStat, CodStatCalc) and the label LabOrbStat is set to "zeroed" if all BPM read zero, to "minimized" if correction converged with some residual (this is the case if there are more BPMs than correctors), to "too many it." if correction did not converge within the limited number of iterations, and to "failed!" if it diverged and the orbit became too large. Statistics results are written to the table in the statistics panel panctr, showing mean, rms and max value for BPM readings, orbit everywhere and corrector strengths.
Finally MakePlot is called to show the corrected orbit, it contains the procedure WeightPlot which plots a histogram of weight factors, showing in darker color weight factors which were set to zero (in case a more robust but incomplete correction is tried).

Loop → butLoopClick performs a series of orbit corrections to get statistical data. When pressing the button, at first the seed field changes to **Nseed** and the number inside is the

number of seeds to do, and the button caption changes to **Run**.
Pressing again starts the loop and the caption is set to **Break**. Pressing again while running aborts the loop, otherwise it continues to the end and the caption becomes **Done**. Pressing again sets the caption back to **Loop**. Internally this is controlled by variable `loopstatus`. During running, the seed index is shown, and in the OPA log window in the main GUI `opamenu` the results of the seeds are listed. When done, the statistics data have a different meaning than for single orbit correction (`LoopStatCalc`): "mean" is the mean rms orbit from all seeds, "rms" is the maximum of the rms values, and "max" is the absolute maximum.

Corr=0 → `butCorZeroClick` and **BPM=0** → `butBPMZeroClick` set all correctors or all BPM references to zero.

X → `butOcoWplotClick` toggles betwenn horizontal and vertical weight factor plot, button caption changes between **X** and **Y**.

(slider) → `sliderWChange` sets all weighting factors above the slider to zero.

Actions available in injection mode only

"Pulsed magnets" panel `pankick`:

ON → `butKickClick` activates the kickers, sets the caption of the button and enables the multi-turn buttons if the lattice is circular. Then the orbit is calculated and shown.

Sync → `butSyncClick` synchronizes the kicker by calculating the time of flight from the start of the lattice to the entry to the kicker and sets its field `delay` to the same value. If the kicker is of finite length, `elemlib`>`Kicker` takes care of internal time of flight.

−,+ → `butKickT(m,p)Click` increase the number of turns (max 3) to be calculated and plotted in a circular lattice. `linoplib`>`OrbitCalc` is called with a negative number used as a switch to count turns and start at begin of lattice in successive turns.

9 Tracking

There are three GUIs for phase space tracking, dynamic aperture and Touschek lifetime, and a physics unit used by all three.

9.1 Phase space

opatrackps

Uses: `tracklib`, `globlib`, `opatunediag`, `mathlib`, `../com/asfigure`, `../com/vgraph`, `../com/asaux`

The GUI for phase space tracking contains four plot windows for the transverse phase spaces (`psx`, `psy`) and for the particle spectra (`ftx`, `fty`), and six panels to set parameters or show results. Single particles are started from coordinates entered manually in edit fields or passed from mouse clicks in the phase space plots. Amplitude dependent tune shifts (ADTS) are

obtained by stepping up particle coordinates. For simulation of injection, a beam ellipse populated with many particles can be tracked.

The six panels are PanParam to set tunes and apertures, PanCtrl to set start coordinates, PanRes to show a table of tunes found in the spectra, PanFFT to set parameters of the Fast Fourier Transform (FFT), PanTush to set parameters for the ADTS loop and PanBeam to set parameters for the beam ellipse. From the last four only two are visible at a time.

Public procedures

Start receives a handle to the tune diagram opatunediag. The combobox comTurns is populated with powers of 2 for the available number of turns. User settings are restored. The four ../com/asfigure instances for plotting are assigned and handles are passed to the edit fields for starting coordinates in order to later read coordinates from clicking into the plot. Flags and fields are initialized. Finally, tracklib>Initdpp is called, which provides the periodic solution (it will exist, otherwise the tracking option would not be enabled in opamenu) and the physical aperture limitations projected to the track point. The (onPaint) event of the plot areas will then show empty beam ellipses corresponding to the linear physical limit. BUG: at start the plot sometimes appears too small, ok after touching (resizing) the GUI.

Actions

Note: a common event handler ediAction is used for most edit fields in this GUI. It is called by KeyPress or Exit events, i.e. when a key is pressed in the field or the cursor leaves the field.

Panel PanParam

Nturns → comTurnsChange gets the number of turns from the selected item. It is always a power of 2 as required by the FFT.

Element apertures → cbxElAperClick toogles between using the apertures of the elements or the values in the **Ax[mm]**, **Ay[mm]** edit fields before calling tracklib>Initdpp again to calculate the physical limit for tracking. The psx,psy figures are initialized again.

A(x,y)[mm] → ediAction[edi(x,y)aper] has the same effect when entering a value in the edit fields.

Trackpoint → ediAction[edistartpos] takes the value from the edit field as trackpoint position (can be anywhere in the lattice, even inside an element) and calls tracklib>Initdpp again to get the optics and aperture limit at that location. Before the input is checked to be within the length of the active lattice segment.

4D → butT6Click sets tracking mode from 4- to 6-dimensional and changes the button caption. Experimental mode, incomplete implementation!

Panel PanCtrl

dp/p[%] → ediAction[edidpp] calls tracklib>Initdpp again for the momentum offset as entered. The periodic solution may be lost, if the value is too large, then a warning pops up.

Run → butRunClick reads the coordinates from the four edit fields **X[mm]** etc. These values may have been entered manually or by a pMouseDown event in the psx,psy plot areas (see ../com/asfigure above). Before starting tracking, the particle amplitude is calculated using the local beta functions, and an ellipse is drawn for comparison with the aperture limitation. Then the function Track is called to track the given number of turns by calling tracklib>Oneturn in a loop, and to plot the coordinates during tracking. When finished it will return false, when the particle was lost it will return true. If succesfull, private procedure Spectrum is called, which applies a sine-window to the data for peak interpolation before calling tracklib>TWOFFT and FFTtoAmp to get horizontal and vertical particle spectra as real amplitudes. Then private FFTplot is called: it plots the spectra, tries to guess the resonances corresponding to the peaks found by calling tracklib>FindPeaks and ResoGuess, and shows the results in the spectrum plots and in the table resgrid in panel PanRes. The tune indentified as fundamental is plotted in the tune diagram opatunediag.

More → butMoreClick continues tracking with the same number of turns and updates the spectra with the new points.

Clear → butClearClick clears the plot areas by initializing again and also clears the tune diagram.

Beam → butBeamClick sets panel PanBeam visible (and hides PanTush, or vice versa).

Exit → butExitClick saves the user settings and closes the GUI.

Panel PanTush

Run → butTushClick starts an ADTS scan by stepping up particle amplitude and tracking the tune. At first the number of steps is taken from the **Nsteps** field, and the **Break** button is enabled to abort the scan. Then three arrays for tunes and amplitudes are declared, for horizontal, vertical and coupled motion. Maximum amplitudes are given by element apertures or by manually entered apertures (see above) and calculated by tracklib>AmpKappa assuming elliptical apertures [3]. If manually entered apertures are used, the acceptance from element apertures are alsos shown in the plot. Since FFT does not distinguish between below and above half integer, following the tune has to take into account on which side the tune is located.

During tracking label labtprog is updated to inform about the progress, the start coordinates are shown in the edit fields of panel PanCtrl and the Poincareé plot of the particle motion is shown in the psx,psy plot areas. When the number of turns is completed and the particle was not lost, then the spectra are calculated and plotted, and identification of the fundamental tune is tried. The lines in the code following the call of Spectrum are an attempt to still follow the particle tune if it moves over half integer. It sometimes works but often doesn't – to be improved. The found tunes are plotted in the tune diagram calling opatunediag>AddTushPoint. If manually entered apertures are used, particles outside the acceptance from element apertures are plotted in grey color in the tune diagram.

When the scan is done, the tune as function of amplitudes is plotted for the three series

horizontal, coupled and vertical, and the button **Amplitude** is enabled. If analytical ADTS had been calculated in non-linear optimization, `opachroma`, then arrays are prepared for plotting them too.

Amplitude → `TSplotButClick` toogles between plotting tune shifts vs. betatron amplitude, pinger kick and transverse max. position and changes the caption of the button accordingely. During the scan transverse kicks were stepped up simulating a pinger, so this plot is most realistic. Amplitudes had been calculated by linear interpolation up to the maximum amplitude, and position is calculated from the amplitude using the local beta function. (In a previous version amplitudes were taken from the peak height in the spectrum, but this turned out to be confusing in case of strong non-linearity or coupling.)

−−> EPS → `butExportClick` exports the ADTS plots to two `.eps` files. Note that the Poincaré plots have no eps export because this would require to store too much data. (Anyway, screenshots can always be taken by couble-clicking the plot.)

−−> TXT → `butTXTClick` writes ADTS data to a text file.

Panel `PanBeam`

X → `ButBeamXYClick` toggles the edit field between accepting data for horizontal or vertical beam size. If vertical, caption changes to **Y**.

Set → `ButBeamShowClick` reads the edit fields for beta and alpha function and emittance and the number of particles and creates the particle ensemble centered at the initial coordinates as given in the `PanCtrl` panel. Note, that the ensemble is generated only in one dimension, while the coordinates in the other dimension are the same for all particles. The start ensemble is plotted in green.

Run → `ButBeamRunClick` reads the **show turns** field, which tells the first N turns to be plotted (useful for injection studies). The last turn is plotted anyway. The function `TrackBeam` performs tracking. It tracks all particles of the ensemble for one turn after the other, counts how many are left, and plots the ensemble if plotting this turn was requested. And it writes the the number of turns done and the number of surviving particles to label `LabT` in panel `PanParam`. If there are surviving paricles, the button caption is set to **More** and pressing it again will continue tracking.

Break → `ButBeamBreak` aborts tracking by setting flag `BeamBreak` which is checked in the `TrackBeam` loop.

Other actions

Resonance Guess – Parameters → `butParamClick` switches visibility from panel `PanRes` to `PanFFT` and fills the edit fields. Panel `PanRes` is passive, it contains only a table listing found peaks and resonance guesses, and panel `PanFFT` contains edit fields to set parameters for peak and resonance identification. Label and button captions change to **FFT parameters – Update**. Pressing the button again switches back to panel `PanRes`, updates the filter, repeats the FFT plot and evaluation, and sets back label and button captions.

(resize) → `FormResize` calls `ResizeAll` to dynamically resize the complete GUI and initialize

the plot areas again.

9.2 Dynamic aperture

opatrackda

There are four modes of dynamic aperture tracking, three 2-dimensional scans for the areas $(\pm x, y)$, $(\pm x, \pm\Delta p/p)$ and $(y, \pm\Delta p/p)$, and one 3-dimensional scan for the volume $(\pm x, \pm x', \pm\Delta p/p)$. They may be combined with three scan methods, grid probing (GP), binary search (BS) and flood fill (FF), however FF is not available for the 2-d $\Delta p/p$ scans, whereas the 3-d scan works only with FF. GP probes every point in a rectangular grid which is successively refined to get an early impression. BS search finds separatrix points along a set of rays from the origin. And FF from computer graphics fills the unstable area outside the separatrix. FF was implemented in OPA by Bernard Riemann.

The FF 3-d scan provides data for "Fast Touschek Tracking" in `opatracktt`. A detailed description is given in [17].

The GUI contains a plot window `fig`, two radio button groups `rgmode` and `rgmeth` to set mode and method, two panels `PanNray` and `pangrid` to set parameters for BS and GP/FF modes, and edit fiels for turns and momentum offset.

Uses: `tracklib`, `globlib` `mathlib`, `../com/asfigure`, `../com/vgraph`, `../com/asaux`

Public procedures

`Start` reads user settings and fills the GUI with these values. The corresponding components are shown or hidden, and incompatible settings are corrected. `tracklib`>`Acc_dpp` and `tracklib`>`Initdpp` are called, which provide the momentum dependant physical acceptance and the periodic solution for the given $\Delta p/p$ (= 0 at start, has to exist, otherwise launching the GUI would be blocked in `opamenu`).

Actions

(mode/method radio groups) → `rbClick` reacts to setting the mode or the method. Mode and method are encoded in the `Tag` fields of the components. Incompatible combinations of mode and method are corrected. Labels and edit field captions are set depending on mode/method calling private procedure `rbsetfields`. Then the grids for probing are updated calling `UpdateGridParams`: the grid dimension are numbers like $2^N + 1$ in order to successively refine the grid during execution. Finally `UpdateApertures` is called to calculate the available apertures to be used for testing on particle loss. Depending on if the **Element apertures** field is checked or not, these are either defined by the element apertures or by values set in the **Ax,Ay** fields. Depending on the selected mode, `UpdateApertures` calls `tracklib`>`Acc_dpp` to calculate the momentum dependent horizontal and vertical apertures, or `tracklib`>`Initdpp` and `Silhouette` to calculate the transverse aperture for

the set momentum offset as it appears at the trackpoint (in linear transformation). Finally MakePlot plots the yet empty apertures.

dpp offset/range → eddppaction performs the same steps to calculate the transverse aperture at the set momentum offset or the momentum dependent apertures, depending on the mode. If the periodic solution is not found for the given momentum offset, an error message is launched and the offset is set to zero.

Nturs → edturns(KeyPress,Exit) takes the given number of turns.

TrackPoint → esposaction should set the track point anywhere but was never implemented because it was not needed yet. However, this option is used in opatrackps and opatracktt, so it could be implemented here too.

In BS mode panel PanNRay is visible:

Number of rays → butnrclick reacts to the +− buttons to step up or down the number of rays for binary search calling UpdateGridParams and shows it in label labnr.

Resolution → edreso(KEyPress,Exit) just reads the required resolution when to stop the binary search.

In GP and FF mode panel PanGrid is visible:

Cells... → bugNgClick reacts to the +− buttons to step up or down the number of grid points calling UpdateGridParams and shows it in the labels. The meaning of the two lines in the panel depends on the selected mode; captions are set when clicking the mode/method radio buttons. In case of the 3-d mode is checked, if the number of momentum slices agrees with the number of points used for calculating the momentum dependent physical apertures, and if this is not the case, the latter is recalculated again, because both tracked and physical apertures at same momentum values are required if results are to be used later in opatracktt.

Element Apertures → cbxAperClick switches between using element apertures or set apertures as given in the eda(x,y) fields to calculate the physical acceptance calling UpdateApertures.

Ax,Ay → eda(x,y)(KeyPress,Exit) read the apertures, which are assumed to be at all elements, and calculates the physical acceptance based on these values.

plot style → butplotStyleClick toggles between a simple and a nicer plot style for the DA result.

Start → butStartClick starts tracking by calling procedure DATracking: it first changes the button caption to **Stop** and its Tag to 1, so pressing the button again will abort tracking. Then, depending on the method, either the rays for BS method or the grid for GP and FF method are set up calling DARaySetup or DAGridSetup. Depending on the mode the koordinates (variables ko...) to be plotted are assigned ($1, 2, 3, 5$ corresponding to $x, x', y, \Delta p/p$). Different tracking procedures are used depending on mode and method. They all make use of DASingleTracking, which also marks the particles by green or red symbols in the plot if they survived or not.

DATrackingClassicGrid for GP method adresses the grid cells as stored in array dagord,

which was established by DAGridSetup. The idea behind is too start with a coarse probing and refine it iteratively in order to quickly get an impression if DA is good or bad.

DATrackingBinRays for BS method performs a binary search along the rays established by DARaySetup by increasing or decreasing the relative coordinate along the ray depending on survival or loss, where in case of loss the decrement is halved until it becomes smaller than the given resolution.

DATrackingFillToolGrid for the FF method is an implementation of the flood fill algorithm from computer graphics, written by Bernard Riemann [17]: it finds the separatrix between stable and unstable regions by checking if adjacent pixels have different results for loss and survival. Inside the stable areas no further testing is done, inside the unstable areas all particles are tested, which is fast, because they usually get lost quickly. Particles results are saved in an array queue of class TIndexQueue which contains its own methods for management.

The 3-d FF scan for momentum dependent separatrix in horizontal phase space requires pre- and postprocessing: arrays and variables named Flo... defined in tracklib contain parameters of the separatrix polygons for the momentum slices. Procedure FloPinit sets them up before tracking, and procedure FloPoly, called after tracking each slice, finds a polygon approximating the separatrix, i.e. the boundary between stable and unstable pixels, centered at the closed orbit. These polygons are overplotted to the grid for each slice. After termination all polygons are shown together with color changing from blue to red indicating the momentum. The polygon data are saved and may be re-used for Fast Touschek Tracking by opatracktt.

The MakePlot procedure shows a variety of DA-plots depending on mode, method and status. The code may be understandable without further comment. The only complication it the flag MonitorXXP introduced to show results during 3-d scan in a different way than after termination.

--> **EPS** → butExportClick exports the current plot as .eps file.

Exit → Exit saves the user settings and closes the GUI.

9.3 Touschek lifetime

opatracktt

Touschek tracking: Lifetime is calculated as described in Appendix **??** from bunch volume and momentum acceptance (MA). The MA is the minimum of the linear MA given by the beam pipe apertures >CalcMALin, of the RF-MA which is derived from input parameters or has been calculated previously by opabucket, and of the dynamic MA obtained from tracking and binary search for min./max. stable momentum offset >MADyn, Trackdbins, TrackLat.

The bunch volume is calculated from the periodic optics solution with its emittance and energy spread >CalcSigma.

The GUI has two panels for several input values affecting lifetime and for derived values >Output.

Coulomb lifetime is calculated from the effective acceptance, hower this includes only the physical, not the dynamic aperture limits >CalcAccEL. Bremsstrahlung lifetime uses the negative MA as used for Touschek lifetime too. Total lifetime is given as the inverse of the sum of the loss rates >CalcLifetime.

Uses: tracklib, linoplib, globlib mathlib, ../com/asfigure, ../com/vgraph, ../com/asaux

9.4 Calculations

tracklib

At start, all linear elements are concatenated to matrices alternating with non-linear (or time dependent) kicks in order to speed up tracking >TrackinMatrix. Physical acceptances are calculated either from element apertures or from given apertures to estimate the maximum range for tracking >Acceptances. Both calculations have to be re-done when changing the reference momentum >Init_dpp. The available aperture in the (x, y)-plane is calculated for a set of rays to obtain the silhouette, i.e. projection of beam pipe apertures to the trackpoint as described in Appendix **??** >AmpKappa. Optics parameters at the Trackpoint are calculated, which may be located anywhere, even inside an element >TrackPoint.

Tracking proceeds turn by turn at fixed momentum >OneTurn (or with changing momentum based on a simple model of synchtron oscillation >OneTurn_S). Tracking one turn proceeds by repeated application of the matrix for a series of linear elements, followed by a non-linear (or time dependent) kick and a check for particle loss >TMatKick.

The procedures in the last third of the unit (from SineWindow to the end) are for signal processing, to calculate the FFT, interpolate frequencies and guess the related resonances.

Uses: linoplib, elemlib, globlib mathlib

10 Special Element Editors

10.1 Longitudinal gradient bend

opalgbedit

10.2 LGB calculations

lgbeditlib

opalgbedit.pas/.lfm (19.1k)

---

uses globlib, mathlib, ASfigure, ASaux

lgbeditlib.pas (13.9k)

---

uses globlib, ASfigure, ASaux

These two procedures optimize the field profile of a longitudinal gradient bend in order to minimize quantum excitation, i.e. the I_5 radiation integral. Magnet type, length, peak field and number of slices are set before starting a Powell minimizer. The first procedure is mainly the GUI and the minimizer, the second one contains physics and plotting.

11 Layout, Currents etc

11.1 Machine Layout

opageometry

opageometry.pas/.lfm (63.2k)

---

uses globlib, linoplib, mathlib, ASaux.

This unit displays the lattice layout, peforms geometric matching and exports various files. The orbit is calculated in 3D-space from element lengths, deflection and rotation angles >CalcOrbit. The orbit as curve in space is rotated and translated depending on the initial conditions >setDrawMode. The elements are made from faces, i.e. polygons in 3D-space >CalcFaces, which become polygons when projected to 2D-space of the image plane >CalcPoly. Changing the initial conditions does not change the faces themselves but applies the same translationa and rotation to all of them >TransPoly.

Up to now the elements are "flat", i.e. represented by a face of some length and width in the midplane. If changing the initial angle, they look ugly. It would be straightforward to define boxes made from several faces, however this would also require some rendering algorithm for 3D-display.

Several files can be exported, among them a .geo file of polygons >butListClick, which can also be read >ReadFiles in order to show several lattice structures in one plot.

Geometric matching looks for lattice variables with names starting with "G", which control lengths and deflection or rotation angles of elements (in >Start) and runs an SVD minimizer to adjust the final (or initial) coordinates and angles of the lattice structure to a target value >butMatchClick.

Geometric matching works but is not well implemented yet. More checks are required, and also an "undo" button should be added.

11.2 Magnet currents

opacurrents

getKfromI, getdKdIfac, getIfromK convert magnet strength in current and vice versa taking into account non-linear magnet saturation.

opacurrents.pas/.lfm (10.3k)

---

uses globlib, ASaux.

This unit calculates magnet currents using two tables for allocation (i.e. hardware type of the magnet in the lattice) and magnet calibration, which may be read with the lattice in globlib> ReadAllocation, ReadCalibration. Calculating the current from the strength parameters and v.v. is done in globlib> getIfromK, getKfromI. Due to non-linearity of the calibration curve, a bisection root finder is used [14].

The opacurrents unit provides a GUI and writes a .snap file to be read by the storage ring control system. Among other data, this file also includes the matrices for tune change, if previously calculated by otunematrix, and for chromaticity change, calculated by chromlib> UpdateChromMatrix.

The current calculation should be done here, not in globlib.

11.3 Test procedures

testcode

testcode.pas (37.3k)

---

uses globlib, linoplib, mathlib, ASaux.

This unit is for temporarily needed procedures or for testing new stuff, which then, if it works well, is moved to another place.

In the present (September 3, 2025) version the unit contains, among others, three procedures for reading MAD .mad, MAD sequence .seq and ELEGANT .lte files. The procedures are yet incomplete but nevertheless facilitate reading these files.

References

[1] OPA Tutorial, https://andreas-streun.de/opa/tutorial2.pdf

[2] OPA userguide, https://andreas-streun.de/opa/opa4.pdf

[3] Inside OPA, https://andreas-streun.de/opa/inside.pdf

[4] Semantic versioning, https://semver.org/

[5] G. Slabaugh, Computing Euler angles from a rotation matrix, https://eecs.qmul.ac.uk/~gslabaugh/publications/euler.pdf.

[6] V. Ziemann and A. Streun, Equilibrium parameters in coupled storage ring lattices and practical applications, Phys. Rev. Accel. Beams, 25, 050703, 2022, https://link.aps.org/doi/10.1103/PhysRevAccelBeams.25.050703.

[7] K. Wille, Physik der Teilchenbeschleuniger und Synchrotronstrahlungsquellen, 1996.

[8] D. Edward and L.Teng, *Parametrization of linear coupled motion in periodic systems,* IEEE Trans.Nucl.Sci. 20, 885 (1973), https://ab-abp-rlc.web.cern.ch/AP-literature/Edwards-Teng-1973.pdf.

[9] D. Sagan, D. Rubin, *Linear Analysis of coupled lattices,* Physical Review Special Topics–Accelerators and Beams 2 (1999) 074001, https://doi.org/10.1103/PhysRevSTAB.2.074001.

[10] A. Streun, A non-linear bunch compression scheme for SwissFEL, https://andreas-streun.de/phys/nlbuco.pdf

[11] J. Bengtsson, The sextupole scheme for the SLS, SLS-Note 9/97, https://ados.web.psi.ch/slsnotes/sls0997.pdf.

[12] Chun-xi Wang, Explicit formulas for 2nd-order driving terms due to sextupoles and chromatic effects of quadrupoles, ANL/APS/LS-330, 2012, https://www.aps.anl.gov/files/APS-sync/lsnotes/files/APS_1429490.pdf.

[13] S. C. Leemann and A. Streun, Perspectives for future light source lattices incorporating yet uncommon magnets, Phys. ST Rev. Accel. Beams, 14, 030701, 2011 https://doi.org/10.1103/PhysRevSTAB.14.030701.

[14] W. H. Press et al., Numerical Recipes in Pascal, Cambridge 1989.

[15] P. Bourke, CONREC, a contouring subroutine, https://paulbourke.net/papers/conrec/.

[16] Rosetta Code, Arithmetic Evaluation, https://rosettacode.org/wiki/Arithmetic_Evaluator/Pascal.

[17] B. Riemann, M. Aiba, J. Kallestrup, A. Streun, Efficient algorithms for dynamic aperture and momentum acceptance calculation in synchrotron light sources, Phys. Rev. Accel. Beams, 27, 094002,2024, https://doi.org/10.1103/PhysRevAccelBeams.27.094002.