User Guide for IR Chess Statistical Analyzer

1 Setup and Introduction

The name "IR" stands for "Intrinsic Ratings"—per the title of my AAAI 2011 stem paper "Intrinsic Chess Ratings" with the late Guy Haworth (who had a separate Bayesian approach that is *not* implemented by this code). The items needed to run the program are:

1. The compiled executable of the source `IRall1file.cpp` or the unpacking of that file into its constituent `.h` and `.cpp` code units (which are described further in the "non-quick" part of this guide). We will use `IRW.exe` as its name. Its location is called the *home folder*.
2. Some *reference files* `RefSF11.aif`, `RefKom13.aif`, `RefSF7.aif`, and/or `RefKom10.aif`, preferably in the same location as the executable. `RefSF15.aif` is in preparation. These files are 86–100 megabytes each.
3. One or more other files in the custom `.aif` format to use as main data. These can be located in the home folder or in a separate *main data folder*. The program offers a pre-set path for the latter at startup; you can say `no` followed by giving a path to change it.

The code produces numerous files in the home folder, the most important being `IRcommandLog.txt` and `IRsessionData.txt`. The former logs every command entered, and (after some editing) enables re-playing any session. The latter preseves all data output—beware that it can grow large with frequent use, though on modern computers the couple hundred megabytes I accumulate a year is no problem. Additional output files can be created on the fly for specific tests. Sessions are labeled and UTC-timestamped in both files.

The extension `.aif` for "Analysis Interchange Format" (which is described in the *Gödel's Lost Letter* article https://rjlipton.wpcomstaging.com/2015/01/20/a-computer-chess-analysis-interchange-format/) can clash with `.aiff` for "Audio Interchange File Format." The clash helps avoid `.aif` files being rejected in some cases, but some machines change `.aif` to `.aiff` on download. Only the reference files are looked for by name—so change `RefSF11.aiff` to `RefSF11.aif` and so on if that happened.

The program should be run from a console, such as the Windows Command Prompt, not by double click. Give the window height at least 40 lines and a scrollback buffer of at least 3,000 lines. Navigate the console to the home folder and enter the executable name plus two command-line arguments, e.g.:

```
IRW SF11 UW
```

This loads the reference file for Stockfish 11 and invokes unit weight for all positions. The alternative engine labels are `Kom13` (which gives Komodo 13.3), `Kom10` (the original Komodo 10), and `SF7` (Stockfish 7, in all cases with default settings except for 512MB hash table size). The alternative to `UW` is `EWN` for "expectation weights normalized." The latter gives more weight to positions that are more complex and difficult, and thus is targeted to "smart cheaters," but tends to reduce the z-scores of "non-smart cheaters" by dint of how the data is clumped. *The program is case-sensitive* and runs entirely in text mode—no GUI.

The program makes one screen-prompt query before showing the main menu. Any answer other than `n`, `N`, `no`, `No`, or `NO` is interpreted as 'yes' plus giving a label for the current session. Then comes the main menu, after a pause to read the reference file (twice) and display numerous messages:

```
 [1] Change (equation model of) Focus Trial <changeTrial>
 [2] Fixed Settings <fixedSettings>
 [3] Show Active Trial and Filters <showTrial>
 [4] Skill-contingent scaling; outer parameters <slideScale>
 [5] Add More Game Turns <addTurns>
 [6] Clear Turns And Filtered Tuples <clearTurns>
--------------------------------------------------------------------------
 [7] Define New Move Filter <newFilters>
 [8] Attach Filter(s) to Focus Trial <attach>
 [9] Detach Filter(s) from Focus Trial <detach>
[10] Clear Filters from Focus Trial <clearFilters>
[11] Hide Filters <hideFilters>
--------------------------------------------------------------------------
[12] Define New Move Selector <newSelectors>
[13] Toggle Move Selector <toggleSelectors>
--------------------------------------------------------------------------
[14] Define and Load New Trial Spec <newTrialSpec>
[15] Show Active Trial Specs and Load One <loadTrialSpec>
[16] Hide Trial Specs <hideSpecs>
--------------------------------------------------------------------------
[17] Run Fit to Find Best (s,c,...) <runFit>
[18] Run Performance Test on Focus Trial <perfTest>
[19] Iterate to calculate IPR <runIPR>
--------------------------------------------------------------------------
[20] Attach Output File <addOutputFile>
[21] Close Output File <closeOutputFile>
[22] Process Commands from File <readCommands>
[23] Quit <quit>
```

Options 1,2,4 are at research level and should not be changed in normal use; 4 and 14 will be invoked as-needed by other routines. The program has a *single focus* for data and operations. (A feature that allowed storing multiple trials and loading them into focus was disabled to keep things simple; the lone vestige is the "reference trial" used for IPRs and skill-contingent scaling.) Option 3 describes what is currently in focus, after a `y/n` prompt for whether to output the loaded move data using one line per position to the screen and all output files (this is refused if there are more than 1,000 positions anyway). Option 5 is followed by a "glob" argument specifying AIF files to load. The program filters out duplicate games. If any "Game Filters" (marked 'GF' in displays) are loaded, the program also skips games that do not match them. Option 6 clears all data and takes no argument.

All menu items can be invoked by number or by the name in brackets. Interaction is fully streamed so that multiple *space-separated* commands can be typed ahead before hitting carriage-return to invoke them. This enables a rudimentary "scripting language" that can also be invoked from stored text files via option 22. Examples are given below that can be readily customized to execute performance tests (option 18) and Intrinsic Performance Rating (IPR) calculations (option 19, which invokes options 17 and 4 alternately to give skill-contingent scaling). **These options are the main workflow of the program.** Option 20 allows saving the output to a new or existing file, 21 for discontinuing output to a named file.

Option 7 leads to a large menu of basic database functions for creating *filters* to select subsets of the loaded data. Option 8 shows a long list of predefined filters and exemplifies the "catalog interface" in

which active elements are marked with a `*`. Catalog elements can likewise be accessed by number or by name. Option 9 allows making individual filters inactive. Option 10 clears all filters but leaves all defined ones in the displayed catalog. Option 11 allows moving never-/no-longer-wanted filters off the displayed catalog (to cut down the lines displayed onscreen) to the "hidden catalog"—which can be accessed anyway via the menu option `-2`, `moreOptions`, on main menu option 8.

Options 12 and 13 define special selections of moves in any given position, thus widening the notion of a "match" in a performance test. The interface is similar to that of filters, but simplified by allowing one menu to toggle them active/inactive rather than the separate options 8 and 9 with filters. The reason for the difference is that filters are more of a safety concern, so their state is absolute rather than "modal." User-defined selectors become additional z-tests in the "Performance Test" display; numerous selectors are permanently active. Another difference is that whereas loaded filters act like Boolean `and`, selection rules are conjoined only when an `AndSelector` is explicitly created. In normal workflow, no additional selectors are needed—so these options may be ignored.

Options 14 and 15 (and 4) manage values of configurable model parameters. Although there are currently 43 parameters in all, only 3 of them are activated as independent characteristics of players:

- `s` for *sensitivity* to small differences in the values of moves (lower is better);
- `c` for *consistency* of avoiding mistakes, especially in tactical positions (higher is better);
- `ev` for influence of *equal-value* moves (lower means less swayed by impulse).

A sizable subset of "slide-scale" parameters are treated as characteristics of the *level* of play but not of individual players. The basic points are depicted graphically in my *Gödel's Lost Letter* blog articles "When Data Serves Turkey" and "Sliding-Scale Problems":

- https://rjlipton.wpcomstaging.com/2016/11/30/when-data-serves-turkey/
- https://rjlipton.wpcomstaging.com/2018/09/07/sliding-scale-problems/

The program is initially set up to treat `ev` as tied to `s` and `c`. It was treated independently in the model-building stage. Another parameter, `co`, is tied equal to `s`; this makes two fractions in the model's utility function share a common denominator.

Different combinations of `s` and `c` correspond to the same Elo rating R, much as players of the same strength can have different mixes of strategic and tactical skill. The regressions over thousands of games at each rating (in steps of 25 from 1000 to 2825, only a few hundred games per "bucket" at 2700+) yield uniform progressions s_R and c_R in terms of R, representing the "central tendencies" of players of that rating. This *central fit* is how the performance test (option 18) sets parameters for a given rating R. Issues of players having different tradeoffs of `s` and `c` for the same rating `R` are addressed in my ICGA ACG 2015 paper "A Comparative Review of Skill Assessment: Performance, Prediction and Profiling" with Haworth and my PhD graduate Dr. Tamal Biswas:

- https://cse.buffalo.edu/ regan/papers/pdf/BHR2015ACG.pdf.

More simply put, options 14 and 15 can be left alone, being manipulated by options 17–19. Option 17 allows a host of choices of active parameters and "loss functions" for the model-fitting regression, but unless some "slide-scale parameters" are expressly freed in the parameters submenu, this is best left to control by option 19. Option 18, it is important to note, does not involve regression. It also has a

number of configurable options, but aside from the rating to test (plus slack allowance set to add 25), they can be left alone.

Thus, the main user action items are options 5–10 and 18–22. Usage can be further streamlined by working in terms of pre-scripted blocks of commands.

2 Examples of Commands

Suppose we have placed the following files in the main data folder:

```
CandidatesMadridJune2022cat21_Kom133d20-30pv64.aif
CandidatesMadridJune2022cat21_SF11d20-30pv64.aif
Candidates2020cat21C24_Kom131d20-30pv64.aif
Candidates2021cat21C24_Kom133d20-30pv64.aif
Candidates2020cat21C24_SF11d20-30pv64.aif
```

A note on nomenclature: `pv64` means that the data uses the engine's Multi-PV analysis mode. The program **will not handle** data in the usual Single-PV engine playing mode, which is notated `pv1`. (To make the program give errors on such data is a deliberate safety choice.) The `d20-30` part means a variable-depth mode where the engine is run until depth 20, then is allowed higher depths within a given time budget (measured by number of nodes) budget, but stops at depth 30 even if time remains. The engine used is after the underscore; the 2020 games were run with version 13.1 not 13.3 of Komodo but the differences are minor enough to combine. I used to mark files from chess24.com with 'C24' to warn that that became the `Site` field, but I've not found reason to continue caring, nor do I manually change it to e.g. `Chennai IND` because the ease of forgetting breeds inconsistency in the `Site` field. Here are some examples of commands and globs:

```
clearTurns addTurns Candidates*Kom13*aif
```

This clears out previous data and loads the three Candidates files with Komodo 13. Command lines must always have a final carriage return to be acted on, but commands within a line can be separated by spaces.

```
newFilters TurnNum from14 geq 14 TurnNum to60 leq 60 done
```

This creates and loads two filters that combine to isolate turns 14–60 of all games. Creation always begins with the type of the filter (always a capitalized name—following the conventions of a "function object" class in the C++ code) followed by the particular name—which can be arbitrary. Names cannot have internal spaces but may have hyphens, underscores, dots, etc. Multiple filters can be created in a chain as shown above.

```
newFilters PlayerToMove Nepomniachtchi2m Nepomniachtchi done
```

This selects the moves made by GM Nepo. The part of the player's name must be spelled out in full exactly as it appears in the AIF file (which replicates PGN game headers), not counting spaces and commas. One must be careful that hyphen is `not` a delimiter, thus "Vachier-Lagrave" is treated as an indivisible unit and will not match "Vachier" by itself. (This is a pain because sources vary

between hyphen and space. But making hyphen a delimiter would be worse.) The name-matching system does not accept "globs" like `Nepo*` the way file systems do. But provided no other player in the loaded data has 'Ian' as a block, one could get the same results by `newFilters PlayerToMove Nepomniachtchi2m Ian done`; nothing constrains the name of the filter to match the content. The "`2m`" is my habitual way of saying "to move."

If you want to distinguish the Candidates games that were played in 2020 versus 2021, you can do

```
newFilters YearIs yeq2020 eq 2020 done
```

and similarly for 2021. All numerical comparison filters have similar syntax where the element after the class and name is one of `leq`, `eq`, or `geq`. Strict $<$ and $>$ are not provided but can be simulated in one of two ways, say for "$x > 1.00$" figuratively meaning "advantage more than a pawn":

(a) Use the `NotFilter` construction applied to the comparison $x \leq 1.00$. There are also `OrFilter` and `AndFilter` constructors giving full Boolean logic.

(b) For comparisons involving decimal numbers, you can simulate something like "$x > 1.00$" as "$x \geq 1.0001$." With unscaled centipawn units, which work in strict steps of 0.01, there cannot be any difference. The program's *scaled* evaluations can be arbitrary decimals—which the program rounds to 2, 3, or 4 decimal places for displays—so in rare cases a position could have scaled value between 1.00 and 1.0001 and be mis-classified. Notwithstanding this, the program only claims 3-place precision of values and 4-place precision of averages, so this kind of difference must be regarded as immaterial.

There are limitations to the Boolean logic. Put another way, the program does not have fully *relational* database capabilities, only what is sometimes called a "flat-file" or "card box" database. For example, one cannot make a query to select those players who were common to both the 2020 and 2022 Candidates, or any two tournaments, automatically. One can "manually" determine that they were Caruana, Ding Liren, and Nepomniachtchi, and use Boolean logic to disjoin them:

```
newFilters
PlayerToMove Caruana2m Caruana
PlayerToMove Nepo2m Ian
PlayerToMove DingLiren2m Ding
OrFilter CommonPlayers Caruana2m y Nepo2m y DingLiren2m n
done
detach Caruana2m y Nepo2m y DingLiren2m n
```

Some points about this: Each filter construction begins with the always-capitalized class of the filter, then a name you supply, then its body. The name of the filter does not matter for the program logic—thus it was fine to abbreviate "Nepomniachtchi" to "Nepo" in that name—but you must be consistent with that name in later commands. Filters exist as soon as they are created, so it was not necessary to exit with `done` and do `newFilters` again before creating the `OrFilter`. They are all loaded at top level (unless created in nested fashion as exemplified below), so one must `detach` the three individual filters to avoid what is otherwise a logical contradiction creating a "null filter."

The use of `Ding` will not clash if the data included moves by the German FM Florian Dinger, because the match is delimited by spaces, commas, and string boundaries. But in case Ding Liren and the Chinese WGM Yixin Ding were in the same tournamernt, one would have to create separate filters

called (say) `Ding2m` and `Liren2m` and then define `AndFilter DingLiren2m Ding2m y Liren2m n`. It is possible to nest this so that the components `Ding2m` and `Liren2m` are *not* exposed at the top level:

```
newFilters
AndFilter DingLiren2m moreOptions PlayerToMove Ding2m Ding y moreOptions
      PlayerToMove Liren2m Liren n
done
```

To understand this: The `y` and `n` go with `AndFilter`. After the overall name `DingLiren2m`, the syntax expects either the name or the catalog number of an existing filter. If you say `moreOptions`, however (or give `-2` as the catalog number), the program goes into creation mode. This internal creation is not ended by `done`; rather, once you finish `PlayerToMove` by giving the name `Ding`, it jumps out to ask if you want to define another conjunct for the `AndFilter`. The `y` signifies yes, so we repeat with the "Liren" part and then the final `n` stops the `AndFilter`. This pops execution up to the top level of `newFilters`, and when you say `done`, only the name `DingLiren2m` is added to the global catalog.

One cannot achieve this in a shorter way by writing, say, `PlayerToMove DingLiren2m "Ding Liren"` instead. This would fail to match the form `Ding, Liren` with internal comma in a PGN file anyway, not to mention the order of `Liren` and `Ding` being switched. This is also a reason why the program *never* allows spaces within blocks of input; space is always only a delimiter. Carriage return is equivalent to space when reading data but causes a line of commands to be acted upon.

Because Boolean logic can be clunky, it is useful to bear in mind that loading multiple filters naturally works like Boolean *and*, and there are shortcuts for some cases of *or*. For example, if we are happy that "Liren" is unique in the data, we can define:

```
newFilters
PlayerOrToMove CommonPlayers Caruana Ian Liren .
done
```

The period `.` *necessarily preceded by a space* terminates the list of players. Now there is no issue of having to `detach` the three components.

To distinguish games within a tournament, there are two main ways: by round or by date. The PGN standard specifies that dates must be in the Euro `YYYY.MM.DD` format, so it is right for the `DateIs` filter in my program to require it. There is no such standard for the PGN `Event` and `Site` fields (despite TWIC strictly giving the latter as city followed by the 3-letter country code with no comma), and this feeds into why my program has no support for cataloguing tournaments. For rounds, my program tries to handle multiple conventions; indeed, I extended it to handle notation like "`23-07`" for the Women's Speed and the double-decimal monstrosity "7.24.3" where 7 is the round, 24 the team pairing, and 3 the board within the team match (or vice-versa?). There are some onscreen prompts about these matters.

Once you have determined the appropriate filters—and maybe double-checked the selection by doing `showTrial y` if there are at most 1,000 selected positions—the workhorse commands are simple:

- `perfTest useRating` R `goTest`
- `runIPR` R_0 *name*

In the former, you fill in a value for the rating R. They system will actually test $R + 25$ unless you expressly set the slack to zero, i.e., do `perfTest ratingSlack 0 useRating` R `goTest` instead. For

safety reasons, the `ratingSlack` field does *not* stay set to 0 when you do this. (You could also just give `useRating` $R - 25$ to get rating R.) There are theee special `useRating` values:

- `0` preserves the rating entered in a previous test (or creation of a "`TrialSpec`" via the `rating` field in main-menu option 14).
- `1` computes and uses the average rating R of the player(s) to move in the data. Again, you actually get $R + 25$ unless you include `ratingSlack 0` before `goTest`.
- `-1` computes and uses the opponents' average rating instead.

The option `1` would enable you to test Caruana, Nepo, and Ding as if they were one player using their average rating (weighted by how many moves they made). One can use it to test all players from a given country by filtering the PGN `WhiteCountry` and `BlackCountry` fields; the `newFilters` menu has a `PlayerCountry` option which handles the white/black determination. Whether it is appropriate to average data this way, as opposed to combining separate tests of each player by the Fisher-Stouffer rule—is a thorny technical matter. In the case of multiple players from a round-robin or knockout finals stage of a tournament, I would bless doing so. But for evaluating Team Norway's performance at the Olympiad, where Carlsen has such a higher rating, maybe not.

For `runIPR`, the R_0 is only an initial guess to help speed the iterative process. The final answer should be the same unless R_0 is weirdly extreme. This option sanitizes the now immensely complicated main menu option 17, `runFit`, which is now targeted for technical research. Both routines create a set of parameters that best fit the data. The set is called a `TrialSpec` and has a "`ScaleSpec`" subset which you will see the program chatter about. It is good to give your spec a *name* in advance, but if you don't, the system will prompt you for the name before you see the final results.

3 Scripted Runs

What all this works toward is the principle that it is *safer to script a formal test entirely in advance rather than work interactively.* Thus, for instance, to do a performance test of Ding Liren in Madrid using his 2806 June rating, we can type the following into a text file and then paste it into the console window:

```
IRW SF11 UW
y
clearTurns addTurns Candidates*2022*SF11*.aif
newFilters
PlayerToMove Liren2m Liren
done
perfTest useRating 2806 goTest
runIPR 2800 LirenMadridSF11IPR
```

The carriage return after the first line is necessary because it is executed at console level. The one after the second line is advisable because it is the opening prompt. The rest could be all one one line with spaces (or tabs) in between, but a final carriage return is needed for the last line to be acted upon. It is fine (IMHO) to mouse-copy this and paste it into the console window, but remember either to include the last line's carriage return in the copied text or hit return after pasting.

This is without removing book. In big tournaments I often resort to approximating the removal of book by using the `TurnNum` filter constructor to compare move numbers. To test only turns 14–60, do:

```
IRW SF11 UW
y
clearTurns addTurns Candidates*2022*SF11*.aif
newFilters
TurnNum from14 geq 14 TurnNum to60 leq 60
PlayerToMove Liren2m Liren
done
perfTest useRating 2806 goTest
runIPR 2800 LirenMadridSF11IPR
```

(The turn-60 cutoff comes from a highly technical phenomenon that I and certain others have been unable to "model away.") The filter logic can build turn selections for each individual game. I have a pair of Perl utilities to help with the complex syntax, `pgn2CT.pl` and `ct4.pl`. The former converts a PGN file into a form where you can enter the novelty moves by hand. For the G/3+1' segment of the match between WGMs Kateryna Lagno and Humpy Koneru from the Women's Speed quarterfinal, it creates a file in a format best explained after filling in the needed by-hand information:

```
WSCC Main Event 2022 : LagnoVsKoneru Chess.com Online, 2022
9 2022.07.19 29-01 Lagno, Kateryna -- Koneru, Humpy: 13...Qd7
10 2022.07.19 29-02 Koneru, Humpy -- Lagno, Kateryna: 10.Rb2
11 2022.07.19 29-03 Lagno, Kateryna -- Koneru, Humpy: 18...Be5
12 2022.07.19 29-04 Koneru, Humpy -- Lagno, Kateryna: 10.h4
13 2022.07.19 29-05 Lagno, Kateryna -- Koneru, Humpy: 15.a4
14 2022.07.19 29-06 Koneru, Humpy -- Lagno, Kateryna: 10...Nf6  % repeat of game #12
15 2022.07.19 29-07 Lagno, Kateryna -- Koneru, Humpy: 15.Re2
16 2022.07.19 29-08 Koneru, Humpy -- Lagno, Kateryna: 10...Nfd7
```

I've inserted "LagnoVsKoneru" by hand because it becomes part of the names of filters created. Otherwise, you could get multiple filters named `Chess.comOnline2022` which would clash. The initial numbers on the succeeding lines are used only as labels within the names of the filters for individual games and do not need to start from 0 or 1 or even be in sequence. The column after the date, however, must have the round number *exactly* as it appears in the PGN file, including any decimals. After the colon, the important elements are the turn number *and* either `.` or `...` with no intervening space, to tell whether it is a White or Black move. The move itself, or any following text (after a percent sign) is not used internally—but of course, this helps when I include this literal text into my cheating reports. The script `ct4.pl` translates a file `foo.txt` of this form into `foo.cmd` with commands that my program can read via main menu option 22, called `readCommands`. So is the PGN file is called `LagnoVsKoneruWSCCFinalsG4Blitz2022.pgn` then we will get `LagnoVsKoneruWSCCFinalsG4Blitz2022CT.txt`, which we edit manually. Then applying `perl ct4.pl` to that file gives `LagnoVsKoneruWSCCFinalsG4Blitz2022CT.cmd`, usable as follows:

```
IRW SF11 UW
LagnoG4testSF11
addOutputFile LagnoG4Results.txt LagnoG4testSF11
clearTurns addTurns WomensSpeedFinals*SF11*.aif
readCommands LagnoVsKoneruWSCCFinalsG4Blitz2022CT.cmd
newFilters
PlayerToMove Lagno2m Lagno
done
```

```
attach pnew5norm n detach pnew4norm n
perfTest useRating 1912 goTest
runIPR 2500 LagnoG4vKoneruSF11IPRNB
```

The third-from-last line widened the cap on advantage for the player to move from 4.00 to 5.00 because Stockfish 11's evaluation function is "inflated" even compared to versions through Stockfish 9. The IPR is standardly defined without removing book (as above), so I use `IPRNB` in names to record its removal. All output gets logged the the file `IRsessionData.txt` *and* all user commands get logged to `IRcommandLog.txt`, so the exact course of a session can be audited. (The latter also shows actions taken by some commands, which need to be manually edited out if you want to actually copy and replay the sequence.)

Note that I changed the simple `y` on the second line to a descriptive name for the session. The third line creates a dedicated output file (besides all data getting dumped to `IRsessionData.txt`); the syntax requires giving a session name too, which I made the same as the name of the session overall. Any session name other than `no`, `No`, `NO`, `n`, or `N` is legal and interpreted as a yes answer to the suggested main data path. The main z-score is after `Combined:  adj` on a line by itself. The z-score of the alternate "predictivity" test is the number after `adj` on the line beginning `CombMultiWtd`; for technical reasons it is negated. The final IPR is on the line beginning `IPR-` at the end; I generally prefer to quote the "simple" form on the next line which is rounded to the nearest 05.

4 Backdoor Features and Design Quirks

Some of these have even fooled me when I've forgotten them for a time:

- The rating 2001, if specified by the user, has the special action of turning off the "sliding scale" hyperparameters. The same happens if you test a 1976 rating with adding 25. This is OK *at* the rating 2001, so does not disturb a test, and goes away as soon as you test another rating. But if you do certain other things after testing 2001 (especially with menu option 17 without using 19 as its front end) you can get unexpected results. The program says its results are "`(ref2001)`" as a warning. (You can create other fixed-reference ratings; this is so wonky it is not even yet in my material below.)

- AIF file names beginning `Ref` are treated in "magic" manner as additions or substitutions to the reference file. Long story short: please avoid. There are 18 players on the August 2022 FRL with surnames from Refaat to Refuveille; do not begin AIF files with their names.

- Filters marked "`(GF)`" depend only on characteristics of a game, not on any positions in that game. For example, whether Black's Elo rating is at least 2000. If any GameFilters are active at the time `addTurns` is executed, they *prevent the loading* of games that fail the filter. If you wish to load such games, you must `detach` them first, load the data, then `attach` them again. If you are interested in only one player in a large tournament, this feature can greatly save memory and loading time—but if you are frequently changing both filters and data, this can mislead. A tricky case is the `PlayerElo` filter class. If you define `PlayerElo moveBy2000plus geq 2000` then the filter can depend on the position insofar as whether White or Black is to move, if only one player is 2000+. But it is still marked `GF` because it will suppress loading games in which *both* players are under 2000.

- The Ctrl-C key combination kills the program except when the regression is running. From option 17, it aborts the regression and leaves the last TrialSpec computed in the regression in-

place. This is handy if progress is slow and a non-fit is evident. From option 19, `runIPR`, it aborts to the last iteration of option 17, which is done at higher precision. A further Ctrl-C then aborts that regression as before.

- One can configure parameters differently for `runIPR` by going into `runFit`, making the desired changes, and instead of selecting `go`, do `-3` which is the code for `cancelChoice`. You will get a prompt that asks whether you wish to save the new settings. Answer `y`, then do option 19, `runIPR`.

- If you give option 5 a file name or glob without '/' or 'ṕath information, it will look for matching file(s) first in your home directory (where the executable file is). If any files are found there, the search stops. If not, then the *main data path* is polled. It is neatest to keep only the reference files in the home folder, so polling the main data path is the normal mode of operation. If your argument to 5 includes a path (which can be relative including use of `..` once), and matching file(s) are found at that location, this supersedes looking up the main data path. If not, and if the path is relative, then *maybe* it will be read relative to the main data folder.

- Command files cannot have commands to open other command files. All reading of command files must be at top level. This is one of several self-limitations for safety, and is the one I may soonest liberalize.

- Some menus allow toggling choices on.off at the same screen, but some do not—most notably, the options for attaching and detaching filters are completely separate entries in the main menu. The point is that toggling is *modal*, and modality creates an unsafe dependence on earlier commands.

- Comparisons are always inclusive of their endpoints. This owes more to my experience with Constructive Mathematics than with quantum theory as in the 'linearization' point.

- The main modeling concept is that every set of values for the parameters—especially "sensitivity s" and "consistency c"—defines a *virtual player* Q. The IPR regression works by finding the Q that is closest to the performance of the actual player P. Then the IPR is calculated by calculating how Q is expected to perform on a fixed set of positions, namely the reference file for the engine. The actual T1-match and centipawn loss in the games by player P is not tallied directly (as the screening does) but rather factored into the regression. The regression is how the *difficulty* of the positions faced by a player is accounted, which the simple "box-score" tallying of *PGN Spy* and other screening-like utilities simply misses.

- Depths are *virtual* and *smoothed*. As with the screening files, the Multi-PV AIF files tend to go to higher depths in endgames and simpler positions, under the supposition that players are able to calculate more moves ahead. Unlike with screening, the Multi-PV AIF files go to depth at least 20 even with the newer engines. Whatever the highest depth $D \geq 20$ reached in the analysis, it is mapped to depth 20 (or to whatever is specified as the "judgment depth" in the model-configuring parts of the code) on a virtual scale. A depth $d < D$ will be mapped to $d^* < 20$—or more precisely, the engine's value at depth d will be split up into the virtual value at depth d^* and the depth $d^* - 1$ and/or depth $d^* + 1$. What thereby gets *smoothed* is the tendency of engines to have a "brain fart" about a move at one depth d which gets corrected at the next depth d'. Then the values at virtual depths d^* and $d^* + 1$ will each have a little of both, and the over-sharp difference between depths will be muffled.

- Evaluations are virtualized too—by what is essentially the same scaling as plotting stock prices on log-log charts where percentage change rather than raw points change is what matters. Matej Guid and Ivan Bratko were aware of the need but acted only as far as making a ± 2.00 cutoff

in advantage for the player to move. That the phenomenon goes stringly all the way down to 0.00 is graphically demonstrated in my "When Data Serves Turkey" article linked above. An independent analyst gave me a demonstration that my scaling has equivalent effect to standard general statistical noise-reduction procedures. Theoretically, the scaling should correspond to the win-lose-draw expectation, which is how AlphaZero and other neural engines calculate evaluations, but the nub of my "Sliding Scale" article is that what works hummingly for AlphaZero loses its simplicity for modeling human players of all ratings.

- The program is *fully linearized*. This means that all individual choices of numerical items are treated as potentially being linear combinations of multiple choices—quite literally the "Schrödinger's Cat" view of the world. This is most transparent in main menu option 17 where the user can compose "loss functions" with arbitrary weights, but it operates in some places where I followed the design philosophy without really craving it. The above treatment of depths is less transparent. The pivotal case is the weighting of positions. I've wound up using simple unit weighting—which is the case where every coefficient has value 1.0—over 99% of the time. I've tried numerous other weighting schemes, but only my recent idea of "expectation weights, normalized" (`EWN`) has shown internal signs of good behavior. Before my code bulked up, I used to display results under unit weights and general wights side-by-side. Now you see only the "`...Wtd`" version of everything, but when the weights are unit, it's the same as the unweighted version anyway.

I hasten to explain a point that I always put in my long-form resports for over-the-board cases: Moves where there is an obvious recapture are already discounted by my model in a manner separate from giving them tiny or zero weight. Namely, my model will generate a prediction probability for the recapture of basically 1.00, so the fact of the player matching it will add almost nothing to the z-score.

But it would be nice to give obvious-move positions small weight, which EWN will do organically. EWN up-weights positions where not only is there more at stake, but there are enough ponderable choices to create a substantial likelihood of error. By dint of being normalized, EWN will give those positions weights higher than 1.0, to offset the obvious-move cases. It also gives small weight to positions with many choices but where they all keep the evaluation at 0.00, say. That is more problematic as policy, and I'll just say that the use of depth weights and other volatile features attempts to detect things such as "drifting into the zone of one msitake" as the saying goes. The use of EWN is to try to isolate "smart cheating" on critical positions by up-weighting them, and in a non-cheating context, to measure how difficult the positions are—e.g. so as to calculate the amount of challenge a player creates for opponents. The downside is that thus "chunking" the weights tends to increase the denominator of the z-scores, thus blunting the instrument in cases of "dumb cheating"—as has been seen all too often online during the pandemic, alas.

5 Code Organization

The program has nine major sections. Much of the code is inlined in .h files only. Some parts have long descriptive comments. The basic organization is:

1. Utilities, math functions, chess notation:
   - Files `IRincludes.h`, `IRutil.h`, `IRfunctions.h`, `Position.h`
   - Classes SimpleDate, MoveParse, Move, Position; namespaces IRutil, IRfunctions
2. The text-based, extensible menuing and logging system.

- File `Menus.h` starts with global interfaces Logging and Catalogable.
- Classes EnumMenu, GoMenu, DynamicMenu, the last composed into Catalog, ValueCatalog, and RefCatalog.

3. Model settings, model parameters, configuring the utility function, generating likelihoods.

   - File `TrialSpec.h` with class TrialSpec (Catalogable) holds 40 model parameters to fit. The parameters are divided into an "inner tier" and an "outer tier"; the latter are referred to as if a class ScaleSpec existed but are not coded separately.
   - File `Scaler.h` with class Scaler (Catalogable) and subclasses NonScaler, LinearScaler, LogScaler, LogisticScaler, which scale down engine-given move values to reduce extremes.
   - File `EvalHandler.h` with class EvalHandler (Catalogable) to manage model settings that are fixed rather than fitted.
   - File `ForwardSpec.h` with class ForwardSpec: Translates TrialSpec into model equations according to EvalHandler settings. For instance, the 's' parameter can be inverted and/or made to multiply $v_1 - v_i$ rather than just v_i.
   - File `FitDataTuple.h` with classes CurveApplication, which holds move utilities u_i, likelihoods L_i, and probabilities p_i, and FitDataTuple, which evaluates quantities to be multiplied by model parameters.

4. Chess game and analysis data read, filtered, and modified.

   - Files `GameInfo.h`, `TurnInfo.h`, `FilteredTurn.h`, and `DecisionInfo.h`
   - Classes GameInfo, TurnInfo, FilteredTurn, DecisionInfo. Use is described below.
   - (Also `Shuffle.h` with class Shuffle which is currently disabled.)

5. Apply parameters to data to generate probabilities.

   - File `IRmodels.h` with namespace IRmodels and a class MoveProjection which holds probability projections according to user-chosen model equations.
   - Enum MODELS governs the choice of model, which includes (proto-)linear, log-linear, and loglog-linear models.

6. Turn filters and move selectors determine which positions and move predicates are tested.

   - File `MoveSelector.h` has abstract base class MoveSelector (Catalogable) with a hierarchy of function-object classes that select subsets of moves for any position, such as all Knight moves.
   - File `TurnFilter.h` has abstract base class MoveSelector (Catalogable) with a hierarchy of function-object classes that govern which positions are loaded for a test.
   - Users may dynamically create TurnFilters (and Selectors to a more-limited extent) and form Boolean combinations of them.

7. Performance data is collected and compiled into test results

   - Files `PerfData.{h,cpp}` with classes SimpleStats, PerfData, IPRstats, BootstrappedItem and the following hierarchy:
   - Abstract class StatsItem deriving virtual base classes AggregateStat and TestItem which converge into the bellwether class AggregateTest and subclasses MoveIndexTest and SelectionTest.

- The test classes automatically manage projected and actual means and variances for z-tests etc.

8. Manager for data and fitting according to user-configured loss function
   - Files `Trial.{h,cpp}` and `Minimizer.h`, plus `TrialFilters.h` for filters that involve fitting-dependent criteria, and third-party optimization package files.
   - Class Trial (which is Catalogable, but the feature of multiple Trials—each with its own model equation and data—is disabled) holds data, executes statistical fitting, and executes performance tests with fitted or assigned parameters.
   - Class Minimizer provides user-configured loss functions and interfaces with minimization algorithms in GSL and other libraries.
   - Files `Heap.h` and `HeapNM.h` with an original implementation of Nelder-Mead minimization via a heap data structure, replacing the GSL implementation.
9. Ensemble coordinates menus and actions and streams and session logging.
   - Files `Ensemble.{h,cpp}` have the omnibus single-instance class Ensemble(Logging).
   - File `IRmain.cpp` merely builds an Ensemble and sets the stream interaction going.

6 Setup and Dataflow

Here is what happens on startup when the program is run:

1. The lone Ensemble instance is created along with both the "focus trial" and the "reference trial"; the latter trial is used only to set a fixed benchmark for statistical tests (which represents a particular master fitting of the model). System paths are set up—the user confirms them by giving any response other than 'n' or 'no' and this becomes the recorded name for the session. Then a script that is in the user language but coded within the program as strings above `main` is run to create initial settings for all configurable quantities in EvalHandler and TrialSpec.
2. Data from the Analysis Interchange Format (AIF) file `SF7Turns.aif` is read into both trials. This is held in raw form by GameInfo and TurnInfo.
3. The data flow is TurnInfo →(TurnFilters)→ FilteredTurn →(EvalHandler)→ DecisionInfo → (EvalHandler)→ FitDataTuple.
4. TurnInfo holds the raw values in the AIF file(s) from White's point of view in centipawn (CP) units. Mate values are stored as 100,000 centipawns minus the number of moves to mate. Indications for move values that are not available because they were pruned by the engine (PRUN), not recorded in lesser-PV mode (NREC), or otherwise not applicable (NA) are stored as-such.
5. FilteredTurn flips values around to the player-to-move's view and divides them by 100.0 to put them into two-place decimal ("Pawn") units. PRUN values are replaced by the worst recorded value of a move at that depth; then NREC values use the value at the highest-available lower depth for the move; and then NA values (which are usually for depths before the first depth at which the search reports values) use the value at the lowest-available higher depth for the move. FilteredTurn thus stores an "Unscaled Value" for every move/depth pair. It also maintains the highest and lowest recorded evaluations at each depth of search. This is done *only* for the turns selected by the "attached" TurnFilters.

6. DecisionInfo creates and stores a parallel table of "Scaled Values" for every move/depth pair, using both the fixed settings in EvalHandler and outer-tier "sliding-scale" parameter settings. Also per optional fixed settings regarding the handling of lower-depth values, it creates "swing" values needed for fitting (see next item). Extreme values are first capped (using user-configurable settings) by Scaler::capEval. Then they are scaled and otherwise mapped by Scaler::mapEval. Access to the unscaled values in FilteredTurn also goes only through this class, so it serves as the main value gateway.

7. FitDataTuple sifts down a DecisionInfo object into just the values actually used for statistical fitting and testing. Those values depend on the "judgment depth" selected by the user to *test* and the "telescope window" of (weighted) depths on which *predictions* are based. The latter window may consist solely of the judgment depth but is notionally apart from it. The FitDataTuple object is created in the body of DecisionInfo but not exported until the user begins a statistical fit or performance test.

Each TurnInfo, FilteredTurn, and DecisionInfo object holds an $L \times D$ array of values, where the number L of legal oves averages in the mid-30s and the number D of data depths is in the range 20–30. Since each array is derivable from the previous one, holding them in triplicate wastes space, but the latter two are generated only for the TurnFilter-selected data and are not altered during fitting. The FitDataTuple objects, however, have size proportional only to L (though some settings cause fitting methods to create up to D of them for each data point—this is real slow and we won't use them unless forced to). They are freely created and recycled on the fly—they are treated as temporaries with lifespan only to generate one set of projections for one game-position data point.

The initial script creates many more "available" TurnFilters than are attached by default and also creates a suite of MoveSelectors, all of which are active for inclusion in a performance data (PerfData) object. The difference between a TurnFilter and a MoveSelector is the former determines which turns in games are tested or regressed over, whereas a selector determines which moves in each position "pass" a certain test. There are also special selectors that represent numerical attributes of moves such as value and ordinal best-to-worst rank—they are configurable only from the fitting menu.

7 Main Menu

Most of the user options build resources or change settings without carrying out any "statistical activity." As grouped on the main menu, they are:

1. Change the main fitting equation between a log-linear model (called "Shares") or a loglog-linear model ("Power Shares") and a dozen other choices.

2. Change settings that are supposed to be fixed rather than fitted: scaling, whether a "patch" is used when moves are tied for equal (optimal) value, prediction depth(s), the judgment depth, the handling of "swing" (described below), and the mapping of how the user sees parameters to the fixed way the evaluator treats them.

3. Show the current settings and loaded data.

4. Edit parameters in the "outer tier" that are ultimately supposed to be fixed but may be fitted while the intended model upgrade is still in "R & D mode." Any edit to most these parameters causes all DecisionInfo objects to be remade.

5. Add more game data from files that may be specified using "glob" wildcards.

6. Clear the loaded game data.

The settings from these six items used to be savable as a catalogable "Trial" object, so that multiple trials each with its own data could be maintained, but this was unwieldy so the feature is currently disabled. Thus only one Trial (apart from the reference trial) exists at any one time, let along be "in focus"—but it can be change at will. The next group of user actions is:

7. Design a new TurnFilter. It is automatically attached after creationbut can be detached at will.

8. Attach one or more existing TurnFilters. This and detaching cause all FilteredTurn as well as DecisionInfo objects to be remade. When filteres are being swapped in and out and many game turns are loaded, it is quicker to attach first (even if it causes a warning that all the turns are filtered out) and then detach.

9. Detach one or more existing TurnFilters. They are kept in a Catalog which allows access by number (good for live use) or name (important for writing scripts since numbers may change). The numbers do not change when a filter is detached.

10. Clear all attached filters—this is even more useful in scripts than interactively.

11. Hide a filter from the "available" list—which in particular allows redefining it by the same name if the original definition was bungled. Filters on the hidden list (and there are some initially by default) can be made available by using the global "moreOptions" feature (numbered -2) in the menuing system.

The next block has numbers 12–13 with similar precedures for MoveSelectors, except that they toggle rather than have separate attach/detach operations and cannot be hidden. Then come:

14. Define a new TrialSpec. This menu shows all the "inner tier" parameters, with option to set (some of) them according to the Elo rating. This is used both to set parameters for performance tests and to choose a reasonable starting point for the fitting algorithms. This and the outer-tier parameter menu are also reachable directly from the fitting menu.

15. Load an existing TrialSpec from the catalog. Unlike filters or selectors, only one can be loaded at a time, so there are no separate "attach" or "detach" operations.

16. They can, however, be hidden. The current method of displaying them takes a lot of vertical screen space so this is handy. Also handy is that whereas TurnFilters and MoveSelectors disallow overwrites (one must hide the original first even if it is detached), TrialSpecs and EvalHandlers allow them.

Items 17–19 are the main statistical actions described above. All results are automatically stored in a local file named `IRsessionData.txt` but menu option 20 allows also directing them to a namable file for the current session. Option 21 can be used to stop this recording.

Option 22 reads and runs scripts from a file of commands. Commands can also be just pasted at the screen prompt up to the allowed buffer size, but it will be best to use files. The "scripting language" is simply whitespace-separated menu commands, preferably by name rather than number so they won't break if numbers are changed. Option 23 closes all files for a clean exit.