



Cuaderno de Tesis de Magíster

N.B.: The following is a template research diary written by Franco López Usquiano and intends to display the numerous capabilities of LaTeX to successfully register and track research projects. It is mostly oriented to Computer Science research in general and Machine Learning projects in particular but the template can be used for any kind of project.

PROJECT SUBTITLE

1 Entrada del 2021-11-20

Se compiló y generó la primera entrada del cuaderno de Tesis del Magíster. Como tip, hay que instalar el paquete **cm-super** para solucionar errores en la compilación.

Fuentes

- Plantilla original en GitHub
<https://github.com/JJG0/research-logbook.git>
- Fork en mi cuenta de GitHub
<https://github.com/flopezus/research-logbook>

2 Entrada del 2021-11-22

Recuento de códigos:

Tenemos el proyecto **sapito-program** alojado en mi cuenta de GitLab en el siguiente enlace:

- <https://gitlab.com/flopus/sapito-program.git>

cuyo contenido es el siguiente:









Name	Last commit	Last update
 .gitignore	Se agrega .gitignore	2 months ago
 README.md	Actualizacion	6 months ago
 SAPITO.C	Se agrega la funcion SingleIntegralShotOffSet	2 months ago
 SAPITO_C.d	Se agrega un ejemplo para generar un archiv...	2 months ago
 SAPITO_C.so	Se agrega un ejemplo para generar un archiv...	2 months ago
 SAPITO_C_ACLiC_dict_rdict.pcm	Se agrega un ejemplo para generar un archiv...	2 months ago
 cernstaff.dat	Se agrega un ejemplo para generar un archiv...	2 months ago
 sapito_tree.C	Se crearon los BRANCHES	2 months ago

Figure 2.1: Estructura del proyecto sapito-program en GitLab.

el cual a la fecha de la presente entrada, tiene 12 commits siendo el último de SHA **4582213d** con nombre de commit “Se crearon los BRANCHES”, donde se modifica el archivo de nombre **sapito_tree.C** el cual es un programa que genera un archivo **.root** llamado **sapito_shot.root**. Este es un archivo con estructura tipo TTree, que representa un conjunto de datos en forma de columnas (que en la jerga de *root* se le llama también *tree*), donde cualquier tipo de variable de **C++** puede ser almacenado en sus columnas. Este *tree* de nombre **T**, posee 16 *branches* (ramas o columnas). Las cuales son mostradas a continuación como extracto del código.

```

1  /*ANALYSIS PARAMETERS*/
2  Int_t shot;
3  Int_t ch;
4  double_t integral;
5  double_t neutrons;
6  Int_t saturated;
7  std::string step;
8
9  /*PLASMA FOCUS CONDITIONS*/
10 std::string gas;
11 double_t voltage;
12 double_t pressure;
13
14 /*DETECTOR PARAMETERS*/
15
16 std::string orientation;
17 std::string moderator;
18 Int_t angle;
19 Int_t shadowC;
20 Int_t test;
21 double_t distance;
22 double_t detector;
23
24 /*EXPERIMENTAL DATE*/
25 Int_t day;
```

```

26  Int_t hour;
27  Int_t min;
28
29
30  TFile *myfile = new TFile("sapito_shot.root","CREATE"); //Creamos un archivo root
31
32  TTree *mytree = new TTree("T","SHOTS ANALYSIS"); //Creamos un tree de nombre "T"
33
34  mytree->Branch("Shot",&shot,"shot/I");
35  mytree->Branch("Channel",&ch,"Channel/I");
36  mytree->Branch("Integral",&integral,"Integral/I");
37  mytree->Branch("Neutrons",&neutrons,"Neutrons/I");
38  mytree->Branch("Saturated",&saturated,"Saturated/I");
39  mytree->Branch("Step",&step,"Step/I");
40  //mytree->Branch("Gas",gas,"Gas/C");
41  mytree->Branch("Voltage",&voltage,"Voltage/I");
42  mytree->Branch("Pressure",&pressure,"Pressure/I");
43  // mytree->Branch("Orientation",orientation,"Orientation/C");
44  //mytree->Branch("Moderator",moderator,"Moderator/C");
45  mytree->Branch("Angle",&angle,"Angle/I");
46  mytree->Branch("Shadow_Cone",&shadowC,"Shadow Cone/I");
47  mytree->Branch("Test",&test,"Test/I");
48  mytree->Branch("Distance",&distance,"Distance/I");
49  mytree->Branch("Detector",&detector,"Detector/I");
50  mytree->Branch("Day",&day,"Day/I");
51  mytree->Branch("Hour",&hour,"Hour/I");
52  mytree->Branch("Min",&min,"Minute/I");
53

```

Posteriormente (hoy) se hizo una actualización al programa **SAPITO.C** agregándole dos nuevas funciones (actualización que había quedado atrasada) de nombres:

- `double_t SingleIntegralShotOffset_0(TString shot, int ch, int binx1, int binx2)`
- `void IntegralShotOutput(TString shot)`

Donde la primera retorna la integral de un canal de un disparo con el offset ajustado usando `FitAOSmovilSp`, en un intervalo determinado. La segunda retorna el archivo `integral_neutrones_shot.txt` donde en tres columnas se muestra el disparo, canal e integral correspondiente al canal. Las integrales se calculan con la función descrita anteriormente.

También se hizo una actualización del programa `sapito_tree.C`, resolviendo unos errores de tipo y agregándole una nueva rama tipo `Int_t` y de nombre `step` la cual entrega información correspondiente al fenómeno de pulso invertido.

Por último se actualizó el archivo `.gitignore` para que excluya el archivo `integral_neutrones_shot.txt` al hacer `git push`.

De esta manera, el proyecto en GitLab queda de la siguiente manera:

Name	Last commit	Last update
 .gitignore	Actualizacion	1 hour ago
 README.md	Actualizacion	1 hour ago
 SAPITO.C	Se agregan dos nuevas funciones SingleInte...	1 hour ago
 SAPITO_C.d	Se agregan dos nuevas funciones SingleInte...	1 hour ago
 SAPITO_C.so	Se agregan dos nuevas funciones SingleInte...	1 hour ago
 SAPITO_C_ACLiC_dict_rdict.pcm	Se agregan dos nuevas funciones SingleInte...	1 hour ago
 cernstaff.dat	Se agrega un ejemplo para generar un archiv...	2 months ago
 sapito_shot.root	Se actualizan los branches	1 hour ago
 sapito_tree.C	Se actualizan los branches	1 hour ago

Figure 2.2: Estructura del proyecto sapito-program en GitLab.

TIP

Para hacer un `git push` a archivos modificados con distintos `commits`, hacemos lo siguiente:

- Nos movemos al directorio del programa, donde ya se encuentra creado el directorio oculto `.git`
- `git status` Verificamos el estado del branch
- `git add programa.C` Agregamos el archivo a actualizar
- `git status` Verificamos el estado del branch, debería aparecer el archivo agregado, listo para hacer el commit
- `git commit -m "comentario" programa.C` Hacemos el commit y agregamos un comentario
- `git push` Subimos la actualización al servidor de GitLab
- Repetir lo mismo para cada archivo a actualizar.

A partir del proyecto anterior, y en específico del programa **SAPITO.C** se creó una clase que agrupa todas las funciones de este programa, permitiendo así una mejor y simple implementación de códigos futuros. El proyecto de la clase tiene por nombre en GitLab **Class ANPulses** y cuyo repositorio es:

- <https://gitlab.com/flopus/class-angepulses>

La clase tiene por nombre **ANPulses** y consta de los archivos **ANPulses.h** donde se declaran las clases, y del archivo **ANPulses.cc** donde se implementan. La última actualización en GitLab fue el 2021-11-09 donde se agregaron todas la mayoría de las funciones de **SAPITO.C**, además la clase se puede probar ejecutando el programa **test.C**.

Para ejecutar el código, se necesita generar antes el diccionario de la clase **ANPulses**, para esto hacemos lo siguiente:

- **root -l** Ingresamos a root
- **.L ANPulses.cc++** Compilamos la macro con ACLiC, con el primer + generamos la biblioteca compartida (shared library) de la clase **ANPulses**, que recibe el nombre **ANPulses.cc.so** haciendo que los símbolos del programa (**ANPulses.cc**) estén disponibles para ser cargados desde el **.so** en la memoria y puedan así ser utilizados desde dentro del interprete por otros programas, además el comando **+** también reconstruye (rebuild) la librería solo si el script o algunos de los archivos que incluye son más nuevos que la biblioteca. Para asegurarnos que la biblioteca partida sea reconstruida usamos el **++**, generando el archivo **ANPulses.cc.d**. También se genera un pre-compiled module (**.pcm**) de nombre **ANPulses.cc-ACLiC_dict_rdict.pcm**.
- **.L test.C++** Compilamos la macro con ACLiC donde haremos la prueba de la clase (se generarán los mismos archivos explicados anteriormente).
- **test()** Ejecutamos la función **test()** que está definida dentro de **test.C**

Par entender mejor lo anterior, de la guía de usuario de root <https://root.cern.ch/root/html/doc/guides/users-guide/Cling.html#aclic-compiling-scripts-into-libraries> tenemos lo siguiente :

9.4 ACLiC: Compiling Scripts Into Libraries

Instead of having Cling interpret your script there is a way to have your scripts compiled, linked and dynamically loaded using the C++ compiler and linker. The advantage of this is that your scripts will run with the speed of compiled C++ and that you can use language constructs that are not fully supported by Cling. On the other hand, you cannot use any Cling shortcuts (see “C++ Extensions To Ease Scripting” above) and for small scripts, the overhead of the compile/link cycle might be larger than just executing the script in the interpreter.

ACLiC will build a dictionary and a shared library from your C++ script, using the compiler and the compiler options that were used to compile the ROOT executable. You do not have to write a Makefile remembering the correct compiler options, and you do not have to exit ROOT.

9.4.1 Usage

Before you can compile your interpreted script you need to add include statements for the classes used in the script. Once you did that, you can build and load a shared library containing your script. To load it use the command **.L** and append the file name with a **+**.

```
root[] .L MyScript.C+
```

The **+ option generates the shared library** and names it by taking the name of the file “filename” but replacing the dot before the extension by an underscore and by adding the shared

library extension for the current platform. For example on most platforms, **hsimple.cxx will generate hsimple_cxx.so**.

The + command rebuild the library only if the script or any of the files it includes are newer than the library. When checking the timestamp, ACLiC generates a dependency file which name is the same as the library name, just replacing the 'so' extension by the extension 'd'. For example on most platforms, **hsimple.cxx will generate hsimple_cxx.d**.

To ensure that the shared library is rebuilt you can use the ++ syntax:

```
root[] .L MyScript.C++
```

To build, load, and execute the function with the same name as the file you can use the `.x` command. This is the same as executing a named script; you can also provide parameters. The only difference is you need to append a `+` or a `++`.

```
root[] .x MyScript.C+(4000)
```

Creating shared library /home/./MyScript_C.so

You can select whether the script is compiled with debug symbol or with optimization by appending the letter 'g' or 'O' after the '+' or '++'. Without the specification, the script is compiled with the same level of debugging symbol and optimization as the currently running ROOT executable. For example:

```
root[] .L MyScript.C++g
```

will compile MyScript.C with debug symbols; usually this means giving the `-g` option to compiler.

```
root[] .L MyScript.C++O
```

will compile MyScript.C with optimizations; usually this means giving the `-O` option to compiler. The syntax:

```
root[] .L MyScript.C++
```

is using the default optimization level.

3 Entrada del 2021-11-23

Se hizo una actualización de la clase **ANPulses** agregando varias funciones nuevas para implementar una nueva función importante, llamada **Test_Unfolding()** la cuál devuelve en pantalla el número de neutrones simulados o calculados, a partir de un flujo inventado y eficiencias absolutas, pero ahora noté que en realidad le pasamos a la función las eficiencias intrínsecas (correspondientes a seis matrices-detectores) como archivos `.root` resultantes de simulaciones en Geant4. Por lo tanto, en la función **Test_Unfolding()** necesitamos implementar un cálculo previo al cálculo de los neutrones simulados.

Por ahora, en GitLab actualizaremos la clase a este estado, donde en **test.C** le agregamos la función **Test_Unfolding()** para probarla. Además agregamos el programa de prueba **flujos_inventados.C** en el cual generamos un canvas de 4x4 para visualizar los distintos flujos inventados variando los parámetros **par1** y **par2**, que corresponden al centroide y ancho de una distribución gaussiana.

Se procedió a actualizar el repositorio origin/main desde el repositorio local main del pclin5 que se encuentra en el lab LIN de la cchen, quedando el repositorio en GitLab como:








Name	Last commit	Last update
 .gitignore	Se agrega y actualiza	29 minutes ago
 ANPulses.cc	Se agregan funciones nuevas como Test_Un...	19 minutes ago
 ANPulses.h	Se agregan funciones nuevas como Test_Un...	19 minutes ago
 Campagna_bajo_scatt_200123_...	c++ class for analysis of neutron yield from ...	2 weeks ago
 flujos_inventados.C	Nuevo programa que genera un canvas para ...	25 minutes ago
 readme	to compile	2 weeks ago
 test.C	Actualizacion para probar Test_Unfolding()	23 minutes ago

Figure 3.1: Estructura del proyecto class-angepulses en GitLab.

Al tratar de hacer un `git pull` desde mi repositorio local en mi thinkpad-t430, arrojó el siguiente error en la terminal:

```

1 hint: Pulling without specifying how to reconcile divergent branches is
2 hint: discouraged. You can squelch this message by running one of the following
3 hint: commands sometime before your next pull:
4 hint:
5 hint:   git config pull.rebase false  # merge (the default strategy)
6 hint:   git config pull.rebase true   # rebase
7 hint:   git config pull.ff only       # fast-forward only
8 hint:
9 hint: You can replace "git config" with "git config --global" to set a default
10 hint: preference for all repositories. You can also pass --rebase, --no-rebase,
11 hint: or --ff-only on the command line to override the configured default per
12 hint: invocation.
```

TIP

Cuya solución fue la siguiente (vista en <https://stackoverflow.com/a/32877954>):

```
git reset --hard
git pull
```

3.1 Pruebas con la función Test Unfolding()

En la clase `ANPulses` tenemos implementada la función `Test_Unfolding(double_t par1, double_t par2)` que toma seis eficiencias intrínsecas simuladas:

- `Efi_219_M_V4.root`
- `Efi_284_M_V4.root`

- Efi_C12_V4.root
- Efi_C15_V4.root
- Efi_C20_V4.root
- Efi_919_M_V4.root

correspondientes a distintas matrices moderadoras.

Luego haciendo la siguiente operación:

$$\tilde{N}_i^{(s)} = \sum_{j=1}^G \varepsilon_{ij}^{abs} \Upsilon_j^{(s)} \quad (3.1)$$

donde $\tilde{N}_i^{(s)}$ corresponde al número de neutrones calculados para cada paso s de la deconvolución asociado al detector i , ε_{ij}^{abs} es la eficiencia absoluta del detector i para el grupo de energía j , G es el número de grupos de energías y $\Upsilon_j^{(s)}$ es el Yield simulado (inventado) para el grupo de energía j en el paso s .

Dentro de la implementación de la función `Test_Unfolding(double_t par1, double_t par2)` las definiciones anteriores se asocian a objetos de `c++` tal y como lo muestra la siguiente tabla:

Definición	Notación	c++ Type	Objeto
Eficiencia intrínseca (simulada)	ε_{ij}^{abs}	<code>vector< vector<Double_t>></code>	<code>R[i][j]</code>
Yield inventado	$\Upsilon_j^{(s)}$	<code>vector<double></code>	<code>Phi_inv[j]</code>
Número de neutrones calculados o simulados	$\tilde{N}_i^{(s)}$	<code>vector<double></code>	<code>N[i]</code>

Table 3.1: Implementación de los elementos de la ecuación 3.1 mediante objetos de `c++`.

Donde la parte del código donde se implementa esto corresponde a:

```

1  /*Numero de neutrones calculados: convolucionados, simulados o inventados por detector (i):
   N_i = sum efi_abs_ij*flux_j */
2  for (int i = 0; i < ndet; i++)
3  {
4      double sum=0;
5      for (int j = 0; j < binnum; j++){
6          sum += R[i][j]*Phi_inv[j];
7      }
8      N[i] = sum;
9      sum = 0;
10 }
```

De esta manera, la salida del programa `test()` para `Test_Unfolding(2.5, 0.2)` es:

Neutrones calculados para cada sistema matriz-detector (son seis):


```

1 0.213356
2 0.000185711
3 0.219751
4 0.231793
5 0.187708
6 0.0808738

```

Flujo inventado:

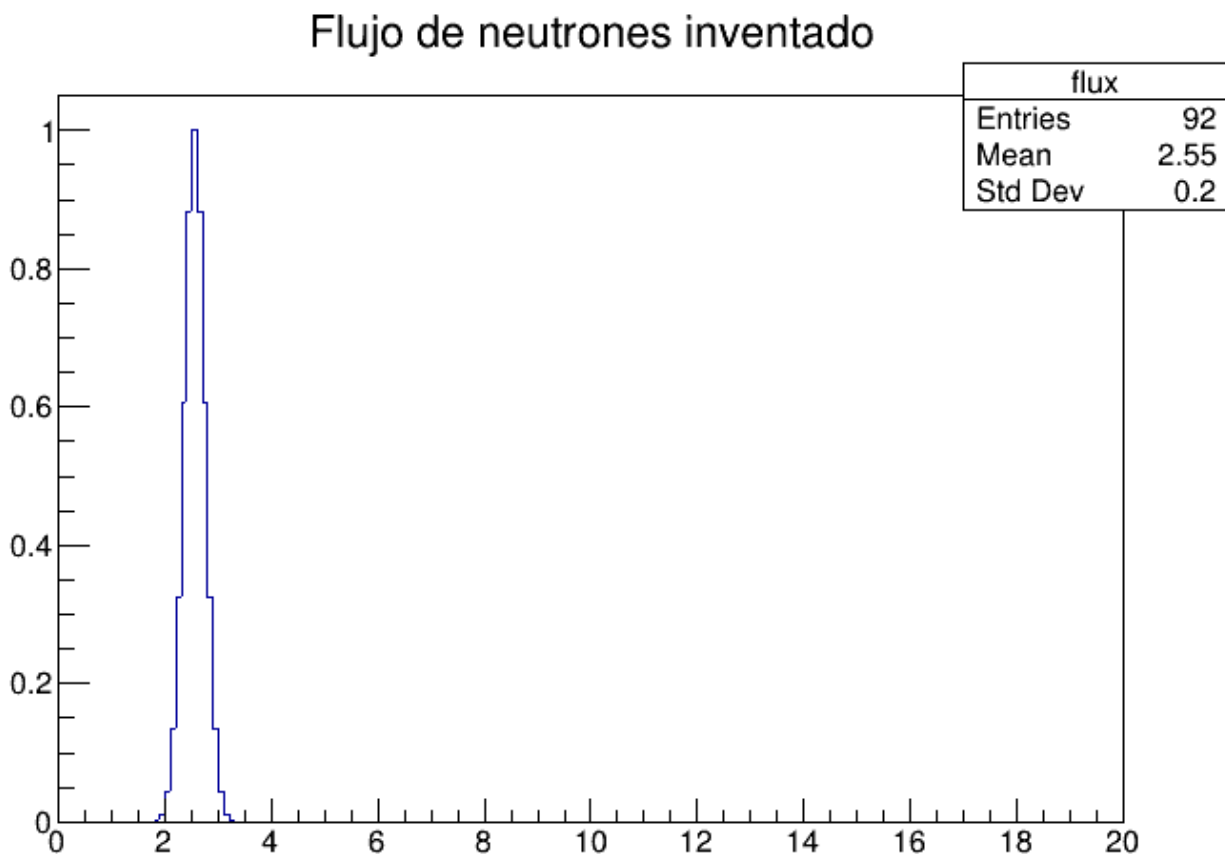


Figure 3.2: Flujo de neutrones inventado con valor de centroide 2.5 y sigma 0.2.

4 Entrada del 2021-11-24

Como notamos anteriormente, en la entrada (3), en la función `Test_Unfolding()` necesitamos implementar el cálculo de la eficiencia absoluta, el cuál lo realizamos de la siguiente manera:

```

1 Double_t SolAng_Dec1 = SolidAngle(200, 10, 42.5); // G219_M
2 Double_t SolAng_Dec2 = SolidAngle(200, 20, 42.5); // G284_M
3 Double_t SolAng_Dec3 = SolidAngle(200, 12, 42.5); // C12
4 Double_t SolAng_Dec4 = SolidAngle(200, 15, 42.5); // C15

```

```

5 Double_t SolAng_Dec5 = SolidAngle(200, 20, 42.5); // C20
6 Double_t SolAng_Dec6 = SolidAngle(200, 15, 42.5); // G919_M
7
8 /*Cargamos las eficiencias absolutas(?) al vector de vectores R*/
9 R.push_back( Readfile("/home/flopez/Data/FuncionesRespuestaByron2020/Efi_219_M_V4.root",
10 "E219_M", SolAng_Dec1/(4*M_PI), cutbinlow, cutbinup) );
11 R.push_back( Readfile("/home/flopez/Data/FuncionesRespuestaByron2020/Efi_284_M_V4.root", "
12 E284_M", SolAng_Dec2/(4*M_PI), cutbinlow, cutbinup) );
13 R.push_back( Readfile("/home/flopez/Data/FuncionesRespuestaByron2020/Efi_C12_V4.root", "
14 EC12", SolAng_Dec3/(4*M_PI), cutbinlow, cutbinup) );
15 R.push_back( Readfile("/home/flopez/Data/FuncionesRespuestaByron2020/Efi_C15_V4.root", "
16 EC15", SolAng_Dec4/(4*M_PI), cutbinlow, cutbinup) );
17 R.push_back( Readfile("/home/flopez/Data/FuncionesRespuestaByron2020/Efi_C20_V4.root", "
18 EC20", SolAng_Dec5/(4*M_PI), cutbinlow, cutbinup) );
19 R.push_back( Readfile("/home/flopez/Data/FuncionesRespuestaByron2020/Efi_919_M_V4.root", "
20 E919_M", SolAng_Dec6/(4*M_PI), cutbinlow, cutbinup) );

```

Obteniendo un número de neutrones calculados por el sistema matriz-detector igual a (para un centroide de 2.5 y ancho 0.2):

```

1 0.000179329
2 3.11895e-07
3 0.000221615
4 0.000292125
5 0.000315249
6 0.000101924

```

Una cantidad considerablemente menor de neutrones, en comparación con el caso anterior. Entonces, necesitamos un flujo de mayor intensidad, es decir, subir el yield total (la integral). Para esto le ponemos un factor A a la distribución gaussiana, es decir $A \cdot \exp$, y lo vamos variando.

Notemos que la integral del flujo inventado antes de aplicar el factor A corresponde a 5.013, tal y como se ve en la Fig. 4.1:

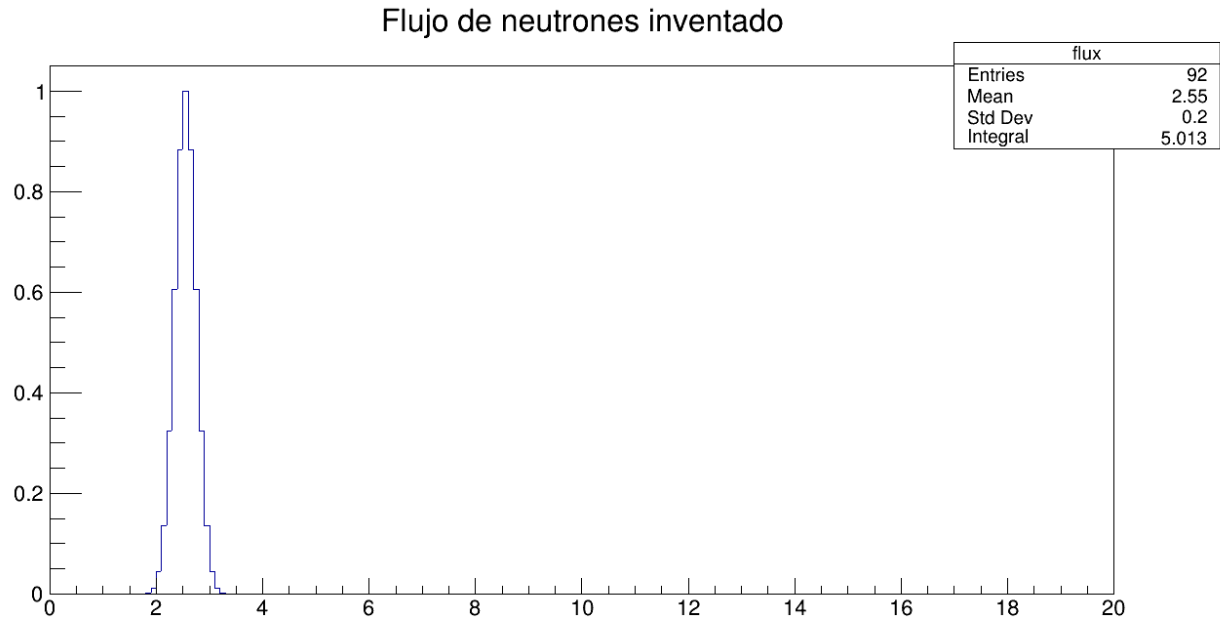


Figure 4.1: Integral del flujo de neutrones inventado con valor de centroide 2.5 y sigma 0.2.

Entonces, modificamos la función `SeedGenerator()` de la clase `ANPulses` para incorporar un factor que controle la intensidad del yield total del flujo inventado. Este factor se llamara A y pasa a ser un nuevo parámetro de la función:

```

1  ...
2  vector<Double_t> ANPulses::SeedGenerator(double* bins, double A=1.0, int binnum, double par1
    , double par2)
3  ...
4  double content = 0;
5  vector<Double_t> seed(binnum, 0);
6
7  ...
8
9  // Gauss //
10
11  for (int i = 0; i < binnum; i++){
12      content = 0;
13      content = A*TMath::Gaus(bins[i], par1, par2);
14      seed[i] = content;
15  }
```

De esta manera, en la función `Test_Unfolding()` definimos el factor como $A = 1000.$, resultando así el número de neutrones calculados como:

```

1  0.179329
2  0.000311895
3  0.221615
4  0.292125
```

5 0.315249
6 0.101924

Y la integral del flujo corresponde a 5013 (aumenta en un factor de 1000 con respecto al anterior, tal y como se esperaba):

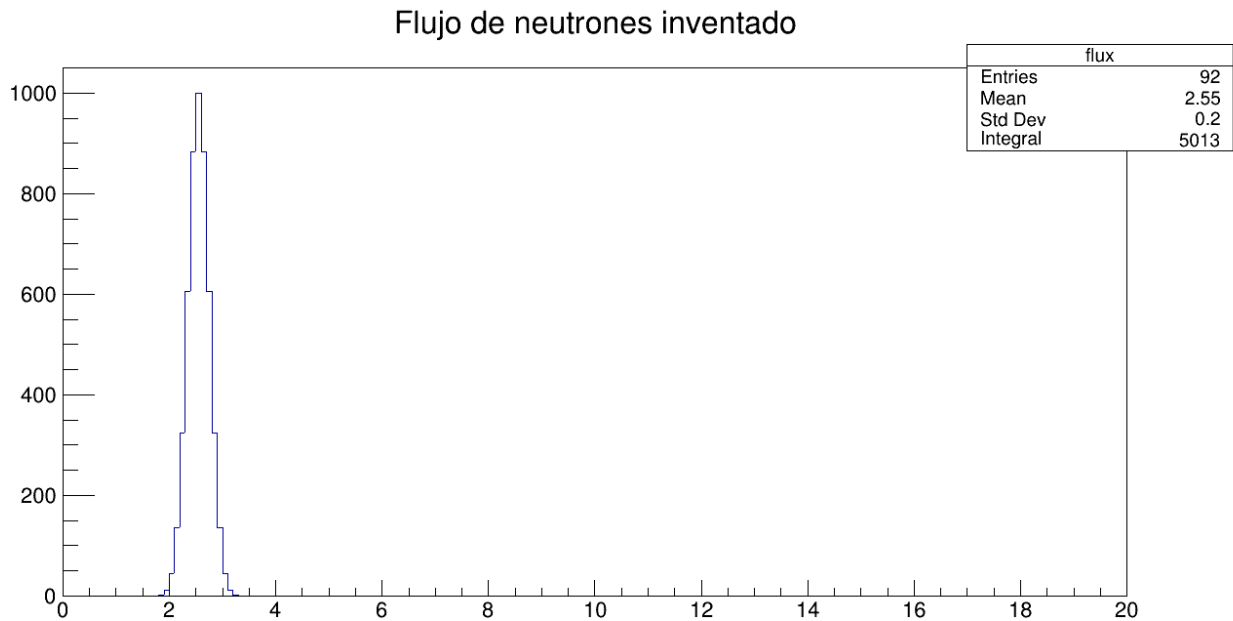


Figure 4.2: Integral del flujo de neutrones inventado con valor de centroide 2.5 y sigma 0.2. con factor $A = 1000$.

Ahora haremos unas modificaciones a la función `Test_Unfolding()`, de tal manera que genere un archivo como salida donde se almacenen los neutrones calculados para cada sistema matriz-detector.

```
1 ofstream neutrones_calculados("neutrones_calculados.dat"); // archivo de salida
2 ...
3 /*Generamos un archivo con los neutrones calculados*/
4 for (int i = 0; i < ndet; i++)
5 {
6     neutrones_calculados<<setw(5) << setfill(' ') << N[i] << " ";
7 }
8 neutrones_calculados.close();
9 ...
```

Finalmente en la clase `ANPulses`, implementamos la función que realiza la deconvolución de los neutrones calculados, llamada `Pretty_EMsimple()`, la cuál toma varias parámetros de entrada:

```
1 void ANPulses::Pretty_EMsimple(int shot, int steps=100, double par1=2.5, double par2=0.02,
    TString Type="Dif", int rebin=1, int bin_l=12)
```

Y en su implementación necesita un archivo con los neutrones calculados y uno con los errores de ellos, el cual lo llenamos de valores 1:

```
1 ...
2 //Le pasamos un numero de neutrones simulados o calculados del Test_Unfolding()
3 ...
4 N = ReadData("neutrones_calculados.dat", shot, ndet+1);
5 ...
6 N_Err = ReadData("neutrones_calculados_err.dat", shot, ndet+1, kFALSE);
7 ...
```

Necesita también las eficiencias intrínsecas, para que internamente se calculen las eficiencias absolutas a partir de ellas. Además, se define el flujo semilla del algoritmo, que corresponde a una distribución gaussiana:

```
1 ...
2 /*Flujo semilla*/
3 Seed = SeedGenerator(bins, binnum, 1.0, par1, par2);
4 ...
```

De esta manera, en el programa `test.C` generamos la instancia de la función con los siguientes parámetros:

```
1 ...
2 cout << "Test de deconvolucion" << endl;
3 pulse.Pretty_EMsimple(1, 100, 2.5, 0.2, "Dif", 1, 12);
4 ...
```

Produciendo entonces al ejecutar `test()`, la siguiente salida en la terminal de root:

```
1 Test de deconvolucion
2 Binning   V4
3 Steps    100
4 cutbinlow 1
5 cutbinup  1
6 Detectors 6
7
8 Shot number: 1
9 Detectors Status and Counting
10 Detector   1    0.179329
11 Detector   2    0.000311895
12 Detector   3    0.221615
13 Detector   4    0.292125
14 Detector   5    0.315249
15 Detector   6    0.101924
16
17 Neutrones Deconvolucionados
18    0.150513
19    0.000214608
20    0.18498
21    0.242506
22    0.25991
```

```

23  0.0822296
24  Diferencia porcentual
25  16.0688
26  31.1923
27  16.5309
28  23.0747
29  155.003
30  19.3226
31
32  Integral: 4129.57
33  Xi^2 : 0.438394
34  St Dv : 0
35  Warning in <TCanvas::Constructor>: Deleting canvas with same name: c1
36  FCN=45.2331 FROM MIGRAD STATUS=CONVERGED 64 CALLS 65 TOTAL
37  EDM=1.61303e-08 STRATEGY= 1 ERROR MATRIX ACCURATE
38  EXT PARAMETER STEP FIRST
39  NO. NAME VALUE ERROR SIZE DERIVATIVE
40  1 Constant 7.75833e+02 1.47213e+01 4.03053e-02 5.87828e-06
41  2 Mean 2.49841e+00 3.29275e-03 1.09120e-05 -3.56939e-02
42  3 Sigma 2.10022e-01 2.25391e-03 9.70061e-06 -2.09220e-02
43  Error in <TPad::TPad>: illegal height: -0.420000
44  Info in <TCanvas::Print>: file EMsimple_Data_S1.png has been created
45  Info in <TCanvas::SaveAs>: ROOT file EMsimple_Data_S1.root has been created

```

Además genera un gráfico con la comparación de neutrones calculados y deconvolucionados:

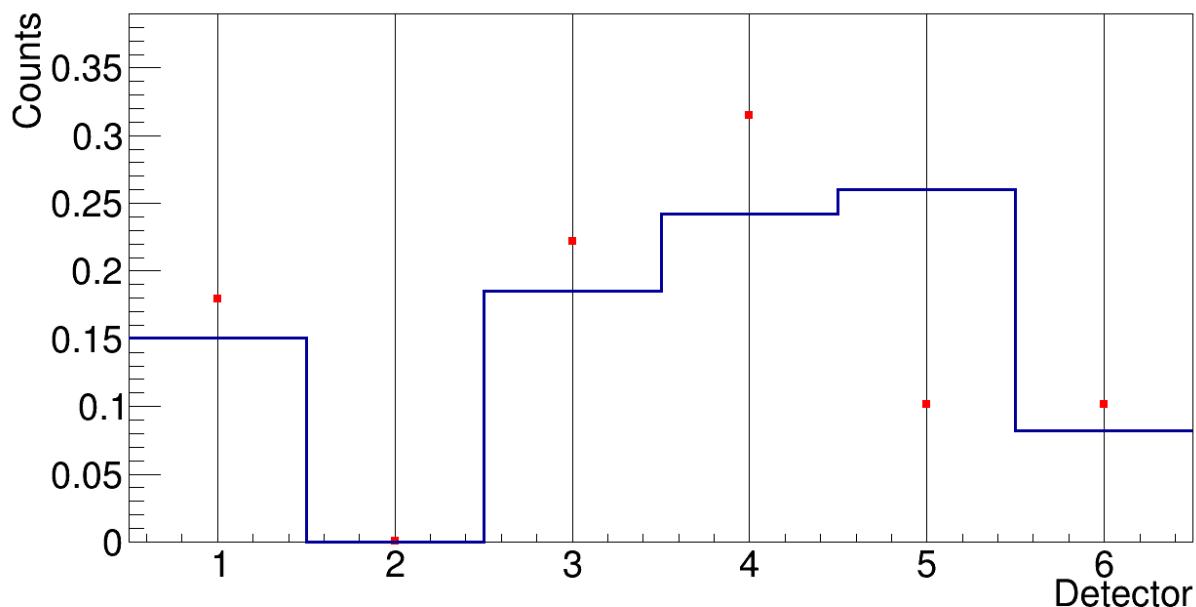


Figure 4.3: Comparación entre neutrones calculados y deconvolucionados.

Al revisar la diferencias relativas porcentuales entre los neutrones deconvolucionados y calculados, note que las diferencias desde el Detector 4 al 5 son incorrectas.

Detector	Neutrones calculados	Neutrones deconvolucionados	Dif relativa % código	Dif relativa % correcta
Detector 1	0.179329	0.150513	16.0688	16.06879
Detector 2	0.000311895	0.000214608	31.1923	31.192228
Detector 3	0.221615	0.18498	16.5309	16.530921
Detector 4	0.292125	0.242506	23.0747	16.985537
Detector 5	0.315249	0.25991	155.003	17.55406
Detector 6	0.101924	0.0822296	19.3226	19.322633

Table 4.1: Comparación de neutrones calculados y deconvolucionados mediante la dif rel %.

Notemos de la fig (4.3) que los puntos de los neutrones calculados del 4 al 5 están incorrectamente graficados.

El error anterior, venía por las siguientes líneas de código en la función `Pretty_EMsimple()`, las cuales eliminaban la entrada 4 del vector de neutrones calculados N:

```
1 N.erase(N.begin()+3);
2 N_Err.erase(N_Err.begin()+3);
```

Al comentar estas líneas, pasa algo curioso, y es que el número de neutrones deconvolucionados es el mismo que el número de neutrones calculados:

```
1 Test de deconvolucion
2 Binning V4
3 Steps 100
4 cutbinlow 1
5 cutbinup 1
6 Detectors 6
7
8 Shot number: 1
9 Detectors Status and Counting
10 Detector 1 0.179329
11 Detector 2 0.000311895
12 Detector 3 0.221615
13 Detector 4 0.292125
14 Detector 5 0.315249
15 Detector 6 0.101924
16
17 Neutrones calculados
18 v = { 0.179329, 0.000311895, 0.221615, 0.292125, 0.315249, 0.101924, };
19 Neutrones Deconvolucionados
20 0.179329
21 0.000311895
22 0.221615
23 0.292125
24 0.315249
25 0.101924
26 Diferencia porcentual
27 5.34767e-05
28 9.77985e-05
29 8.45083e-05
```

```

30 3.49242e-05
31 3.73356e-05
32 6.25612e-05
33
34 Integral: 5013.25
35 Xi^2 : 3.81291e-13
36 St Dv : 0
37 Warning in <TCanvas::Constructor>: Deleting canvas with same name: c1
38 FCN=1.07128e-10 FROM MIGRAD STATUS=CONVERGED 37 CALLS 38 TOTAL
39 EDM=1.11912e-13 STRATEGY= 1 ERROR MATRIX ACCURATE
40 EXT PARAMETER STEP FIRST
41 NO. NAME VALUE ERROR SIZE DERIVATIVE
42 1 Constant 1.00000e+03 1.72980e+01 3.00000e-02 -1.93530e-08
43 2 Mean 2.55000e+00 2.82469e-03 7.65000e-05 9.99301e-09
44 3 Sigma 2.00000e-01 1.99740e-03 1.72186e-06 -1.74058e-04

```

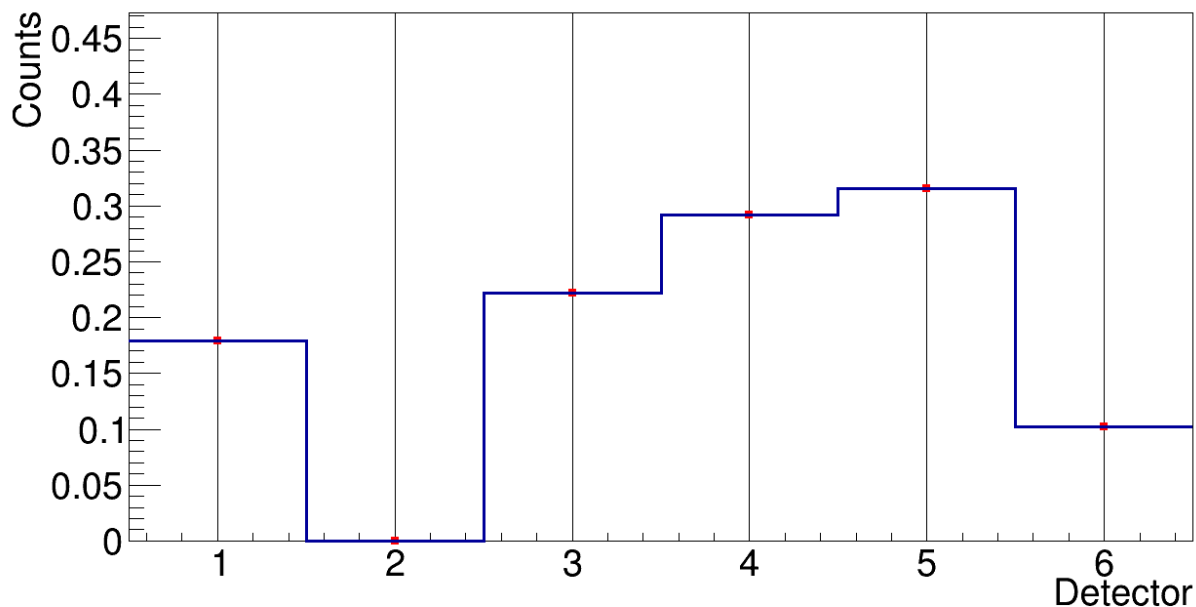


Figure 4.4: Comparación entre neutrones calculados y deconvolucionados.

TIP

Después de dar muchas vueltas a la situación, me di cuenta que estábamos usando como flujo semilla del algoritmo de deconvolución la misma distribución gaussiana que usamos como flujo inventado para calcular los neutrones calculados. Por lo que era esperable que obtuvieramos el mismo número de neutrones deconvolucionados y calculados, ya que **queremos que el algoritmo sea capaz de hacer eso** pero cuándo después le pasemos neutrones medidos.

5 Entrada del 2021-11-25

flopez: 2021-11-25: Hacer pruebas con neutrones calculados a partir de flujos inventados no gaussianos, usando como flujo semilla de la deconvolución una distribución gaussiana.

Llenar con la actualización del código.

Queremos encontrar un criterio de stop del χ^2 reducido, donde la definición de este último corresponde al χ^2 dividido por el número de grados de libertad (ndl) del problema a tratar:

$$\chi_{red}^2 = \frac{\chi^2}{ndl} \quad (5.1)$$

en nuestro caso, corresponde al bineado de las eficiencias intrínsecas: 92 (**binx**). Las eficiencias al pasar a objetos de **c++** se define como un vector. Entonces, si revisamos el siguiente código:

```
1 R.push_back( Readfile("/home/flopez/Data/FuncionesRespuestaByron2020/Efi_219_M_V4.root", "
    E219_M", SolAng-Dec1/(4*M_PI), cutbinlow, cutbinup)) ;
```

La función **Readfile()** tal y como está implementado en la clase **ANPulses**, retorna un vector de largo 92, que corresponde al bineado de las eficiencias intrínsecas. Luego, al hacer **push_back** estamos llenando el vector **R** con vectores como entrada, en específico, 6 vectores, correspondientes a las eficiencias absolutas de cada sistema de detección, es decir, **R** corresponde a una matriz de 6×92 . Este vector de vectores es el que será usado para realizar la deconvolución.

El criterio de stop lo establecemos al ver las diferencias entre flujos, para esto denotamos primero los flujos a considerar:

Flujos	Definición
Φ_j^{inv}	Valor de flujo inventado para la región j de energía
$\Phi_j^{(s)}$	Valor de flujo correspondiente a cada paso s de la deconvolución en la región j de energía
$\Phi_j^{(seed)}$	Valor del flujo semilla en la región j de energía

Table 5.1: Flujos a considerar en el criterio de stop χ^2

Entonces, definiremos χ^2 como

$$\chi^2 = \sum_{j=1}^{M=92} \left(\frac{\Phi_j^{inv} - \Phi_j^{(s)}}{(\Phi_j^{inv}/100)} \right)^2 \quad (5.2)$$

deberíamos entonces cortar en $\chi^2 \approx N$, con N número de detectores.

6 Entrada del 2021-11-29

A continuación mostraremos el algoritmo a implementar a partir de la función `EM_Pretty_EMsimple()`

Algorithm 1: EM1

Input: \vec{n} neutrones calculados, $\Phi^{(0)}$ flujo semilla, R matriz de eficiencias absolutas (función respuesta)

Output: $\Phi^{(s)*}$ flujo de salida que satisface $\chi^2 < N$ (número de detectores)

```

1 while  $\chi^2 > N$  do
2   for  $s = 0$  to  $Steps$  do
3     Algoritmo EM ;
4      $(\Phi^s)$  Flux = FluxNext ;
5     
$$\chi^2 = \sum_{i=1}^N \frac{\left( \sum_{j=1}^M R_{ij} \Phi_j^s - n_i \right)^2}{\Delta n_i^2};$$

6   end
7 end
```

A continuación calculamos:

Algorithm 2: EM2

Input: $\Phi^{(s)*}$ flujo de salida que satisface $\chi^2 < N$ (número de detectores), $\Phi^{(inv)}$ flujo inventado

Output: χ_c^2

```

1 return 
$$\chi_c^2 = \sum_{j=1}^M \frac{(\Phi_j^{s*} - \Phi_j^{inv})^2}{(\Delta \Phi_j^{(inv)})^2};$$

```

Con los resultados de la implementación de estos algoritmos, la idea es realizar un gráfico de $\Phi^{(inv)}$ y $\Phi^{(0)}$, donde el tercer eje corresponderá al valor de χ_c^2 asociado a cada par $(\Phi^{(0)}, \Phi^{(inv)})$, determinando así la dependencia del flujo deconvolucionado con respecto de la semilla inicial.

7 Entrada del 2021-12-01

Se implementó en `researchdiary_config.tex` una caja de color amarillo para las preguntas a partir del paquete `todonotes` (manual: <http://tug.ctan.org/macros/latex/contrib/todonotes/todonotes.pdf>)

```

\newcommand{\question}[1]{\todo[inline,backgroundcolor=yellow!90!yellow!80!white]
{\textbf{\userId}: #1}}
```

La cuál se utiliza de la siguiente manera: `\question{2021-12-01 Pregunta}` y además aparece en el índice de Todo list.

8 Entrada del 2021-12-02

Hasta la presente fecha teníamos atrasada la actualización en GitLab de la clase **ANPulses** y demás macros nuevas, por lo que realizaremos un update del repositorio. Se agregará entonces la macro **neutron_sim_generator.C**, que al ser cargada en root para luego ejecutar la función **neutron_sim_generator()**, se tiene que esta itera sobre los anchos (**sig**) en las funciones **Test_Unfolding_TH1D(2.5,sig)** y **Test_Unfolding(2.5,sig)**. Las cuales, respectivamente generan un canvas con todos los flujos inventados para distintos anchos y generan diversos archivos **.dat** con los neutrones calculados correspondiente a cada flujo, que luego son guardados en una carpeta de nombre **./neutrones_calculados**.

Las actualizaciones realizadas a la clase **ANPulses**, primero tiene que ver con los nombres de los archivos de neutrones calculados:

```
1 void ANPulses::Test_Unfolding(double_t par1, double_t par2)
2 {
3     /*double to String with Custom Precision using ostream*/
4     // Create an output string stream
5     std::ostringstream stream_centroide;
6     std::ostringstream stream_ancho;
7     // Set Fixed -Point Notation
8     stream_centroide << std::fixed;
9     stream_ancho << std::fixed;
10    // Set precision to 1 digits
11    stream_centroide << std::setprecision(1);
12    stream_ancho << std::setprecision(1);
13    //Add double to stream
14    stream_centroide << par1;
15    stream_ancho << par2;
16    // Get string from output string stream
17    std::string centroide = stream_centroide.str();
18    std::string ancho = stream_ancho.str();
19
20    ofstream neutrones_calculados("./neutrones_calculados/neutrones_calculados.C"+centroide+"
        _S"+ancho+".dat"); // archivo de salida
21 ...
```

También actualizamos la función **SeedGenerator()**, por lo que ahora además del factor *A* de intensidad agregado (ver entrada 4) tiene un nuevo parámetro tipo **TString** llamado *dist* que me permite elegir el tipo de distribución: "G" para gaussian, "AG" para asymeric gaussian. Se adicionarán más a futuro.

Se agrega también la macro **file_generator.C** la cual me servirá a futuro para generar el archivo **.root** de la campaña. Por último notemos que las barras de errores de las Fig. 4.3 y 4.4 se deben a que en el archivo **neutrones_calculados_err.dat** habían solo unos en cada columna. Para evitar estas barras de error, el archivo se llena de ceros.

TIP

Revert commits: (<https://stackoverflow.com/a/14281090/17548388>)

- **IF** you have **NOT** pushed your changes to remote:

```
git reset HEAD~1
```

Check if the working copy is clean by git status. After restart from git add ...

- **ELSE** you have pushed your changes to remote:

```
git revert HEAD
```

This command will revert/remove the local commits/change and then you can push. Besides, deleted all files that I had ready to push. Be careful!

La actualización del repositorio en GitLab quedó como:

Name	Last commit	Last update
EM_SIMPLE_png	Carpeta que guarda la salida .png de Pretty_...	7 minutes ago
EM_SIMPLE_root	Carpeta que guarda la salida .root de Pretty_...	7 minutes ago
neutrones_calculados	Nueva carpeta con los archivos generados d...	11 minutes ago
.gitignore	Se agrega y actualiza	1 week ago
ANPulses.cc	Actualizacion: se agregan funciones como P...	22 minutes ago
ANPulses.h	Actualizacion: se agregan funciones como P...	22 minutes ago
Campagna_bajo_scatt_200123_...	c++ class for analysis of neutron yield from ...	3 weeks ago
file_generator.C	Macro que servirá a futuro para generar el ar...	2 minutes ago
flujos_inventados.C	última actualización, se pasa a la macro neu...	15 minutes ago
neutron_sim_generator.C	Nueva macro	14 minutes ago
neutrones_calculados_err.dat	Archivo con los valores de incerteza para los...	5 minutes ago
readme	to compile	3 weeks ago
test.C	Actualizacion: instancia de Pretty_EMsimple()	19 minutes ago

Figure 8.1: Estructura del proyecto class-arpulses en GitLab.

Ahora implementaremos en la clase el chi-square y chi-square reducido.

```

1 ...
2 double_t ANPulses::Xi_Square_red(vector<Double_t> N_i, vector<Double_t> N_rec, vector<
  Double_t> Err, double ndet)
3 {
4   double_t xi=0;
5   double_t partial_sum=0;
```

```

6  double_t nprom=0;
7  double_t variance=0;
8
9  for (int i = 0; i < ndet; i++){ partial_sum = (N_i[i] - N_rec[i])/N_i[i]; xi +=
    partial_sum*partial_sum; }
10
11
12  /*chi-square reducido: chi-square dividido por el numero de grados de libertad (ndl) del
    problema, en esta caso el
    * numero de detectores*/
13
14  return xi/(double (ndet));
15
16 }
17
18 double_t ANPulses::Xi_Square(vector<Double_t> N_i, vector<Double_t> N_rec, vector<Double_t>
    Err, double ndet)
19 {
20     double_t xi=0;
21     double_t partial_sum=0;
22     double_t nprom=0;
23     double_t variance=0;
24
25     for (int i = 0; i < ndet; i++){ partial_sum = (N_i[i] - N_rec[i])/N_i[i]; xi +=
        partial_sum*partial_sum; }
26
27
28     return xi;
29
30 }
31 ...

```

Por lo que se hizo un `git push` al repositorio de commit SHA 2478ed9eae44ec7438853297012eb4e6ae155132.

Ahora necesitamos implementar los nuevos chi squares de los algoritmos EM1 y EM2. Para esto haremos los cambios en una nueva función `Pretty_EMsimple_em1()`. En esta función seteamos el parámetro A de intensidad de flujo de `SeedGenerator()` en 1000. Luego se probaron distintos valores de A para la función semilla (1,10,100,1000,10000,100000), pero no hubo cambios en el número de neutrones deconvolucionados, se mantuvo constante.

9 Entrada del 2021-12-03

Se modificaron las funciones `Test_Unfolding()` y `Test_Unfolding_TH1D()` añadiendo en ambas los parámetros nuevos siguientes:

Parámetro	Descripción
TString dist	Establece el tipo de distribución inventada a usar: "G": gaussian, "AG": asymmetric gaussian
double_t SIGL	Cuándo el parámetro dist toma el valor "AG", entonces SIG_L establece el ancho izquierdo de la distribución.
double_t SIGR	Cuándo el parámetro dist toma el valor "AG", entonces SIG_R establece el ancho derecho de la distribución.

Table 9.1: Nuevos parámetros.

Además la función `Test_Unfolding()` ahora genera los archivos de salida con los neutrones calculados con la siguiente estructura:

Se hicieron distintas actualizaciones a la clase **ANPulses**. Se agregó la función **Pretty_EMsimple_em1()** la cuál servirá como función de prueba para implementar los algoritmos EM1(1) y EM2(2). Esta función por ahora genera un canvas con los histogramas del flujo semilla, flujo deconvolucionado y flujo inventado. Así también muestra el valor de χ^2 y $\chi^2/(ndl)$ por paso s de iteración del *Expectation maximisation method* (EM). Junto con esto genera un archivo de nombre `xi2.txt` con los valores de χ^2 y $\chi^2/(ndl)$ por paso s del algoritmo. Por último acepta ahora tres parámetros más, los mismos mostrados en la tabla (9.1), los cuáles en principio están pensados para usarse en el flujo semilla.

flopez: 2021-12-03 Implementar parámetros diferenciados para la semilla y para el flujo inventado en `Pretty EMsimple em1()`, tanto para el tipo de distribución como de valor de anchos. Esto para hacer la iteraciones más fáciles de programar.

TIP

No agregar nunca el formato `\verb` en `\todo` todo ya que explota el código. Si pasa esto, hay que borrar los archivos `.aux` `.log` `.out` y `.tdo` antes de volver a compilar. Para agregar ecuaciones, se debe usar el comando `\protect`.

Por último la macro `neutron_sim_generator.C` tiene implementada tres funciones más de tipo `void`:

Función	Descripción
<code>neutron_sim_generator_AG_R()</code>	Genera 16 flujos inventados del tipo "asymmetric gaussian" donde el parámetro sobre el que se itera corresponde al ancho de la gaussiana derecha y dejando el ancho izquierdo fijo. Generando entonces 16 archivos con neutrones calculados.
<code>neutron_sim_generator_AG_L()</code>	Genera 16 flujos inventados del tipo "asymmetric gaussian" donde el parámetro sobre el que se itera corresponde al ancho de la gaussiana izquierda y dejando el ancho derecho fijo. Generando entonces 16 archivos con neutrones calculados.
<code>neutron_sim_generator_AG_RL()</code>	Genera 16 flujos inventados del tipo "asymmetric gaussian" donde los parámetros sobre los que se iteran corresponden al ancho derecha e izquierdo de la distribución gaussiana, 4 iteraciones para cada ancho. Generando entonces 16 archivos con neutrones calculados.

Table 9.2: Nuevos parámetros.

10 Entrada del 2021-12-06

Actualización del repositorio en GitLab con todo lo nombrado en la entrada anterior (9).

Quiero generar una nueva función `ReadData()` que lea un archivo de texto, donde en la primera columna de este archivo se encuentre el número de disparo y seguido, los neutrones calculados correspondientes a cada detector, y por último, la información del flujo inventado: `dist`, `par1`, `par2`, `SIG_L` `SIG_L` según corresponde. Algo como lo siguiente:

```
1 ...
2 9 0.1675 0.000212405 0.205418 0.268571 0.286972 0.089015 AG 2.5 1.0 1.3 0.5
3 ...
```

flopez: 2021-12-06: Hacer pruebas variando los flujos inventados y los flujos semilla. **HECHO, ver entrada (14)**

11 2021-12-07

Actualización del repositorio en GitLab, donde se crea la función:

```
1 ...
2 void ANPulses::Test_Unfolding_shots(int shot, double_t par1, double_t par2, TString dist,
   double_t SIG_L, double_t SIG_R)
3 ...
```

la cuál genera un archivo de nombre `neutrones_calculados_merge.dat` donde se guarda la información

de una simulación (“shot”) tal y como se muestra en la entrada 10). La idea de esta función es poder iterar sobre ella generando neutrones simulados correspondientes a distintos flujos inventados (gaussianos y gaussianos asimétricos).

Luego se agregó a la clase **ANPulses** la función:

```
vector<Double_t> ANPulses::ReadDataShot(TString fname, int shot, int ndet=3)
```

la cuál lee un archivo creado por la función anterior, donde están ordenados los disparos por filas, junto con el número de neutrones por detector y parámetros del flujo inventado con los cuáles se calculan estos neutrones. La función retorna un vector de neutrones calculados, correspondientes a un disparo determinado.

En la macro **neutron_sim_generator.C** se agrega la función **void neutrones_sim_generator_merge()** la cuál itera sobre la función **Test_Unfolding_shots()** para distintos valores de ancho de una distribución gaussiana. Llenando así el archivo **neutrones_calculados_merge.dat**.

Flujo diferencial es el dividido por dE .

Método de Deconvolución: Máxima Expectación

$$\Upsilon_b^{(s+1)} = \frac{1}{\sum_r \epsilon_{rb}^{(abs)}} \sum_r \frac{\epsilon_{rb}^{(abs)} \Upsilon_b^{(s)} N_r}{\sum_k \epsilon_{rk}^{abs} \Upsilon_k^{(s)}} \quad (11.1)$$

donde $\Upsilon_b^{(s+1)}$ corresponde al Yield obtenido para la iteración $(s+1)$ en el grupo (bin) de energía b , $\epsilon_{rb}^{(abs)}$ corresponde a la eficiencia absoluta del detector r para el grupo de energía b , N_r es el número de neutrones detectados por el detector r , $binnum$ corresponde al de grupo de energías y $ndet$ corresponde a la cantidad de detectores.

Algoritmo de deconvolucion EM implementado en c++:

```
1  for(int it=0; it<steps*1.5; it++){ /*Al multiplicar: steps*1.5, hacemos 1.5 veces mas pasos
   en la iteracion*/
2  FluxNext = Flux;
3
4  for(Int_t b=0; b<binnum; b++)
5  {
6      Sum_R=0.;
7      for(Int_t r=0; r<ndet; r++) Sum_R += (R[r][b]); /* \sum_r \epsilon_{rb}^{(abs)} */
8
9      Sum_N_R_Flux=0.;
10
11     for(Int_t r=0; r<ndet; r++)
12     {
13         Sum_R_Flux=0.;
14
15         for(Int_t k=0; k<binnum; k++)
16         {
```



```

17         Sum_R_Flux += (R[r][k] * Flux[k] * dE[k]); // *dE  $\sum_k^{\text{binnum}} \epsilon_{rk}^{(abs)} \Upsilon_k^{(s)}$  */
18     }
19     Sum_N_R_Flux += ((R[r][b]* Flux[b]) * N[r] )/Sum_R_Flux; /*  $\sum_r \frac{\epsilon_{rb}^{(abs)} \Upsilon_b^{(s)} N_r}{\sum_k^{\text{binnum}} \epsilon_{rk}^{(abs)} \Upsilon_k^{(s)}}$  */
20 }
21     FluxNext[b] = Sum_N_R_Flux/Sum_R; /*  $\Upsilon_b^{(s+1)}$  */
22 }
23     Flux=FluxNext; /*flujo deconvolucionado*/
24 }

```

flopez: 2021-12-07: Imprimir en pantalla o en un archivo todos los valores que intervienen en el método EM, ya que al monitorear el Break segmentation violation del algoritmo se ve que hay valores casi iguales a cero. **HECHO, ver entrada (13)**

12 2021-12-08

Se hicieron cambios en la función `SeedGenerator()` sacando `par2` de la función `asymetric gaussian`:

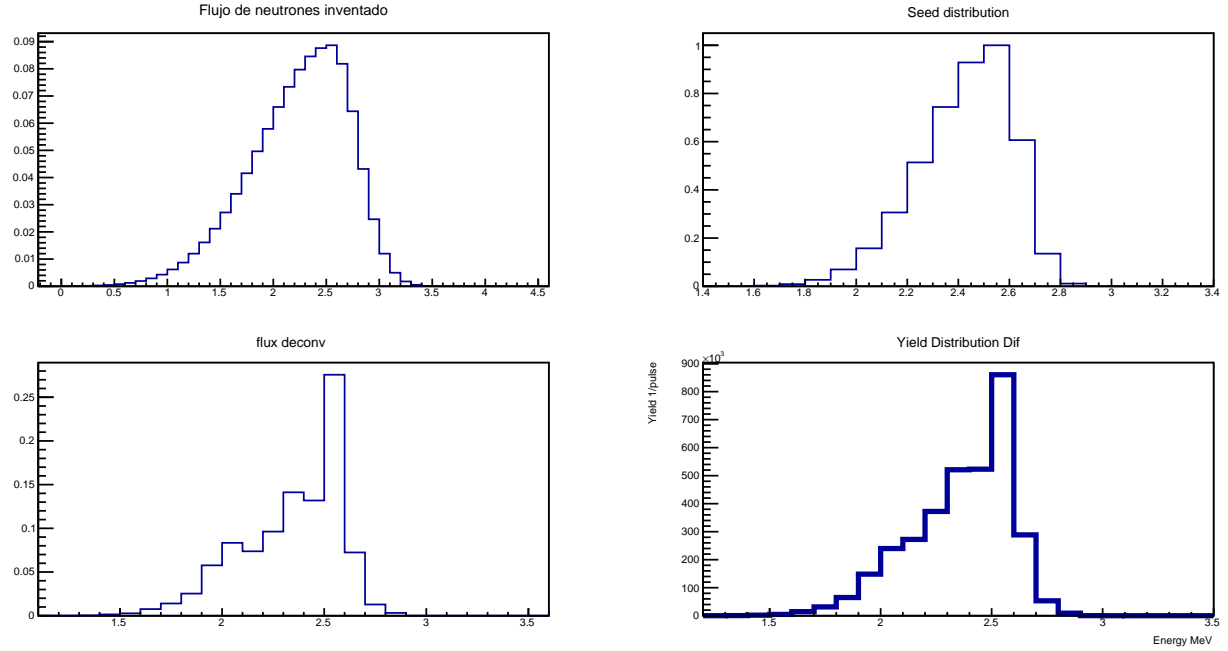
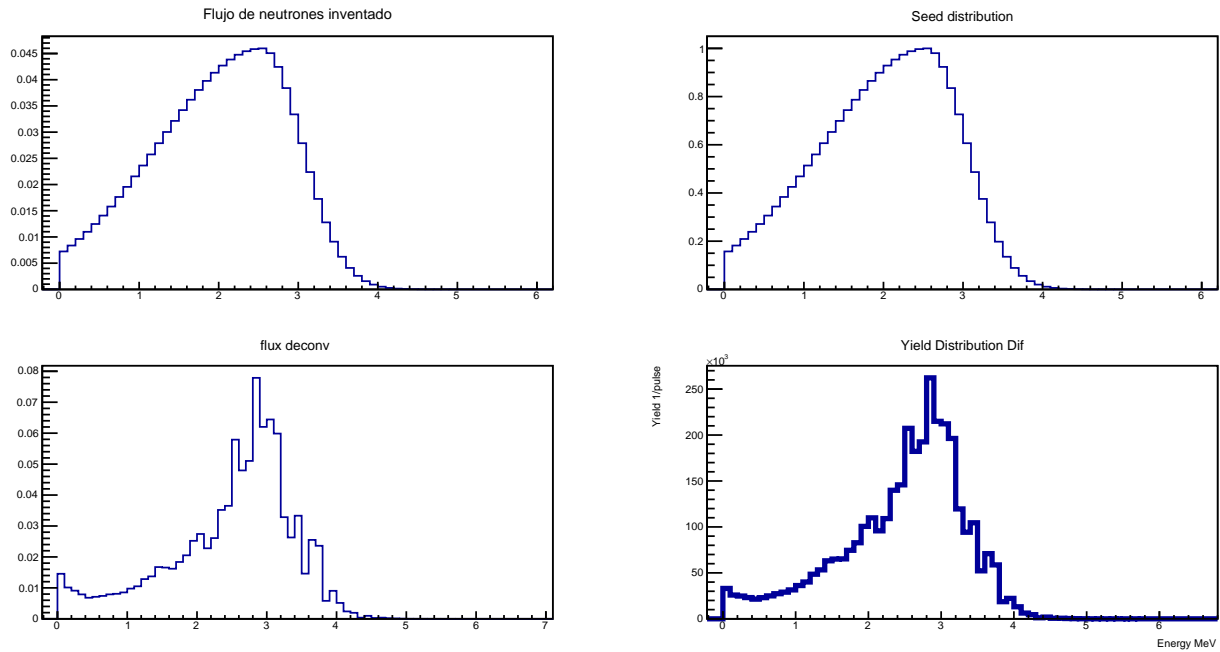
```

1  else if (dist=="AG")
2  {
3      // Assymetric Gauss //
4
5      ...
6
7      for (int i = 0; i < ctd; i++){
8          content = 0;
9          //~ content = A*TMath::Gaus(bins[i], par1, par2*SIG_L); //1.5 ANTES
10         content = A*TMath::Gaus(bins[i], par1, SIG_L); //1.5 DESPUÉS
11         seed[i] = content;
12     }
13     for (int i = ctd; i < binnum; i++){
14         content = 0;
15         //~ content = A*TMath::Gaus(bins[i], par1, par2*SIG_R); //0.9 ANTES
16         content = A*TMath::Gaus(bins[i], par1, SIG_R); //1.5 DESPUÉS
17         seed[i] = content;
18     }
19
20 }

```

Luego en la macro `test.C` mostramos la salida de la función `test()`:

```
pulse.Pretty_EMsimple_em1(15, 1000, 2.5, 0.2,"Dif", 1, 12, "AG",1.3,0.5);
```

Figure 12.1: Salida antes de arreglar `SeedGenerator()`Figure 12.2: Salida después de arreglar `SeedGenerator()`

Se cambio el parámetro `dist` de tipo `TString` a tipo `int` en `SeedGenerator()`, ahora 1 corresponde a `G` y 2 a `AG`. Esto para hacer más manejable la conversión a vectores de las líneas del archivo `neutrones_calculados_merge.dat` en la función `ReadDataShot()`. Así, el parámetro `dist` queda como tipo `int` en las funciones:

```

void Pretty_EMsimple()
void Pretty_EMsimple_em1()
void Test_Unfolding
TH1D* Test_Unfolding_TH1D()
void Test_Unfolding_shots()

```

Además, de esta manera en `void Pretty_EMsimple_em1()` ahora graficamos de forma correcta el flujo inventado que genera los neutrones calculados, los cuales sirven de entrada para el método EM. Quedando la línea de código que genera el histograma de flujo inventado como:

```
TH1D* flux_inv = Test_Unfolding_TH1D(par1_inv, par2_inv, dist_inv, SIG_L_inv, SIG_R_inv);
```

y donde los parámetros vienen de:

```

1  /*Trabajamos con el vector Neutrons de ReadDataShot*/
2  vector<double_t> Neutrons_raw = ReadDataShot("./neutrones_calculados/
    neutrones_calculados_merge.dat", shot, ndet); /*vector de 12 entradas con toda la info
    */
3  int neutrons_size = Neutrons_raw.size();
4  /*Parametros del flujo inventado asociado a un shot*/
5  vector<double_t> Neutrons_param(Neutrons_raw.begin()+neutrons_size-5, Neutrons_raw.begin()
    +neutrons_size);
6  std::cout << "Parametros de Neutrones calculados  " << endl;
7  std::cout << "n = { ";
8      for (Double_t n : Neutrons_param) {
9          std::cout << n << ", ";
10     }
11     std::cout << "}; \n";
12     std::cout << std::endl;
13     int dist_inv = Neutrons_param[0];
14     double_t par1_inv = Neutrons_param[1];
15     double_t par2_inv = Neutrons_param[2];
16     double_t SIG_L_inv = Neutrons_param[3];
17     double_t SIG_R_inv = Neutrons_param[4];
18
19
20 /*Neutrones calculados*/
21 vector<double_t> Neutrons_cal(Neutrons_raw.begin()+1, Neutrons_raw.begin()+ndet+1);
22 N = Neutrons_cal;

```

Visualización correcta del flujo inventado para la línea de código:

```
pulse.Pretty_EMsimple_em1(15, 1000, 2.5, 0.2,"Dif", 1, 12, 2 ,1.3,0.5);
```

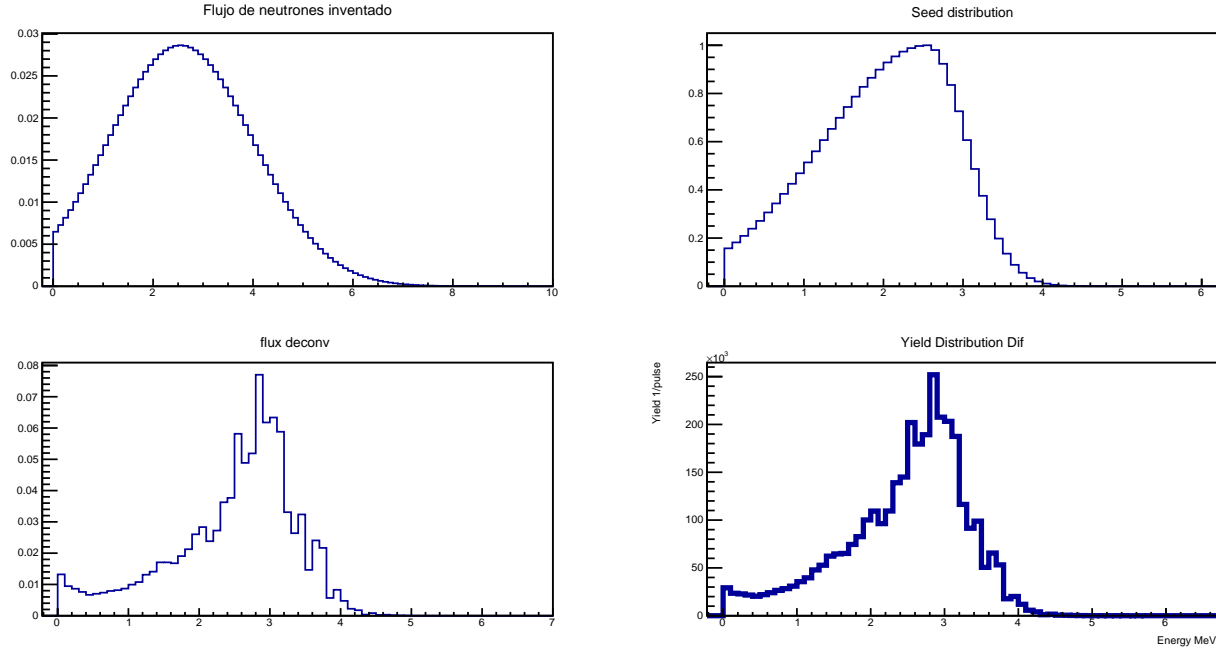


Figure 12.3: Salida después de arreglar los parámetros del flujo inventado en `void Pretty_EMsimple_em1()`

13 2021-12-09

Actualización del repositorio en GitLab con lo mencionado en la entrada anterior (12) de commit **3c84a7d8** que de acuerdo a GitLab se hicieron: 4 changed files with 306 additions and 90 deletions:

ANPulses.cc	+273 -62
ANPulses.h	+6 -6
neutron_sim_generator.C	+11 -11
test.C	+16 -11

Figure 13.1: Actualización de archivos en el repositorio en Gitlab de commit **3c84a7d8**.

Además de estos cambios, se mejoró el algoritmo (1), para esto, debugueamos esta implementación guardando un archivo de nombre `objets_debug.dat` con todas las variables de entrada de la función `Pretty_EMsimple_em1()`:

- `vector<double_t> N` : Neutrones calculados .
- `vector<double_t> Neutrons_raw`: Neutrones calculados raw (con informacion de # shot, y parámetros del flujo inventado).

- `Double_t SolAng_Dec1` : Valores de ángulos sólidos por detector.
- `vector<double_t> Seed` : Flujo semilla
- `vector<double_t> dE` :Ancho de bin energía
- `vector<double_t> Flux` : Flujo diferencial: (Flujo semilla)/(ancho de bin): $Flux[i] = Seed[i]/dE[i]$
- `R[i]` : Entradas de la matriz eficiencias absolutas ("Funciones respuesta")
- `R[i]` Entradas de la matriz de eficiencias absolutas luego de reemplazar ceros por $1.E - 23$

Es importante destacar que para generar correctamente este archivo, en la macro `test.C` tiene que ir nuevamente definido:

```
ofstream objets_debug("objets_debug.dat");
```

seguido por:

```
pulse.Pretty_EMsimple_em1(15, 1000, 2.5, 1.0,"Dif", 1, 12, 2 ,1.3,0.5);
```

En la función `Pretty_EMsimple_em1()` también se implementó el guardar un histograma de flujo deconvolucionado por cada paso del algoritmo EM (creando un vector de histogramas), con la idea posterior de producir un canvas donde se muestre la deformación del flujo por cada paso del método, esto es, visualizar a la izquierda el flujo inventado y a la derecha de este el flujo en constante actualización por paso.

TIP

C++:

- `vector.size()` cuando el vector no es un puntero
- `vector->size()` cuando el vector es un puntero.

Avance del código en LIN

Posterior a la actualización anterior, en la CCHEN, continué trabajando en el código, realizando las siguientes actualizaciones:

Se crea una macro de nombre `deconvoluciones.C` con dos funciones:

- `void neutrones_deconv()` la cuál para distintos disparos, es decir, para distintos flujos inventados, itera el método EM, `Pretty_EMsimple_em1()` para distintos valores de ancho del flujo semilla.
- `void neutrones_deconv_plot()` grafica un mapa de calor con la comparación del flujo inventado y flujo deconvolucionado, utilizando los valores de archivo `"chi2_test.dat"`

En la clase `ANPulses` se creó la función `double_t Xi_Square_em2()` la cuál calcula el χ_c^2 del algoritmo (2), comparando el flujo deconvolucionado con el flujo inventado. Pero en este caso se cambió el denominador $(\Delta\Phi_j^{(inv)})^2$ por el cuadrado de la integral del flujo deconvolucionado (ERROR, debería ser la integral del flujo inventado) con una tolerancia del 10%. A continuación mostramos la implementación de la función nombrada:

```

1 double_t ANPulses::Xi_Square_em2(vector<Double_t> N_i, vector<Double_t> N_rec, double_t
    intgl, double ndet)
2 {
3     double_t xi=0;
4     double_t partial_sum=0;
5     double_t nprom=0;
6     double_t variance=0;
7     double_t tolerance = 0.1;
8
9     for (int i = 0; i < ndet; i++){ partial_sum = (N_i[i] - N_rec[i])/(intgl*tolerance); xi +=
        partial_sum*partial_sum; }
10
11
12
13 return xi;
14
15 }
16

```

Dentro de la función `Pretty_EMsimple_em1()` se crea el archivo de nombre `chi2_test.dat`, el cuál es un archivo de salida para analizar la dependencia del flujo inventado con la semilla en la deconvolucion. Para cada iteración de `Pretty_EMsimple_em1()` en `neutrones_deconv()` se va a agregando una línea en el archivo:

```
ofstream chi2_test("chi2_test.dat", ios_base::app);
```

Donde las filas corresponden a columnas con la siguiente información:

- **shot**: número de disparo
- **it**: paso de iteración para el cuál se satisface EM1 (1)
- **par2**: ancho del flujo semilla
- **par2_inv**: ancho del flujo inventado
- **chi2_em2**: valor de χ_c^2 del algoritmo EM2 (2)

Ejemplo de la salida:

```

1 1      1      0.1  0.05  19.2782
2 1     351     0.5  0.05  53.4334
3 1      1      0.9  0.05  58.4065
4 1     990     1.3  0.05  58.634
5 2      1      0.1  0.15  2.75981
6 2      1      0.5  0.15  9.16272
7 ...
8 16    1124    0.9  1.55  0.138689
9 16     1      1.3  1.55  0.036807

```

A la salida "`objets_debug.dat`" se le agrega al final, lo siguiente:

- `vector<double_t> vec_fluxinv`: flujo inventado
- `vector<double_t> vec_fluxdeconv`: flujo deconvolucionado

14 Entrada del 2021-12-14

Nueva actualización del repositorio en GitLab con el avance realizado en el laboratorio LIN, sistematizado en la entrada anterior. Además se agregó al `readme` las instrucciones para generar una simulación experimental y poder testear así la dependencia del método con la semilla inicial.

```

1 ...
2 RUN SIMULATION EXPERIMENT:
3
4 root -l
5
6     .L ANPulses.cc++
7     .L neutron_sim_generator.C
8     neutrones_sim_generator_merge() # genera los neutrones simulados
9     .L deconvoluciones.C
10    neutrones_deconv()              #aplicamos el metodo EM para cada disparo
11    neutrones_deconv_plot()         #graficamos un mapa de calor con la comparacion entre
                                     flujo deconvolucionado y flujo inventado, donde el eje z es Xi2 de EM2.

```

De esta manera, para la primera simulación realizada, se tienen las siguientes condiciones:

Tenemos 64 disparos, donde cada uno corresponde a disparos generados por una distribución inventada de tipo gaussiana centrada en 2.5 y con anchos variables desde 0.01 hasta 0.63 en pasos de 0.01 (ver archivo `neutrones_calculados_merge.dat`). Luego, hasta el disparo 16 anterior, se aplicó la función `Pretty_EMsimple_em1()` con los algoritmos (1) y (2) implementados. Por cada disparo se itero una semilla inicial del método EM de tipo gaussiano centrada en 2.5 y de ancho variable entre 0.1 y 1.3 en pasos de 0.4.

```

1 ...
2 void neutrones_deconv(){
3
4
5     ANPulses pulse;
6
7     ofstream objetos_debug("objetos_debug.dat"); //importante para sobrecribir el archivo.
8     ofstream chi2_test("chi2_test.dat"); //importante para sobrecribir el archivo.
9
10    /*Parametros Pretty_EMsimple_em1(): int shot, int steps, double par1, double par2, TString
       Type,
11    * int rebin, int bin_l, int dist, double_t SIG_L, double_t SIG_R */
12    //~ pulse.Pretty_EMsimple_em1(15, 1000, 2.5, 1.0, "Dif", 1, 12, 1, 1.3, 0.5);
13    //~ int num_ite = 16; /*numero de iteraciones*/
14
15    for ( int shot=1; shot <= 16; shot++)
16        {
17            for (double sig=0.1; sig <=1.55; sig+=0.4)

```

```

18     {
19         pulse.Pretty_EMsimple_em1(shot, 1000, 2.5, sig,"Dif", 1, 12, 1 ,1.3,0.5);
20     }
21 }
22 }
23
24 }
25 ...

```

Luego con el el archivo de salida de esta función "chi2_test.dat" se realiza un mapa de calor, utilizando la función siguiente de la macro **deconvoluciones.C**:

```

1 ...
2 void neutrones_deconv_plot(){
3
4     //~ vector<double_t > Neutrons;
5     ifstream file("chi2_test.dat");
6     vector< vector<double_t>> vectors; //~matriz de disparos*/
7     std::string line;
8     while (std::getline(file, line))
9     {
10         std::istringstream ss(line);
11         std::vector<double_t> new_vec;
12         double_t v;
13         while (ss >> v)                //~ populate the new vector hasta donde haya doubles
14         {
15             new_vec.push_back(v);
16         }
17         vectors.push_back(new_vec);    //~ append it to the list of vectors
18     }
19
20
21     int vectors_size = vectors.size();
22     cout << vectors_size << endl;
23     TCanvas *c1 = new TCanvas("c1","c1",600,400);
24     TH2F *hcol1 = new TH2F("Xi2","Xi2",15,0,1.5,17,0,1.7);
25     //~ float px, py;
26
27     for (Int_t i = 0; i < vectors_size; i++)
28     {
29         //~ for (Int_t j = 2; j <= 3; j++)
30         //~ {
31             hcol1->Fill(vectors[i][2],vectors[i][3],vectors[i][4]);
32             //~ cout << vectors[i][j] << " " << vectors[i][j] ;
33         //~ }
34
35     }
36     hcol1->Draw("COLZ");
37
38 }

```

resultando el gráfico siguiente (14.1):

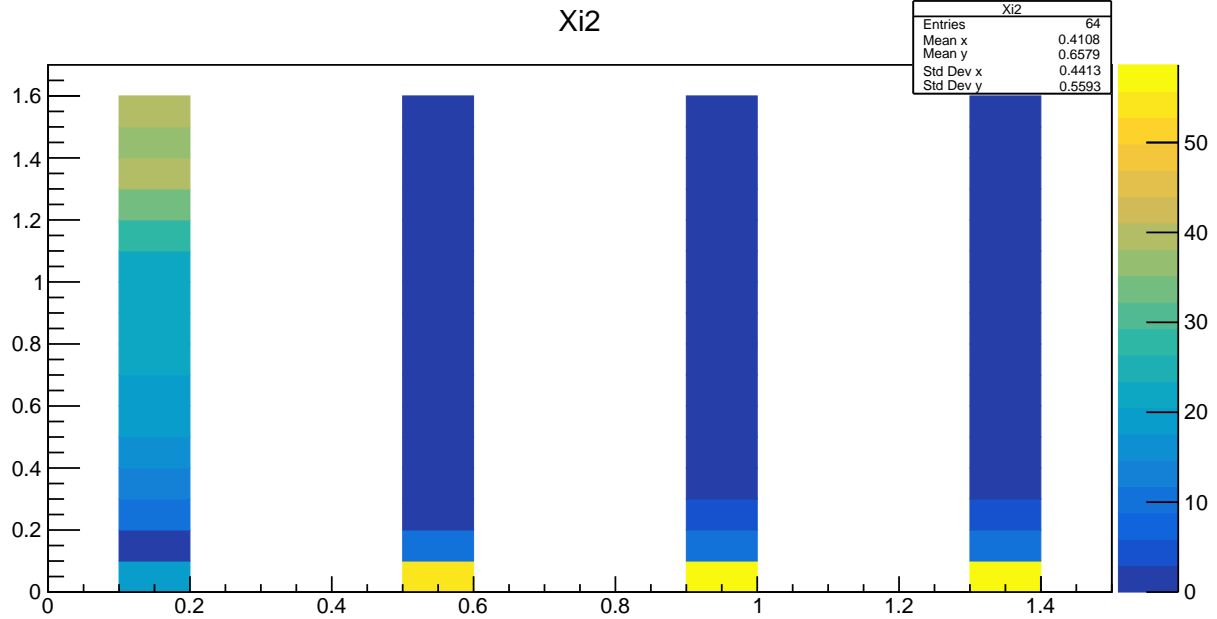


Figure 14.1: Simulación experimental de 64 disparos. Comparación entre el flujo inventado y deconvolucionado usando χ_c^2 .

donde el eje horizontal corresponde a los valores de ancho del flujo semilla del método EM y el eje vertical corresponde a los anchos de los flujos inventados. De esta manera, el eje z corresponde al valor de χ_c^2 que resulta de la comparación entre el flujo inventado y flujo deconvolucionado.

Por lo tanto, a partir de este ejercicio se busca determinar que grupos de flujo semilla son los mejores en el método EM para variados tipos de flujo inventado, con tal de que el flujo deconvolucionado no se deforme tanto con respecto al flujo inventado (parámetro χ_c^2). De esta manera, en el gráfico (14.1) las semillas que cumplen esto deberían formar un *atractor* de valores mínimos en este mapa de calor.

A su vez, queremos establecer un balance entre el número de pasos (algoritmo 1) y la deformación del flujo deconvolucionado (algoritmo 2). Por experiencias de trabajos previos (Selene y Pablo) se sabe que a medida que el número de pasos es muy grande, en el flujo resultante del método EM aparecen artefactos matemáticos indeseados y sin relación con la física del problema.

flopez: 2021-12-14: Generar una clase que inicialice todas las variables que necesito para el algoritmo EM, y así me evito el problema de que al iterar sobre Pretty EMsimple em1() se produzca un segmentation fault debido a que no se pueden abrir muchas veces seguidas un archivo .root, como es el caso de las eficiencias intrínsecas.

flopez: 2021-12-14: Establecer un balance entre el número de pasos del método EM y la deformación del flujo deconvolucionado. FMolina sugirió ocupar como parámetro de stop del algoritmo, el producto de χ^2 (del algoritmo 2) y χ_c^2 .

15 Entrada del 2021-12-20

TIP

Valores por defecto de funciones: Observar que es en la declaración (`clase.h`) solamente que se indica el valor default, no en la implementación (`clase.cc`).

flopez: 2021-12-20: Arreglar el orden de los parámetros por defecto de la clase `ANPulses`, ya que estos tienen que estar en la declaración (`clase.h`) y no en la implementación (`clase.cc`)

16 Entrada del 2021-12-23

Cambios en la clase `ANPulses`. Se agrega `f->Close()` y `delete f` para evitar el break segmentation por abrir muchos archivos `.root` de manera seguida en los loops, en las siguientes funciones:

- `Test_Unfolding()`: se agrega `f->Close()` y `delete f`
- `Test_Unfolding_TH1D()`: se agrega `f->Close()` y `delete f`
- `Test_Unfolding_shots()`: se agrega `f->Close()` y `delete f`

Se agrega la clase `InitVar` para inicializar las variables de archivos `.root` necesarios para las deconvoluciones. Esta clase contiene cuatro funciones, donde las tres primeras se remueven de la clase `ANPulses` y la última se adapta desde la función `Pretty_EMsimple_em1()` sacando las líneas que componen la función `R_effiAbs()` de una parte de ella.

- `vector<Double_t> Readfile()`
- `vector<Double_t> ReadBining()`
- `Double_t SolidAngle()`
- `vector<vector<double_t> > R_effiAbs()`: Función que retorna un vector de vectores o matriz de eficiencias absolutas, donde cada entrada corresponde a un vector de eficiencias absolutas asociada a cada sistema de detección.

Además en estas funciones se hacen los siguientes cambios que no estaban en `ANPulses`:

- `vector<Double_t> Readfile()`:
 - Se cambia `auto f= new TFile(fname)` por `TFile* f = TFile::Open(fname,"read")`
 - Se agrega `delete f` antes de `return R`
- `vector<Double_t> ReadBining()`
 - Se cambia `auto f= new TFile(fname)` por `TFile* f = TFile::Open(fname,"read")`

– Se agrega `delete f` antes de `return bins`

Estos cambios junto con introducir el comando `ulimit -n unlimited` (revisar este comando ya que en realidad debería ser `ulimit -d unlimited` o mejor probar con `ulimit -n 2048`) antes de correr los programas lograron solucionar el `break segmentation fault`. Fuentes: <https://root-forum.cern.ch/t/too-many-open-files/9237/5> y <https://root-forum.cern.ch/t/file-open-limit/6349/2>.

En `Pretty_EMsimple_em1()` Se comentan los gráficos para disminuir el tiempo de ejecución de los loops. Además las entradas cero del flujo semilla las reemplazamos por 1.0×10^{-23} . También se hacen casos para generar el archivo `chi2_test.dat` de acuerdo a la distribución de los flujo semilla y flujo inventado:

```

1  ...
2  //////////////////////////////////////
3  /*Llenamos el archivo chi2_test.dat*/
4
5
6  if( (dist==1) && (dist_inv==1))
7  {
8
9      chi2_test << left <<setw(5)<< setfill(' ') << shot << " "
10         <<setw(5) << setfill(' ') << it << " "
11         <<setw(5) << setfill(' ') << par2 << " " //ancho flujo semilla
12         <<setw(5) << setfill(' ') << par2_inv << " " //ancho flujo
13         inventado
14         <<setw(5) << setfill(' ') << chi2_em2 << endl;
15  }
16
17  else if ( (dist==1) && (dist_inv==2))
18  {
19      chi2_test << left <<setw(5)<< setfill(' ') << shot << " "
20         <<setw(5) << setfill(' ') << it << " "
21         <<setw(5) << setfill(' ') << par2 << " "
22         <<setw(5) << setfill(' ') << SIG_L_inv << " "
23         <<setw(5) << setfill(' ') << SIG_R_inv << " "
24         <<setw(5) << setfill(' ') << chi2_em2 << endl;
25  }
26  }
27
28  else if ( (dist==2) && (dist_inv==1))
29  {
30      chi2_test << left <<setw(5)<< setfill(' ') << shot << " "
31         <<setw(5) << setfill(' ') << it << " "
32         <<setw(5) << setfill(' ') << SIG_L << " "
33         <<setw(5) << setfill(' ') << SIG_R << " "
34         <<setw(5) << setfill(' ') << par2_inv << " "
35         <<setw(5) << setfill(' ') << chi2_em2 << endl;
36  }
37  }
38
39  else
40  {
41  {

```

```

42     chi2_test << left <<setw(5)<< setfill(' ') << shot << " "
43         <<setw(5) << setfill(' ') << it << " "
44         <<setw(5) << setfill(' ') << SIG_L << " "
45         <<setw(5) << setfill(' ') << SIG_R << " "
46         <<setw(5) << setfill(' ') << SIG_L_inv << " "
47         <<setw(5) << setfill(' ') << SIG_R_inv << " "
48         <<setw(5) << setfill(' ') << chi2_em2 << endl;
49
50 }
51
52
53 ///////////////////////////////////////////////////

```

La macro `neutron_sim_generator.C` se actualiza, agregando descripciones de las funciones implementadas. Además se crea una nueva función llamada `neutrones_sim_generator_merge_fluxinvAG()`, la cuál genera neutrones calculados correspondientes a una distribución de flujo inventada tipo gaussiana asimétrica, donde se varia el ancho del lado izquierdo y derecho, manteniendo el centroide de la distribución en 2.5.

De esta manera, con esta macro podemos generar neutrones calculados para cada sistema de detección en un solo archivo, usando dos tipos de distribuciones.

Función	Descripción
<code>neutron_sim_generator_merge_fluxinvG()</code>	Genera neutrones calculados correspondientes a flujos inventados del tipo gaussiano donde el parámetro sobre el que se itera corresponde al ancho de la gaussiana. Generando entonces un solo archivo: <code>neutrones_calculados_merge.dat</code> con los neutrones calculados asociados a un disparo (ancho determinado) y sistema de detección.
<code>neutron_sim_generator_merge_fluxinvAG()</code>	Genera neutrones calculados correspondientes a flujos inventados del tipo gaussiano asimétrico donde el parámetro sobre el que se itera corresponde al ancho izquierdo y derecho de dos distribuciones gaussianas. Generando entonces un solo archivo: <code>neutrones_calculados_merge.dat</code> con los neutrones calculados asociados a un disparo (anchos determinados) y sistema de detección.

Table 16.1: Nuevas funciones.

Por último se actualiza la macro `deconvoluciones.C`, reestructurando los nombres de sus funciones de la siguiente manera:

Función	Descripción
<code>neutrones_deconv_11()</code>	Función que realiza las deconvoluciones utilizando semillas gaussianas simétricas (1) y flujos inventados gaussianos simétricos (1). En ambas distribuciones los pasos de energía para las iteraciones pueden ser ajustados. Como prueba inicial se usaron pasos de de 0.01 o 10 keV.
<code>neutrones_deconv_12()</code>	Función que realiza las deconvoluciones utilizando semillas gaussianas simétricas (1) y flujos inventados gaussianos asimétricos (2). En ambas distribuciones los pasos de energía para las iteraciones pueden ser ajustados. Como prueba inicial se usaron pasos de de 0.05 o 50 keV
<code>neutrones_deconv_21()</code>	Función que realiza las deconvoluciones utilizando semillas gaussianas asimétricas (2) y flujos inventados gaussianos simétricos (1). En ambas distribuciones los pasos de energía para las iteraciones pueden ser ajustados. Como prueba inicial se usaron pasos de de 0.05 o 50 keV para la primera y pasos de 0.01 o 10 keV para la segunda.

Table 16.2: Actualización del nombre de las funciones.

Se actualizan y agregan también funciones que realizan mapas de calor con los valores de las deconvoluciones:

Función	Descripción
<code>neutrones_deconv_plot()</code>	Función que realiza el grafico de mapa de calor de χ_c^2 en función de los anchos del flujo semilla (gaussiano) y flujo inventado (gaussiano).
<code>neutrones_deconv_plotSteps()</code>	Función que realiza el grafico de mapa de calor de pasos que le toma en cumplir al algoritmo em1 (1) en quebrar: $N < \chi^2$. Esto en función de los anchos del flujo semilla (gaussiano) y flujo inventado (gaussiano).
<code>neutrones_deconv_plot_11()</code>	Función que realiza el grafico de mapa de calor de χ_c^2 (pad 11) y de pasos que le toma en cumplir al algoritmo em1 (1) en quebrar: $N < \chi^2$ (pad 12). Esto en función de los anchos del flujo semilla (gaussiano) y flujo inventado (gaussiano). Luego en los pad 21 y pad 22, se muestran los mismos graficos anteriores pero en escala logarítmica: <code>gpad->SetLogz()</code>
<code>neutrones_deconv_plot_21()</code>	Función que realiza el grafico de mapa de calor de χ_c^2 (pad 11) y de pasos que le toma en cumplir al algoritmo em1 (1) en quebrar: $N < \chi^2$ (pad 12). Esto en función de los anchos del flujo semilla (gaussiano asimetrico) y flujo inventado (gaussiano). Luego en los pad 21 y pad 22, se muestran los mismos graficos anteriores pero en escala logarítmica: <code>gpad->SetLogz()</code>

Table 16.3: Funciones que realizan graficos de las deconvoluciones.

Por último se agregan al repositorio los archivos siguientes:

- `chi2_test_1-149_fluxseed_G_fluxinv_G.dat`
- `chi2_test_1-196_fluxseed_G_fluxinv_AG.dat`
- `neutrones_calculados/neutrones_calculados_merge_fluxinv_AG.dat`

Reunión con los profesores Francisco Molina y Víctor Muñoz. (4ta)

Se muestran los avances, en específico el gráfico de la simulación `neutrones_deconv_plot_11()`. El profesor Víctor sugirió utilizar un algoritmo genético para encontrar las mejores semillas para el método EM. Para esto, los parámetros de las distribuciones semilla que tendremos en consideración serán el centroide, ancho e inclinación. Dio como consejo explorar esta posibilidad en un plazo de dos semanas.

Estas simulaciones experimentales (loops), me darán una idea del error de σ_0 , es decir, me permitirán tomar una decisión para los anchos del flujo semilla, de tal manera que sean los que mejor se ajustan al método EM, para variados rangos de ancho del flujo inventado.

También en la discusión, se habló de realizar una exploración del efecto de la variación del centroide, tanto en el flujo semilla como en el flujo inventado. Esto de la manera en que se ha venido realizando, es decir, usando los loops.

El profesor Víctor habló también de realizar pruebas del método con neutrones medidos del experimento e ir viendo que resultados se obtienen para ciertos valores de semilla. La idea es comparar el número de neutrones deconvolucionados con los neutrones medidos. Para lograr esto, antes se necesita calcular el valor de área de un pulso single de un neutrón, valor que se obtiene de las mediciones con fuente de calibración y el sistema de adquisición en modo trigger interno. Por lo que necesitamos convertir estos archivos .root usando el briken offline.

Del gráfico mostrado, el profesor Víctor sugirió normalizar el gráfico de χ_c^2 en función de los anchos de los flujos semilla e inventado, es decir, dividir por el mayor valor encontrado de χ_c^2 . Lo mismo para el gráfico de pasos. Sin embargo el profesor Francisco comentó que tal vez para este último no sea tan necesario.

El profesor Francisco, habló de hacer pruebas variando el valor de la tolerancia en χ_c^2 , que para las simulaciones hasta ahora, corresponde a un 10% del valor de la integral del flujo inventado. Entonces, se sugirió tomar valores de 10%, 5% y 1%, de esta manera, se espera que en el gráfico de χ_c^2 , la diagonal se vaya cerrando a medida que disminuye el valor de la tolerancia.

Se propuso tener un bosquejo del proyecto de tesis para el 20 de enero.

La quinta reunión quedó agendada para el 6 de enero a las 10 am.

17 Entrada del 2021-12-28

Se actualiza el repositorio `class-anpulses` de GitLab con los cambios descritos en la entrada anterior (16).

18 Entrada del 2021-12-30

Se termina de actualizar el repositorio `class-anpulses` de GitLab con los cambios descritos en la entrada (16).

19 Entrada del 2022-01-03

A continuación describiremos y presentaremos los resultados de las simulaciones experimentales realizadas hasta ahora:

Simulación 11

Instrucciones para su ejecución:

- `root -l`
- `.L InitVar.cc++`

- `.L ANPulses.cc++`
- `.L neutron_sim_generator.C`
- `neutrones_sim_generator_merge_fluxinvG()`: Se genera el archivo `neutrones_calculados_merge.dat` con los neutrones calculados asociados a un disparo (ancho determinado) y sistema de detección. Los pasos de ancho para el flujo corresponden a 0.01 o 10 keV, con un máximo de 1.49.
- `.L deconvoluciones.C`
- `neutrones_deconv_11()`: Input: `neutrones_calculados_merge.dat` Output: `chi2_test.dat`.
Función que realiza las deconvoluciones utilizando semillas gaussianas simétricas (1) y flujos inventados gaussianos simétricos (1). En ambas distribuciones los pasos de energía para las iteraciones pueden ser ajustados. Como prueba inicial se usaron pasos de de 0.01 o 10 keV para ambos flujos. Se genera entonces el archivo `chi2_test.dat` donde los datos de las columnas corresponden a los explicados en la entrada del (13).
- `neutrones_deconv_plot_11()`: Input: `chi2_test.dat` Output: Salida gráfica de root.
Ver tabla (16.3)

El gráfico generado es el siguiente:

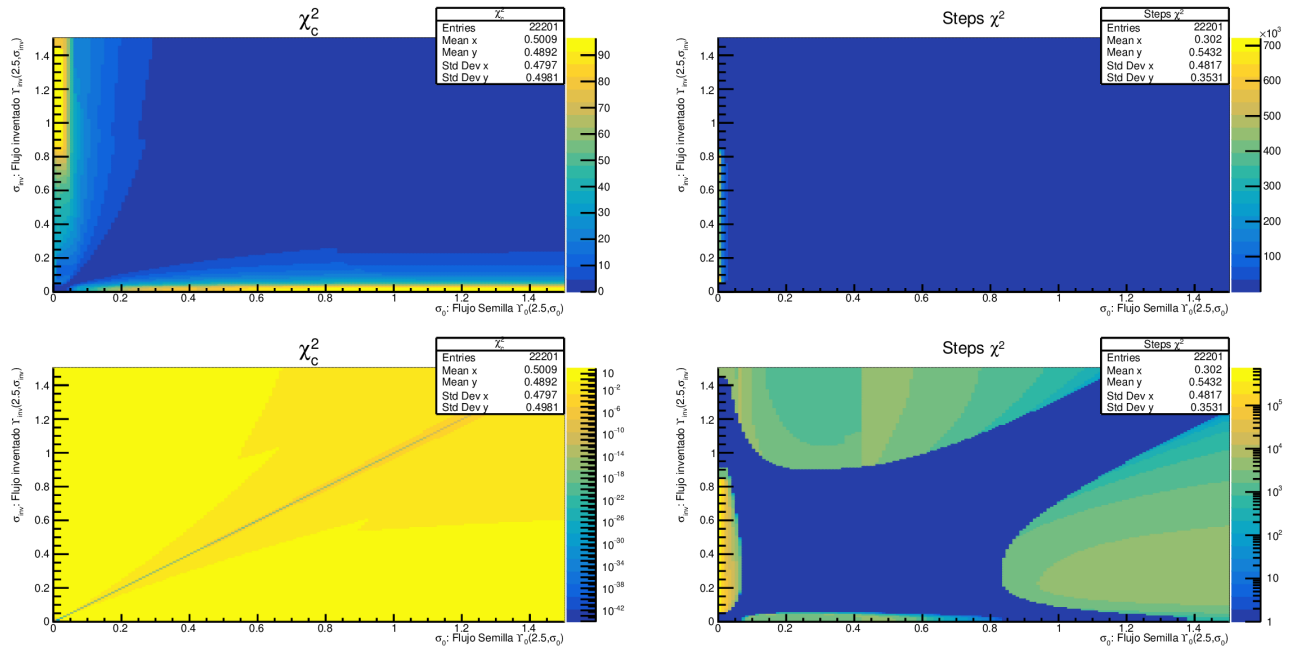


Figure 19.1: Simulación experimental de 149 disparos. Comparación entre el flujo inventado y deconvolucionado usando χ^2_c .

Donde los pads (21) y (22) de la Fig. 19.1 corresponden a representaciones en escala logarítmica en el eje z de los respectivos gráficos superiores. Podemos manipular cada pad del gráfico mediante el menú de root, usando las funciones de la clase TH2F llamadas `SetMaximum()` y `SetMinimum()`, que tienen por definición (https://root.cern/doc/v612/TH1_8h_source.html#l00390):


```

1  /*
2   * Set the minimum / maximum value for the Y axis (1-D histograms) or Z axis (2-D
   histograms)
3   * By default the maximum / minimum value used in drawing is the maximum / minimum
   value of the histogram
4   * plus a margin of 10%. If these functions are called, the values are used without any
   extra margin.
5   */
6  virtual void SetMaximum(Double_t maximum = -1111) { fMaximum = maximum; }; // *MENU*
7  virtual void SetMinimum(Double_t minimum = -1111) { fMinimum = minimum; }; // *MENU*

```

Por ejemplo, en el pad (12) hacemos:

- Click derecho sobre la imagen
- Seleccionamos la función **SetMinimum**
- Se abre una ventana, donde ingresamos el valor de 100 y le damos OK.

Obteniendo el siguiente resultado:

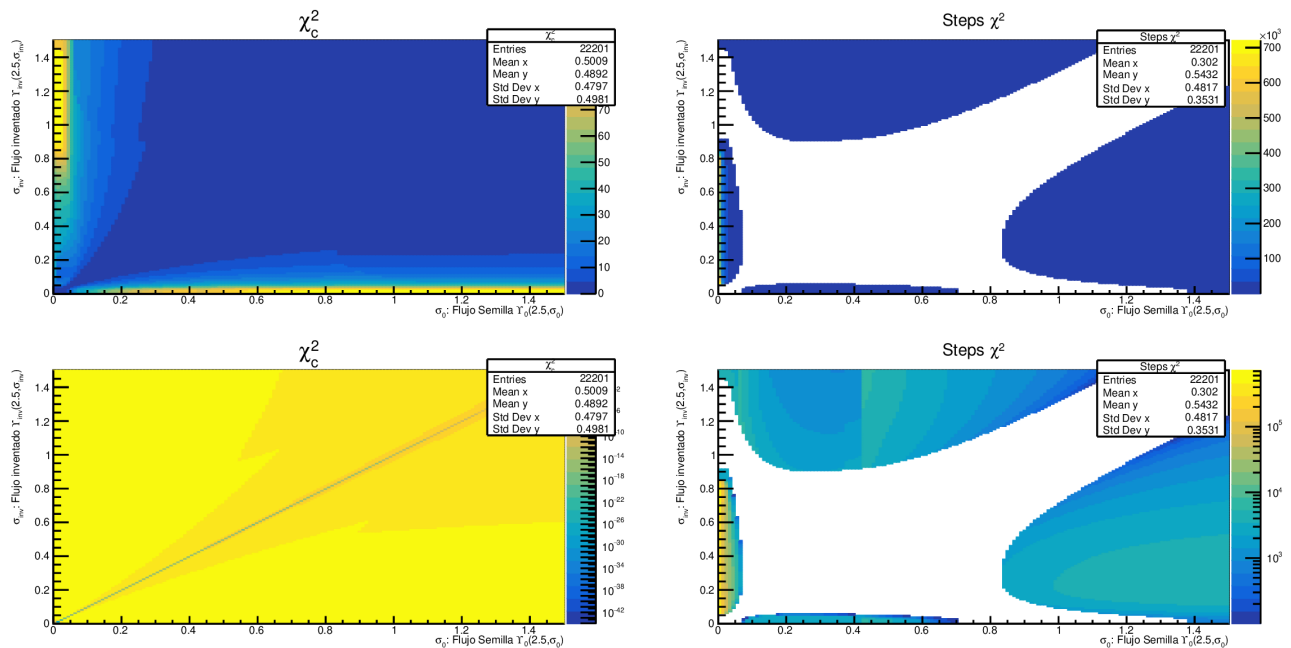


Figure 19.2: Grafico 19.1 donde el SetMinimum() es 100

Donde se muestran los valores de pasos que son mayores a 100, y los menores se eliminan.

Otro ejemplo es el siguiente, donde se estableció el máximo (SetMaximum) en 3000 para el pad (12):

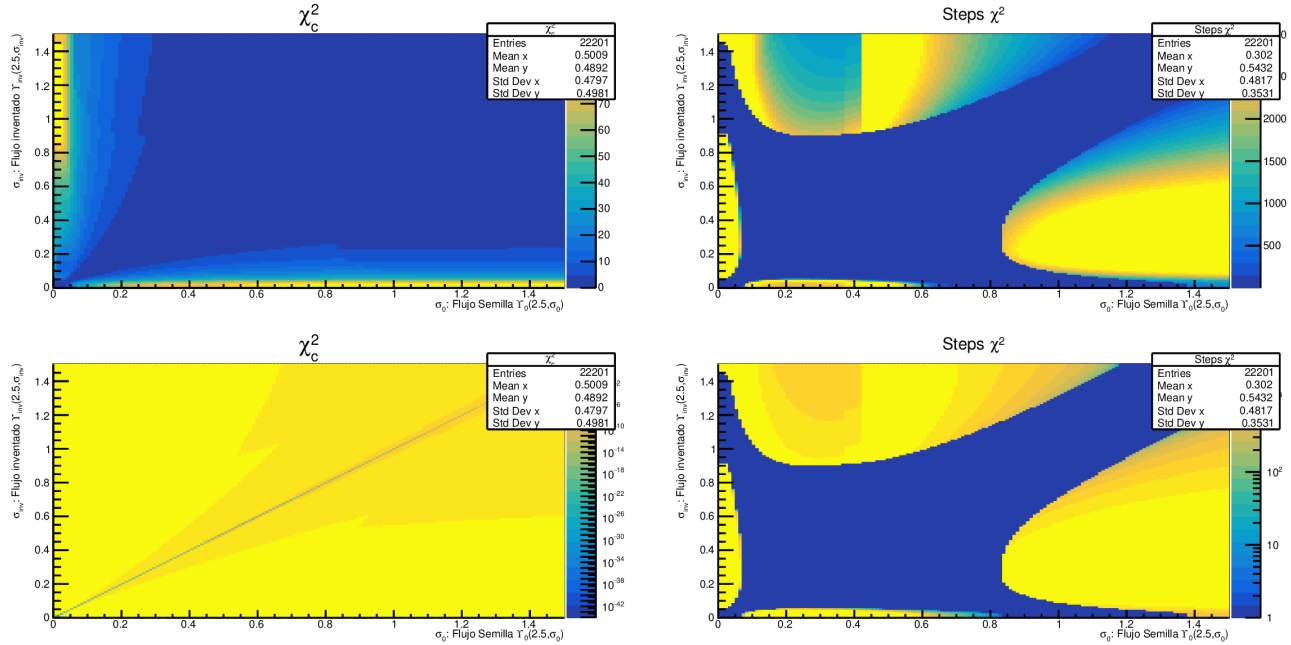


Figure 19.3: Grafico 19.1 donde el SetMaximum() es 3000

20 Entrada del 2022-01-04

Simulación 21

Instrucciones para su ejecución:

- `ulimit -n 30000`: donde 29204 es el número total de filas en `chi2_test_deconv_21.dat`
- `root -l`
- `.L InitVar.cc++`
- `.L ANPulses.cc++`
- `.L neutron_sim_generator.C`
- `neutrones_sim_generator_merge_fluxinvG()`: Se genera el archivo `neutrones_calculados_merge.dat` con los neutrones calculados asociados a un disparo (ancho determinado) y sistema de detección. Los pasos de ancho para el flujo corresponden a 0.01 o 10 keV, con un máximo de 1.49.
- `.L deconvoluciones.C`
- `neutrones_deconv_21()`: Input: `neutrones_calculados_merge.dat` Output: `chi2_test.dat`.

Función que realiza las deconvoluciones utilizando semillas gaussianas asimétricas (2) y flujos inventados gaussianos simétricos (1). En ambas distribuciones los pasos de energía para las iteraciones pueden ser ajustados. Se usaron pasos de 0.05 o 50 keV para los anchos izquierdo y derecho de los flujos semilla asimétricos (como máximo 0.7 por cada lado). Se genera entonces el archivo `chi2_test.dat`

donde los datos de las columnas corresponden a (visto como vector en c++) a lo mostrado en la entrada del 16.

- `neutrones_deconv_plot_21()`: Input: `chi2_test.dat` Output: Salida gráfica de root.

Ver tabla (16.3)

De esta manera, el gráfico generado es el siguiente:

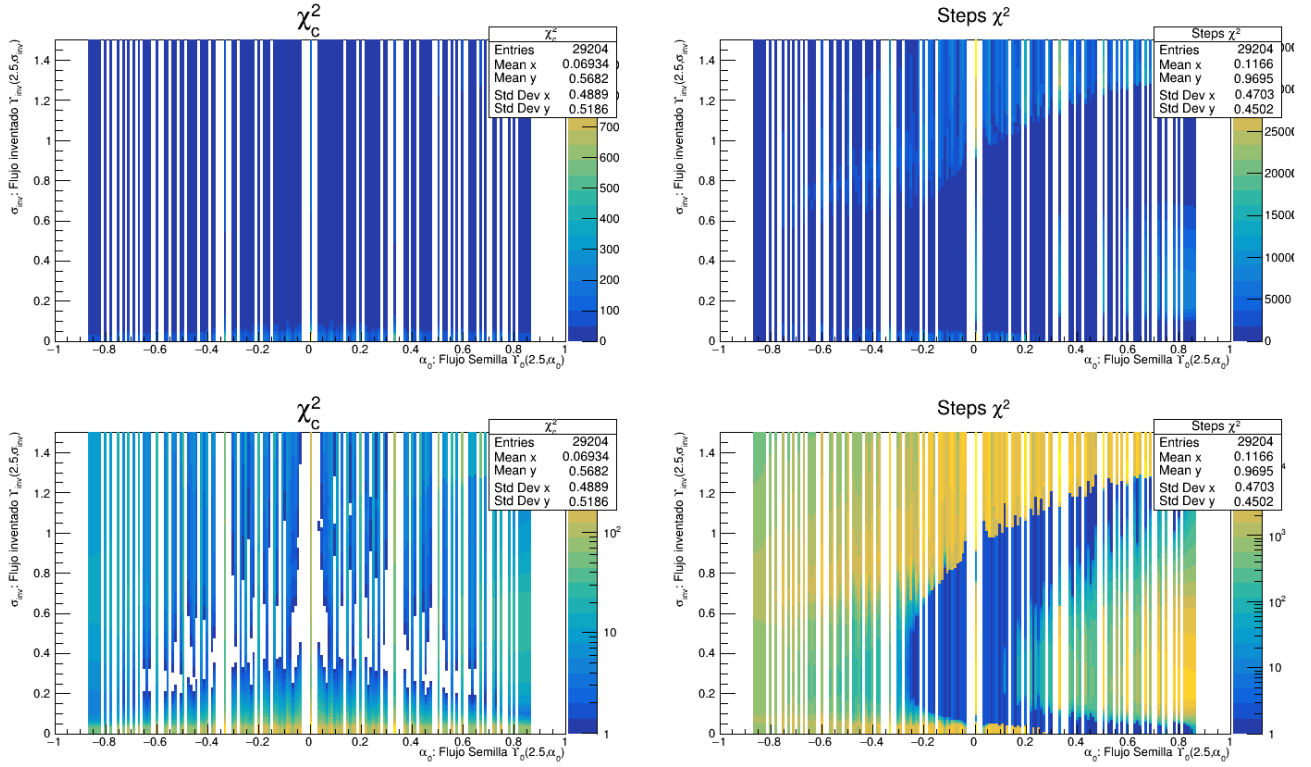


Figure 20.1: Grafico salida de la función `neutrones_deconv_plot_21()`.

Del gráfico (20.1) vemos claramente que hay un problema con el bineado en el eje horizontal. Es por esto que aparecen líneas en blanco en ese eje. El bineado es establecido al crear los histogramas TH2F en la macro `deconvoluciones.C` de la siguiente manera:

```

1 ...
2 TH2F *hcol1 = new TH2F("#chi_{c}^{2}", "#chi_{c}^{2}", 196, -1.0, 1.0, 149, 0, 1.5);
3 TH2F *hcol2 = new TH2F("Steps #chi^{2}", "Steps #chi^{2}", 196, -1.0, 1.0, 149, 0, 1.5);
4 ..

```

En los dos histogramas anteriores, al establecer para el eje vertical el bineado como `..., 149, 0, 1.5)` estamos diciendo que los anchos de los bins corresponden a anchos fijos de valor $(1.5 - 0)/149 = 0.010067114$. Y en el eje horizontal establecemos un bineado de ancho fijo de valor $|-1.0 - 1.0|/196 = 0.010204082$. Sin embargo, para este último, dados los datos, se tiene que los anchos de los bins deberían de ser variables, ya que el valor de α es oscilante.

Notemos que en el gráfico al cambiar ligeramente el bineado del eje vertical, por el siguiente:

```

1 ...
2 TH2F *hcol1 = new TH2F("#chi_{c}^{2}", "#chi_{c}^{2}", 196, -1.0, 1.0, 150, 0, 1.5);
3 TH2F *hcol2 = new TH2F("Steps #chi^{2}", "Steps #chi^{2}", 196, -1.0, 1.0, 150, 0, 1.5);
4 ..

```

es decir, a un ancho fijo de 0.01, el cual corresponde al paso exacto de los valores de la variable σ_{inv} (que va desde 0.01 hasta 1.49) se obtiene lo siguiente:

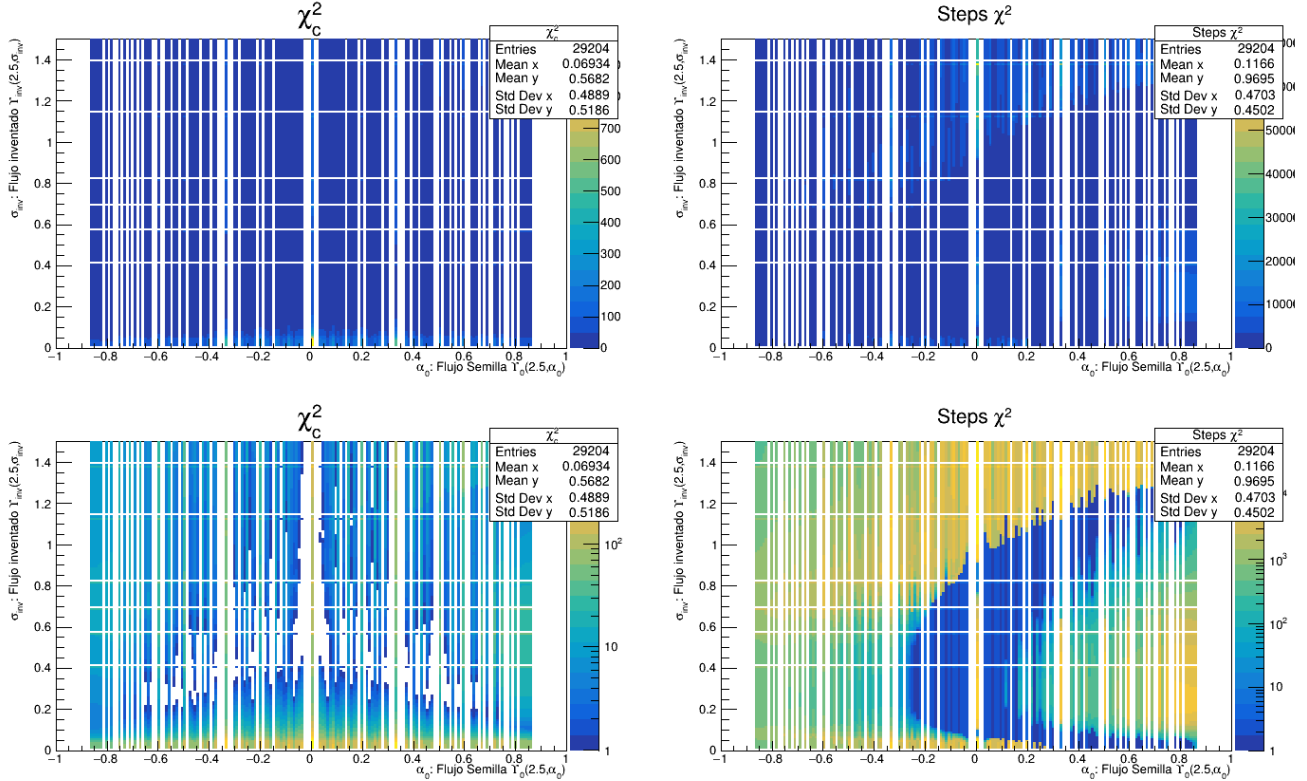


Figure 20.2: Grafico salida de la función `neutrones_deconv_plot_21()`.

Donde obtenemos líneas blancas horizontales, lo cual indica un binnedo incorrecto(?) en el eje vertical.

Luego, haciendo distintas pruebas con el binnedo del eje vertical, notamos que anchos fijos de bin de valores 0.01 ejem: (151,0,1.51), (149,0,1.49), generan líneas blancas horizontales. Mientras que binnedos de anchos con mayores decimales: 0.010068493 ejem: (146,0,1.47) no generan líneas blancas.

El valor de α al que llamamos *skewness*, u factor de oblicuidad o deformación de la distribución, lo calculamos de la siguiente manera:

$$\alpha = \frac{\sigma_R - \sigma_L}{\sigma_R + \sigma_L} \quad (20.1)$$

donde σ_R y σ_L corresponden a los anchos derecho e izquierdo de las distribuciones gaussianas usadas para construir una distribución semilla asimétrica. α así definida toma valores pertenecientes al intervalo $]-1, 1[$,

que para valores negativos me indica que la distribución tiene una inclinación hacia la izquierda y para valores positivos, tiene una inclinación hacia la derecha.

Sin embargo, notamos que al hacer un `SetMaximun()` en el `pad1` aparece un valor de 983.307, lo cuál es incorrecto para el valor de χ^2 , ya que de acuerdo al archivo `chi2_test.dat` el valor máximo de χ^2 no es mayor a 100. De esta manera, el valor incorrecto mencionado, se debe a que estamos asignando para un mismo valor de coordenadas (α, σ_{inv}) en el mapa de calor, distintos valores de χ^2 , luego estos se van sumando en el eje z .

Para comprobar esto, definimos la función `neutrones_deconv_plot_21_debug()` en la macro `deconvoluciones.C`. Esta función genera un archivo de nombre `objets_debug_test.dat` en el cual además de imprimir todos los parámetros de la simulación 21, se muestra también en la última columna el valor de skewness (α). Lo que haremos ahora será cambiar la salida al archivo `objets_debug_test.dat` para que el par (α, σ_{inv}) sean las columnas finales y así mediante las herramientas del procesador de texto `geany` buscar las repeticiones de pares ordenados. Así, encontramos lo siguiente:

- 4 repeticiones del par (0.5, 0.01), donde la suma de los χ^2 es 293.72.
- 14 repeticiones del par (0, 0.04) donde la suma de los χ^2 es 811.29807.

Y así sucesivamente, lo cuál está dentro del orden del valor máximo de χ^2 encontrado con `SetMaximun()`.

Este problema queda de manifiesto también al graficar el ancho izquierdo y derecha de la distribución gaussiana asimétrica de la semilla como mapa de calor, donde el eje z corresponde al valor de α (ver Fig. 20.3):

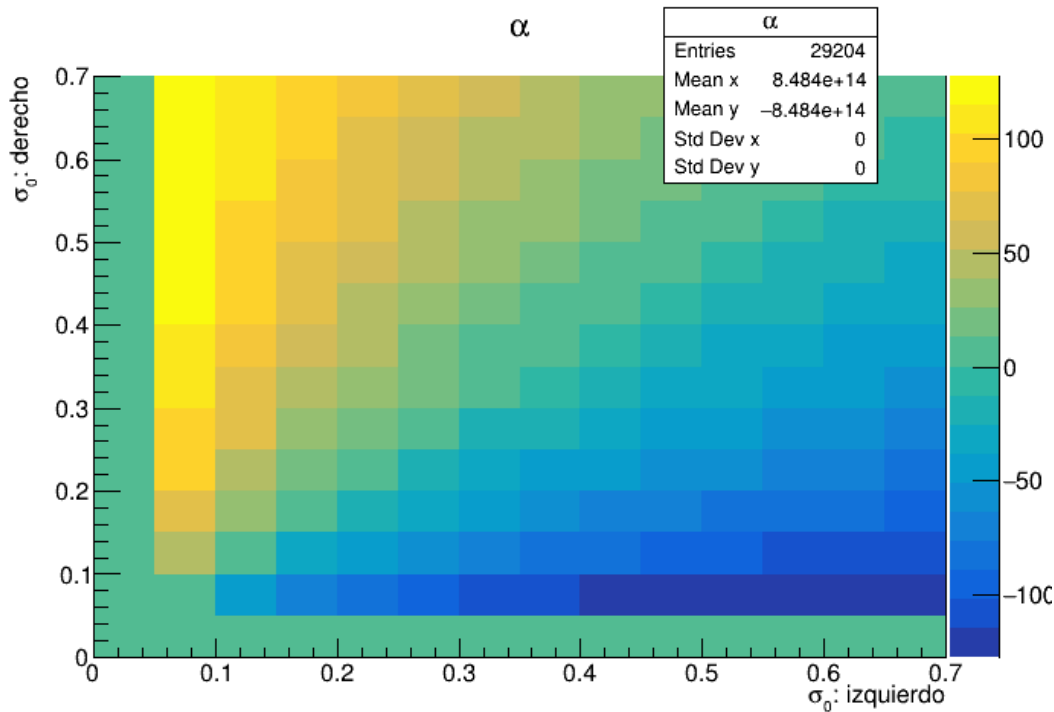


Figure 20.3: Grafico salida de la función `neutrones_deconv_plot_21_debug()`.

donde el máximo valor de α corresponde a 127.714 y el mínimo corresponde a -127.714, lo cual contradice la definición dada de α , ya que sus valores deben oscilar entre -1 y 1, manifestando entonces que hay una repetición de pares σ izquierdo y derecho (de manera que el valor de α se va sumando). Esto se puede ver al utilizar la herramienta de Find Document Usage en geany, mostrando por ejemplo que el par (0.05, 0.05) se repite al menos 100 veces en `objs_debug_test.dat`.

Simulación 22

Instrucciones para su ejecución:

- `ulimit -n 39000`: donde 38416 es el número total de filas en `chi2_test_deconv_22.dat`
- `root -l`
- `.L InitVar.cc++`
- `.L ANPulses.cc++`
- `.L neutron_sim_generator.C`
- `neutrones_sim_generator_merge_fluxinvAG()`: Se genera el archivo `neutrones_calculados_merge.dat` con los neutrones calculados asociados a un disparo (ancho determinado) y sistema de detección. Los pasos de ancho para el flujo corresponden a 0.05 o 50 keV, tanto como para el ancho izquierdo y derecho. Así, fijamos primero el ancho izquierdo en 0.05 y el ancho derecho aumenta en pasos de 0.05 comenzando desde 0.05 hasta el valor máximo de 0.7. Una vez logrado este valor para el ancho derecho, aumentamos el ancho izquierdo en 0.05 e iniciamos nuevamente con la variación del ancho derecho. La iteración termina cuando ambos anchos logren el máximo de 0.7
- `.L deconvoluciones.C`
- `neutrones_deconv_22()`: Función que realiza las deconvoluciones utilizando semillas gaussianas asimétricas (2) y flujos inventados gaussianos asimétricos (2). En ambas distribuciones los pasos de energía para las iteraciones pueden ser ajustados. Se usaron pasos de 0.05 o 50 keV para los anchos izquierdo y derecho de los flujos semilla asimétricos (como máximo 0.7 por cada lado). Se genera entonces el archivo `chi2_test.dat` donde los datos de las columnas corresponden a (visto como vector en c++) a lo mostrado en la entrada del 16.
- `neutrones_deconv_plot_22()`:
Input: `neutrones_calculados_merge.dat` Output: `chi2_test.dat`.

De esta manera, el gráfico generado es el siguiente:

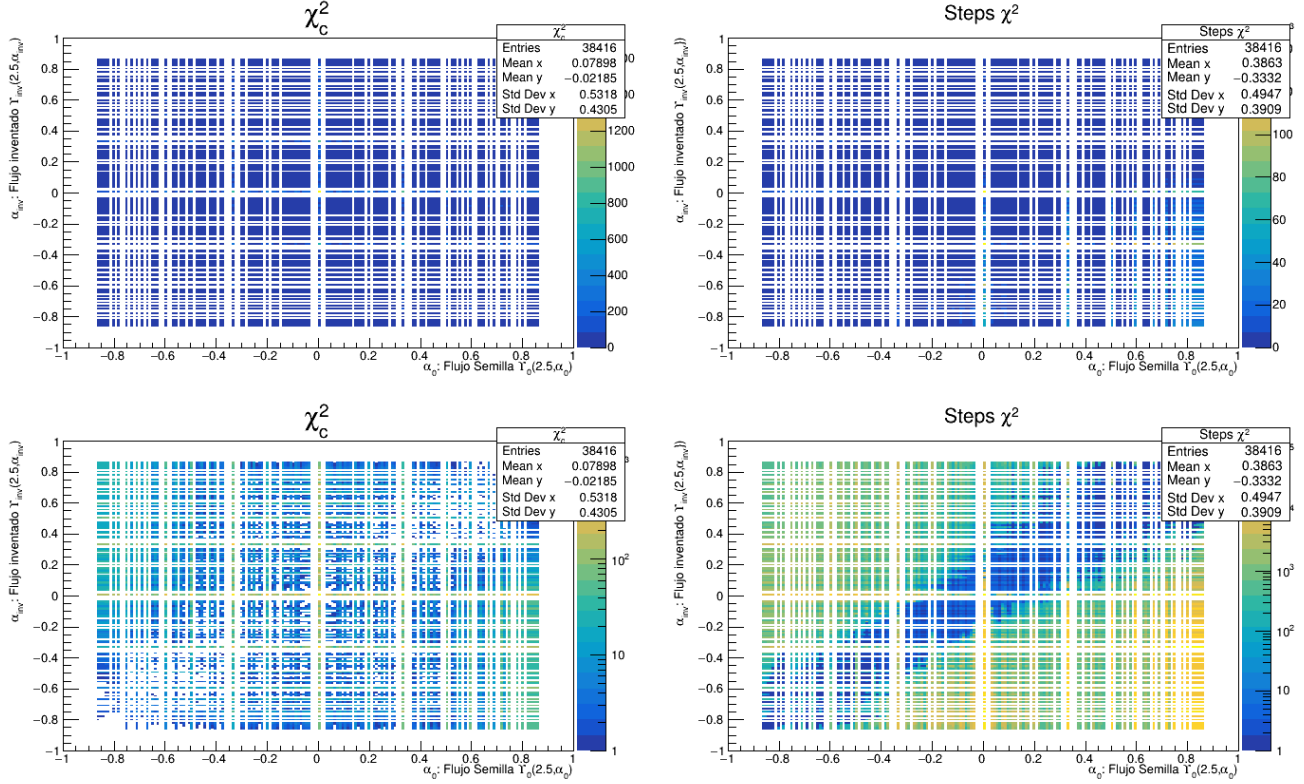


Figure 20.4: Grafico salida de la función `neutrones_deconv_plot_22()`.

Notamos nuevamente en la Fig. 20.4 un error en el bineado tanto en el eje vertical como horizontal. Además, al hacer un `SetMaximum` en el pad11 obtenemos un valor de χ^2 igual a 1714.14, el cual es incorrecto, ya que este no debería superar el valor 100. Esto se debe a un problema idéntico al de la simulación 21, es decir, tenemos pares (α_0, α_{inv}) repetidos, por lo que se van sumando los valores de χ^2 .

Las salidas de las deconvoluciones de las simulaciones 21 y 22 serán renombradas respectivamente como:

- `chi_test_deconv_21_fail.dat`
- `chi_test_deconv_22_fail.dat`

21 Entrada del 2022-01-11

Para solucionar el problema debido al uso de la definición propia de α dada en la ecuación (20.1). Utilizaremos la definición de α dada por el conjunto de librerías de `c++` de nombre `boost` (<https://www.boost.org/>), para esto instalamos la version 1.71.0 de `libboost-all` para Ubuntu. En específico, usaremos el set de herramientas (toolkit) `math` (https://www.boost.org/doc/libs/1_71_0/libs/math/doc/html/index.html) de `boost`, en su versión 2.10.0. Dentro la librería `math` en la sección `distribuitons` usaremos la llamada **Skew Normal Distribution** (https://www.boost.org/doc/libs/1_71_0/libs/math/doc/html/math_toolkit/dist_ref/dists/skew_normal_dist.html).

Luego, guiados por el ejemplo en `stackoverflow` (<https://stackoverflow.com/questions/43450880/>)

`using-boost-skew-normal-distribution`) creamos una macro de `root` de nombre `skew_normal.C` donde haremos pruebas de la distribución *skew normal* de `boost`. En esta macro definimos la función `skew()` como un objeto tipo puntero `TF1`, la cual toma como parámetro un `double` de nombre `alpha` que es el valor de la oblicuidad o skewness que tendrá la distribución que definiremos dentro de la función:

```
1 #include <boost/math/distributions/skew_normal.hpp>
2 ...
3 using namespace boost::math;
4 using namespace std;
5
6 TF1* skew(double_t alpha){
7
8 ...
9
10 }
11 ...
```

Luego, dentro de `skew()` en las primeras líneas convertimos la variable `alpha` en una tipo `string`, para poder imprimirla como título en un gráfico de la distribución:

```
1 ...
2 // Create an output string stream
3 std::ostringstream stream_alpha;
4 std::ostringstream stream_skewness;
5 // Set Fixed -Point Notation
6 stream_alpha << std::fixed;
7 stream_skewness << std::fixed;
8
9 // Set precision to 1 digits
10 stream_alpha << std::setprecision(1);
11 stream_skewness << std::setprecision(3);
12 //Add double to stream
13 stream_alpha << alpha;
14
15 // Get string from output string stream
16 std::string alpha_shape = stream_alpha.str();
17 ...
```

Luego hacemos uso de la función `skew_normal()` y la transformamos a una pdf vía la función `pdf()`, ambas de la librería `math` de `boost` (`boost::math::function()`), y a partir de esta creamos el objeto `TF1` (función en 1-D) como puntero, de nombre `skewNormal`:

```
1 TF1 *skewNormal = new TF1("skewNormal", "pdf(skew_normal([0], [1], [2]), x)",0,7);
```

La distribución skew normal es una variante de la bien conocida distribución normal o distribución de gauss. De hecho una distribución skew normal con el parámetro de forma o *shape* igual a cero corresponde a la distribución normal, por lo que esta última puede ser considerada un caso especial de la más genérica distribución skew normal.

Sabemos que la distribución normal es de la forma:

$$\varphi(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \quad (21.1)$$

luego considerando los valores de los parámetros *mean*: $\mu = 0$ y la desviación estándar o *scale*: $\sigma = 1$, entonces la distribución queda como

$$\varphi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \quad (21.2)$$

Ahora, calculamos la función de distribución acumulada o cumulative distribution function (CDF):

$$\Phi(x) = \int_{-\infty}^x \varphi(t) dt = \frac{1}{2} \left[1 + \operatorname{erf} \left(\frac{x}{\sqrt{2}} \right) \right] \quad (21.3)$$

Entonces, la función de densidad de probabilidad o probability density function (PDF) de la distribución skew normal con parámetro de shape α corresponde a:

$$f(x) = 2\varphi(x)\Phi(\alpha x), \quad (21.4)$$

Ahora, necesitamos agregar los parámetros *location* (ξ) y *shape* (ω), para esto realizamos el siguiente cambio de variable a la función $f(x)$:

$$x \rightarrow \frac{x - \xi}{\omega} \quad (21.5)$$

resultando:

$$\begin{aligned} f(x) &= \frac{1}{\omega} \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\xi}{\omega}\right)^2} \int_{-\infty}^{\left(\alpha\left(\frac{x-\xi}{\omega}\right)\right)} \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}} dt \\ f(x) &= \frac{1}{(\omega\pi)} e^{-\frac{1}{2}\left(\frac{x-\xi}{\omega}\right)^2} \int_{-\infty}^{\left(\alpha\left(\frac{x-\xi}{\omega}\right)\right)} e^{-\frac{t^2}{2}} dt \end{aligned} \quad (21.6)$$

Se puede verificar que la distribución normal se recupera cuando $\alpha = 0$, y que el valor absoluto de la asimetría aumenta a medida que aumenta el valor absoluto de α . La distribución está sesgada a la derecha si $\alpha > 0$ y está sesgada a la izquierda si $\alpha < 0$.

Esta definición (21.6) es utilizada en la línea de código (21) que escribimos anteriormente, donde los parámetros de la distribución skew normal tienen que ser establecidos:

- location (ξ): [0]

- scale (ω): [1]
- shape (α): [2]

De la definición de `skew_normal_distribution` en `boost` tenemos que *location*, *scale* y *shape* corresponden a:

```

1 // Constructor:
2 skew_normal_distribution(RealType location = 0, RealType scale = 1, RealType shape = 0);
3 // Accessors:
4 RealType location()const; // mean if normal.
5 RealType scale()const; // width, standard deviation if normal.
6 RealType shape()const; // The distribution is right skewed if shape > 0 and is left
   skewed if shape < 0.
7 // The distribution is normal if shape is zero.
```

Siguiendo con la función `TF1` definido anteriormente, seteamos los valores antes nombrados:

```

1 ...
2 skewNormal->SetParameters(2.5,1,alpha);
3 ...
```

Luego para obtener el valor de skewness o γ_1 , utilizaremos la función `boost::math::skewness()`, de la siguiente manera:

```

1 cout<< skewness(skew_normal_distribution<double>(2.5,1.,alpha)) << endl;
2
3 /*Valor de la oblicuidad */
4 double_t sk = skewness(skew_normal_distribution<double>(2.5,1.,alpha));
5
6 stream_skewness << sk;
7 std::string boost_skewness = stream_skewness.str();
8
9
10 //~ skewNormal->SetTitle(alpha_shape.c_str());
11 skewNormal->SetTitle(boost_skewness.c_str());
```

Finalmente, la función `skew()` regresa una función de root tipo `TF1` definida como una distribución skew normal con los siguientes parámetros:

- location (ξ): [0] = 2.5
- scale (ω): [1] = 1
- shape (α): [2] = `alpha`

Luego utilizamos esta función (`skew()`) para generar un canvas con 16 distribuciones skew normal (ver Fig. 21.1) para distintos valores del parámetro shape (α), y por lo tanto, distintos valores de skewness

(γ_1). Ambos parámetros mostrados en el título de cada pad junto con el valor de location y scale de cada distribución:

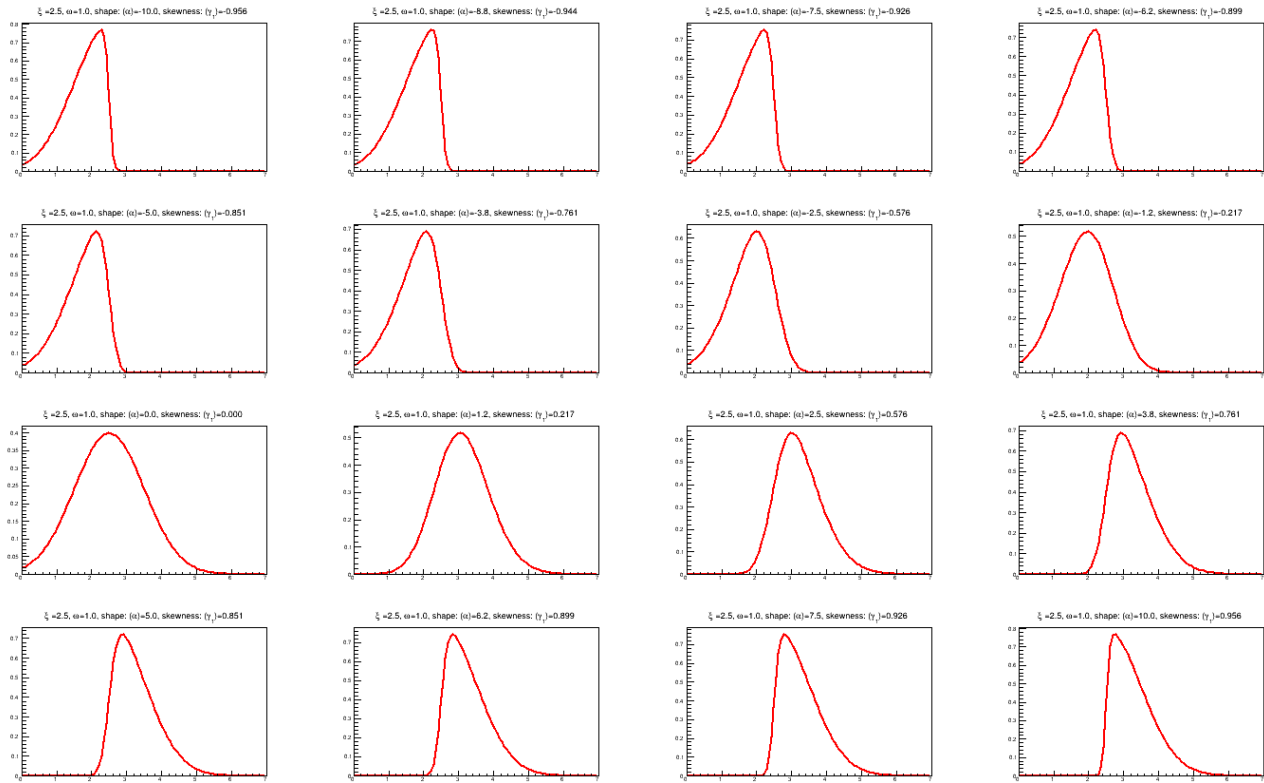


Figure 21.1: Grafico salida de la macro `skew_normal.C`

También generamos otro canvas con las 16 distribuciones superpuestas en un solo gráfico:

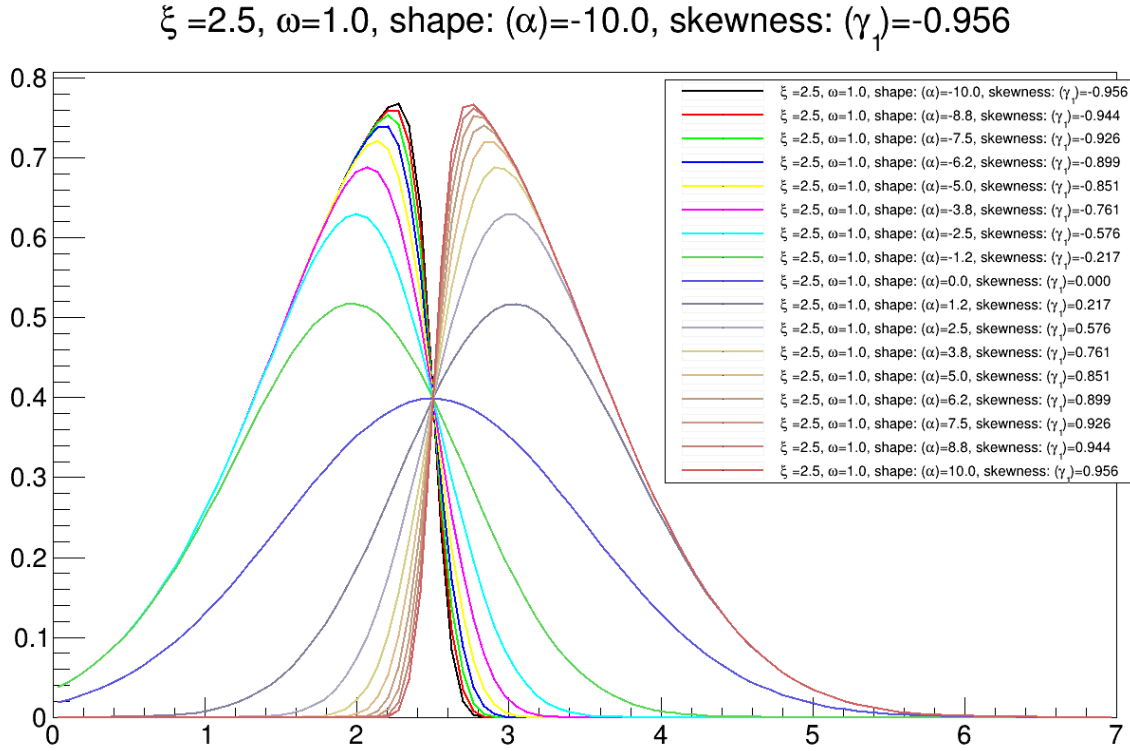


Figure 21.2: Grafico salida de la macro `skew_normal.C` con las 16 distribuciones en un solo gráfico. En el título se muestran los parámetros de la distribución de mayor amplitud de la izquierda.

De la Fig. 21.2 notamos que el valor del skewness de la distribución varía de izquierda a derecha desde el valor $\gamma_1 = -0.956$ pasando por $\gamma_1 = 0$ que es cuando la distribución corresponde a una distribución normal de promedio 2.5 y de desviación estándar igual a 1, hasta llegar al valor máximo de $\gamma_1 = 0.956$. En esta evolución el valor de la *moda* de la distribución (ver Fig. 21.3) es distinto para cada distribución. Entonces, si queremos hacer uso de estas en las deconvoluciones, tienen que tener la misma *moda* y *altura* pero distinto valor de skewness, ya que se quiere estudiar el efecto en las deconvoluciones al mover un solo parámetro a la vez en la semilla.

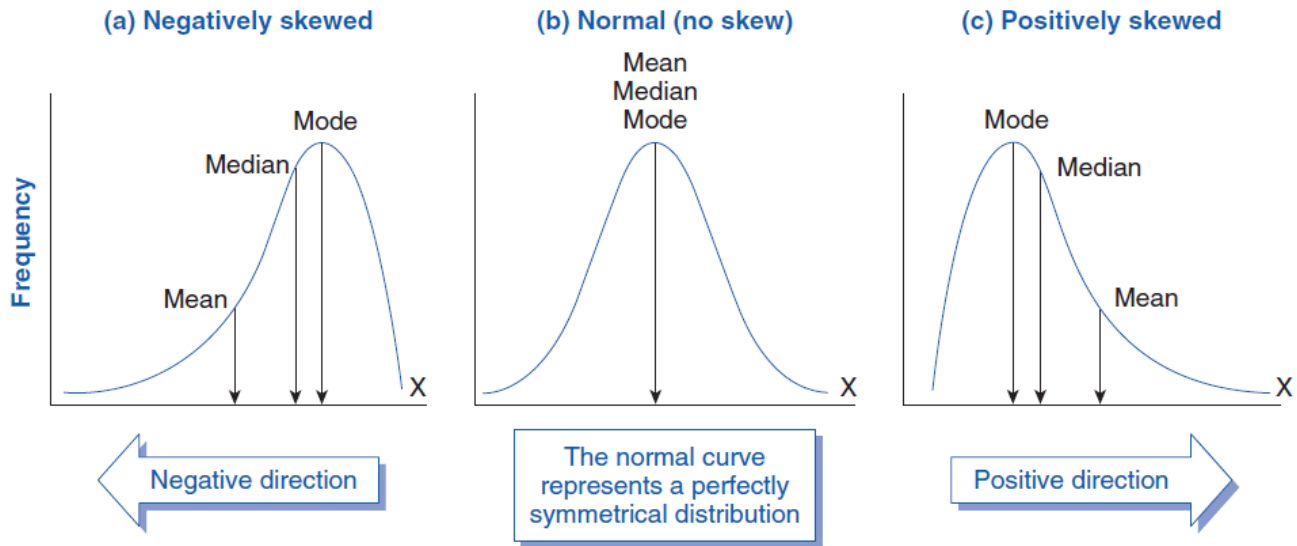


Figure 21.3: Caracterización de un distribución skew normal. Fuente: <https://www.biologyforlife.com/skew.html>

De acuerdo a la librería **math** de **boost** tenemos varias funciones implementadas para la distribución skew normal, de las cuales seleccionamos las que nos servirán para nuestro código, y que son mostradas a continuación:

Function	Implementation Notes
pdf	Usando: $f(x) = \frac{1}{(\omega\pi)} e^{-\frac{1}{2}(\frac{x-\xi}{\omega})^2} \int_{-\infty}^{\alpha(\frac{x-\xi}{\omega})} e^{-\frac{t^2}{2}} dt$
location	ξ
scale	ω
shape	α
quantile	Maximum of the pdf is sought through searching the root of $f'(x) = 0$
median	quantile(1/2)
mean	$\xi + \omega\delta\sqrt{\frac{2}{\pi}}$ donde $\delta = \frac{\alpha}{\sqrt{1+\alpha^2}}$
mode	Maximum of the pdf is sought through searching the root of $f'(x) = 0$
variance	$\omega^2 \left(1 - 2\frac{\delta^2}{\pi}\right)$
skewness	$\gamma_1 = \frac{(4-\pi)}{2} \frac{\left(\delta\sqrt{\frac{2}{\pi}}\right)^3}{\left(1-2\frac{\delta^2}{\pi}\right)^{3/2}}$

Table 21.1: Nombre de las funciones y su implementación en la librería math para distribuciones skew normal

Entonces, si queremos generar distintas distribuciones semilla o inventadas con valores de skewness distintos, se tiene que elegir un conjunto de α 's que generen un conjunto de γ_1 . Para esto, necesitamos como se comporta γ_1 en función de α . Por lo que graficaremos $\gamma_1(\alpha)$ dada la relación matemática de la tabla 21.1:

$$\gamma_1(\alpha) = \frac{(4 - \pi)}{2} \frac{\left(\frac{\sqrt{2}\alpha}{\sqrt{(1+\alpha^2)\pi}} \right)^3}{\left(1 - \frac{2\alpha^2}{(1+\alpha^2)\pi} \right)^{3/2}} \quad (21.7)$$

Obteniendo:

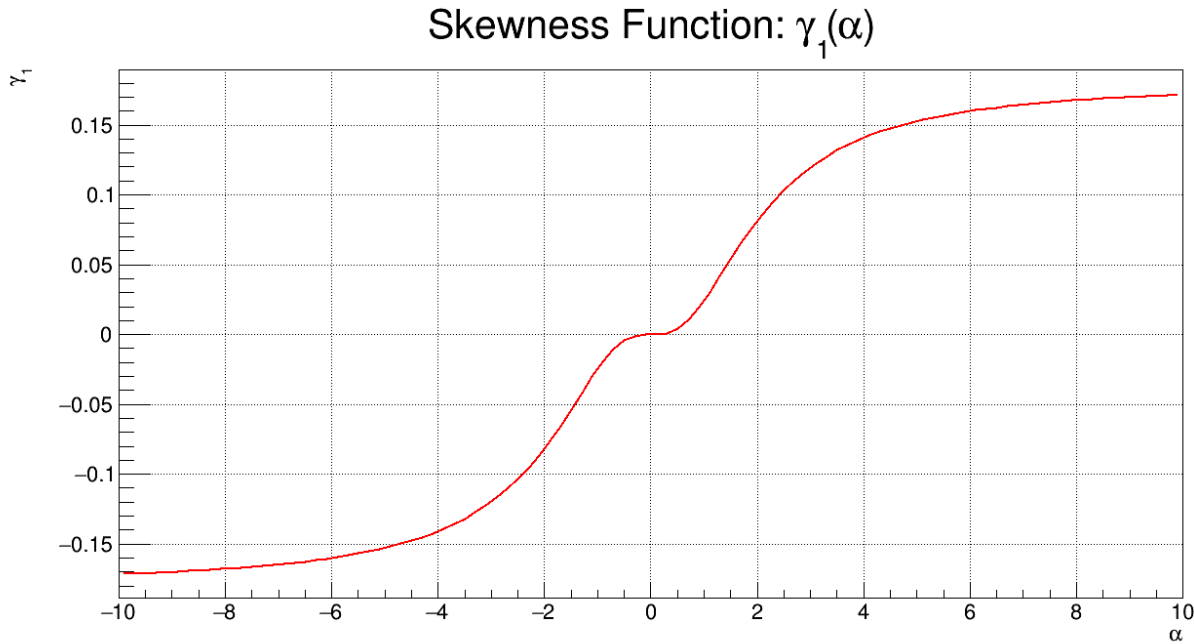


Figure 21.4: Gráfico de $\gamma_1(\alpha)$.

En la vecindad de $\alpha = 0$ la variación de γ_1 es pequeña. Luego desde $\alpha = 0.5$ la función crece (y decrece) de manera rápida tipo logaritmo, hasta lograr un valor permanente cercano a 1 y -1 respectivamente.

22 Entrada del 2022-01-13

Luego de haber explorado las distribuciones skew normal definidas con la librería **math** de **boost**, las utilizaremos para repetir las simulaciones 21 y 22, con variaciones de γ_0 que se correspondan con un solo valor de σ_{inv} y variaciones de γ_0 que se correspondan con un solo valor de γ_{inv} respectivamente.

Se modificó la clase **ANPulses.cc** en específico la función **SeedGenerator()**, esto para incluir una nueva forma de definir una distribución asimétrica, usando el conjunto de librerías **boost** (opción **dist=3**).

La inclusión se hace en las siguientes líneas de **SeedGenerator()**:

```

1 ...
2 else if (dist==3)
3 {
4     /*Skew Normal Distribution*/

```

```

5     double_t location = par1;
6     double_t scale = 1.2;
7     double_t shape = 1.0;
8
9     for (int i = 0; i < binnum; i++)
10
11     {
12         content = 0;
13
14         content = A*boost::math::pdf(boost::math::skew_normal(location, scale, SIG_L),bins[i]
15 ); //1.5
16         seed[i] = content;
17     }
18
19 }
20 ...

```

A continuación mostraremos los histogramas generados con esta opción, usando la nueva función de la macro `neutron_sim_generator.C`:

- `neutrones_sim_generator_merge_skewnormal_L_test()`

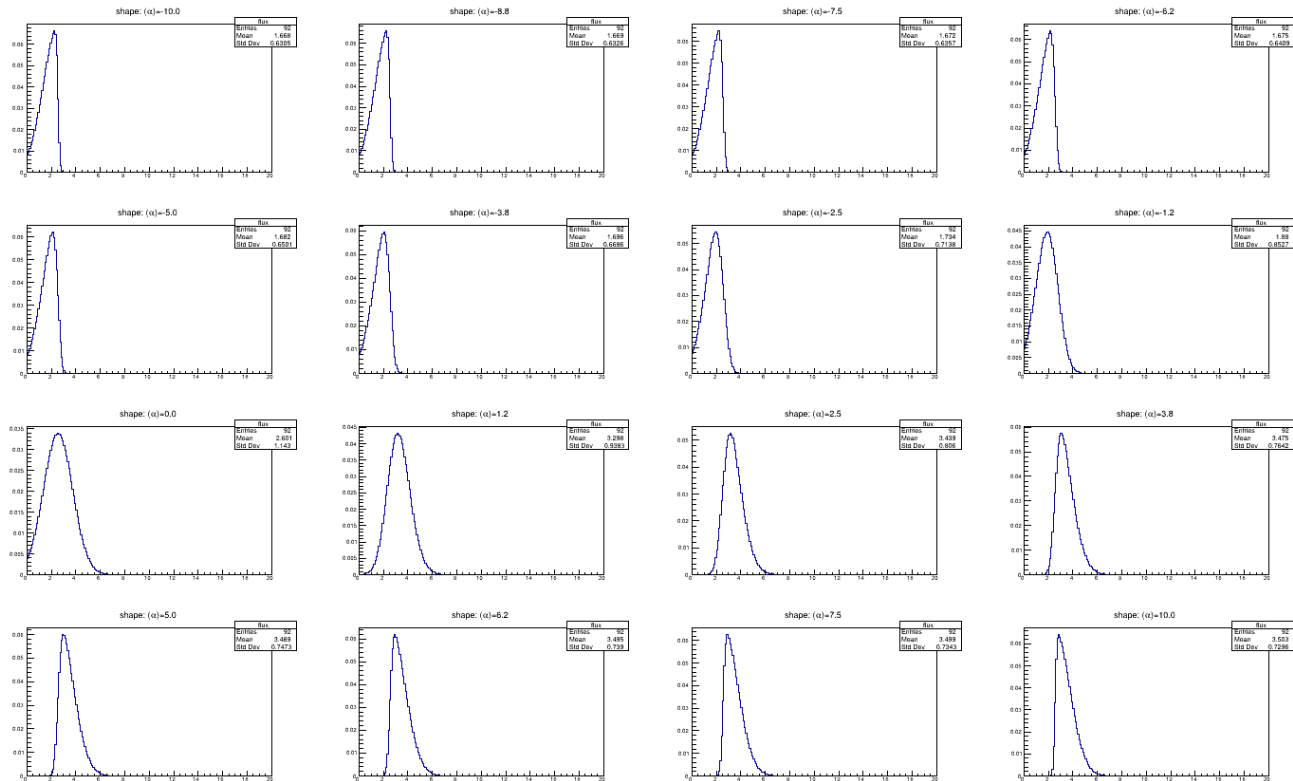


Figure 22.1: Histogramas de 16 distribuciones generados con `SeedGenerator()` para la opción `dist=3` en `Test_Unfolding_TH1D(2.5,0.2,3,sig1,1.0)`

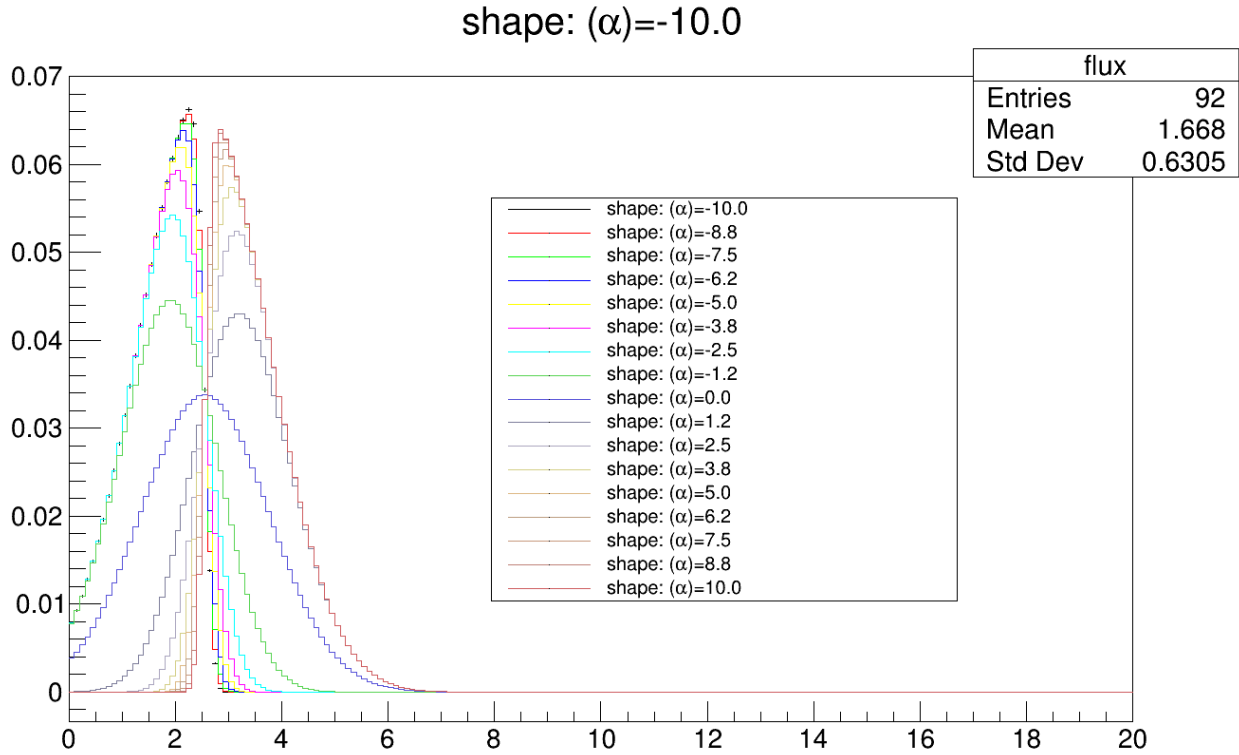


Figure 22.2: Superposición de histogramas de 16 distribuciones generados con `SeedGenerator()` para la opción `dist=3` en `Test_Unfolding_TH1D(2.5,0.2,3,sigl1,1.0)`

En la Fig 22.1 al hacer click en cualquier caja de estadística y establecer la opción `SetOptStat()` en 1001111, se puede ver que todos los histogramas están normalizados a 1, es decir, la integral de cada uno de estos es igual a 1.

En la Fig 22.2 mostramos el valor del parámetro *shape* para cada histograma. Notamos además que los histogramas del lado derecho tienen una amplitud ligeramente menor en comparación a sus histogramas espejo del lado izquierdo. Pero estos deberían ser idénticos de acuerdo a la Fig 21.2.

De esta manera, realizamos las siguientes simulaciones:

Simulación 31 (usando boost)

Instrucciones para su ejecución:

- `ulimit -n 20000`: donde 18029 es el número total de filas en `chi2_test.dat` (121 distribuciones semillas \times 149 distribuciones gaussianas inventadas, es decir, por cada distribución inventada se itera (deconvolucionada) usando una semilla, generando un total de $121 \times 149 = 18029$ iteraciones.)
- `root -l`
- `.L InitVar.cc++`
- `.L ANPulses.cc++`

- `.L neutron_sim_generator.C`
- `neutron_sim_generator_merge_fluxinvG()`: Se genera el archivo `neutrones_calculados_merge.dat` con los neutrones calculados asociados a un disparo (ancho determinado) y sistema de detección. Los pasos de ancho para el flujo corresponden a 0.01 o 10 keV, con un máximo de 1.49. Generando de esta manera 149 distribuciones
- `.L deconvoluciones.C`
- `neutrones_deconv_31_skewnormal()`:
Input: `neutrones_calculados_merge.dat` Output: `chi2_test.dat`.
Función que realiza las deconvoluciones utilizando semillas skew normal distribution (3) y flujos inventados gaussianos simétricos (1). En ambas distribuciones los pasos de energía para las iteraciones pueden ser ajustados, para la primera se ajusta el parámetro **shape** (α) y para la segunda el ancho de la distribución σ .
Dada la forma del skewness (ver Fig 21.4), el parámetro α para las semillas se varía en cuatro intervalos:
 - 1er intervalo: $[-10, -5[$ con pasos de 0.5
 - 2do intervalo: $[-5, -0.09[$ con pasos de 0.1
 - 3er intervalo: $[0, 5]$ con pasos de 0.1
 - 4to intervalo: $[5.5, 10]$ con pasos de 0.5

Esta elección, genera 121 valores de α uniformemente distribuidos en cada uno de los intervalos anteriores. Así como 121 distribuciones skew normal (semilla), y por lo tanto, 121 valores distintos de skewness (con valores idénticos pero negativos con respecto al punto espejo $\gamma_1 = 0$).

Se genera entonces el archivo `chi2_test.dat` donde los datos de las columnas corresponden a la salida generada por estas líneas de código (ver `Pretty_EMsimple_em1()` de la clase `ANPulses`).

```

1 ...
2 else if ( (dist==3) && (dist_inv==1))
3 {
4 double_t sk = boost::math::skewness(boost::math::skew_normal_distribution<double
      >(2.5,1.2,SIG_L));
5 double_t mode = boost::math::mode(boost::math::skew_normal_distribution<double
      >(2.5,1.2,SIG_L));
6 double_t median = boost::math::median(boost::math::skew_normal_distribution<double
      >(2.5,1.2,SIG_L));
7 double_t mean = boost::math::mean(boost::math::skew_normal_distribution<double
      >(2.5,1.2,SIG_L));
8 double_t variance = boost::math::variance(boost::math::skew_normal_distribution<double
      >(2.5,1.2,SIG_L));
9
10     chi2_test << left <<setw(5)<< setfill(' ') << shot << " "
11     <<setw(5) << setfill(' ') << it << " "
12     <<setw(5) << setfill(' ') << SIG_L << " " /*shape*/
13     <<setw(5) << setfill(' ') << sk << " " /*skewness de boost libray*/
14     <<setw(5) << setfill(' ') << mode << " "
15     <<setw(5) << setfill(' ') << median << " "
16     <<setw(5) << setfill(' ') << mean << " "
```

```

17      <<setw(5) << setfill(' ') << variance << " "
18      <<setw(5) << setfill(' ') << par2_inv << " "
19      <<setw(5) << setfill(' ') << chi2_em2 << endl;
20
21  }
22  ...

```

- `neutrones_deconv_plot_31_tgraph()`: Input: `chi2_test.dat` Output: Dos mapas de calor generados con root.

Los mapas de calor generados son los siguientes:

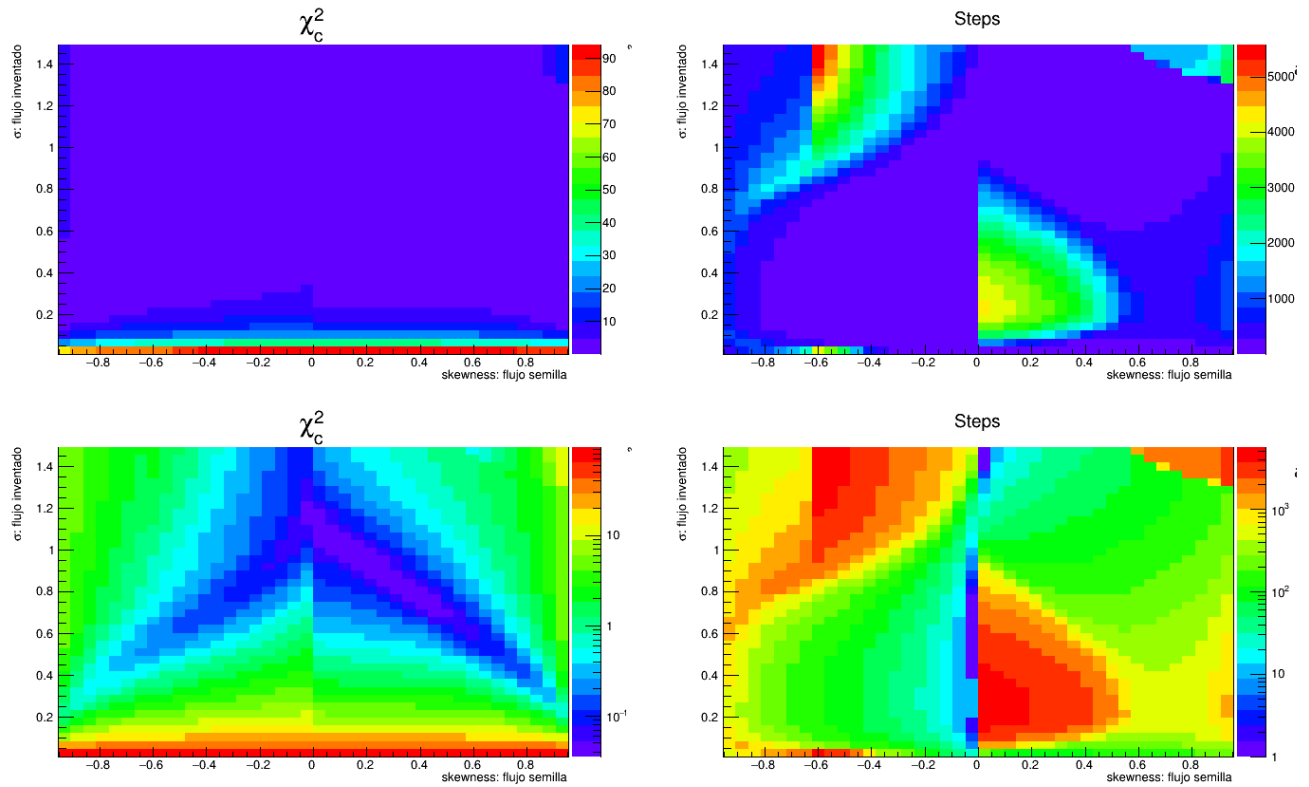


Figure 22.3: Grafico salida de la función `neutrones_deconv_plot_31_tgraph()`. En los pad11 y 21 se grafica χ_c^2 en función del valor del skewness del flujo semilla y el valor de σ del flujo inventado. En los pad12 y 22 se grafica los pasos que le toma converger a χ^2 en función del valor del skewness del flujo semilla y el valor de σ del flujo inventado.

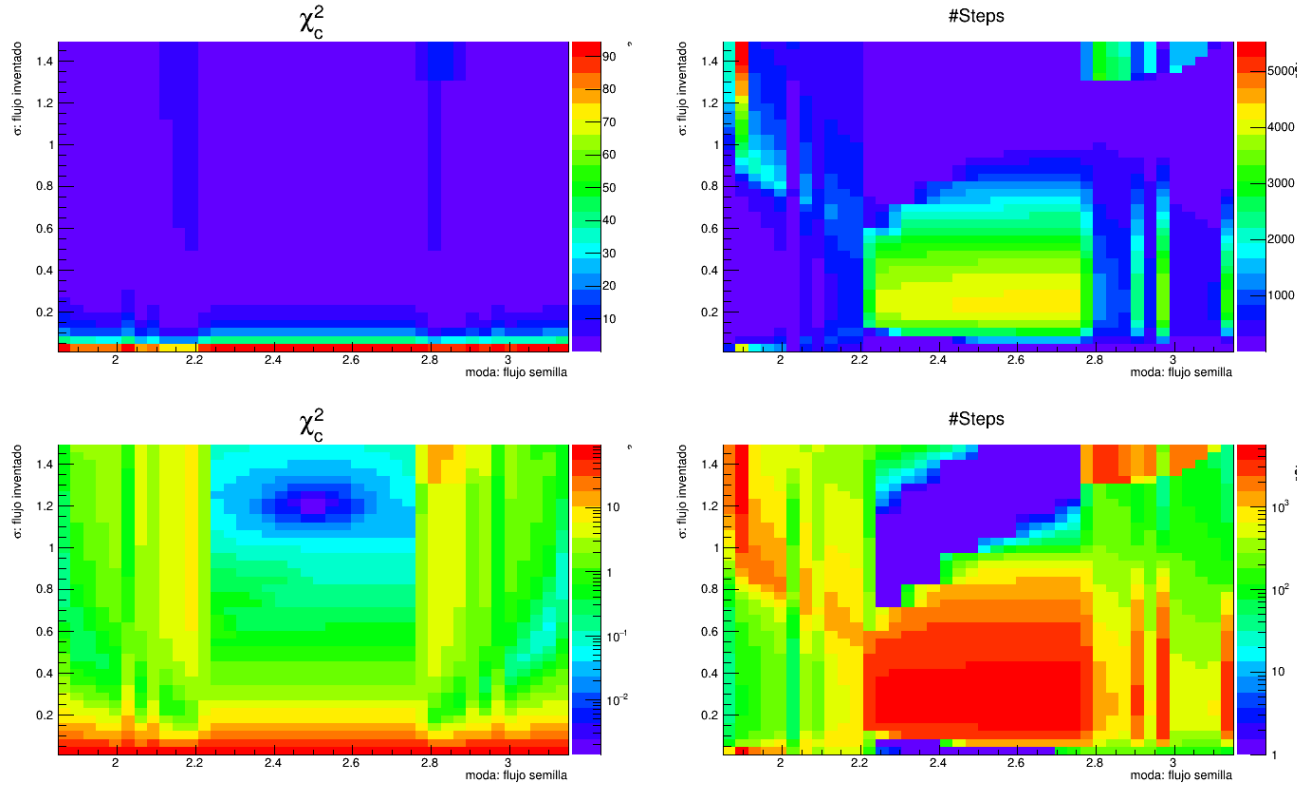


Figure 22.4: Grafico salida de la función `neutrones_deconv_plot_31_tgraph()`. En los pad11 y 21 se grafica χ^2_c en función del valor de la moda del flujo semilla y el valor de σ del flujo inventado. En los pad12 y 22 se grafica los pasos que le toma converger a χ^2 en función del valor de la moda del flujo semilla y el valor de σ del flujo inventado.

El archivo de salida de la simulación 31 de nombre `chi2_test.dat` es renombrado como `chi2_skewnormal_31_allst.dat`, donde en las distintas columnas y en orden ascendente se almacena

- **shot**: número de disparo (asociado a un único parámetro σ del flujo inventado)
- **it**: pasos que le toma al algoritmo `em1(1)` χ^2 satisfacer su condición de salida o quiebre ($\chi^2 > N$).
- **SIG_L**: Cuando en `Pretty_EMsimple_em1()` el valor de `dist=3`, tenemos que el parámetro para la distribución semilla (**SIG_L**) corresponde al shape (α) de una distribución skew normal generada con boost de location $\xi = 2.5$ y scale $\omega = 1.2$. El parámetro **SIG_R** no se considera en la salida ni tampoco para generar la distribución semilla.
- **sk**: corresponde al skewness de la distribución semilla obtenida con `boost`.
- **mode**: corresponde a la moda de la distribución semilla obtenida con `boost`.
- **median**: corresponde a la mediana de la distribución semilla obtenida con `boost`.
- **mean**: corresponde al promedio de la distribución semilla obtenida con `boost`.
- **variance**: corresponde a la varianza de la distribución semilla obtenida con `boost`.

- `par2_inv`: corresponde al σ de la distribución de flujo inventado.
- `chi2_em2`: corresponde al valor de χ_c^2 del algoritmo em2 [2](#), que resulta de comparar las formas del flujo deconvolucionado con el flujo inventado.

Simulacion 33 (usando boost)

Instrucciones para su ejecución:

- `ulimit -n 15000`: donde 14641 es el número total de filas en `chi2_test.dat` (121 distribuciones semillas \times 121 distribuciones skew normal inventadas, es decir, por cada distribución inventada se itera (deconvoluciona) usando una semilla, generando un total de $121 \times 121 = 14641$ iteraciones.)
- `root -l`
- `.L InitVar.cc++`
- `.L ANPulses.cc++`
- `.L neutron_sim_generator.C`
- `neutrones_sim_generator_merge_skewnormal_L()`:
Se genera el archivo `neutrones_calculados_merge.dat` con los neutrones calculados asociados a un disparo (valor de shape α) y sistema de detección. Los pasos de α para el flujo corresponden a cuatro los dado en cuatro intervalos:
 - 1er intervalo: $[-10, -5[$ con pasos de 0.5
 - 2do intervalo: $[-5, -0.09[$ con pasos de 0.1
 - 3er intervalo: $[0, 5]$ con pasos de 0.1
 - 4to intervalo: $[5.5, 10]$ con pasos de 0.5

Generando así 121 distribuciones skew normal como flujos inventados.

- `.L deconvoluciones.C`
- `neutrones_deconv_33_skewnormal()`:
Input: `neutrones_calculados_merge.dat` Output: `chi2_test.dat`. Función que realiza las deconvoluciones utilizando semillas skew normal distribution (3) y flujos inventados skew normal distribution (3). En ambas distribuciones los pasos de energía para las iteraciones pueden ser ajustados, siendo en las dos el parámetro **shape** (α). El cual se distribuye para las semillas de igual manera que para los flujos inventados que se mostró anteriormente.
- `neutrones_deconv_plot_33_tgraph()`: Input: `chi2_test.dat` Output: Dos mapas de calor generados con root.

Se genera entonces el archivo `chi2_test.dat` donde los datos de las columnas corresponden a la salida generada por estas líneas de código (ver `Pretty_EMsimple_em1()` de la clase `ANPulses`):

```

1 else if ( (dist==3) && (dist_inv==3))
2 {
3 double_t sk = boost::math::skewness(boost::math::skew_normal_distribution<double>(2.5,1.2,
    SIG_L));
4 double_t mode = boost::math::mode(boost::math::skew_normal_distribution<double>(2.5,1.2,
    SIG_L));
5 double_t median = boost::math::median(boost::math::skew_normal_distribution<double>(2.5,1.2,
    SIG_L));
6 double_t mean = boost::math::mean(boost::math::skew_normal_distribution<double>(2.5,1.2,
    SIG_L));
7 double_t variance = boost::math::variance(boost::math::skew_normal_distribution<double
    >(2.5,1.2,SIG_L));
8
9 double_t sk_inv = boost::math::skewness(boost::math::skew_normal_distribution<double
    >(2.5,1.2,SIG_L_inv));
10 double_t mode_inv = boost::math::mode(boost::math::skew_normal_distribution<double>(2.5,1.2,
    SIG_L_inv));
11 double_t median_inv = boost::math::median(boost::math::skew_normal_distribution<double
    >(2.5,1.2,SIG_L_inv));
12 double_t mean_inv = boost::math::mean(boost::math::skew_normal_distribution<double>(2.5,1.2,
    SIG_L_inv));
13 double_t variance_inv = boost::math::variance(boost::math::skew_normal_distribution<double
    >(2.5,1.2,SIG_L_inv));
14
15     chi2_test << left <<setw(5)<< setfill(' ') << shot << " "
16     <<setw(5) << setfill(' ') << it << " "
17     <<setw(5) << setfill(' ') << SIG_L << " " /*shape semilla*/
18     <<setw(5) << setfill(' ') << sk << " " /*skewness de boost libray*/
19     <<setw(5) << setfill(' ') << mode << " "
20     <<setw(5) << setfill(' ') << median << " "
21     <<setw(5) << setfill(' ') << mean << " "
22     <<setw(5) << setfill(' ') << variance << " "
23     <<setw(5) << setfill(' ') << SIG_L_inv << " " /*shape flujo inventado*/
24     <<setw(5) << setfill(' ') << sk_inv << " " /*skewness de boost libray*/
25     <<setw(5) << setfill(' ') << mode_inv << " "
26     <<setw(5) << setfill(' ') << median_inv << " "
27     <<setw(5) << setfill(' ') << mean_inv << " "
28     <<setw(5) << setfill(' ') << variance_inv << " "
29     <<setw(5) << setfill(' ') << chi2_em2 << endl;
30 }

```

Que de manera ordena corresponden a:

- **shot**: número de disparo (asociado a un único parámetro γ del flujo inventado)
- **it**: pasos que le toma al algoritmo `em1`(1) χ^2 satisfacer su condición de salida o quiebre ($\chi^2 > N$).
- **SIG_L**: Cuando en `Pretty_EMsimple_em1()` el valor de `dist=3`, tenemos que el parámetro para la distribución semilla (`SIG_L`) corresponde al shape (α) de una distribución skew normal generada con boost de location $\xi = 2.5$ y scale $\omega = 1.2$. El parámetro `SIG_R` no se considera en la salida ni tampoco para generar la distribución semilla.

- **sk**: corresponde al skewness de la distribución semilla obtenida con **boost**.
- **mode**: corresponde a la moda de la distribución semilla obtenida con **boost**.
- **median**: corresponde a la mediana de la distribución semilla obtenida con **boost**.
- **mean**: corresponde al promedio de la distribución semilla obtenida con **boost**.
- **variance**: corresponde a la varianza de la distribución semilla obtenida con **boost**.
- **SIG_L_inv**: corresponde al α de la distribución skew normal de flujo inventado obtenida con **boost**.
- **sk_inv**: corresponde al skewness de la distribución skew normal de flujo inventado obtenida con **boost**.
- **mode_inv**: corresponde a la moda de la distribución skew normal de flujo inventado obtenida con **boost**.
- **median_inv**: corresponde a la mediana de la distribución skew normal de flujo inventado obtenida con **boost**.
- **mean_inv**: corresponde al promedio de la distribución skew normal de flujo inventado obtenida con **boost**.
- **variance_inv**: corresponde a la varianza de la distribución skew normal de flujo inventado obtenida con **boost**.
- **chi2_em2**: corresponde al valor de χ_c^2 del algoritmo em2 [2](#), que resulta de comparar las formas del flujo deconvolucionado con el flujo inventado.

Los mapas de calor generados por la simulación 33 son los siguientes:

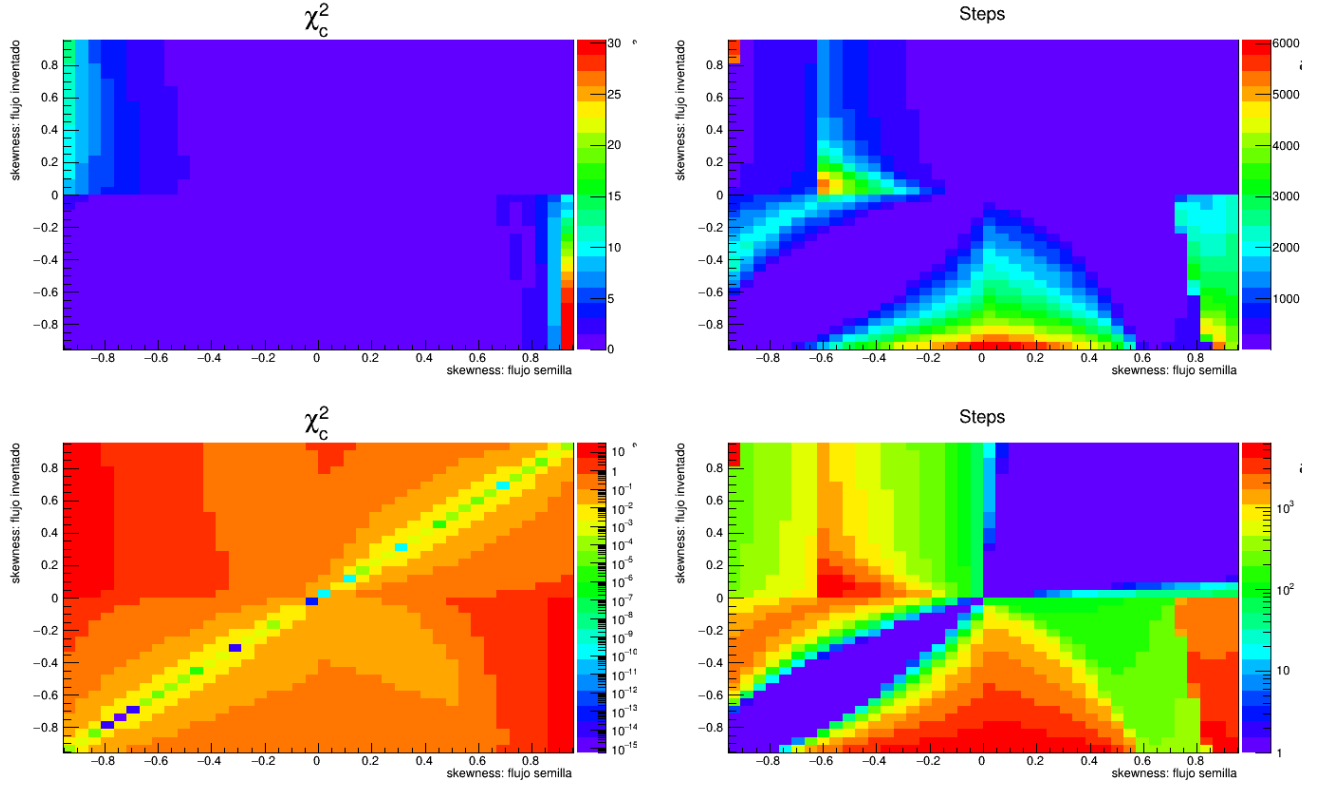


Figure 22.5: Grafico salida de la función `neutrones_deconv_plot_33_tgraph()`. En los pad11 y 21 se grafica χ_c^2 en función del valor del skewness del flujo semilla y el valor del skewness del flujo inventado. En los pad12 y 22 se grafica los pasos que le toma converger a χ^2 en función del valor del skewness del flujo semilla y el valor del skewness flujo inventado.

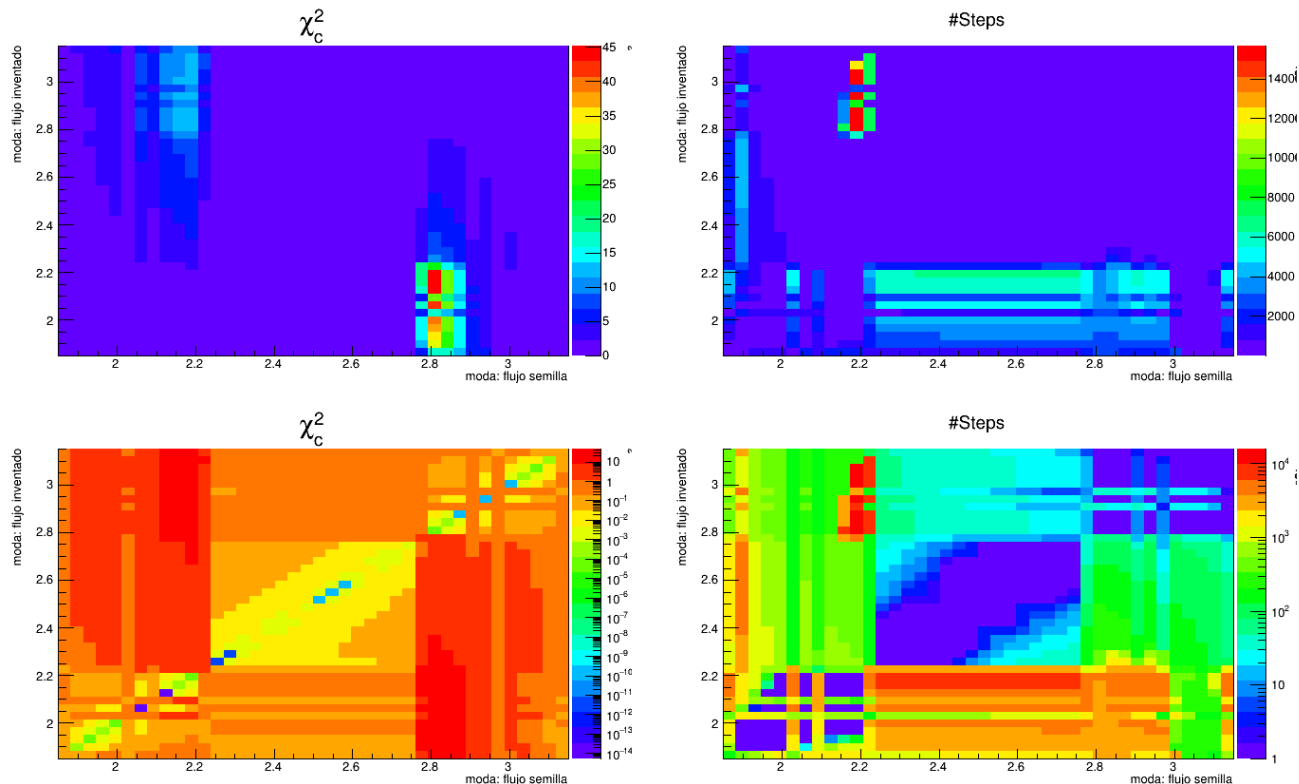


Figure 22.6: Grafico salida de la función `neutrones_deconv_plot_33_tgraph()`. En los pad11 y 21 se grafica χ_c^2 en función del valor de la moda del flujo semilla y el valor de γ del flujo inventado. En los pad12 y 22 se grafica los pasos que le toma converger a χ^2 en función del valor de la moda del flujo semilla y el valor de γ del flujo inventado.

Antes de continuar, revisando el código, se encontró en la línea 1883 de la clase `ANPulses.cc` de la última actualización del repositorio en gitlab, en específico en la función `Pretty_EMsimple_em1()`, una doble definición de `chi2_em2`:

```
1  chi2_em2 = Xi_Square_em2(vec_fluxinv, vec_fluxdeconv, flux_inv->Integral() , flux_inv->
    GetNbinsX());chi2_em2 = Xi_Square_em2(vec_fluxinv, vec_fluxdeconv, flux_deconv->Integral
    () , flux_inv->GetNbinsX());
```

En principio la única repercusión que debería tener es en el tiempo de ejecución del código, ya que hace un cálculo repetido e innecesario. Por lo tanto, en la actualización siguiente del repositorio se elimina la definición repetida.

23 Entrada del 2022-01-14

De la Fig. 22.5 podemos ver en el pad21 que los menores valores de χ_c^2 corresponden a la diagonal, esto es así ya que el método de la maximización de la expectación obtiene un mejor resultado de flujo deconvolucionado cuando la semilla es muy parecida al flujo inventado.

Reunión con los profesores Francisco Molina y Víctor Muñoz. (5ta)

Se muestran los resultados obtenidos de las simulaciones 31 (Fig. 22.3) y 33 (Fig. 22.3). Se comenta la extraña discontinuidad que se ve en el pad 21 de la Fig. 22.3, una de las posibles causas de esta es que al definir un valor de shape α para cada flujo semilla (y por lo tanto un valor de skewness γ asociado) se está moviendo al mismo tiempo tanto el centroide de la distribución (la moda para una distribución skew normal) como la amplitud de la distribución semilla (ver Fig. 22.2). Es decir, estamos observando el efecto en el algoritmo EM para el flujo deconvolucionado con respecto de la variación de dos valores, cuando en principio queremos estudiar el efecto de una sola variación, que es la deformación o skewness de la distribución semilla.

El profesor VM sugiere utilizar la nomenclatura de *momentos* para referirnos a las propiedades de una distribución, como lo son: la media, varianza y skewness. También se sugiere seguir explorando en aplicar algoritmos de redes neuronales para obtener el flujo de neutrones medido por los detectores.

24 Entrada del 2022-02-24

Actualización del fork del `research-logbook` en github hasta la entrada anterior (23):

- Repositorio: <https://github.com/flopezus/research-logbook>
- commit: 9a2493fba25f0f103e1009ca5d29136ff92de778

25 Entrada del 2022-02-25

Se realizó una actualización del repositorio `class-anpulses` en gitlab con las modificaciones más recientes:

- `ANPulses.cc` commit 8cb306b84528f87fae1ec5ec06c3d83474fb94a5: Se agrega el header de boost para usar sus librerías, en específico la distribución skew normal de la librería math. En `SeedGenerator()` se agrega una nueva opción (3) para dist, la cual genera un histograma usando la distribución skew normal. En `Pretty_EMsimple_em1()` se agregan nuevas salidas al archivo `chi2_test.dat`, que dan cuenta de las simulaciones 31 y 33.
- `deconvoluciones.C` commit 56ddde7b6feeb905f6cd1f56b6868dfc8fad8bcf: Se agrega la función `neutrones_deconv_plot_21_debug()` para debugear. Se agregan las funciones `neutrones_deconv_31_skewnormal()` y `neutrones_deconv_33_skewnormal()`, las cuales realizan las deconvoluciones usando la opción `dist=3` en `Pretty_EMsimple_em1()`. Se agregan las funciones `neutrones_deconv_plot_31_tgraph()` y `neutrones_deconv_plot_33_tgraph()` para graficar los mapas de calor de las simulaciones 31 y 33. Además se agregan las funciones `neutrones_deconv_22()` y `neutrones_deconv_plot_22()`. Así como también la función `neutrones_deconv_plot_21_tgraph()`.
- `readme` commit 4ae18483cc0795f1f80fc143e472298382a96fe8: Actualización de las instrucciones para realizar las simulaciones 11, 21, 22, 31 y 33.

- `neutron_sim_generator.C` commit 430748499a4ee349655b3808c4568e4b8a20ea79: Se agrega la función `neutrones_sim_generator_merge_skewnormal_L()` para generar neutrones correspondientes a distribuciones asimétricas del tipo skew normal.
Se agrega la función `neutrones_sim_generator_merge_skewnormal_L_test()` para realizar pruebas (gráficos).
- `test.C` commit 4477a70ee237dac30ffba2201d0298269338187c: Actualización de pruebas.
- `skew_normal.C` commit b3b998268e8f2c67a19e1701a5b0826526a64481: Se agrega esta nueva macro con pruebas del conjunto de librerías boost.
- Se agregan los archivos:
 - `chi2_test_deconv_21_fail.dat`
 - `chi2_test_deconv_22_fail.dat`
 - `chi2_test_deconv_12.dat`
 - `chi2_test_deconv_11.dat`
 - `chi2_skewnormal_33_allst.dat`
 - `chi2_skewnormal_31_allst.dat`
- `neutrones_calculados/deconv`: Se agregan los archivos de neutrones calculados correspondientes a distribuciones gaussianas, gaussianas asimétricas y skew normal:
 - `neutrones_calculados/deconv/neutrones_calculados_merge_fluxinv_G.dat`
 - `neutrones_calculados/deconv/neutrones_calculados_merge_fluxinv_AG.dat`
 - `neutrones_calculados/deconv/neutrones_calculados_merge_skewnormal.dat`
- `img`: Se agrega esta carpeta conteniendo un repositorio de imágenes de los mapas de calor de las distintas simulaciones.
- `objs_debug_test.dat`: Se agrega este archivo de prueba que es salida de la función `neutrones_deconv_plot_21_debug()` de la macro `deconvoluciones.C`, para comprobar si es que hay repeticiones de los valores del skewness en la simulación 21.
- `objs_debug.dat`: Se agrega este archivo que se genera en `Pretty_EMsimple_em1()` con la salida de distintos objetos del algoritmo: Anchos de bins de energía, Flujo semilla, Eficiencias absolutas, etc. Sirve para debugear.

Tratar de hacer un gráfico de pasos (it) versus valor del error del método $\text{em1 } \chi^2$ (para descartar un overfitting del método.)

26 Entrada del 2022-03-03

Si queremos que los flujos inventados tengan la misma amplitud para las distribuciones gaussianas y gaussianas asimétricas no hay que normalizar por el área (no hacer scale de root al histograma en el código). Para los flujos del tipo skew normal, tenemos que establecer el parámetro scale (ω) de tal forma que esté relacionado con el valor de varianza de las distribuciones gaussianas con las que se quiere comparar.

Tenemos que para una distribución skew normal, la varianza se escribe como

$$\sigma^2 = \omega^2 \left(\frac{\alpha}{\sqrt{1 + \alpha^2}} \right) \quad (26.1)$$

Luego si el shape es cero, es decir, $\alpha = 0$, entonces tenemos una distribución normal.

Pero, en realidad ¿la amplitud de los flujos de neutrones es un parámetro importante a considerar en las simulaciones?. El PF por disparo emite cierto número de neutrones, los cuáles se relacionan con la distribución o flujo inicial mediante el valor de la integral total de esta distribución, es decir, si aumento la amplitud del flujo generador voy a tener mayor número de neutrones, pero el aumento del número de neutrones es proporcional al aumento de esta amplitud. Entonces, en realidad el parámetro fundamental es la integral total, ya que esta me determina el número de neutrones emitidos. Entonces, si queremos estudiar cuál es el efecto de un cambio en el ancho de a distribución semilla en el flujo deconvolucionado, no es necesario que todas tengan la misma amplitud, si no que es importar normalizar las integrales.

Para mantener la misma moda de todas las distribuciones skew normal (semillas) de la librería boost en función de distintos parámetros de skewness (ya que quiero variar solo el ancho de la distribución) hay varias posibles soluciones. Una es trasladar toda la distribución a la moda de la distribución gaussiana generadora. Esto es, establecer a que bin corresponde la moda de la distribución skew normal, establecer el bin de la moda de la gaussiana (la cual está centrada en 2.5), luego calcular la diferencia entre esos bins y trasladar todo el bineado del vector skew normal (a la derecha o izquierda) con respecto a este valor ese valor.

27 Entrada del 2022-03-07

El método de las esferas de Bonner para determinar el espectro de energía de un campo neutrónico, tiene varias ventajas y desventajas [1], siendo las dos desventajas principales: la pobre resolución en energía y los problemas al hacer el unfolding. Para el primero, un sistema ideal tiene que tener tantos detectores como regiones de energía (para que la matriz sea cuadrada y por lo tanto resoluble de manera única), por ejemplo, si queremos resoluciones de 100 keV hasta el valor de energía de 14 MeV, se requieren 140 detectores (cada uno con una función respuesta distinta). Por lo que la resolución de energía es insalvable, sin embargo, al compararla con la técnica de tiempo de vuelo, esta dificultad se ve mejor frente a la incertidumbre del tiempo cero de medición que necesita el tof (que generalmente viene de las señales de rayos x duros), ya que hay indicios de emisión de neutrones incluso en las etapas anteriores a la máxima compresión del plasma (Cristián Pavez, buscar referencias). Ver en que etapa se producen los rayos x que el tof toma como tiempo cero.

Para la calibración de los detectores proporcionales de manera de obtener del yield de neutrones, no se usará la metodología de Ariel-Tarifeño que dependen de calibraciones anteriores [2] (como yo pensaba) si no que la reducción de las incertezas vienen de la mano de conocer bien las funciones respuesta de detección de cada detector.

28 Entrada del 2022-03-11

Los histogramas usados en el algoritmo de maximización de la expectación si tienen el bineado de las eficiencias generadas con GEANT4 (al generarlos se copia el bineado). Esto es, isoletargico hasta el valor 8 del eje x (no el número de bin), luego lineal.

29 Entrada del 2022-03-15

Las actualizaciones realizadas al código hasta ahora son las siguientes:

Dado que teníamos el problema de que al generar flujos usando la distribución skew normal de las librerías boost, las posiciones de los centroides variaban (con respecto al valor 2.5) para distintos valores de forma α (ver Fig 22.2). Para arreglar esto, se prueba una posible solución en la macro `test.C` mediante la función `dist_inv_test()`. Con esta generamos todos los tipos de distribuciones implementadas en la clase `ANPulses.cc`, y las mostramos en un canvas, de manera independiente y en superposición. También se genera un segundo canvas donde se muestra en el pad superior dos distribuciones skew normal de distinto α superpuestas, dando cuenta de su distinto centroide (para igual parámetro ω y ξ). Luego en el pad inferior se muestra la implementación de la traslación de una la distribución skew normal con respecto del centroide de la distribución gaussiana, para que así ambas queden centradas en el mismo valor.

Entonces, la salida gráfica de `dist_inv_test()` corresponde a:

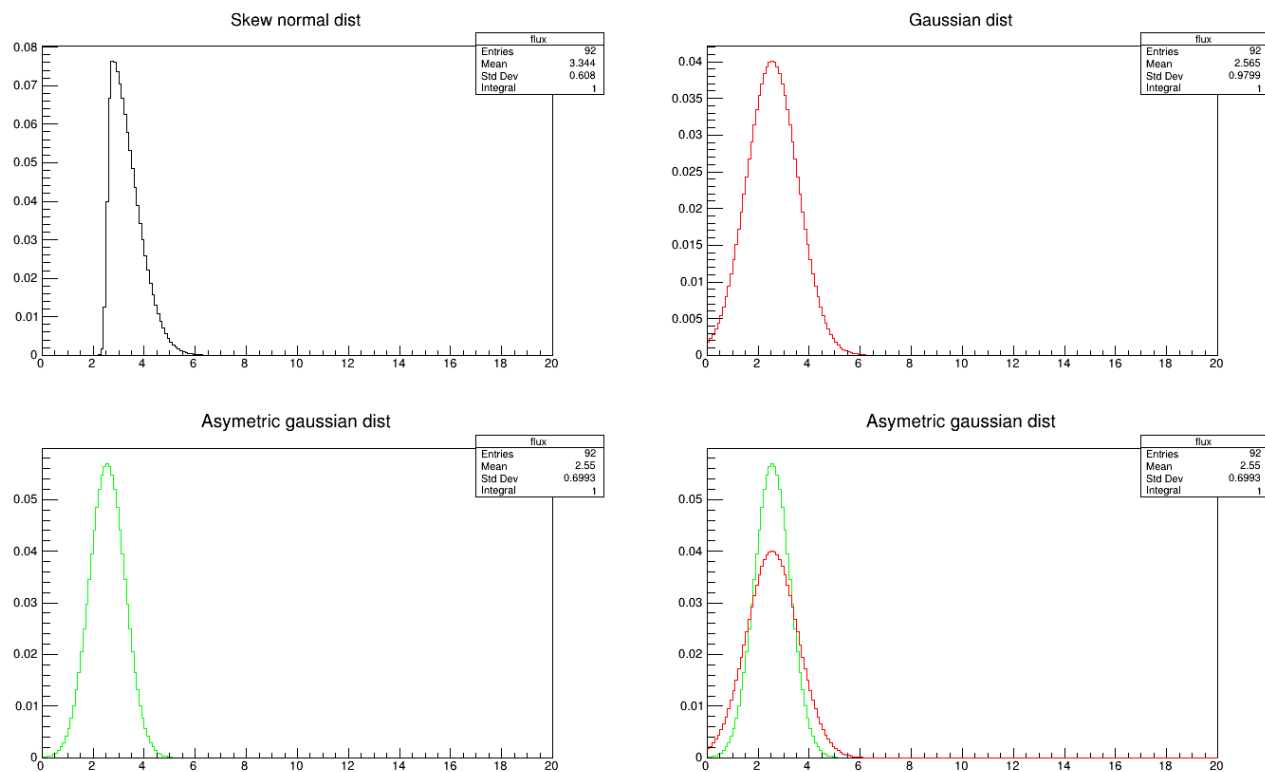


Figure 29.1: Canvas 1 con las distintas distribuciones.

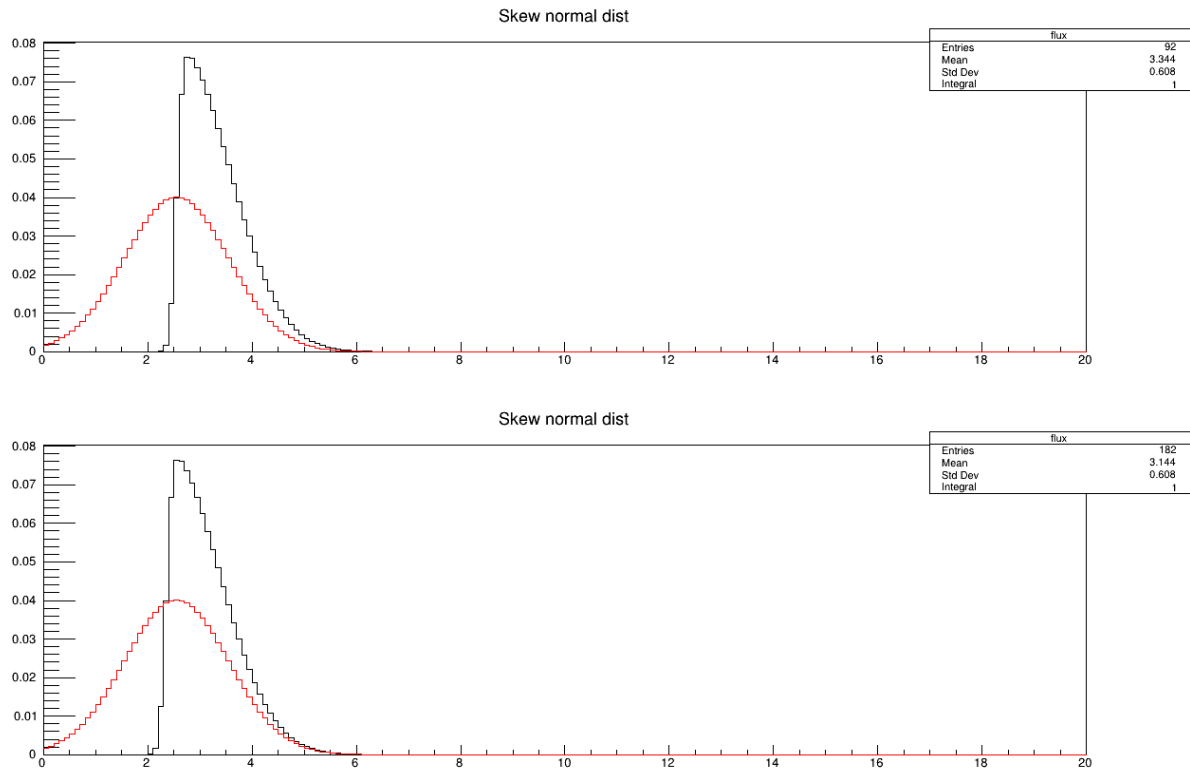


Figure 29.2: Canvas 2 con la comparación de la traslación.

Además, con respecto al canvas 2, la función muestra en el terminal el número de bin cuyo contenido es el máximo para cada distribución y la posición en el eje x del bin máximo, esto antes y después de la traslación:

```

1 ...
2 binmax gaussian dist: 26
3 x position of maximum content G 2.55
4 binmax skew normal dist: 28
5 x position of maximum content G 2.75
6 AFTER SHIFT
7 binmax skew normal dist: 26
8 x position of maximum content 2.55

```

Como vemos, esta solución arregla el problema del movimiento del centroide para las distribuciones skew normal, por lo que lo implementaremos en la clase `ANPulses.cc`.

A continuación mostramos el gráfico asociado a la salida de (Simulación 11):

```

1 Pretty_EMsimple_em1(120,1000,2.5,1.0,"Dif",1,1,12,1,10,0.5)

```

con la distribución semilla gaussiana sin normalizar y con la traslación implementada:

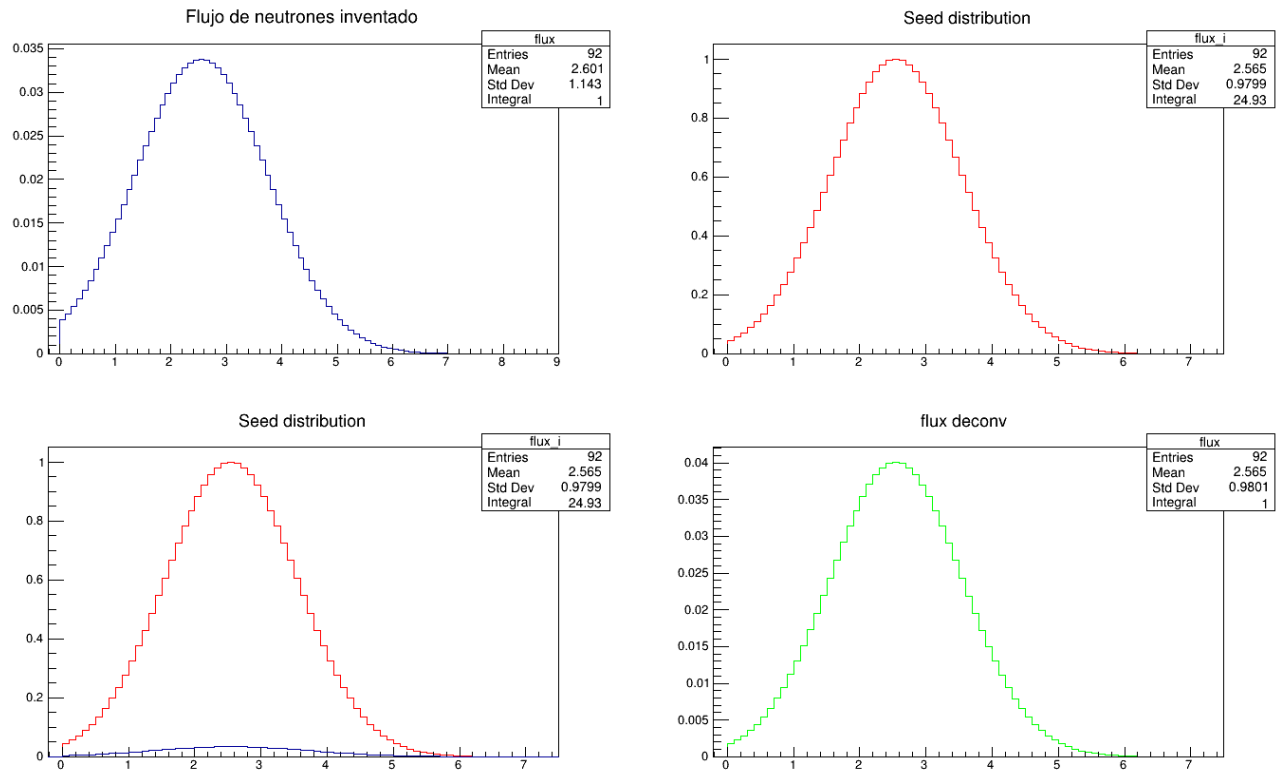


Figure 29.3: Salida de la deconvolución em1 antes de normalizar la distribución semilla.

Ahora mostramos la salida del mismo comando anterior, pero con la semilla gaussiana normalizada.

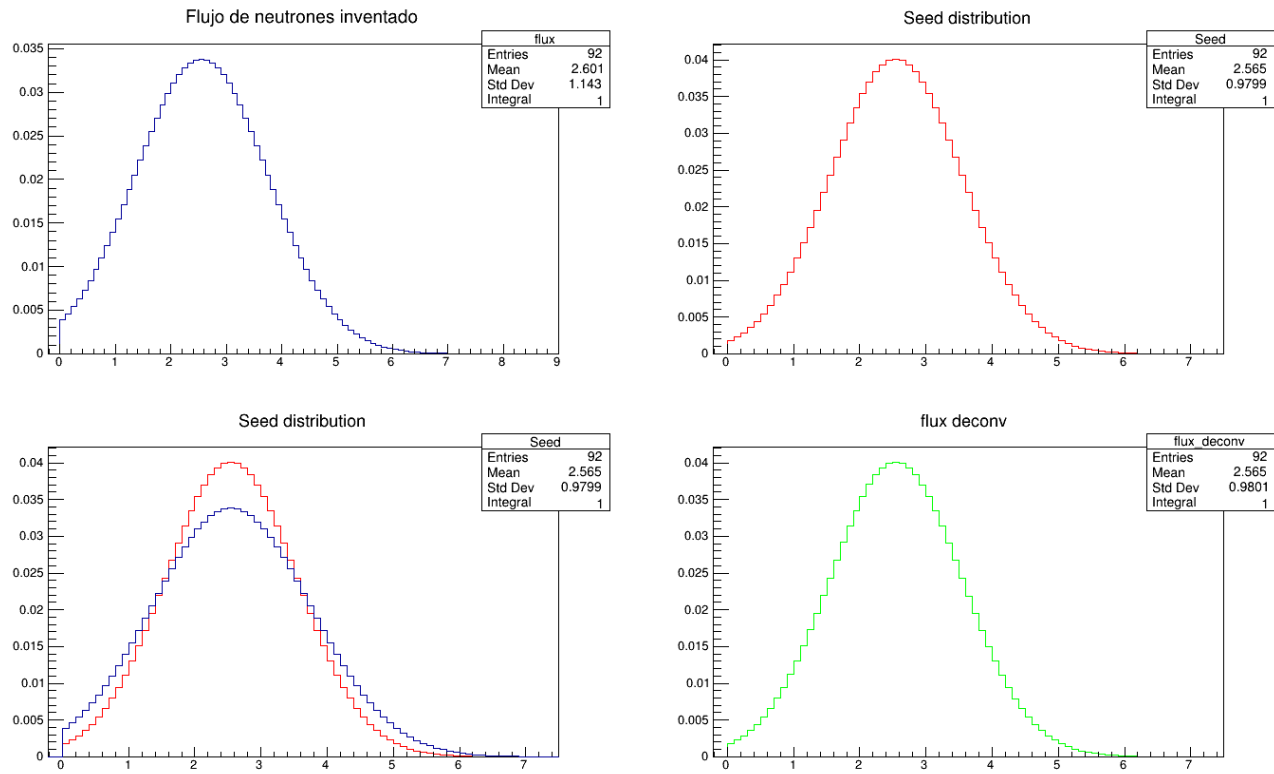


Figure 29.4: Salida de la deconvolución em1 después de normalizar la distribución semilla.

Veamos como funciona esto para las distribuciones skew normal, de la salida del comando (Simulación 31):

```
1 Pretty_EMsimple_em1(120,1000,2.5,1.0,"Dif",1,1,12,3,10,0.5)
```

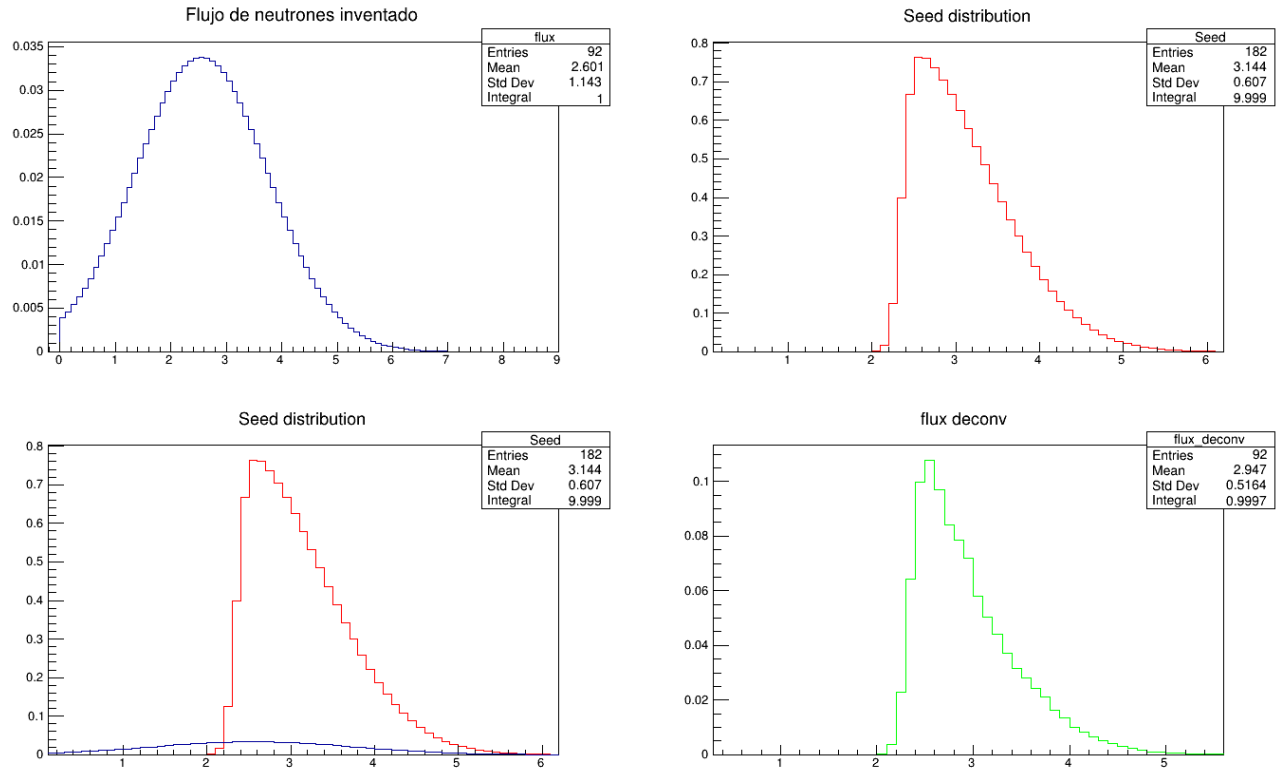


Figure 29.5: Salida de la deconvolución em1 para una semilla skew normal.

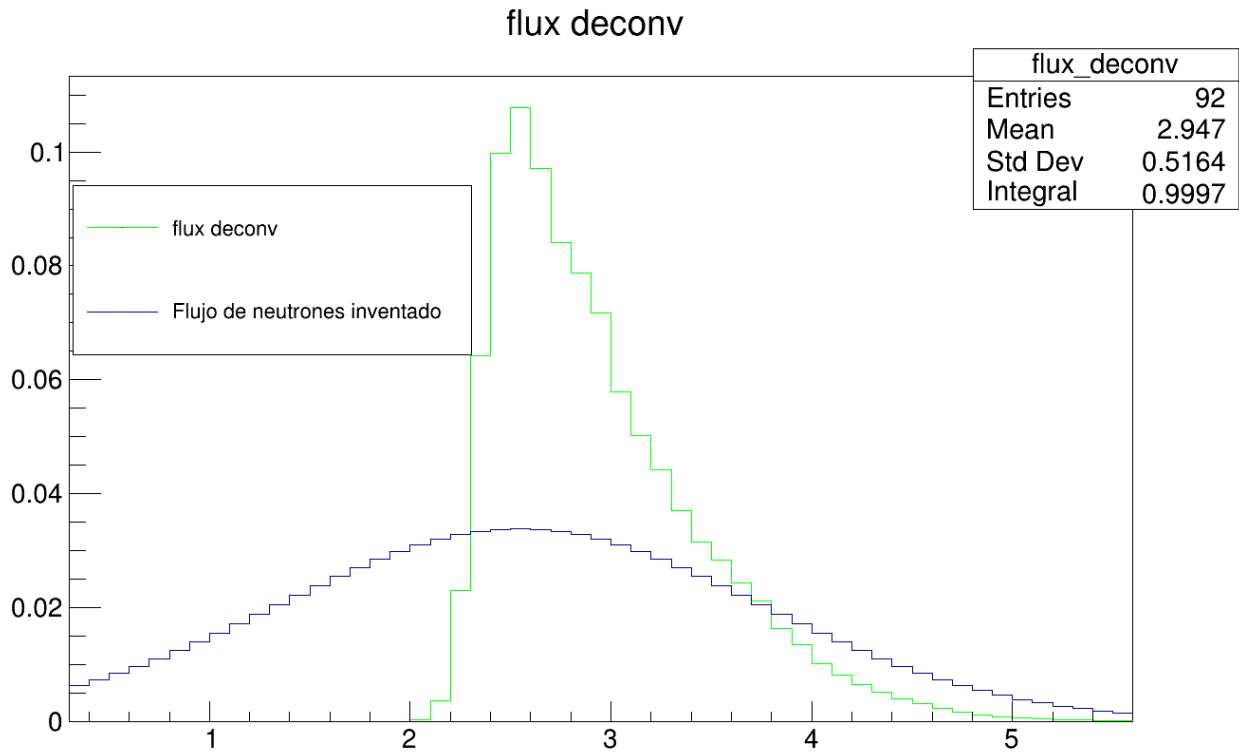


Figure 29.6: Comparación de flujo deconvolucionado para una semilla skew normal y flujo inventado tipo gaussiano

De la Fig 29.5 pad 12 vemos que hay un problema con la normalización, ya que debería ser 1 en vez de 9.999. Para el caso anterior Fig 29.4 pad 12, vemos que la normalización funciona. Esto nos sugiere que el problema puede venir de la traslación del flujo skew normal, es especial del SetBin, ya que solo para este tipo de distribuciones hacemos esta traslación.

De todas maneras realizamos una simulación tipo 33, obteniendo el siguiente resultado:

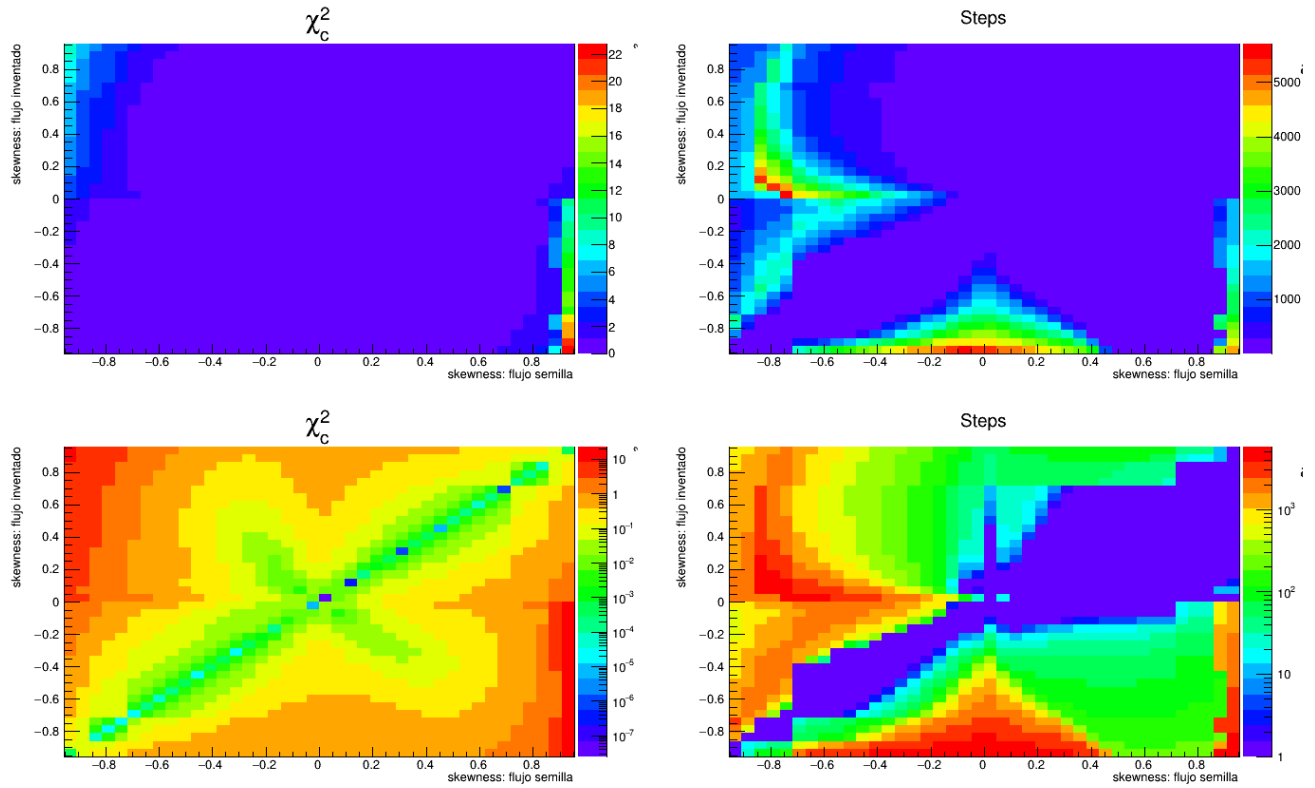


Figure 29.7: Simulación 33

Gráfico Simulación 31:

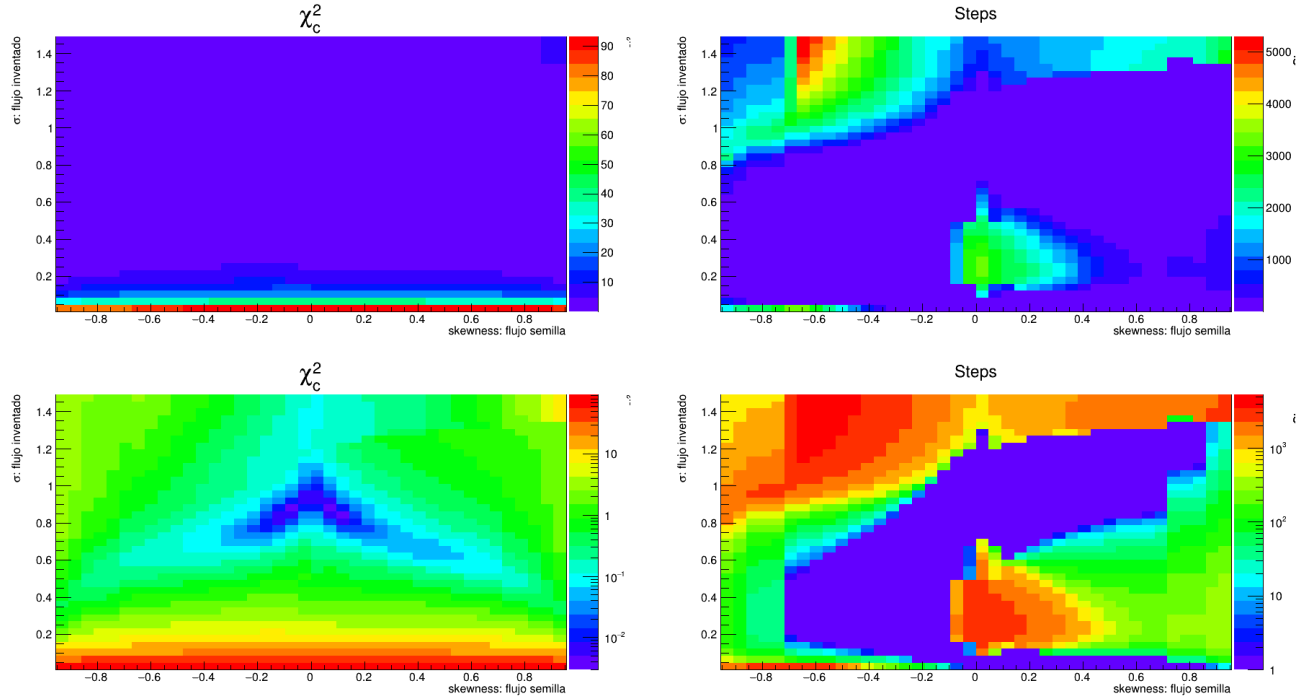


Figure 29.8: Simulación 31 - Semillas skew normal y distribuciones inventadas gaussianas

flopez: 2022-03-16: Generar un gráfico de valor de integrales tanto para la simulación 31 y 33, donde en los ejes horizontal y vertical estén los valores correspondientes de la integrales y en el eje z el valor de χ^2_c .

flopez: 2022-03-16: Generar mapas de calor, pero variando el centroide del flujo inventado (fijando el ancho) e ir probando con distintos valores de semillas.

Para generar una simulación 13, es decir, semillas gaussianas y distribuciones inventadas skew normal, debemos trasladar todas las funciones skew normal al centroide 2.5.

Necesitamos implementar que la función `Pretty_EMsimple_em1()` seleccione de forma automática el archivo con los neutrones generados por distribuciones gaussianas o skew normal, esto para no tener que cambiar a mano en la misma función.

A Resources

It is a good idea to record sources that explain concepts or provide tools so the research is both better documented and if someone has to continue with it, there is enough supporting documentation.

B TO DO

Here you will have all your TODOs grouped with anchor links to the parts of the document where they are. Really handy if you do not know where to continue with your project.

Todo list

■ flopez: 2021-11-25: Hacer pruebas con neutrones calculados a partir de flujos inventados no gaussianos, usando como flujo semilla de la deconvolución una distribución gaussiana.	17
■ flopez: 2021-12-03 Implementar parámetros diferenciados para la semilla y para el flujo inventado en Pretty EMsimple em1(), tanto para el tipo de distribución como de valor de anchos. Esto para hacer la iteraciones más fáciles de programar.	22
■ flopez: 2021-12-06: Hacer pruebas variando los flujos inventados y los flujos semilla. HECHO, ver entrada (14)	23
■ flopez: 2021-12-07: Imprimir en pantalla o en un archivo todos los valores que intervienen en el método EM, ya que al monitorear el Break segmentation violation del algoritmo se ve que hay valores casi iguales a cero. HECHO, ver entrada (13)	25
■ flopez: 2021-12-14: Generar una clase que inicialice todas las variables que necesito para el algoritmo EM, y así me evito el problema de que al iterar sobre Pretty EMsimple em1() se produzca un segmentaion fault debido a que no se pueden abrir muchas veces seguidas un archivo .root, como es el caso de las eficiencias intrínsecas.	33
■ flopez: 2021-12-14: Establecer un balance entre el número de pasos del método EM y la deformación del flujo deconvolucionado. FMolina sugirió ocupar como parámetro de stop del algoritmo, el producto de χ^2 (del algoritmo 2) y χ_c^2	33
■ flopez: 2021-12-20: Arreglar el orden de los parámaretros por defecto de la clase ANPulses, ya que estos tienen que estar en la declaración (clase.h) y no en la implementación (clase.cc) . . .	34
■ flopez: 2022-03-16: Generar un gráfico de valor de integrales tanto para la simulación 31 y 33, donde en los ejes horizontal y vertical estén los valores correspondientes de la integrales y en el eje z el valor de χ_c^2	75
■ flopez: 2022-03-16: Generar mapas de calor, pero variando el centroide del flujo inventado (fijando el ancho) e ir probando con distintos valores de semillas.	75

C Conceptos

C.1 Sistemas de adquisición de datos (DAQ) C.2 Física Nuclear

1. **Dead time:** Para un sistema de detección, es el tiempo mínimo de separación de dos eventos para que sean medidos como dos pulsos independientes. En algunos casos las limitaciones para este tiempo mínimo está determinada por los procesos en el detector mismo y en otros casos este límite está asociado a la electrónica del sistema de detección.
2. **Gain**
3. **Rango dinámico (de una señal)**
4. **Sampleo** La tarjeta digitalizadora samplea a 62.5 MHz (max 125 MHz), lo cual sirve para guardar el EFIR, obteniendo así una buena resolución en el espectro de altura de pulso, pero para guardar samples (pulsos) al samplear con esta frecuencia estaríamos sobre sampleando la señal, lo cuál se traduce en una mayor información digitalizada por sample, es decir, a un mayor tamaño del archivo .dlt. Es por esta razón, que para el experimento de neutrones pulsados se utilizó un clock externo que sampleaba la señal a 25 MHz.
5. **Ancho de ventana de adquisición de samples** Para la tarjeta digitalizadora Struck SIS3316-125-16, se tiene que cada bin corresponde a un ancho de tiempo de 40 ns, de esta manera, para un signal length (en GASIFIC) de $2^{16} - 1 = 65535$ se tiene que la máxima ventana posible de adquisición de una señal (sample, para disparos) corresponde a $40 \text{ ns} \times 2^{16} - 1 \approx 2.6 \text{ ms}$. Luego para movies (samples de neutrones de fondo) no necesitamos una ventana tan ancha de adquisición, ya que nos importa más la subida (para hacer el análisis de rise time usando TDC), de esta manera, para un signal length de 5000 en GASIFIC, tenemos que la ventana de adquisición corresponde a $40 \text{ ns} \times 5000 \approx 0.2 \text{ ms}$ ($200\mu\text{s}$), la cuál es suficiente para capturar la forma de la subida de la señal.

1. **Material Fisible:** ^{235}U
2. **Material Fisionable** ^{238}U
3. **Eficiencia Intrínseca de detección (ε_{int}):** Corresponde a la probabilidad de detectar una partícula en un detector. No depende de la distancia a la fuente.
4. **Eficiencia Absoluta de detección (ε_{abs}):** Corresponde a la probabilidad de detectar una partícula emitida desde la fuente. Depende de la distancia fuente-detector.

$$\varepsilon_{abs} = \varepsilon_{int} \frac{\Omega}{4\pi} \quad (\text{C.1})$$

5. **Neutron Recoil**
6. **Flujo diferencial:**

$$\frac{d\Phi}{dE} \longrightarrow \text{Flux}[i] = \text{Seed}[i]/dE[i] \quad (\text{C.2})$$

7. **Flujo integral:**

$$\frac{d\Phi}{dE} \cdot dE \rightarrow \text{Flux}[i] = (\text{Seed}[i] / dE[i]) * dE[i] \quad (\text{C.3})$$

C.3 Física de Plasmas

8. **Drive parameter:** Este parámetro se define como:

$$\frac{I}{a\sqrt{\rho}} \quad (\text{C.4})$$

donde I es la corriente, a es el radio del ánodo, y ρ es la densidad de gas o presión.

Es proporcional a la velocidad del plasma tanto en su fase axial como radial. Es uno de los parámetros más importantes que determina el rendimiento de un PF como fuente de neutrones de fusión. Su valor constante en una amplia gama de PF's indica realmente que todos estos

dispositivos funcionan a las mismas velocidades axial y radial y, por lo tanto, por inferencia, todos tienen las mismas temperaturas en las fases axial y radial [3] .

9. **Pinch lifetime** (t_{pf}): Definido como el tiempo desde la primera compresión a un radio mínimo hasta el momento de la ruptura violenta de la columna de plasma. Con valores entre decenas de ns a 100-200 ns [4]. De acuerdo a CP, el t_{pf}

del PF400J es de 3 a 4 ns.

C.4 Estadística

10. **Skewness**: Es el tercer momento estándar de una distribución de probabilidad y da cuenta de la asimetría de esta. El grado de asimetría de una distribución se denomina sesgo hacia la derecha o hacia la izquierda

D References

Do not forget to cite the papers that you are using in your research, this way the **Previous Work** part in your paper will be infinitely easier to write when the time comes.

References

- [1] D. Thomas and A. Alevra, “Bonner sphere spectrometers—a critical review,” Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment, vol. 476, pp. 12–20, 1 2002.
- [2] A. Tarifeño-Saldivia, R. E. Mayer, C. Pavez, and L. Soto, “Calibration methodology for proportional counters applied to yield measurements of a neutron burst,” Review of Scientific Instruments, vol. 85, 1 2014.
- [3] T. Zhang, R. Rawat, S. Springham, V. Gribkov, T. Tan, J. Lin, S. Hassan, S. Mahmood, P. Lee, and S. Lee, “Drive parameter as a design consideration for mather and filippov types of plasma focus,” pp. 288–288, IEEE, 2006.
- [4] S. Lee and A. Serban, “Dimensions and lifetime of the plasma focus pinch,” IEEE Transactions on Plasma Science, vol. 24, pp. 1101–1105, 6 1996.