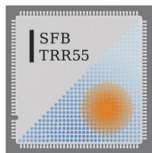


Data analysis exercises

Christian Hoelbling
Bergische Universität Wuppertal

New Horizons in Lattice Field Theory
Mar. 18-19, 2013, IIP Natal



please copy the directory
octave_routines
please start octave



BERGISCHE
UNIVERSITÄT
WUPPERTAL

Octave basics

Octave is an interpreter:

```
octave:1> a=1
```

```
a = 1
```

Don't print results:

```
octave:1> a=1;
```

Print an expression:

```
octave:1> a
```

```
a = 1
```

Octave arrays

Octave can handle arrays well:

```
octave:1> a=[1,2,3,4,5]
```

```
a =
```

```
    1    2    3    4    5
```

Simpler:

```
octave:1> a=1:5
```

```
a =
```

```
    1    2    3    4    5
```

Also with step sizes:

```
octave:1> a=5:-0.5:3
```

```
a =
```

```
    5.0000    4.5000    4.0000    3.5000    3.0000
```

Octave treats objects as matrices:

```
octave:1> a=[1,1;0,2]
```

```
a =
```

```
1    1  
0    2
```

Arithmetic operations are natively matrix:

```
octave:1> a^2
```

```
ans =
```

```
1    3  
0    4
```

If you want it element-wise:

```
octave:1> a.^2
```

```
ans =
```

```
1    1  
0    4
```

Octave vectors

Octave distinguishes between row vectors

```
octave:1> a=[1,2]
```

```
a =
```

```
1    2
```

and column vectors

```
octave:1> b=[1;2]
```

```
b =
```

```
1
```

```
2
```

Transposition is simple:

```
octave:1> b'
```

```
ans =
```

```
1    2
```

Octave matrix multiply

If we define a row vector

```
octave:1> a=[1,2]
```

```
a =
```

```
1    2
```

there is of course a difference between

```
octave:1> a*a'
```

```
ans = 5
```

and

```
octave:1> a'*a
```

```
ans =
```

```
1    2
```

```
2    4
```

Octave eigenvalues

Let us define a symmetric matrix:

```
octave:1> m=[2,1;1,4]
```

```
m =
```

```
    2    1  
    1    4
```

We can find the eigenvalues:

```
octave:1> eig(m)
```

```
ans =
```

```
    1.5858  
    4.4142
```

Octave example

To find the eigenvectors as well, use:

```
octave:1> [vec, val]=eig(m)
```

```
vec =
```

```
   -0.92388    0.38268
```

```
    0.38268    0.92388
```

```
val =
```

```
Diagonal Matrix
```

```
   1.5858         0
```

```
         0   4.4142
```

So we can reconstruct the original matrix:

```
octave:1> vec*val*vec'
```

```
ans =
```

```
   2.00000   1.00000
```

```
   1.00000   4.00000
```

Octave matrix functions

Initializing matrix (zeros, rand, diag) example:

```
octave:1> m=rand(2)
m =
    0.089507    0.047387
    0.718905    0.878561
```

Sum (and prod) reduces one dimension:

```
octave:1> sum(m)
ans =
    0.80841    0.92595
octave:2> sum(m, 2)
ans =
    0.13689
    1.59747
```

Octave cell arrays

One last detail: cell arrays

```
octave:1> a={1,"string"}  
a =  
{  
    [1,1] = 1  
    [1,2] = string  
}
```

They may contain different objects and are accessed as:

```
octave:1> a{1}  
ans = 1  
octave:2> a{2}  
ans = string
```

Octave cell arrays

Cell arrays can easily be enlarged:

```
octave:1> a=[a [1,2]]
```

```
a =
```

```
{  
    [1,1] = 1  
    [1,2] = string  
    [1,3] =  
        1    2  
}
```

```
octave:2>a{3}
```

```
ans =
```

```
1    2
```

Let's start!

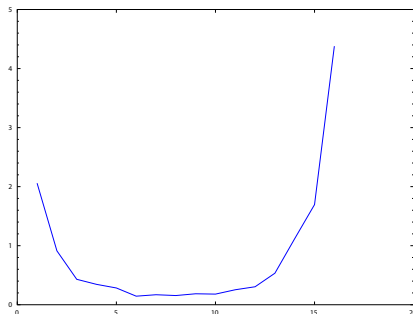
Please open the file “readme.m”

Produce one fake “pion” propagator:

```
octave:1> myprop1=piprop(16)
```

and plot it:

```
octave:1> plot(myprop1{1})
```



directory:

octave_routines

Displaying data

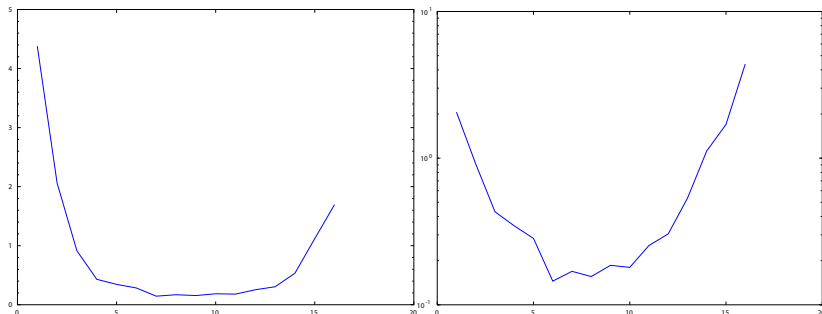
Please don't look at "piprop.m" or any "[..]prop.m" file yet!

If you like the origin at 0 instead of 16:

```
octave:1> plot(myprop1{1}([end 1:end-1]))
```

For a logarithmic y-scale plot:

```
octave:1> semilogy(myprop1{1})
```



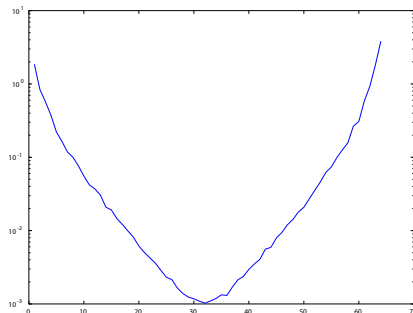
The first ensemble

Now 100 configs with $N_T = 64$:

```
octave:1> prop=piprop(64,100);
```

For a logarithmic y-scale plot:

```
octave:1> semilogy(prop{1})
```



Bootstrapping

Produce 200 bootstrap samples:

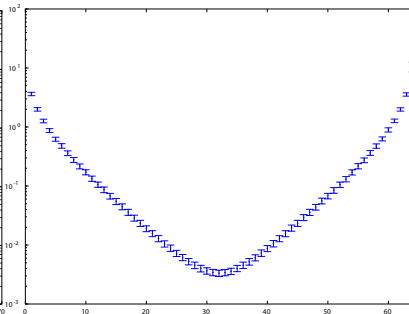
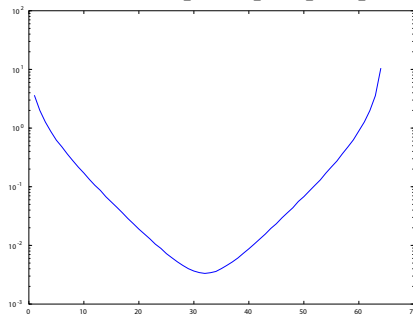
```
octave:1> bp=boot(prop,200);
```

Plot the average (column 201):

```
octave:1> semilogy(bp(:,end))
```

Routine for plotting with errors:

```
octave:1> plotprop(bp)
```



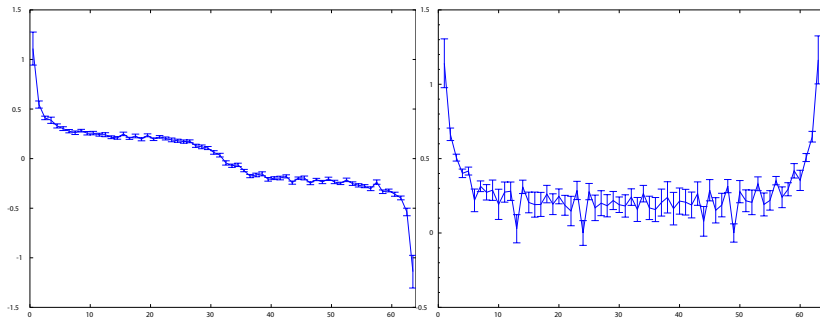
Effective mass

Effective mass with simple exp:

```
octave:1> plotplat(bp)
```

Effective mass with cosh:

```
octave:1> plotsymplat(bp)
```



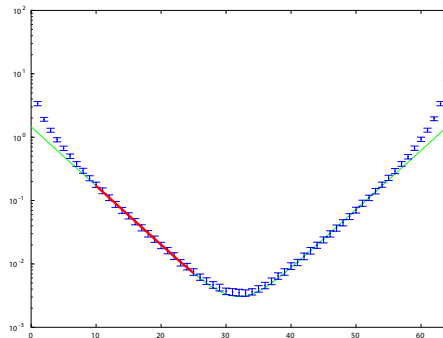
Mass fit

```
uncorrelated_fit=0  
fully_correlated_fit=-1
```

```
central_value_only=1  
full_bootstrap_fit=0
```

```
do_plot=1  
dont_plot=0
```

```
massfit(bp,uncorrelated_fit,10,25,  
        central_value_only,do_plot)
```



Multiple output variables

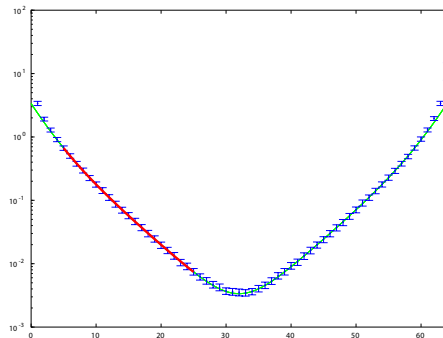
```
octave:1> [m,em,c,ec,fitq]=massfit(bp,0,10,25)
m = 0.21448
em = 0.0024118
c = 1.4575
ec = 0.13362
fitq = 0.99984
```

Excited state fit

You might want to try a fit with 2 coshs:

```
octave:1> [m,em,c,ec,fitq,m2,em2,c2,ec2]=  
           exsfit(bp,0,5,25,0,1)
```

```
m = 0.20403  
em = 0.0028877  
c = 1.1466  
ec = 0.15216  
fitq = 1.00000  
m2 = 0.43349  
em2 = 0.026877  
c2 = 2.2185  
ec2 = 0.28166
```



Find the mass!

- Explore the fitting routine
- Try varying nconf, nboot, NT, fit range, correlation
- Find the “pion” mass and its error
- Give your results to me (with error!)
- If you are done, you can try “rhoprop” instead of “piprop”

have fun!

email me your results until tomorrow, noon:
hch@physik.uni-wuppertal.de

Going to the physical point

Please open the file “day2.m”

generate one well behaved ensemble at “bare quark mass” m_q :

```
NT=64
```

```
NBOOT=100
```

```
NCONF=100
```

```
 $m_q = -0.01$ 
```

```
octave:1> prp=easy_piprop(NT, $m_q$ ,NCONF);
```

Bootstrap it and look at the pion mass:

```
octave:1> bpr=boot(prp,NBOOT);
```

```
octave:2> m=massfit(bpr,0,15,30,1,1)
```

Only central value computed!

```
m = 0.57282
```

The physical point

We are at $1/a = 1.5$ GeV, so the target “physical” $m_\pi a = 0.09$

Produce ensembles that allow you to go to the physical point

- You can either interpolate or extrapolate
- Near the “physical point” propagators are more noisy

Generating ensembles

Template ensemble generation:

```
# a random table of quark masses
masstab=[0,-0.04,-0.06,-0.08]

for i=1:length(masstab)
    prp=easy_piprop (NT,masstab(i),NCONF);
    bp{i}=boot (prp,NBOOT);
# mb and cb are bootstrap results for m and c
    [m,em,c,ec,q,mb{i},cb{i}]=massfit (bp{i},0,15,30);
# extract f from c=f^2*m^2
    fb{i}=sqrt (abs (2*cb{i} ./mb{i}));
end
```

Going to the physical point

Please open the file “chiralfit.m”

This is a long fit routine with a very simple core:

```
stp=[d0,k0];  
# do the constrained fit  
[par,ch2c,info]=sqp(stp',@constrainedfit,  
                    [],[],[],[],1000);
```

A χ^2 function is minimized, x-errors are yet ignored

Function to minimize

```
function chisq=constrainedfit(par)
# chi^2 function for constrained fpi-mpi^2 fit
  global _f _m _er _n

  chisq=0;
  for i=1:_n
    mm=_m(i);
    ff=fps(mm,par);
    ex=[_m(i)-mm,_f(i)-ff];
    chisq+=ex*_er{i}*ex';
  end
  chisq;
endfunction
```

Full function

```
function chisq=fullfit(par)
# chi^2 function for full fpi-mpi^2 fit
  global _f _m _er _n

  npar=2;
  chisq=0;
  for i=1:_n
    mm=par(npar+i);
    ff=fps(mm,par(1:npar));
    ex=[_m(i)-mm,_f(i)-ff];
    chisq+=ex*_er{i}*ex';
  end
  chisq;
endfunction
```

Fit function

x-values (`_m`), y-values (`_f`) and the inverse covariance matrix (`_er`) are global:

```
global _f _m _er
```

```
octave:1> global _f
```

```
octave:2> _f
```

```
_f =
```

```
    0.068717    0.072484    0.075524    0.073040
```

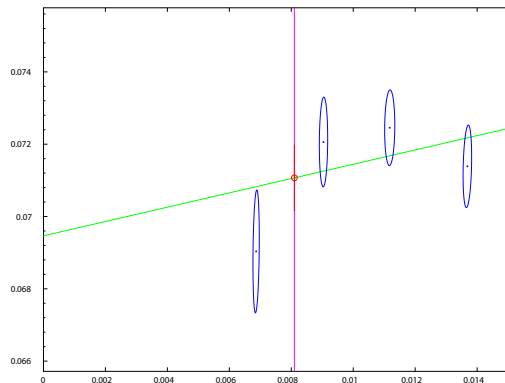
The actual linear fit function:

```
function ff=fps(m,c)
# the actual fit function
    ff=c(1)+c(2)*m;
endfunction
```

Doing the fit

Using the chiral fit:

```
> [fpi,efpi,q,qc]=chiralfit(fb,mb,-1,0,1)
fpi = 0.071069
efpi = 9.1972e-04
q = 0.51122
qc = 0.49685
```



Different fit forms

Setting the x-axis as m^2 (line 33):

```
# use mpi^2 as x-axis  
mb{i}=mb{i}.^2;
```

Vary the 2-parameter fit form (end of file):

```
function ff=fps(m,c)  
# the actual fit function  
ff=c(1)+c(2)*m;  
endfunction
```

Do a 3-parameter fit:

```
[fpi,efpi,q,qc]=chiralfit_quadratic(fb,mb,-1,1,1)
```

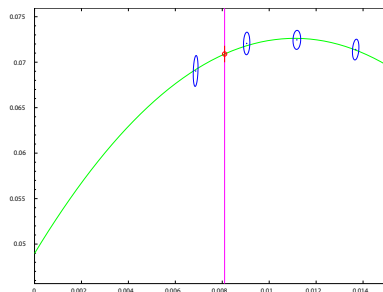
Quadratic fit

Using the chiral fit:

```
> [fpi,efpi,q,qc]=chiralfit_quadratic(fb,mb,-1,0,1)
fpi = 0.070900
efpi = 9.0830e-04
q = 0.97827
qc = 0.97816
```

Vary the 3-parameter fit form (end of file):

```
function ff=fps(m,c)
# the actual fit function
    ff=c(1)+c(2)*m+c(3)*m.^2;
endfunction
```



Find f at the physical point!

- Explore the fitting routines - change fit forms
- Try varying ensembles and fit parameters
- Find the “decay constant” f and its error
- If you are done, try estimating the systematics
- Give your results to me (including full error estimate!)
- If you want the solution, look at “cheat.m” and “physpoint.m”

have fun!

if you have not looked at cheats, email me your results until tomorrow, noon: hch@physik.uni-wuppertal.de