cboot Introduction

Tomoki Ohtsuki

April 10, 2016

Contents

1 The basic functionality and several important classes

1.1 Context objects

The first thing you do in `cboot` is to create a *context object*, which stores several truncation parameters and parameter-dependent constants and methods. They include e.g.

- derivative cutoff Λ: it appears e.g. in the truncation of the Ising model bootstrap study as

$$\begin{aligned}&\text{find } \{a_{m,n}\} \text{ with}\\ &\sum_{\substack{m,n\geq 0\\ m+n\leq\Lambda}} a_{m,n}\partial_x^m\partial_y^n F^{(-)}_{\Delta,l}(\Delta_\phi;x,y)\geq 0 \text{ for } \Delta\geq\Delta_{\min}.\end{aligned}$$

  One of the motivations for introducing context object is that it is quite easy encounter a bug once you input inconsistent value of Λ.

- Working numerical precision
- Max expansion order in ρ-series
- Matrix representing coordinate transform $\rho \to x$
- Matrix representing coordinate transform $(z, \bar{z}) \to (x, y)$
- A method to create a matrix which represents convolution with $v^\delta = \{(1-z)(1-\bar{z})\}^\delta$, accompanied by (anti-)symmetrization w.r.t. $z \leftrightarrow (1-z)$.

The most basic (minimal) context object is called `cb_universal_context` and is implemented in the file `context_object.pyx`. However, you will rarely use this class directly, as it has no method to create conformal blocks[1]. The conformal block creation methods are attributes of its descendant classes, introduced in the next subsection.

1.2 Conformal block approximation and `prefactor_numerator` class

Context objects with conformal block computation methods can be created by `context_for_scalar` function:

```
import sage.cboot as cb
context_3d=cb.context_for_scalar(Lambda=13,epsilon=0.5)
```

The parameter argument `Lambda` refers to the derivative cutoff Λ and the `epsilon` parameter specifies the space-time dimensionality, $\epsilon = \frac{d-2}{2}$, so here we constructed a context for $d = 3$ bootstrap study. The expansion order in ρ-series (default:250) and working binary precisions (default:800) too are keyword arguments.

The returned value `context_3d` is an instance of `scalar_cb_context_generic`, which inherits from `cb_universal_context` and is equipped with the method to calculate the conformal blocks (up to given derivative cutoff).

`context_for_scalar` returns different (inherited) classes when called with $\epsilon = 0$ and $\epsilon = 1$, i.e., $d = 2$ and $d = 4$: we prepare different classes in these cases,as the rational approximation of the conformal block is special in the sense that it contains double poles. To avoid this complication, the exact expression by Dolan and Osborn is implemented. The names of methods and attributes relevant for the bootstrap study is common, so code valid In practice, you don't have to care these slight difference

Now this context has the method to compute conformal block. For constant value of Δ, use `gBlock` method

```
import sage.cboot as cb
context_3d=cb.context_for_scalar(Lambda=13,epsilon=0.5)
g_em=context_3d.gBlock(2,3,0,0)
```

The first argument refers to spin and second to operator dimension. The third and fourth arguments, which are both 0 now, specify the dimension difference of external operator Δ_{12} and Δ_{34}. The resulting object `g_em` is a numpy array containing the conformal block and its (x, y)-derivatives as multi-precision numbers.

To approximate the conformal block, use `approx_cb` method:

```
g_approx=context_3d.approx_cb(13,2)
```

The first argument (13 here) is the cutoff $\nu_{\max}$ (defined in [1]) for the number of poles included in the approximation, and the second one (2 here) is the spin of approximated block. In this function, Δ_{12} and Δ_{34} are keyword arguments whose default values are 0.

`g_approx` is an instance of class `prefactor_numerator`, which wraps the information of poles (it is another class, `damped_rational`) and array of polynomials.

Let us see `g_approx` correctly approximates the conformal block: type

```
print float(max(abs(g_approx(3)-g_em)))
```

The argument of `g_approx` (which is `3` here) specifies the value of Δ. You can also verify that increasing $\nu_{\max}$ will lead to a better approximation:

```
g_approx=context_3d.approx_cb(20,2)
print float(max(abs(g_approx(3)-g_em)))
```

There are other methods useful for the conformal bootstrap studies, but they can be better explained in a tutorial fashion.

[1]The reason for the separation is just technical – I wanted to save re-compile time by separating functionality.

2 How actual codes proceed

2.1 3d Bosonic example

Let us play with the simplest examples contained in `examples/3dBosonic.py`.

2.1.1 The Ising model binary search

We have already explained what the first two lines of the code mean:

```
import sage.cboot as cb
context=cb.context_for_scalar(epsilon=0.5,Lambda=13)
```

In the following lines,

```
lmax=25
nu_max=12
cbs={}
for spin in range(0,lmax,2):
    g=context.approx_cb(nu_max,spin)
    cbs.update({spin:g})
```

we are storing rational approximation of conformal blocks in the dictionary `cbs`.

Then, let us proceed to the definition of `make_F`. This is a function to create the convolved conformal block with given spin and extremal operator dimension δ:

```
def make_F(delta,spin,gap_dict):
    mat_F=context.F_minus_matrix(delta)
    try:
        gap=context(gap_dict[spin])
    except KeyError:
        if spin==0:
            gap=context.epsilon
        else:
            gap=2*context.epsilon+spin
    g_shift=cbs[spin].shift(gap)
    F=context.dot(mat_F,g_shift)
    return F
```

In the definition of `mat_F`

```
    mat_F=context.F_minus_matrix(delta)
```

the method `F_minus_matrix` of the context object creates the matrix representing the convolution with $\{(1-z)(1-\bar{z})\}^{\delta}$ together with anti-symmetrization in $z \leftrightarrow (1-z)$.

The next several lines are to shift Δ by `gap`. To determine the amount of shift (`gap`), the code first search the input dictionary `gap_dict`:

```
    try:
        gap=context(gap_dict[spin])
```

So if your input `gap_dict` has `spin` input as a key, `gap_dict[spin]` will be your preferred value, and the action of context (i.e., the `__call__` method of context object) will convert it to a multi-precision number. When there is no such key, `gap` will be set to the unitarity bound.

```
    except KeyError:
        if spin==0:
            gap=context.epsilon
        else:
            gap=2*context.epsilon+spin
```

Then, the `shift` method of `prefactor_numerator` will shift both the numerator polynomials and prefactors of `cbs[spin]`, and the `context.dot` operation performs the matrix product with `mat_F` and `g_shift`, returning the result.

```
    g_shift=cbs[spin].shift(gap)
    F=context.dot(mat_F,g_shift)
    return F
```

The next function `make_SDP` takes its arguments as external operator dimension `delat` and artificial gap specification dictionary `gap_dict` (which is directly passed to `make_F` function defined above).

```
def make_SDP(delta,gap_dict):
    delta=context(delta)
    Fs=[make_F(delta,spin,gap_dict) for spin in cbs.keys()]
    mat_F=context.F_minus_matrix(delta)
    norm=context.dot(mat_F,context.gBlock(0,0,0,0))
    obj=norm*0
    return context.SDP(norm,obj,Fs)
```

Here, `Fs` is the list containing approximated $F^{(-)}$ with desired shift. `norm` is used for the normalization of linear functional, which we take here to be the identity vector conformal block(hence all the arguments are 0). As long as you are interested in binary search of the dimension bound, the optimization target `obj` can be null. `context.SDP` will combine these data into `SDP` class.

Then binary search function is simple, but you have to set `sdpb` to be appropriate path to `SDPB`.

```
from subprocess import Popen, PIPE
import re

sdpb="sdpb"
sdpbparams=["--findPrimalFeasible","--findDualFeasible","--noFinalCheckpoint"]
def bs(delta,upper=3,lower=1,sdp_method=make_SDP):
    upper=context(upper)
    lower=context(lower)
    while upper - lower > 0.001:
        D_try=(upper+lower)/2
        prob=sdp_method(delta,{0:D_try})
        prob.write("3d_Ising_binary.xml")
        sdpbargs=[sdpb,"-s","3d_Ising_binary.xml"]+sdpbparams
        out, err=Popen(sdpbargs,stdout=PIPE,stderr=PIPE).communicate()
        sol=re.compile(r'found ([^ ]+) feasible').search(out).groups()[0]
        if sol=="dual":
            print("(Delta_phi, Delta_epsilon)={0} is excluded."\
            .format((float(delta),float(D_try))))
            upper=D_try
        elif sol=="primal":
            print("(Delta_phi, Delta_epsilon)={0} is permitted."\
            .format((float(delta),float(D_try))))
            lower=D_try
        else:
            raise RuntimeError("Unexpected return from sdpb")
    return upper
```

Now everything should look familiar: if you tried with some `D_try` and obtained and obtained "found dual feasible", `upper` would be replaced with `D_try`, and if you instead obtained "found primal feasible", `lower` would be replaced.

The default result of the execution

```
sage 3dBosonic.py
```

will perform the binary search with `delta`=0.518, the result of which should be ~ 1.413.

2.1.2 Central charge lower bound

It is easy to derive the central charge lower bound as well, modifying `make_SDP` function:

```
def make_SDP_for_cc(delta,gap_dict={0:1}):
    delta=context(delta)
    Fs=[make_F(delta,spin,gap_dict) for spin in cbs.keys()]
    mat_F=context.F_minus_matrix(delta)
    norm=context.dot(mat_F,context.gBlock(2,3,0,0))
    obj=context.dot(mat_F,context.gBlock(0,0,0,0))
    return context.SDP(norm,obj,Fs)
```

The difference from `make_SDP` is that `norm` is taken to be the conformal block coming from the energy-momentum tensor and `obj` is that from the identity.

Since there is no need to perform binary search, SDPB-execution function is simpler than `bs`:

```
def cc(delta):
    prob=make_SDP_for_cc(delta)
    prob.write("3d_Ising_cc.xml")
    sdpbargs=[sdpb,"-s","3d_Ising_cc.xml","--noFinalCheckpoint"]
    out, err=Popen(sdpbargs,stdout=PIPE,stderr=PIPE).communicate()
    sol=re.compile(r'primalObjective *= *([^ ]+) *$',re.MULTILINE)\
            .search(out).groups()[0]
    return -delta**2/float(sol)
```

Note that there are neither `--findDualFeasible` nor `--findPrimalFeasible`, as they will exit the search before the optimal solution turns out.

To perform central charge search, uncomment the corresponding lines in `3dBosonic.py`.

2.1.3 Bound on ϵ'

As a simple exercise, let us derive a bound on ϵ', i.e., the second-lowest operator neutral under any global symmetries, as a function of Δ_ϵ (ϵ is of course the lowest such operator). To do this, we should study $\langle\epsilon\epsilon\epsilon\epsilon\rangle$ as well, but a twist to $\mathbb{Z}_2$-odd four point function (like that of 3d Ising) is that $\epsilon\times\epsilon$ can contain ϵ itself.

To account for this, we have to modify the `make_SDP` function a bit:

```
def make_SDP_epsilon_prime(delta,gap_dict):
    delta=context(delta)
    Fs=[make_F(delta,spin,gap_dict) for spin in cbs.keys()]
    mat_F=context.F_minus_matrix(delta)
    Fs+=[context.dot(mat_F,context.gBlock(0,delta,0,0))]
    norm=context.dot(mat_F,context.gBlock(0,0,0,0))
    obj=norm*0
    return context.SDP(norm,obj,Fs)
```

As you see, the only difference is the addition

```
Fs+=[context.dot(mat_F,context.gBlock(0,delta,0,0))]
```

which represents the positivity requirement on the conformal block coming from ϵ itself. Uncommenting the last two lines of `3dBosonic.py`, you will know that $\Delta_{\epsilon'}$ must be smaller than 2.4 [2]for $\Delta_\epsilon = 0.8$, so for this value of Δ_ϵ, we must have at least two relevant operators singlet under any global symmetries.

[2]In general, we expect this bound to be weaker than the bound without the requirement of the function on ϵ-contribution (i.e., $\mathbb{Z}_2$-odd example presented above). However, these result somehow coincide. The same sort of coincidence takes place in other dimensions like $d = 5$.

2.2 $O(n)$-example

Then we let us look around how $O(n)$ model bootstrap in the script `3dOn.py` works.

2.2.1 Bounds for S-sector and T-sector

Again the code `3don.py` starts with imports and computation of conformal blocks. Note that this time conformal blocks with odd spins are also stored.

```
import sage.cboot as cb
import numpy as np
from subprocess import Popen, PIPE
import re

context=cb.context_for_scalar(epsilon=0.5,Lambda=13)
lmax=25
nu_max=12
cbs={}
for spin in range(0,lmax):
    g=context.approx_cb(nu_max,spin)
    cbs.update({spin:g})
```

This time, to prepare a component in the sum-rule, we have to specify which sector it is in. Thus, we make the following definition

```
def make_F(delta,sector,spin,gap_dict,NSO):
    delta=context(delta)
    try:
        gap=context(gap_dict[(sector,spin)])
    except KeyError:
        if spin==0:
            gap=context.epsilon
        else:
            gap=2*context.epsilon+spin
    g_shift=cbs[spin].shift(gap)

    g_num=g_shift.matrix
    g_pref=g_shift.prefactor
    F=context.F_minus_matrix(delta).dot(g_num)
    H=context.F_plus_matrix(delta).dot(g_num)

    if sector=="S":
        num=np.concatenate((context.null_ftype,F,H))

    elif sector=="T":
        num=np.concatenate((F,(1-2/context(NSO))*F,-(1+2/context(NSO))*H))

    elif sector=="A":
        num=np.concatenate((-F,F,-H))

    return context.prefactor_numerator(g_pref,num)
```

Recall that the conformal block `g_shift` is an instance of `prefactor_numerator`, and the line

```
    g_num=g_shift.matrix
```

picks the `np.ndarray` of polynomials in Δ, which is the `matrix` attribute. Another attribute is a `damped_rational` instance named `prefactor`,

```
g_pref=g_shift.prefactor
```
Now `g_num` is a numpy `ndarray` and can be multiplied in the usual way

```
F=context.F_minus_matrix(delta).dot(g_num)
H=context.F_plus_matrix(delta).dot(g_num)
```
`context.F_plus_matrix(delta)` creates the `ndarray` representing the convolution with v^δ, accompanied by the symmetrization in $z \leftrightarrow (1-z)$.

In the lines

```
if sector=="S":
    num=np.concatenate((context.null_ftype,F,H))

elif sector=="T":
    num=np.concatenate((F,(1-2/context(NSO))*F,-(1+2/context(NSO))*H))

elif sector=="A":
    num=np.concatenate((-F,F,-H))
```
we prepare for the numerator polynomials, depending on the desired sector.`context.null_ftype` is a null-vector with the dimension consistent with that of $F^{(-)}$ (i.e., `F`). Finally the line

```
return context.prefactor_numerator(g_pref,num)
```
constructs the return value, combining `g_pref` and `num` into a `prefactor_numerator` instance.

The `make_SDP` function and binary-search function `bs` is almost parallel.

The default action of `sage 3dOn.py` will perform

```
print bs(0.52)
```
which prints the result of binary search on the Δ_S dimension for $\Delta_\phi = 0.52$. If you uncomment the line

```
#print bs(0.52,sector="T")
```
you will also obtain the Δ_T-bound.

2.2.2 A bit simplified version

In `3dOn2.py` same but a bit simplified functions are provided. In the new `make_F`,

```
def make_F(delta,sector,spin,gap_dict,NSO,Delta=None):
    delta=context(delta)
    if Delta==None:
        try:
            gap=context(gap_dict[(sector,spin)])
        except KeyError:
            if spin==0:
                gap=context.epsilon
            else:
                gap=2*context.epsilon+spin
        g=cbs[spin].shift(gap)
    else:
        Delta=context(Delta)
        g=context.gBlock(spin,Delta,0,0)
    F=context.dot(context.F_minus_matrix(delta),g)
    H=context.dot(context.F_plus_matrix(delta),g)

    if sector=="S":
        return [0,F,H]
```

```
    elif sector=="T":
        return [F,(1-2/context(NSO))*F,-(1+2/context(NSO))*H]

    elif sector=="A":
        return [-F,F,-H]
```

I've added another keyword argument `Delta`, default value of which is `None`. Then, if there is no specification for `Delta`, `g` will be the (shifted) conformal block rational approximation, but if `Delta` is given a value, `g` will be the ($\Delta = constant$) conformal block vector. In the first case,

```
    F=context.dot(context.F_minus_matrix(delta),g)
    H=context.dot(context.F_plus_matrix(delta),g)
```

will automatically multiply the convolution matrix with the numerator numpy vector of `g` and combine it with the prefactor of `g`. If `g` is simply a numpy vector, `context.dot` will perform the usual numpy manipulation. The return value like

```
    if sector=="S":
        return [0,F,H]
```

looks a bit different from the previous example - there are no numpy concatenation, and the return value is a list of `prefactor_numerator`s and 0s.

Note also the change in `make_SDP`

```
def make_SDP(delta,gap_dict,NSO=2):
    delta=context(delta)
    pvms=[]
    for sector in ("S","T","A"):
        if sector is not "A":
            spins=[spin for spin in cbs.keys() if not spin%2]
        else:
            spins=[spin for spin in cbs.keys() if spin%2]
        for spin in spins:
            pvms.append(make_F(delta,sector,spin,gap_dict,NSO))

    norm=make_F(delta,"S",0,None,NSO,Delta=0)
    obj=0
    return context.sumrule_to_SDP(norm,obj,pvms)
```

Note that in the definition of `norm`, we are specifying `Delta` keyword, so in the terminology of the previously defined `make_SDP` function, `norm` here should look like

```
[0,norm_F,norm_H]
```

Note also that `obj` is simply `0` (integer). Then, in the line

```
    return context.sumrule_to_SDP(norm,obj,pvms)
```

`sumrule_to_SDP` will automatically collect the dimensions of all the `prefactor_numerator` entries in the list, and replace all the 0 entries with the null vectors of appropriate dimensions. After that, it will pass the replaced list to `context.join`, which is a function to combine `prefactor_numerator` or numpy ndarray into a single `prefactor_numerator`.[3]

As a simple twist, you can also obtain the lower bound on the current central charge.

2.3 $d = 3$, $\mathcal{N} = 2$ example

`examples/supe_Ising.py` is an example of $d = 3$, $\mathcal{N} = 2$ superconformal bootstrap implementation, which is straightforward but a bit lengthy. First of all, we have to prepare a function to compute the superconformal block for chiral $\times$ anti-chiral OPE:

[3] This operation will also join `prefactor_numerator` with different prefactors by taking the least-common `damped_rational`.

```
def cSCblock(nu_max,spin,Delta=None):
    d=context.epsilon*2+2
    epsilon=context.epsilon
    pole_add_data=[((1,1),[-spin-1],[-spin],(spin+2*epsilon)/4/(spin+epsilon)),\
            ((0,2),[epsilon,epsilon-1,context(-1-spin),spin+d-3],\
            [0,d-3,context(-spin),context(spin+2*epsilon)],1/context(16))]
    if spin > 0:
        pole_add_data.append(((-1,1),[spin+d-3],[spin+2*epsilon]\
                ,context(spin)/context(4*(spin+epsilon))))
    if Delta ==None:
        res_g=context.approx_cb(nu_max,spin,0,0)
        res_g_tilde=res_g
        bosonic = res_g
    else:
        Delta=context(Delta)
        res_g=context.gBlock(spin,Delta,0,0)
        res_g_tilde=np.copy(res_g)
        bosonic=np.copy(res_g)
        if Delta==0 and spin==0:
            return (res_g,res_g_tilde,bosonic)
    for shift,poles,factors,C in pole_add_data:
        if Delta==None:
            g=context.approx_cb(nu_max,spin+shift[0]).shift(shift[1])\
                    .multiply_factored_rational(poles,factors,C)
        else:
            g=context.gBlock(spin+shift[0],Delta+shift[1],0,0)
            for x in poles:
                g=g/(Delta-context(x))
            for x in factors:
                g*=(Delta-context(x))
            g=g*C
        res_g+=g
        if shift[0]%2:
            res_g_tilde=res_g_tilde-g
        else:
            res_g_tilde=res_g_tilde+g
    return (res_g, res_g_tilde, bosonic)
```

The definition for `pole_add_data`

```
    pole_add_data=[((1,1),[-spin-1],[-spin],(spin+2*epsilon)/4/(spin+epsilon)),\
            ((0,2),[epsilon,epsilon-1,context(-1-spin),spin+d-3],\
            [0,d-3,context(-spin),context(spin+2*epsilon)],1/context(16))]
    if spin > 0:
        pole_add_data.append(((-1,1),[spin+d-3],[spin+2*epsilon]\
                ,context(spin)/context(4*(spin+epsilon))))
```

is a list of 4-component tuples, which specify how the superconformal descendants contribute to

the superconformal block

$$\begin{aligned}\mathcal{G}(\Delta,l:z,\bar{z}) =& g(\Delta,l:z,\bar{z})\\ &+\frac{l+d-2}{4(l+\frac{d-2}{2})}\frac{\Delta+l}{(\Delta+l+1)}g(\Delta+1,l+1:z,\bar{z})\\ &+\frac{1}{16}\frac{\Delta(\Delta-d+3)(\Delta+l)(\Delta-l-2\epsilon)}{(\Delta-\epsilon)(\Delta-\epsilon+1)(\Delta+1+l)(\Delta-l-d+3)}g(\Delta+2,l:z,\bar{z})\\ &+\frac{l}{4(l+\epsilon)}\frac{(\Delta-l-2\epsilon)}{(\Delta-l-d+3)}g(\Delta+1,l-1:z,\bar{z})\end{aligned}$$

in our convention (see App.A). Of course, the last contribution is present only for $l>0$ (so this data is appended only if `spin==0`). As a first step, we compute the bosonic conformal block:

```
if Delta ==None:
    res_g=context.approx_cb(nu_max,spin,0,0)
    res_g_tilde=res_g
    bosonic = res_g
else:
    Delta=context(Delta)
    res_g=context.gBlock(spin,Delta,0,0)
    res_g_tilde=np.copy(res_g)
    bosonic=np.copy(res_g)
    if Delta==0 and pin==0:
        return (res_g,res_g_tilde,bosonic)
```

We will need different linear combinations of conformal blocks, $\mathcal{G}(\Delta,l)$, $\tilde{\mathcal{G}}(\Delta,l)$, and $g(\Delta,l)$ in the sum-rule, so separately prepare the variables `res_g`, `res_g_tilde`, and `bosonic`.

When `Delta` is specified, it means we are computing Δ =constant superconformal block numpy vector. In this mode, note that simply setting `res_g_tilde=res_g` doesn't work, and must employ `copy` method of numpy. Note also that `Delta=0` (i.e. the identity conformal block) is exceptional.

Let us take a close look at the `for` loop inside

```
for shift,poles,factors,C in pole_add_data:
```

which will compute the contribution from superconformal descendants. As an example, if you consider

$$\frac{l+d-2}{4(l+\frac{d-2}{2})}\frac{\Delta+l}{(\Delta+l+1)}g(\Delta+1,l+1:z,\bar{z}),$$

this conformal block contribution corresponds to If `Delta==None` (i.e., we compute the rational-approximation)

```
g=context.approx_cb(nu_max,spin+shift[0]).shift(shift[1])\
        .multiply_factored_rational(poles,factors,C)
```

with `shift=(1,1)`, `poles=[context(-spin-1)]`, `factors=[-spin]`, and `C=(spin`d-2)/4/(spin+epsilon)+. This is achieved in the following steps:

1. Compute the conformal block with spin `spin+shift[0]`
2. Shift `Delta` by the `shift[1]` by the usual shift method
3. Add poles, zeros, and overall constants with the `multiply_factored_rational` method of `prefactor_numerator`. The argument `poles` stands for the locations of newly added poles, `factors` for the zeroes of numerator Δ-polynomial, and `C` for the overall constant.

`prefactor_numerator` class supports `__add__` method, and you are allowed to write

```
res_g+=g
```

The remainder of the script should look familiar now.

2.4 A mixed correlator example

The last example exhibits how to reproduce the result of [2], where the assumption that σ and ϵ are the only relevant scalar operators in the 3d Ising model was shown to imply that it is isolated in the space of allowed values of $(\Delta_\sigma, \Delta_\epsilon)$.

The sum-rule constraints coming from the consistency conditions of $\langle\sigma\sigma\sigma\sigma\rangle$, $\langle\epsilon\epsilon\epsilon\epsilon\rangle$, and $\langle\sigma\sigma\epsilon\epsilon\rangle$ is [2]

$$\sum_{O:\mathbb{Z}_2\text{-even}} (\lambda_{\sigma\sigma O}, \lambda_{\epsilon\epsilon O})\vec{V}_{\mathbb{Z}_2\text{-even},\Delta_O,l_O}\begin{pmatrix}\lambda_{\sigma\sigma O}\\ \lambda_{\epsilon\epsilon O}\end{pmatrix} + \sum_{O:\mathbb{Z}_2\text{-odd}} \lambda_{\sigma\epsilon O}\vec{V}_{\mathbb{Z}_2\text{-odd},\Delta_O,l_O}\lambda_{\sigma\epsilon O} = \vec{0}$$

$$\vec{V}_{\mathbb{Z}_2\text{-even},\Delta,l} = \begin{pmatrix} \begin{pmatrix} F^{(-)\sigma\sigma,\sigma\sigma}_{\Delta,l} & 0\\ 0 & 0\end{pmatrix}\\ \begin{pmatrix} 0 & 0\\ 0 & F^{(-)\epsilon\epsilon,\epsilon\epsilon}_{\Delta,l}\end{pmatrix}\\ \begin{pmatrix} 0 & 0\\ 0 & 0\end{pmatrix}\\ \begin{pmatrix} 0 & \frac{1}{2}F^{(-)\sigma\sigma,\epsilon\epsilon}_{\Delta,l}\\ \frac{1}{2}F^{(-)\sigma\sigma,\epsilon\epsilon}_{\Delta,l} & 0\end{pmatrix}\\ \begin{pmatrix} 0 & \frac{1}{2}F^{(+)\sigma\sigma,\epsilon\epsilon}_{\Delta,l}\\ \frac{1}{2}F^{(+)\sigma\sigma,\epsilon\epsilon}_{\Delta,l} & 0\end{pmatrix}\end{pmatrix}, \quad \vec{V}_{\mathbb{Z}_2\text{-odd},\Delta,l} = \begin{pmatrix} 0\\ 0\\ (-1)^l F^{(-)\sigma\epsilon,\sigma\epsilon}_{\Delta,l}\\ F^{(-)\epsilon\sigma,\sigma\epsilon}_{\Delta,l}\\ -F^{(+)\epsilon\sigma,\sigma\epsilon}_{\Delta,l}\end{pmatrix}$$

with

$$F^{(\mp)ij,kl}_{\Delta,l} = v^{\frac{\Delta_j+\Delta_k}{2}} g^{\Delta_{ij},\Delta_{kl}}(\Delta, l : z, \bar{z}) \mp \{z \leftrightarrow \bar{z}\}$$

So let us see around the code `mixed_ising.py` where this sum-rule is implemented. The beginning part should look familiar except for the import of `cached_function`.

```
import sage.cboot as cb
from sage.misc.cachefunc import cached_function
from subprocess import Popen, PIPE
import re

sdpb="sdpb"
sdpbparams=["--findPrimalFeasible","--findDualFeasible","--noFinalCheckpoint"]

context=cb.context_for_scalar(epsilon=0.5,Lambda=11)
lmax=20
nu_max=8
```

This time we also have to prepare the conformal block derivative table for non-identical scalars,

```
@cached_function
def prepare_g_0(spin,Delta=None):
    return context.approx_cb(nu_max,spin)

@cached_function
def prepare_g_se(spin,Delta_se,Delta=None):
    g_se=context.approx_cb(nu_max,spin,Delta_1_2=Delta_se,Delta_3_4=Delta_se)
    return g_se

@cached_function
def prepare_g_es(spin,Delta_se,Delta=None):
```

```
    g_es=context.approx_cb(nu_max,spin,Delta_1_2=-Delta_se,Delta_3_4=Delta_se)
    return g_es
```

`@cached_function` will store the result of the function execution and we will be able to avoid the re-computation when called with the same argument.

`prepare_g_se` will compute the conformal block with

$$g_{\Delta,l}^{\Delta_{\sigma\epsilon},\Delta_{\sigma\epsilon}}(z,\bar{z})$$

with the context's `approx_cb` method. This time, the keyword arguments `Delta_1_2` and `Delta_3_4` (whose default value are 0) are specified to be a non-zero value, i.e., $\Delta_{\sigma\epsilon}$. The following function is prepared just for computing the constant-Δ conformal block vectors:

```
def prepare_g(spin,Delta_se,Delta=None):
    if Delta==None:
        return (prepare_g_0(spin),
                prepare_g_se(spin,Delta_se),
                prepare_g_es(spin,Delta_se))
    else:
        g_0=context.gBlock(spin,Delta,0,0)
        if not (Delta==0 and spin==0):
            g_se=context.gBlock(spin,Delta,Delta_se,Delta_se)
            g_es=context.gBlock(spin,Delta,-Delta_se,Delta_se)
        else:
            g_se=None
            g_es=None
        return (g_0,g_se,g_es)
```

Note that the `gBlock` method will raise an error when called with $\Delta, l = 0$ but $\Delta_{12} \neq 0$.

The structure of `make_F` function is almost the same, but if `sector` argument is `even`, the contribution is a vector of matrices, so the output value in this case is a nested list.

```
def make_F(deltas,sector,spin,gap_dict,Delta=None):
    delta_s=context(deltas[0])
    delta_e=context(deltas[1])
    Delta_se=delta_s-delta_e
    if Delta==None:
        try:
            shift=context(gap_dict[(sector,spin)])
        except KeyError:
            if spin==0:
                shift=context.epsilon
            else:
                shift=2*context.epsilon+spin
        gs=[x.shift(shift) for x in prepare_g(spin,Delta_se,Delta=Delta)]
    else:
        gs=prepare_g(spin,Delta_se,Delta=Delta)

    if sector=="even":
        F_s_s=context.dot(context.F_minus_matrix(delta_s),gs[0])
        F_e_e=context.dot(context.F_minus_matrix(delta_e),gs[0])

        F_s_e=context.dot(context.F_minus_matrix((delta_s+delta_e)/2),gs[0])
        H_s_e=context.dot(context.F_plus_matrix((delta_s+delta_e)/2),gs[0])
        return [[[F_s_s,0],
                    [0,0]],
                  [[0,0],
```

```
                [0,F_e_e]],
                [[0,0],
                 [0,0]],
                [[0,F_s_e/2],
                 [F_s_e/2,0]],
                [[0,H_s_e/2],
                 [H_s_e/2,0]]]

    elif sector=="odd+":
        F_s_e=context.dot(context.F_minus_matrix((delta_s+delta_e)/2),gs[1])
        F_e_s=context.dot(context.F_minus_matrix(delta_s),gs[2])
        H_e_s=context.dot(context.F_plus_matrix(delta_s),gs[2])

        return [0,0,F_s_e,F_e_s,-H_e_s]

    elif sector=="odd-":
        F_s_e=context.dot(context.F_minus_matrix((delta_s+delta_e)/2),gs[1])
        F_e_s=context.dot(context.F_minus_matrix(delta_s),gs[2])
        H_e_s=context.dot(context.F_plus_matrix(delta_s),gs[2])

        return [0,0,-F_s_e,F_e_s,-H_e_s]
    else: raise RuntimeError("unknown sector name")
```

These individual contributions are summed up in the `make_SDP` functions:

```
def make_SDP(deltas):
    pvms=[]
    gaps={("even",0):3,("odd+",0):3}
    for spin in range(0,lmax):
        if not spin%2:
            pvms.append(make_F(deltas,"even",spin,gaps))
            pvms.append(make_F(deltas,"odd+",spin,gaps))
        else:
            pvms.append(make_F(deltas,"odd-",spin,gaps))

    epsilon_contribution=make_F(deltas,"even",0,{},Delta=deltas[1])
    sigma_contribution=make_F(deltas,"odd+",0,{},Delta=deltas[0])
    for m,x in zip(epsilon_contribution,sigma_contribution):
        m[0][0]+=x
    pvms.append(epsilon_contribution)
    norm=[]
    for v in make_F(deltas,"even",0,{},Delta=0):
       norm.append(v[0][0]+v[0][1]+v[1][0]+v[1][1])
    obj=0
    return context.sumrule_to_SDP(norm,obj,pvms)
```

We are assuming that σ and ϵ are the only relevant operators in the $\mathbb{Z}_2$-even and odd sectors, respectively, so we set

```
    gaps={("even",0):3,("odd+",0):3}
```

and consider the isolated contributions of σ and ϵ separately. These are computed in the lines

```
    epsilon_contribution=make_F(deltas,"even",0,{},Delta=deltas[1])
    sigma_contribution=make_F(deltas,"odd+",0,{},Delta=deltas[0])
```

To make the full use of the 3-point symmetry $\langle\sigma\sigma\epsilon\rangle$, these contributions are combined into a sigle positivity requirement by

```
for m,x in zip(epsilon_contribution,sigma_contribution):
    m[0][0]+=x
pvms.append(epsilon_contribution)
```

`norm` is taken to be the identity contribution as usual.

When executed, you will see that how the allowed $(\Delta_\sigma, \Delta_\epsilon)$ are isolated with $\Lambda = 11$ search space.

A Conformal block convention

The conformal block convention used in `cboot` follows that of [3], so in the $z \to 0$ limit with real positive z,

$$g_{\Delta,l}^{\Delta_{12},\Delta_{34}}(z,\bar{z}) = z^{\Delta}(1+O(z)).$$

Thus, compared with the convention of [4], [5], and [6],

$$g_{\Delta,l,\cdots}^{(\text{there})} = \left(\frac{1}{4}\right)^{\Delta} g_{\Delta,l,\cdots}^{(\text{here})},$$

and with [2],

$$g_{\Delta,l,\cdots}^{(\text{there})} = \left(-\frac{1}{4}\right)^{\Delta} g_{\Delta,l,\cdots}^{(\text{here})}.$$

References

[1] D. Simmons-Duffin, "A Semidefinite Program Solver for the Conformal Bootstrap," *JHEP* **06** (2015) 174, `arXiv:1502.02033 [hep-th]`.

[2] F. Kos, D. Poland, and D. Simmons-Duffin, "Bootstrapping Mixed Correlators in the 3D Ising Model," *JHEP* **11** (2014) 109, `arXiv:1406.4858 [hep-th]`.

[3] M. Hogervorst, H. Osborn, and S. Rychkov, "Diagonal Limit for Conformal Blocks in d Dimensions," *JHEP* **08** (2013) 014, `arXiv:1305.1321 [hep-th]`.

[4] F. Kos, D. Poland, and D. Simmons-Duffin, "Bootstrapping the $O(N)$ vector models," *JHEP* **06** (2014) 091, `arXiv:1307.6856 [hep-th]`.

[5] S. El-Showk, M. F. Paulos, D. Poland, S. Rychkov, D. Simmons-Duffin, and A. Vichi, "Solving the 3d Ising Model with the Conformal Bootstrap II. c-Minimization and Precise Critical Exponents," *J. Stat. Phys.* **157** (2014) 869, `arXiv:1403.4545 [hep-th]`.

[6] S. M. Chester, S. Giombi, L. V. Iliesiu, I. R. Klebanov, S. S. Pufu, and R. Yacoby, "Accidental Symmetries and the Conformal Bootstrap," *JHEP* **01** (2016) 110, `arXiv:1507.04424 [hep-th]`.