balthasar Documentation

Release 1.0.0

Olivia Di Matteo

Nov 18, 2016

CONTENTS

CHAPTER

ONE

INTRODUCTION TO BALTHASAR

Balthasar is a set of Python tools I've written to handle objects relevant to my research in quantum tomography. In particular, I work frequently with:

- Mutually unbiased bases and their associated operator tables
- Lines and curves in discrete phase space
- Affine planes and Latin squares and their interplay with curves in phase space
- Discrete Wigner functions (computation and plotting)

The idea was to enable anyone interested to quickly generate and transform tables of MUB operators. In particular, Balthasar was developed with the eventual automation of coarse-grained discrete Wigner functions in mind (paper in preparation).

Balthasar also relies on my package PyniteFields, which handles all the underlying finite field arithmetic.

Currently all development is being done (sometimes poorly) by me, Olivia Di Matteo, a PhD student at the University of Waterloo / Institute for Quantum Computing. Some of the research is being done in collaboration with researchers at the Max Planck Institute for the Science of Light.

Please note that Balthasar is a work in progress, as is this documentation. Things may be incomplete, hastily or unclearly explained, or broken. If you find any bugs, or want to help, shoot me an e-mail at odimatte@uwaterloo.ca.

CHAPTER

TWO

TUTORIAL

2.1 Installation

Balthasar is written in Python 3. The following packages are required:

- PyniteFields
- numpy
- matplotlib, libffi, cairocffi

Balthasar can be installed by running:

```
python setup.py install
```

in the main directory of the program.

To generate the Sphinx documentation, one can run:

```
make html
```

or the desired command from the doc directory (but if you're reading this, you probably already knew that).

2.2 General functionality

Below we will go through some of the simple functionality of Balthasar: making tables of MUB operators, Latin squares, and discrete Wigner Functions. References for the underlying math can be found on the *references page*.

2.2.1 Mutually unbiased bases

To generate a set of MUBs in dimension p^n, you must simply provide a finite field expressed in the self-dual basis.:

```
from pynitefields import *
from balthasar import *

# Construct a finite field and convert to self-dual basis (dimension 8 here)
f = GaloisField(2, 3, [1, 1, 0, 1])
f.to_sdb([3, 5, 6])

# Construct the MUB by feeding it the field
dim8_mubs = MUBs(f)
```

The created table of MUB operators can be viewed using print:

```
dim8_mubs.print()
```

For large dimensions, constructing the MUBs can be quite intensive; by default, matrix forms of all the operators are computed. To speed up initialization, we can tell Balthasar not to compute the matrix, and just handle the operators as names and strings.

```
f256 = GaloisField(2, 8, [1, 0, 1, 1, 1, 0, 0, 0, 1])
f256.to_sdb([5, 18, 30, 44, 106, 135, 147, 249])

dim256_mubs = MUBs(f256) # Head over to reddit, this'll take a while
dim256_mubs = MUBs(f256, matrix = False) # Runs in reasonable time
```

Unfortunately, MUBs constructed without matrix representations cannot be used to numerically evaluate any Wigner functions. They can, however, be used to coarse-grain Wigner functions and compute new surviving displacement operators. Balthasar will produce a warning when you construct MUBs without matrices.

2.2.2 Discrete Wigner functions

One of the main purposes for writing Balthasar was to compute Wigner functions in discrete phases space. A Wigner function can be generated using a table of MUBs expressed in the self-dual basis. A Wigner function can be computed from either a state vector or a density matrix. Here we have an example in dimension 4.

```
# Initialize a field
f = GaloisField(2, 2, [1, 1, 1])
f.to_sdb([1, 2])

# Construct MUB table
dim4_mubs = MUBs(f)

# Construct Wigner function framework by passing it MUBs
my_wf = WignerFunction(dim4_mubs)

# An arbitrary quantum state.
state = (1.0 / math.sqrt(2)) * np.array([[1, 0, 0, 1]])

# Compute the wf of state (returns a numpy array)
wf_bell = my_wf.compute_wf(state)

# Plot the Wigner function of state
my_wf.plot(state)
```

2.2.3 Coarse-grained Wigner functions

Coarse-graining is the result of a foray into research on incomplete quantum tomography. A coarse-grained Wigner function is constructed from a fine-grained one by aggregating the kernel (point) operators over a set of cosets of the finite field. Much of the procedure for coarse graining has been automated, though there are some tunable parameters.

Let us begin with a simple set up::

```
# We will coarse grain a dim 16 system to a dim 4 one
f16 = GaloisField(2, 4, [1, 1, 0, 0, 1])
f16.to_sdb([3, 7, 12, 13])
```

```
f4 = GaloisField(2, 2, [1, 1, 1])
f4.to_sdb([1, 2])

m = MUBs(f16)

fine_wf = WignerFunction(m)
```

There are two coarse-graining schemes we focus on. The first works in the general case; a basis is chosen for the big field with respect to the small field. This is the polynomial basis by default, but it is also possibly to manually specify one using the optional 'basis' argument.

```
# Coarse grain in general
coarse_wf = CoarseWignerFunction(fine_wf, f4, mode='general')
```

For square dimensions, it is also possible to construct cosets using the copy of the small subfield within the big field.

```
# Coarse grain using the subfield as the first coset
coarse_wf = CoarseWignerFunction(fine_wf, f4, mode='subfield')
```

The class CoarseWignerFunction inherits from WignerFunction, so it is possible to compute and plot Wigner functions like normal.

```
from math import sqrt
state = (1.0/sqrt(2))*np.array([[1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]])

coarse_wf.compute_wf(state)
coarse_wf.plot(state)
```

2.3 Advanced functionality

The remaining section of the tutorial pertains more to working with the structure of the underlying phase space than to the structures built on top of it.

In general, the set of 'standard' MUBs is associated to a bundle of linear curves of the form $\beta = \lambda\alpha$. These are called the Desarguesian curves. However, there exist other sets of MUBs associated to different sets of curves. In most cases, these are unitarily equivalent to those of the Desarguesian set, but they may have different entanglement structures.

In what follows, we will discuss how to generate MUB tables using different sets of curves. To these curves we can also plot their striations (sets of parallel lines), and generate their associated Latin squares.

2.3.1 Curves

Balthasar contains a separate class Curve for manipulating curves over the finite field. A curve $\beta(\alpha) = c_0 + c_1\alpha + \cdots + c_k\alpha^k$ where the c_i are elements of the finite field is turned into a Curve object using a list of the form [c_0, c_1, ..., c_k] as in the following example:

```
f = GaloisField(2, 2, [1, 1, 1])
c = Curve([0, f[1], f[3], f[2]], f)

c.print() # Will print the curve in polynomial form
c.print(True) # Will print the points in the curve as tuples
```

Note that it is necessary to specify the field over which the curve is defined. This is because we can simplify cases where the coefficients are just integers:

```
f = GaloisField(5)
c = Curve([0, 1, 3], f)
```

In general, curves are represented in phase space in the form $\beta = f(\alpha)$. However, it is also possible to express a curve in the form $\alpha = f(\beta)$ by passing an extra argument to function.

```
f = GaloisField(5)
cba = Curve([0, 1, 3], f)        # b = a + 3 a
cab = Curve([0, 1, 3], f, "alpha") # a = b + 3 b
```

2.3.2 Striations

Striations are the partitions of the affine plane into groups of parallel lines. They are used to build Latin squares and MUBs, and also to compute the point operators in discrete phase space for the Wigner function under Wootters' quantum net WF formulation. We are no longer using this formalism, but the striations are nevertheless useful to see, in particular when coarse-graining Wigner Functions (where lines from the same striation are bundled together and turned into thick lines).

The set of striations can be generated using the code snippet below. Note that these striations are linear. The striations are stored as a list, with the slopes in order, and the infinite slope last.

You can view a striation graphically by using the plot function and passing in an index. Striations are indexed by their slope, from 0 through each field element to 1, and then the last (-1) striation is the vertical lines with infinite slope.

```
f = GaloisField(2, 2, [1, 1, 1])
s = Striations(f)
s[0] # Rays
s.plot()  # Graphically see the rays
s.plot(2) # Plot the striation with slope x^2, x is the primitive element of f
```

2.3.3 Latin squares

Latin squares can be contructed from non-degenerate curves over finite fields. 'Non-degenerate' means that the curve is something called a permutation polynomial, i.e. putting the entire field through the curve gives us the field back in a permuted order.

As MUBs can be associated with sets of non-degenerate curves (which are also additive and commutative), we can consider that some MUBs can be associated with complete sets of Latin squares. These Latin squares have a special property, that of being a complete, mutually orthogonal set. Some unitary transformations on these MUBs sometimes lead to a new set of mutually orthogonal Latin squares which is isomorphic to the first. These relationships are discussed in detail in previous work, [cite arxiv here].

To generate a Latin square in Balthasar, one must simply pass it a curve over some finite field.

```
f = GaloisField(7)
c = Curve([0, f[1]], f)
l = LatinSquare(c)
l.print() # Prints the Latin square of order 7
```

2.3.4 MUBs and curves

By default, MUBs will be constructed with the set of Desarguesian curves. However, we can specify a set of d + 1 curves with which to produce MUBs. We show here an example in dimension 4. The set of curves is taken from [cite

Andrei's work].

```
f = GaloisField(2, 2, [1, 1, 1])

c1 = Curve([0, 0, f[3]], f)     # beta = alpha^2
c2 = Curve([0, f[3], f[3]], f) # beta = alpha + alpha^2
c3 = Curve([0, f[1], f[3]], f) # beta = sigma alpha + alpha^2
c4 = Curve([0, f[2], f[3]], f) # beta = sigma^2 alpha + alpha^2
c5 = Curve([0, 0], f, True)     # alpha = 0

curves = [c1, c2, c3, c4, c5]

some_mubs = MUBs(f, curves)
```

CHAPTER

THREE

CLASSES IN BALTHASAR

3.1 MUBs (mutually unbiased bases)

class balthasar.**MUBs** (*f*, ***kwargs*)

Class to hold a complete set of mutually unbiased bases (MUBs).

Parameters **f** (*GaloisField*) – The finite field over which the MUBs will be constructed. Must be expressed in self-dual basis.

Class MUBs also takes the following keyword arguments.

Keyword Arguments

- **curves** (*list*) – Advanced functionality. Pass a full set of curves with which to create the MUB table. By default the set of linear curves is used.
- **matrix** (*bool*) – Tells Balthasar whether to construct the full numerical matrices for the MUB operators. True by default.

Class MUBs contains the following public attributes.

field
GaloisField – The finite field of choice

p
int – A prime number, the dimension of a single particle

n
int – The number of particles

dim
int – The dimension of the system dim = p^n

w
pthRootOfUnity – The p^{th} root of unity in the field.

curves
list of Curves – The set of curves used to construct the table

table
The table of operators, in form (name, phase, matrix)

D
dictionary – Table of displacement operators, mapping of points in phase space to the associated (phase, matrix)

matrices
bool – Default true. Constructs the matrices associated with the operators. If set to false, only the string representations of the operators is computed.

Currently, class MUBs fully supports only qubit systems (i.e. systems of dimension 2^n).

build_operator_table()

Construct the table of MUB operators.

MUBs are commonly represented in two forms: collections of mutually unbiased basis vectors, or a table of disjoint, commuting operators whose sets of mutual eigenvectors are mutually unbiased bases. We choose the latter representation, and represent MUBs as a table of tuples of the form (name, phase, matrix); the organization of this table will be specified after what follows.

An operator in the MUB table is represented as a displacement operator of the form

$$D(\alpha, \beta) = \Phi(\alpha, \beta) Z_\alpha X_\beta$$

where α, β are field elements, and $\Phi(\alpha, \beta)$ is a phase factor. The two operators Z and X are defined as having the action

$$Z_\alpha |\ell\rangle = \omega(\alpha\ell)|\ell\rangle, \quad X_\beta |\ell\rangle = |\ell + \beta\rangle.$$

For a single qubit these operators are exactly equal to the Pauli operators; for multiple qubits, tensor products thereof. Similarly for qudits, they are the generalized Paulis and tensor products thereof. These operators satisfy the Heisenberg-Weyl commutation relations, i.e.

$$Z_\alpha X_\beta = \chi(\alpha\beta) X_\beta Z_\alpha$$

where

$$\chi(\alpha) = \omega^{\mathrm{tr}(\alpha)}$$

is the character of the field, and ω is the pth root of unity, $\exp\left(\frac{2\pi i}{p}\right)$.

The trace of a field element is

$$\mathrm{tr}(\alpha) = \alpha + \alpha^p + \cdots + \alpha^{p^{n-1}},$$

and always yields an element of the mother (base prime) field.

We can expand α and β in terms of a self- dual basis $\{\theta_1, \ldots, \theta_n\}$:

$$\alpha = \sum_{i=1}^{n} (a_i \theta_i), \quad \beta = \sum_{i=1}^{n} (b_i \theta_i),$$

where $a_i = \mathrm{tr}(\alpha\theta_i)$ and similarly for the b_i. With this, we can express the $Z_\alpha X_\beta$ in tensor product form:

$$Z_\alpha X_\beta = Z^{a_1} X^{b_1} \otimes \cdots \otimes Z^{a_n} X^{b_n}$$

In a way, we can consider this as assigning each self-dual basis element to a single particle.

Displacement operators will be stored in a dictionary of tuples, indexed by the coordinates in phase space (α, β). The values have the form

(name, $\Phi(\alpha, \beta)$, $Z_\alpha X_\beta$)

Here name will be something like "Z ZX X", or a string representing the tensor product structure above. The phase factor is included separately from the matrix because they're not always needed.

By default, we will produce MUBs of the Desarguesian form, where $\beta = \lambda\alpha$.

f_m(*m*, *x*)
A phase factor which ensures our displacement operators sum to proper projectors.

This portion of the phase is a polynomial over field elements We let $f_0(x) = f_1(x) = 0$. Then we define

$$f_m(x) = \sum_{i=1}^{m-1} \sum_{j=0}^{i-1} x^{2^i+2^j}$$

Parameters

- **m** (*int*) – The level of recursion (this should start as the number of qubits).
- **x** (*FieldElement*) – The argument of the function.

Returns The value of $f_m(x)$ as defined above. This will be a field element and is either 0 or 1. As the output of the polynomial should be an integer rather than a FieldElement, the phase function phi contains some code to make this conversion.

phi(*a*, *b*)
Phase function for the displacement operators.

Parameters

- **a** (*FieldElement*) – Horizontal coordinate in phase-space
- **b** (*FieldElement*) – Vertical coordinate in phase-space

Returns The value of $\Phi(a, b)$.

Currently, the phase functions are expressed as follows:

$$\Phi(\alpha, \beta) = (-1)^{f(\alpha\beta)} i^{\mathrm{tr}(\alpha\beta)}$$

where $f(x)$ is the companion function f_m which determines the sign of the displacement operators (so that we never end up with the negative identity).

print(*matrix_form=False*)
Print a nice(-ish) formatted MUB table.

Parameters **matrix_form** (*bool*) – If set to True, will print out the matrices as well as the operator names.

verify_curves(*curves*)
Check that the properties of user-provided curves are valid.

Curves must be additive and commutative.

TODO: everything. Currently the only thing that is checked is whether there are enough curves.

3.2 WignerFunction

class balthasar.**WignerFunction**(*mubs*)
Class to store and plot a discrete Wigner function.

Parameters **mubs** (*MUBs*) – A set of MUBs. The displacement operators will be used to construct the Wigner function kernel. If the 'matrices' option in MUBs is False, no kernel will be produced.

field
GaloisField – The finite field over which the Wigner function is defined

dim
int – The dimension of the system

mubs
MUBs – The MUBs used to construct this Wigner function

D
dictionary – The dictionary of displacement operators

kernel
dictionary – Operators at each point in discrete phase space ('point operators' according to Wootters)

compute_kernel()
Compute the 'kernel' of the Wigner function, i.e. the set of what Wootters calls point operators.

The kernel is a set of operators associated to each point in phase space, $\Delta(\alpha, \beta)$. The operator at the origin is computed as the sum of all the displacement operators:

$$\Delta(0,0) = \frac{1}{p^n} \sum_{\alpha,\beta} D(\alpha, \beta)$$

The kernel operator at any other point can be computed by translating that of the origin by the appropriate displacement operator:

$$\Delta(\alpha, \beta) = D(\alpha, \beta)\Delta(0,0)D^\dagger(\alpha, \beta)$$

Returns A dictionary containing the mapping from points (α, β) to the operator $\Delta(\alpha, \beta)$.

compute_wf(*state*)
Compute the Wigner function for a given state.

For a state ρ, the value of the Wigner function at a point (α, β) is given by

$$W_\rho(\alpha, \beta) = \text{Tr}(\rho\Delta(\alpha, \beta))$$

Parameters **state** (*np array/matrix*) – The state to compute the Wigner Function of. This is a numpy array and can either be a vector, or a full density matrix.

Returns A numpy matrix which is the Wigner function of the passed state.

plot(*state*, *filename=''*)
Compute and plot the Wigner function of a given state.

The plot is a 3D representation of the phase space with the axes labeled by kets in the computational and +/- basis.

For the CoarseWignerFunction class, the axes are labeled by groups of kets in the same coset.

Parameters

- **state** (*np array/matrix*) – The state to compute the Wigner Function of. This is a numpy array and can either be a vector, or a full density matrix.
- **filename** (*string*) – If a filename is passed, the Wigner function plot will not be displayed on screen, but instead saved as an eps file.

plot_mat(*state*)
A simple matrix plot of the Wigner function.

Parameters **state** (*np array/matrix*) – The state to compute the Wigner Function of. This is a numpy array and can either be a vector, or a full density matrix.

3.3 CoarseWignerFunction

class balthasar.**CoarseWignerFunction**(*wf*, *coarse_field*, ***kwargs*)

Bases: balthasar.wignerfunction.WignerFunction

Class to store and plot a coarse-grained discrete Wigner function.

CoarseWignerFunction inherits many properties from WignerFunction, such as the MUBs, field, dimensions, and the original kernel.

Parameters

- **wf** (WignerFunction) – The Wigner function to coarse-grain.
- **coarse_field** (*GaloisField*) – A field over which to perform coarse-graining.

Keyword Arguments

- **mode** (*string*) – There are two modes for coarse-graining, "general", or "subfield". The default mode is "general"; for square dimensions it is also possible to coset using the subfield.
- **basis** (*list of FieldElement*) – A user-specified basis of the field in which to perform coarse-graining. If unspecified, default is the polynomial basis. Only valid for the "general" mode.
- **cosets** (*list of lists*) – A user-specified partitioning of the finite field to use as the cosets for coarse-graining. Use with caution, as no perfect checking mechanisms are implemeneted.

subfield
list – The effective subfield within the large field

subfield_map
list – Map between elements of the large field with those in the coarse field

cosets
list of lists – The set of cosets of the big finite field

coarse_D
dictionary – Surviving displacement operators, indexed by points in the coarse-grained space.

coarse_kernel
dictionary – The kernel operators for the coarse- grained Wigner function.

compute_coarse_D()
Compute the coarse-grained displacement operators.

This will be a subset of the fine-grained displacement operators which 'survive' the sum of eq. x in our paper.

compute_coarse_kernel()
Compute the kernel of the coarse-grained Wigner function.

This is done by 'globbing' together the fine-grained kernel coset by coset:

$$\Delta_c(C_i, C_j) = \sum_{\alpha \in C_i} \sum_{\beta \in C_j} \Delta(\alpha, \beta)$$

The final kernel will be indexed by points in the subfield here, though really it should be indexed by coset representatives.

compute_cosets()
Coset the large finite field over the coarse-grained one.

Two cases to consider here, one where there is a basis provided to make the cosets, the other where we use the subfield.

Details can be found in our manuscript.

compute_polynomial_basis()
Return the polynomial basis for coarse graining.

When the dimension isn't square, this is just the polynomial basis in the 'big' field. When it is square, we choose the polynomial basis of the big field with respect to the small field. For example, in dimension 16 coarse-graining to dimension 4, we choose the basis $\{1, \sigma\}$ rather than the full poly basis because dim 16 is a 2-dim vector space over the dim 4 case.

compute_subfield()
Compute the subfield from the big field. e.g. if our original system is dimension 16, and we are coarse-graining down to dimension 4, we want to find the copy of F4 in F16.

compute_wf(*state*)
Compute the coarse Wigner function for a given state.

The idea there is the same as for the normal Wigner function, except the matrix will have the dimensions of the coarse field.

Parameters **state** (*np array/matrix*) – The state to compute the Wigner Function of. This is a numpy array and can either be a vector, or a full density matrix.

Returns A numpy matrix which is the coarse Wigner function of the state.

3.4 Striations

class balthasar.**Striations**(*field*)
Striations, sets of parallel lines in the affine plane.

The rays in discrete phase space are the set of non-intersecting lines, save for the point at the origin. Together they cover all of phase space. Given a single one of these rays, and translating it by every element in the field, also fully covers the phase space. This set of parallel lines is a striation, and is the basis for things like Latin squares, as well as the quantum nets defined by Wootters when constructing his Wigner functions.

The Striations class is used to construct rays and their translates. Mostly it is here just to keep this business separate from the more physical aspects of Balthasar.

Striations are essentially stored as a set of Curves over the provided finite field.

Parameters **field** (*GaloisField*) – The field / discrete phase space to work in.

field
GaloisField – The finite field we work in.

dim
int – The dimension of the space.

rays
list – The set of curves in this phase space that pass through the origin. The last ray in the set is the vertical one; the rest can all be accessed by the index which is the power of the primitive element of the slope (e.g. rays of σ^3 is rays[3]).

striations
list of lists – A list containing the full set of translates for each ray.

static generate_rays(*field*)
Generates the set of rays in phase space.

Parameters **field** (*GaloisField*) – The field over which to generate the rays.

Returns The list of rays, as objects of type Curve.

plot(*str_idx=0*, *colours=[]*)
Plot a set of striations in discrete phase space.

Parameters

- **str_idx** (*int*) – The index of the striation to plot. By default this function will plot the rays.
- **colours** (*list*) – A set of colours to use to plot. There are 16 basic ones implemented; if you are plotting striations of a larger phase space, you will need to specify more.

print(*as_points=False*)
Print out all the striations.

Parameters **as_points** (*bool*) – If set to true, will print the curves as a set of points. Otherwise, default behaviour is to print the curves as sets of polynomials over the field elements.

3.5 Curve

class balthasar.**Curve**(*coefs*, *field*, *form='beta'*)
Class to hold all points in a curve.

Curves are sets of points of the form $(\alpha, c(\alpha)$ for all α in a specified GaloisField, where

$$c(\alpha) = c_0 + c_1\alpha + c_2\alpha^2 + ...$$

They are constructed by passing in a list of coefficients in the form $[c_0, c_1, c_2, \ldots]$.

Parameters

- **coefs** (*list*) – The coefficients of the curve.
- **field** (*GaloisField*) – The field over which the curve is defined.

field
GaloisField – The finite field in which is curve is defined

coefs
list – The coefficients $[c_0, c_1, \ldots]$.

form
string – "beta" or "alpha", tells whether to do the curve as :math:'beta = f(alpha) or alpha = f(beta). By default, we use the beta form, beta = f(alpha).

is_ray
bool – A Boolean which indicates whether the curve passes through the point (0, 0) or not

points
list – A list of tuples of field elements which are the points of this curve over the field.

print(*as_points=False*)
Print the curve.

Parameters **as_points** (*bool*) – If True is passed, will print the list of points on the curve as tuples. By default, prints the curves as a polynomial.

3.6 LatinSquare

class balthasar.**LatinSquare**(*curve*)

Class to hold a Latin square.

A Latin square L is computed from a curve c according to:

$$L(\alpha, \beta) = \alpha + c(\beta)$$

Parameters **curve** (Curve) – The curve with which to create the Latin square.

p

int – Prime dimension

n

int – Power of prime

dim

int – The dimension of the system p^n

curve

Curve – The curve with which this square was made

square

array – The actual square (stored as a numpy array)

print()

Prints the Latin square.

CHAPTER

FOUR

REFERENCES

The following papers are a (non-exhaustive) set of resources for various aspects of the current functionality of Balthasar.

Operators used in the MUB tables, and various unitarily equivalent constructions:

- [KSSdG] A.B. Klimov, L.L. Sánchez-Soto, H. de Guise (2005) J. Phys. A: Math. Gen **38** 2747
- [RBKSS] J.L. Romero, G. Björk, A. B. Klimov, L.L. Sánchez-Soto (2005) Phys. Rev. A **72** 062310

Curves in phase space:

- [KRBSS09] A.B. Klimov, J.L. Romero, G. Björk, L.L. Sánchez-Soto (2009) Ann. Phys. **324** 53-72
- [KRBSS07] A.B. Klimov, J.L. Romero, G. Björk, L.L. Sánchez-Soto (2007) J. Phys. A: Math. Theor. **40** 3987-3998

Discrete Wigner functions:

- [GHW] K.S. Gibbons, M.J. Hoffman, W.K. Wootters (2004) Phys. Rev. A **70** 062101

How Latin squares fit in with this:

- [GDMKdG] M. Gaeta, O. Di Matteo, A.B. Klimov, H. de Guise (2014) J. Phys. A: Math. Theor. **47** 435303

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

b