Potapov_interpolation Documentation

Release 0.1

Gil Tabak

Sep 30, 2018

Contents

Contents:

CHAPTER 1

Readme

1.1 Overview

The purpose of this package is to characterize network of optical components when time delays with feedback are present. The core of the package identifies 'trapped' optical modes that result from feedback in the system. These modes are found by identifying the roots/poles of the transfer function containing only time delays and passive linear components. Further analysis of the network yields a linear Hamiltonian written in terms of the identified modes, linear operators coupling the system modes to the environment, and an overall scattering matrix. Nonlinear elements are added as additional Hamiltonian terms.

Our manuscript describing this method can be found on http://arxiv.org/abs/1510.08942 or http://www.epjquantumtechnology.com/content/3/1/3.

1.2 Installation

Simply clone the repository, open a terminal window, type:

```
git clone https://github.com/tabakg/potapov_interpolation
cd potapov_interpolation
python setup.py install
```

1.3 Files

1.3.1 Time_Delay_Network.py

This module includes a class to contain the information of a passive linear network with time delays. Several examples are included in this module. Each example includes matrices that yield a transfer function. This module contains methods to re-construct finite-dimensional approximations of a transfer function of passive systems.

–`make_commensurate_roots()`: A method to determine the roots utilizing the commensurate structure of the roots. The algorithm determines a polynomial based on the gcd of the time delays to describe where poles of the transfer function occur. The roots of the polynomial are found. The periodicity of the exponential e^{-zT} for the gcd delay is then used to find the roots within desired frequency ranges.

–`make_roots()`: A method that find the poles of the transfer function within a contour. This is a very general method in that the function needs only to be meromorphic with poles of order 1. In particular this yields the poles even when the delays are not commensurate.

–`run_Potapov()`: This function will run the Potapov procedure. The instance of `Time_Delay_Network` will update its `roots`, `vecs`, and `T_testing`. These are respectively the poles of the transfer function, a list of vectors (complex-valued column matrices) that contain the information to reconstruct the matrix-valued projectors in the Potapov representation, and an approximating transfer function that has been generated using the Potapov interpolation procedure. This method uses either `make_roots()` or `make_commensurate_roots()`, which the user can specify.

1.3.2 Potapov.py

We implement the procedure for finding Blaschke-Potapov products to approximate given functions near poles. Please see section 6.2 in our manuscript for details. This procedure is used to generate the modes of the passive linear network.

Given a rational matrix-valued function, we also construct the matrices ABCD that give the state-space representation of the system (see `get_Potapov_ABCD()`).

1.3.3 Roots.py

A module for identifying the zeros of a complex-valued function.

1.3.4 functions.py

Miscellaneous functions. Includes:

–`Pade()` Generate a Pade approximation for delays that do NOT feed back.

–`spatial_modes()` Finding the spatial location of modes. This is necessary to generate nonlinear terms. The required inputs to `spatial_modes()` are `roots`,`M1`,and `E`. These are respectively the poles of the transfer function, the directed connectivity matrix of the internal nodes of a network, and a diagonal matrix-valued function whose diagonal values correspond to the delays in the Fourier domain.

–`make_nonlinear_interaction()` Generate the weight of an interaction term due to phase-matching.

1.3.5 Hamiltonian.py

Includes a class `Hamiltonian()` to contain the information needed to construct the Hamiltonian of the system, including nonlinear terms. This class also includes a function `make_eq_motion()` to generate the classical equations of motion from the nonlinear Hamiltonian.

Also includes a class `Chi_nonlin()` which contains the information for a particular chi nonlinearity.

1.3.6 Time_Sims.py

Integrate the dynamics of a passive in time using the ABCD matrices.

1.3.7 Time_Sims_nonlin.py

–`make_f_lin()` generates outputs from ABCD model.

–`make_f()` generates outputs from nonlinear Hamiltonian model.

–`run_ODE()` integrates the equations of motion in time.

–`double_up()` prepares a doubled-up system which can be used for non-classical simulations.

1.4 Sample Usage – Time Domain Simulation

See [Simple code example](https://github.com/tabakg/potapov_interpolation/blob/master/Sample_Code_Usage.ipynb)

CHAPTER 2

Potapov_Code package

2.1 Submodules

2.2 Potapov_Code.Time_Delay_Network module

Created on Mon Mar 2 17:37:37 2015

@author: gil @title: examples

class Potapov_Code.Time_Delay_Network.**Example1**(*max_freq=30.0*, *max_linewidth=1.0*, *N=1000*, *center_freq=0.0*, *tau=0.3*, *r=0.8*)

Bases: *Potapov_Code.Time_Delay_Network.Time_Delay_Network*

Single input, single output with a single delay.

class Potapov_Code.Time_Delay_Network.**Example2**(*max_freq=10.0*, *max_linewidth=10.0*, *N=1000*, *center_freq=0.0*, *r=0.9*, *tau=1.0*)

Bases: *Potapov_Code.Time_Delay_Network.Time_Delay_Network*

Two inputs, two outputs with a delay (i.e. Fabry-Perot).

class Potapov_Code.Time_Delay_Network.**Example3**(*max_freq=60.0*, *max_linewidth=1.0*, *N=5000*, *center_freq=0.0*, *r1=0.9*, *r2=0.4*, *r3=0.8*, *tau1=0.1*, *tau2=0.23*, *tau3=0.1*, *tau4=0.17*)

Bases: *Potapov_Code.Time_Delay_Network.Time_Delay_Network*

Two inputs and two outputs, with four delays and third mirror This corresponds to figures 7 and 8 in our paper.

class Potapov_Code.Time_Delay_Network.**Example4**(*max_freq=100.0*, *max_linewidth=3.0*, *N=5000*, *center_freq=0.0*)

Bases: *Potapov_Code.Time_Delay_Network.Time_Delay_Network*

Two inputs and two outputs, with free delay (i.e. not in a loop). This corresponds to figures 9 and 10 in our paper.

class Potapov_Code.Time_Delay_Network.**Example5**(*max_freq=50.0*, *max_linewidth=3.0*, *N=1000*, *center_freq=0.0*)

Bases: *Potapov_Code.Time_Delay_Network.Time_Delay_Network*

Modified example 4, with analytic term.

class Potapov_Code.Time_Delay_Network.**Time_Delay_Network**(*max_freq=30.0*, *max_linewidth=1.0*, *N=1000*, *center_freq=0.0*)

A class to contain the information of a passive linear network with time delays.

max_freq
maximum height in the complex plane.

Type optional [float]

max_linewidth
maximum width in the complex plane.

Type optional [float]

N
number of points to use on the contour for finding the roots/poles of the network.

Type optional [int]

center_freq
how much to move the frame up or down the complex plane.

Type optional [float]

get_Potapov_ABCD(*z=0.0*, *doubled=False*)
Find the ABCD matrices from the Time_Delay_Network.

Parameters **z** (*optional [complex number]*) – location where to estimate D.

Returns A,B,C,D matrices.

Return type (tuple of matrices)

get_frequency_pertub_func_z(*use_ufuncify=True*)
Generates a function that can be used to perturb roots using Newton's method. This function has form $-f(z)/f'(z)$ when the time delays are held fixed.

We give two ways to generate the perturbative function. One is by directly plugging in numbers into a sympy expression and the second is by using the ufuncify method to creative a wrapper for the function.

Parameters **use_ufuncify** (*optional [boolean]*) – whether to use ufuncify or not.

Returns The function to use for Newton's method in z, $-f(z)/f'(z)$.

Return type Newton's method function (function)

get_minimizing_function_z()
Minimizing this function gives the adjusted roots.

Gives a function to minimize, its arguments are x, y, Ts. Also gives its derivative.

Returns A function of $x, y, *Ts$ to minimize in the two variables, x, y.

Return type Minimizing function (function)

get_outputs()
Get some of the relevant outputs from the Potapov procedure.

Returns The original transfer function, the approximating generated transfer function, the identified poles of the transfer function, and the vectors representing the form of the Potapov factors.

Return type *self.T*,'self.T_testing','self.roots','self.vecs' (tuple)

Raises **Exception** – Must have *self.T,self.T_testing,self.roots,self.vecs*.

get_symbolic_frequency_perturbation_z()
A method to prepare the symbolic expression T_denom_sym for further computations. This expression represents the denominator in terms of the various delays $T_1, ..., T_k$ and the complex variable z.

The inputs should be $z, (T_1 + \Delta T_1, ..., T_k + \Delta T_k)$

This method treats the various delays as separate variables.

Returns T_denom_sym and its derivative in z.

Return type A pair of two symbolic expressions (tuple)

make_T_Testing()
Generate the approximating transfer function using the identified poles of the transfer function.

make_commensurate_roots(*list_of_ranges=[]*)
Assuming the delays are commensurate, obtain all the roots within the frequency ranges of interest. Sets self.roots a list of complex roots in the desired frequency ranges.

Parameters **list_of_ranges** (*optional [list of 2-tuples]*) – list of frequency ranges of interest in the form: (minimum frequency, maximum frequency).

Returns None.

make_commensurate_vecs()

make_roots()
Generate the roots given the denominator of the transfer function.

make_spatial_modes()
Generate the spatial modes of the network.

make_vecs()
Generate an ordered list of vectors representing the form of the Potapov factors.

run_Potapov(*commensurate_roots=False*, *filtering_roots=True*)
Run the entire Potapov procedure to find all important information. The generated roots, vecs, approximated transfer function T_Testing, and the spatial_modes are all stored in the class.

Parameters

- **commensurate_roots** (*optional[boolean]*) – which root-finding method to use.
- **filtering_roots** (*optional[boolean]*) – makes sure the poles of the transfer function all have negative real part. Drops ones that might not.

Returns None.

Potapov_Code.Time_Delay_Network.**example6_pade**()
This example is the same as example 3, but we return a Pade approximation instead of a Potapov approximation. Instead of returnings roots, etc., we return a different kind of function (see below).

This is used for figure 14 of our paper.

Returns An approximation to $T(z)$ using Pade approximation of order n.

Return type T (matrix-valued function in complex number z and integer n)

Potapov_Code.Time_Delay_Network.**plot3D**(*f*, *N=2000*, *x_min=-1.5*, *x_max=1.5*, *y_min=-25.0*, *y_max=25.0*, *name='complex_plane_plot.pdf'*, *title='Complex Frequecy Response Diagram'*, *xlabel='Cavity damping rate ($\\kappa$)'*, *ylabel='Detuning ($\\Delta$)'*)

Make a color and hue plot in the complex plane for a given function.

Used code from http://stackoverflow.com/questions/17044052/mathplotlib-imshow-complex-2d-array

Parameters

- **f** (*function*) – to plot.
- **N** (*optional[int]*) – number of points to use per dimension.

Returns None.

Potapov_Code.Time_Delay_Network.**plot_all**(*L*, *dx*, *labels*, *colors*, *lw*, *name*, **args*)

A method to plot the absolute value and phase of each component for a list of matrix-valued functions in the complex plane along an axis.

Parameters

- **L** (*float*) – plot from 0 to L.
- **dx** (*float*) – distance between points.
- **labels** (*list of str*) – labels to use.
- **colors** (*list of srt*) – indicators of color for different curves.
- **lw** (*float*) – line width to use.
- **name** (*str*) – name of the file to save.
- **args** (***) – A list of functions to plot.

Returns None.

2.3 Potapov_Code.Hamiltonian module

Created on Mon Mar 31 2015

@author: gil @title: Hamiltonian.py

class Potapov_Code.Hamiltonian.**Chi_nonlin**(*delay_indices*, *start_nonlin*, *length_nonlin*, *refraction_index_func=None*, *chi_order=3*, *chi_function=None*)

Class to store the information in a particular nonlinear chi element.

delay_indices
indices of delays to use.

Type list of indices

start_nonlin
location of nonlinear crystal with respect to each edge.

Type positive float or list of positive floats

length_nonlin
length of the nonlinear element.

Type float

refraction_index_func
the indices of refraction as a function of the netural frequency $/omega$ AND polarization pol.

Type function

chi_order
order of nonlinearity.

Type optional [int]

chi_function
strength of nonlinearity. first (chi_order+1) args are frequencies, next (chi_order+1) args are indices of polarization.

Type optional [function]

class Potapov_Code.Hamiltonian.**Hamiltonian**(*roots*, *modes*, *delays*, *Omega=None*, *nonlin_coeff=1.0*, *polarizations=None*, *cross_sectional_area=1e-10*, *chi_nonlinearities=None*, *using_qnet_symbols=False*)

A class to create a sympy expression for the Hamiltonian of a network.

roots
the poles of the transfer function.

Type list of complex numbers

omegas
The natural frequencies of the modes.

Type list of floats

modes
Modes of the network.

Type list of complex-valued column matrices

delays
The delays in the network.

Type list of floats

Omega
Quadratic Hamiltonian term for linear dynamics.

Type optional [matrix]

nonlin_coeff
Overall scaling for the nonlinearities.

Type optional [float]

polarizations
The polarizations of the respective modes. These should match the arguments in Chi_nonlin.chi_func.

Type optional [list]

cross_sectional_area
Area of beams, used to determines the scaling for the various modes.

Type float

chi_nonlinearities
A list of Chi_nonlin instances.

Type list

TODO: Return L operator for QNET. TODO: decide what to do with roots of negative imaginary part (negative freq.) TODO: maybe re-organize the data into a dict of the form mode_inde: (root,mode,etc) TODO: Replace python floats by mpmath variables for arbitrary precision

Dagger(*symbol*)

E_field_weight(*mode_index*)
Make the weights for each field component $E_i(n) = [\text{weight}](a + a^\dagger)$.

Parameters **mode_index** (*int*) – The index of the mode.

Returns It has form: $[\hbar * \omega(n)/2V_{eff}(n)\epsilon]^{1/2}$. Here we set hbar = 1.

Return type Weight of E-field (float)

make_Delta_delays()
Each different frequency will experience a different shift in delay lengths due to all nonlinearities present. We will store those shifts as a list of lists in the class. This list is called Delta_delays. The ith list will be the shifts in all the original delays for the ith root (i.e. frequency).

Returns None.

make_E_field_weights()

Returns A dictionary from mode index to the E-field weight.

Return type Weights (dict)

TODO: In the make_positive_keys_chi2 function, generate and pass the correct polarization functions.

make_H(*eps=1e-05*)
Make a Hamiltonian combining the linear and nonlinear parts.

The term -1j*A carries the effective linear Hamiltonian, including the decay term $-\frac{i}{2}L^\dagger L$. However, this term does not include material effects including dielectric and nonlinear terms. It also does not include a term with contribution from the inputs.

If one wishes to include terms due to coherent input, one can impose a linear Hamiltonian term consistent with the classical equations of motion. This yields the usual term $i(a\alpha^* - a^\dagger\alpha)$.

To obtain the form $A = i\Omega - \frac{1}{2}C^\dagger C$ with $Omega$ Hermitian, we notice A can be split into Hermitian and anti-Hermitian parts. The anti-Hermitian part of A describes the closed dynamics only and the Hermitian part corresponds to the decay terms due to the coupling to the environment at the input/output ports.

Parameters

- **Omega** (*complex-valued matrix*) – Describes the Hamiltonian of the system.
- **eps** (*optional[float]*) – Cutoff for the significance of a particular term.

Note: Omega = -1j*A <— full dynamics (not necessarily Hermitian)

Omega = (A-A.H)/(2j) <— closed dynamics only (Hermitian part of above)

Returns A symbolic expression for the full Hamiltonian.

Return type Expression (sympy expression)

make_chi_nonlinearity(*delay_indices*, *start_nonlin*, *length_nonlin*, *refraction_index_func=None*, *chi_order=3*, *chi_function=None*)
Add an instance of Chi_nonlin to Hamiltonian.

Parameters

- **delay_indices** (*int OR list/tuple of ints*) – The index representing the delay line along which the nonlinearity lies. If given a list/tuple then the nonlinearity interacts the N different modes.
- **start_nonlin** (*float OR list/tuple of floats*) – The beginning of the nonlinearity. If a list/tuple then each nonlinearity begins at a different time along its corresponding delay line.
- **length_nonlin** (*float*) – Duration of the nonlinearity in terms of length. (Units in length)
- **refraction_index_func** (*function*) – The indices of refraction as a function of the natural frequency $/omega$.
- **chi_order** (*optional [int]*) – Order of the chi nonlinearity.
- **chi_function** (*function*) – A function of 2*chi_order+2 parameters that returns the strenght of the interaction for given frequency combinations and polarizations. The first chi_order+1 parameters correspond to frequencies combined the the next chi_order+1 parameters correspond to the various polarizations.
- **TODO** – check units everywhere, including f versus omega = f / 2 pi.

make_dict_H_lin(*Omega*)

Using the current information about the modes and chi_nonlinearities, generate a dictioanry mapping

make_dict_H_nonlin(*filtering_phase_weights=False*, *eps=1e-05*)

Using the current information about the modes and chi_nonlinearities, generate a dictioanry mapping

make_eq_motion()

Input is a tuple or list, output is a matrix vector. This generates Hamilton's equations of motion for a and a^H. These equations are CLASSICAL equations of motion. This means we replace the operators with c-numbers. The orde r of the operators will yield different results, so we assume the Hamiltonian is already in the desired order (e.g. normally ordered).

These equations of motion will not show effects of squeezing. To do this, we will need a full quantum picture.

Returns A function that yields the Hamiltonian equations of motion based on the Hamiltonian given. The equations of motion map $(t, a) \rightarrow v$. where math:*t* is a scalar corresponding to time, a is an array of inputs correpsonding to the internal degrees of freedom, and v is a complex-valued column matrix describing the gradient.

Return type Equations of Motion (function)

make_lin_H(*Omega*)

Make a linear Hamiltonian based on Omega.

Parameters **Omega** (*complex-valued matrix*) – A matrix that describes the Hamiltonian of the system.

Returns A symbolic expression for the nonlinear Hamiltonian.

Return type Expression (sympy expression)

make_nonlin_H(*filtering_phase_weights=False*, *eps=1e-05*)

Make a nonlinear Hamiltonian based on nonlinear interaction terms

Parameters

- **filtering_phase_weights** (*optional[boolean]*) – Whether or not to filter the phase_matching_weights by the size of their values. The cutoff for their absolute value is given by eps.
- **eps** (*optional[float]*) – Cutoff for the significance of a particular term.

Returns A symbolic expression for the nonlinear Hamiltonian.

Return type Expression (sympy expression)

TODO: Make separate dictionaries for values of chi_function, for phase_matching_weights, and for products of E_field_weights. filter the keys before generating terms.

TODO: Make fast function for integrator; combine with make_f and make_f_lin

make_nonlin_term_sympy(*combination*, *pm_arr*)
Make symbolic nonlinear term using sympy.

Example

```
>>> combination = [1,2,3]; pm_arr = [-1,1,1]
>>> print Hamiltonian.make_nonlin_term_sympy(combination,pm_arr)
    a_1*Dagger(a_2)*Dagger(a_3)
```

Parameters

- **combination** (*tuple/list of integers*) – indices of which terms to include.
- **pm_arr** (*tuple/list of +1 and -1*) – Creation and annihilation indicators for the respective terms in combination.

Returns symbolic expression for the combination of creation and annihilation operators.

Return type (sympy expression)

make_phase_matching_weights(*weight_keys*, *chi*, *filtering_phase_weights=False*, *eps=1e-05*)
Make a dict to store the weights for the selected components and the creation/annihilation information.

Parameters

- **weight_keys** (*list of tuples*) – Keys for weights to consider. Each key is a tuple consisting of two components: the first is a tuple of the indices of modes and the second is a tuple of +1 and -1.
- **filtering_phase_weights** (*optional[boolean]*) – Whether or not to filter the phase_matching_weights by the size of their values. The cutoff for their absolute value is given by eps.
- **eps** (*optional [float]*) – Cutoff for filtering of weights.

Returns A dictionary of weights with values corresponding to the phase matching coefficients.

Return type Weights (dict)

make_weight_keys(*chi*, *key_types='all_keys'*, *pols=(1, 1, -1)*, *res=(0.1, 0.1, 0.1)*)
Make a list of keys for which various weights will be determined. Each key is a tuple consisting of two components: the first is a tuple of the indices of modes and the second is a tuple of +1 and -1.

Parameters **chi** (Chi_nonlin) – the nonlinearity for which the weight will be found.

Returns A list of keys of the type described.

Return type Keys (list)

TODO: pass the k(lambda) function from chi to the function called from phase_matching or phase_matching_hash.

mode_volumes()
Find the effective volume of each mode to normalize the field.

Returns A list of the effective lengths of the various modes.

move_to_rotating_frame(*freqs=0.0*, *include_time_terms=True*)
Moves the symbolic Hamiltonian to a rotating frame

We apply a change of basis $a_j \to ae^{-i\omega_j}$ for each mode a_j. This method modifies the symbolic Hamiltonian, so to use it the Hamiltonian sould already be constructed and stored.

Parameters

- **freqs** (*optional [real number or list/tuple]*) – Frequency or list of frequencies to use to displace the Hamiltonian.
- **include_time_terms** (*optional [boolean]*) – If this is set to true, we include the terms $e^{-i\omega_j}$ in the Hamiltonian resulting from a change of basis. This can be set to False if all such terms have already been eliminated (i.e. if the rotating wave approximation has been applied).

TODO: replace the sine and cosine stuff with something nicer. Maybe utilize the _get_real_imag_func method in Time_Delay_Network.

normalize_modes()
Normalize the modes of Hamiltonian.

perturb_roots_z(*perturb_func*, *eps=1e-12*)
One approach to perturbing the roots is to use Newton's method. This is done here using a function perturb_func that corresponds to $-f(z)/f'(z)$ when the time delays are held fixed. The function perturb_func is generated in get_frequency_pertub_func_z.

Parameters

- **perturb_func** (*function*) – The Newton's method function.
- **eps** (*optional [float]*) – Desired precision for convergence.

phase_weight(*combination*, *pm_arr*, *chi*)
The weight to give to each nonlinear term characterized by the given combination and pm_arr.

Parameters

- **combination** (*list/tuple of integers*) – Which modes/roots to pick
- **pm_arr** (*list of +1 and -1*) – creation and annihilation of modes
- **chi** (Chi_nonlin) – The chi nonlinearity for which to compute the phase coefficient.

Returns The weight to add to the Hamiltonian.

2.4 Potapov_Code.Potapov module

Created on Mon Mar 2 13:59:30 2015

@author: Gil Tabak @title: Potapov

The code in this file implements the procedure for finding Blaschke-Potapov products to approximate given functions near poles.

Please see section 6.2 in our manuscript for details: http://arxiv.org/abs/1510.08942 (to be published in EPJ QT).

Potapov_Code.Potapov.**Potapov_prod**(*z*, *poles*, *vecs*, *N*)

Takes a transfer function T(z) that outputs numpy matrices for imaginary $z = i\omega$ and the desired poles that characterize the modes. Returns the Potapov product as a function approximating the original transfer function.

Parameters

- **z** (*complex number*) – value where product is evaluated.
- **poles** (*list of complex numbers*) – The poles of the Potapov product.
- **vecs** (*list of complex-valued matrices*) – The eigenvectors corresponding to the orthogonal projectors of the Potapov product.
- **N** (*int*) – Dimensionality of the the range.

Returns Complex-valued matrix of size $N \times N$.

Return type (matrix)

Potapov_Code.Potapov.**estimate_D**(*A*, *B*, *C*, *T*, *z*)

Estimate the scattering matrix S=D using the ABC matrices the transfer function T at a frequency $z = i\omega$.

Try to satisfy $T(z) = D + C(zI - A)^{-1}B$

Parameters

- **A,B,C** (*matrices*) – The A,B, and C matrices of the state-space representation.
- **T** (*matrix-valued function*) – The input/output function to estimate.
- **z** (*complex number*) – The location at which the scattering matrix will be estimated.

Returns The estimated S=D scaterring matrix based on the value of the function T and the ABC matrices.

Return type D (matrix)

Potapov_Code.Potapov.**finite_transfer_function**(*U*, *eigenvectors*, *eigenvalues*)

Give a rational Blaschke-Potapov product of z with the given eigenvalues and eigenvectors and constant unitary factor U.

Parameters

- **U** (*complex-valued matrix*) – A unitary matrix.
- **eigenvectors** (*list of complex-valued matrices*) – eigenvectors to use.
- **eigenvalues** (*list of complex numebrs*) – eigenvalues to use.

Returns A function that takes a complex number and returns the Potapov product evaluated at that number.

Return type Transfer function (function)

Potapov_Code.Potapov.**get_ABCD**(*val*, *vec*)

Make the ABCD model of a single Potapov factor given some eigenvalue and eigenvector.

The ABCD model can be used to obtain the dynamics of a linear system.

Parameters

- **val** (*complex number*) – an eigenvalue.

- **vec** (*complex-valued matrix*) – an eigenvector.
- **sym** (*optiona[boolean]*) – Modify B and C so that $B = C.H$.

Returns Four matrices representing the ABCD model.

Return type [A,B,C,D] (list)

Potapov_Code.Potapov.**get_Potapov**(*T*, *poles*, *found_vecs*)

Given a transfer function T and some poles, generate the Blaschke-Potapov product to reconstruct or approximate T, assuming that T can be represented by the Blaschke-Potapov product with the given poles. Also match the values of the functions at zero.

If T is a Blaschke-Potapov function and the the given poles are the only poles, then T will be reconstructed.

In general, there is possibly an analytic term that is not captured by using a Blaschke-Potapov approximation.

Parameters

- **T** (*matrix-valued function*) – A given meromorphic function.
- **poles** (*a list of complex valued numbers*) – The given poles of T.
- **vecs** (*list of complex-valued matrices*) – The eigenvectors corresponding to the orthogonal projectors of the Potapov product.

Returns equation to T at z=0 and approximating T using a Potapov product generated by its poles and residues.

Return type Potapov product (matrix-valued function)

Potapov_Code.Potapov.**get_Potapov_ABCD**(*poles*, *vecs*, *T=None*, *z=None*)

Combine the ABCD models for the different degrees of freedom.

Parameters

- **val** (*a list of complex numbers*) – given eigenvalues.
- **vec** (*a list of complex-valued matrices*) – given eigenvectors.

Returns Four matrices representing the ABCD model.

Return type [A,B,C,D] (list)

Potapov_Code.Potapov.**get_Potapov_vecs**(*T*, *poles*)

Given a transfer function T and some poles, compute the residues about the poles and generate the eigenvectors to use for constructing the projectors in the Blaschke-Potapov factorization.

Potapov_Code.Potapov.**normalize**(*vec*)

Normalize a vector.

Parameters **vec** (*complex-valued matrix*) – A vector.

Returns The normalized vector.

Return type (vector)

Potapov_Code.Potapov.**plot**()

A nice function for plotting components of matrix-valued functions.

Parameters

- **L** (*float*) – length along which to plot.
- **dx** (*float*) – step length to take.
- **func** (*function*) – complex matrix-valued function.

- **i,j** (*tuple of ints*) – coordinate to plot.
- **args** (*functions*) – Desired transformations on the inputs.

Potapov_Code.Potapov.**prod**(*z*, *U*, *eigenvectors*, *eigenvalues*)

Return the Blaschke-Potapov product with the given eigenvalues and eigenvectors and constant unitary factor U evaluated at z.

Parameters

- **z** (*complex number*) – where product is evaluated.
- **U** (*complex-valued matrix*) – A unitary matrix.
- **eigenvectors** (*list of complex-valued matrices*) – eigenvectors to use.
- **eigenvalues** (*list of complex numebrs*) – eigenvalues to use.

Returns The Potapov product evaluated at z.

Return type Product (complex-valued matrix)

2.5 Potapov_Code.Roots module

Created on Sat Feb 28 20:15:35 2015

@author: gil @title: Rootfinder

Find the roots of a function f in the complex plane inside of a rectangular region. We implement the method in the following paper:

> Delves, L. M., and J. N. Lyness. "A numerical method for locating the zeros of an analytic function." Mathematics of computation 21.100 (1967): 543-560.

Alternative code using a similar method can be found here:

> http://cpc.cs.qub.ac.uk/summaries/ADKW_v1_0.html

The main idea is to compute contour integrals of functions of the form $z^k f'/f$ around the contour, for integer values of k. Here f' denotes the derivative of f. The resulting values of the contour integrals are proportional to $\sum_i z_i^k$, where i is the index of the roots.

Throughout we denote $f_{frac} = f'/f$.

I have also tried several optimizations and strategies for numerical stability.

Potapov_Code.Roots.**Muller**(*x1*, *x2*, *x3*, *f*, *tol=1e-05*, *N=400*, *verbose=False*)

A method that works well for finding roots locally in the complex plane. Uses three points for initial guess, x1,x2,x3.

Parameters

- **x1,x2,x3** (*complex numbers*) – initial points for the algorithm.
- **f** (*function*) – complex valued function for which to find roots.
- **tol** (*optional[float]*) – tolerance.
- **N** (*optional[int]*) – maximum number of iterations.
- **verbose** (*optional[boolean]*) – print warnings.

Returns estimated root of the function f.

Potapov_Code.Roots.**combine**(*eps=1e-05*, **args*)
chain together several lists and purge redundancies.

Parameters

- **eps** (*optional[float]*) – tolerance for purging elements.
- **args** (*lists*) – several lists.

Returns A list of combined elements.

Potapov_Code.Roots.**count_roots_rect**(*f*, *fp*, *x_cent*, *y_cent*, *width*, *height*, *N=10*, *outlier_coeff=100.0*, *max_steps=5*, *known_roots=None*, *verbose=False*)
I assume f is analytic with simple (i.e. order one) zeros.

TODO: save values along edges if iterating to a smaller rectangle extend to other kinds of functions, e.g. function with non-simple zeros.

Parameters

- **f** (*function*) – the function for which the roots (i.e. zeros) will be found.
- **fp** (*function*) – the derivative of f.
- **x_cent,y_cent** (*floats*) – The center of the rectangle in the complex plane.
- **width,height** (*floats*) – half the width and height of the rectangular region.
- **N** (*optional[int]*) – Number of points to sample per edge
- **outlier_coeff** (*float*) – multiplier for coefficient used when subtracting poles to improve numerical stability. See new_f_frac_safe.
- **max_step** (*optional[int]*) – Number of iterations allowed for algorithm to repeat on smaller rectangles.
- **roots** (*known*) – Roots of f that are already known.
- **verbose** (*optional[boolean]*) – print warnings.

Returns

A list of roots for the function f inside the rectangle determined by the values x_cent,y_cent,width, and height.

Potapov_Code.Roots.**find_maxes**(*y*)
Given a list of numbers, find the indices where local maxima happen.

Parameters **y** (*list of floats*) –

Returns list of indices where maxima occur.

Potapov_Code.Roots.**find_roots**(*y_smooth*, *c*, *num_roots_to_find*)
given the values y_smooth, locations c, and the number to go up to, find the roots using the polynomial trick.

Parameters **y_smooth** (*list of complex numbers*) – poins along smoothed-out boundary.

Potapov_Code.Roots.**get_boundary**(*x_cent*, *y_cent*, *width*, *height*, *N*)
Make a rectangle centered at x_cent,y_cent. Find points along this rectangle. I use the convention that width/height make up half the dimensions of the rectangle.

Parameters

- **x_cent,y_cent** (*floats*) – the coordinates of the center of the rectangle.
- **width,height** (*float*) – The (half) width and height of the rectangle.

- **N** (*int*) – number of points to use along each edge.

Returns A list of points along the edge of the rectangle in the complex plane.

Potapov_Code.Roots.**get_max**(*y*)
return the $IQR + median$ to determine a maximum permissible value to use in the numerically safe function new_f_frac_safe.

Potapov_Code.Roots.**get_roots_rect**(*f*, *fp*, *x_cent*, *y_cent*, *width*, *height*, *N=10*, *outlier_coeff=100.0*, *max_steps=5*, *known_roots=None*, *verbose=False*)
I assume f is analytic with simple (i.e. order one) zeros.

TODO: save values along edges if iterating to a smaller rectangle extend to other kinds of functions, e.g. function with non-simple zeros.

Parameters

- **f** (*function*) – the function for which the roots (i.e. zeros) will be found.
- **fp** (*function*) – the derivative of f.
- **x_cent,y_cent** (*floats*) – The center of the rectangle in the complex plane.
- **width,height** (*floats*) – half the width and height of the rectangular region.
- **N** (*optional[int]*) – Number of points to sample per edge
- **outlier_coeff** (*float*) – multiplier for coefficient used when subtracting poles to improve numerical stability. See new_f_frac_safe.
- **max_step** (*optional[int]*) – Number of iterations allowed for algorithm to repeat on smaller rectangles.
- **roots** (*known*) – Roots of f that are already known.
- **verbose** (*optional[boolean]*) – print warnings.

Returns

A list of roots for the function f inside the rectangle determined by the values x_cent,y_cent,width, and height.

Potapov_Code.Roots.**inside_boundary**(*roots_near_boundary*, *x_cent*, *y_cent*, *width*, *height*)
Takes roots and the specification of a rectangular region returns the roots in the interior (and ON the boundary) of the region.

Parameters

- **roots_near_boundary** (*list of complex numbers*) – roots near the boundary.
- **x_cent,y_cent** (*floats*) – coordinates of the center of the region.
- **width,height** (*floats*) – The (half) width of height of the rectangle.

Returns Roots in the interior and on the boundary of the rectangle.

Potapov_Code.Roots.**linspace**(*c1*, *c2*, *num=50*)
make a linespace method for complex numbers.

Parameters

- **c1,c2** (*complex numbers*) – The two points along which to draw a line.
- **num** (*optional [int]*) – number of points along the line.

Returns a list of num points starting at c1 and going to c2.

Potapov_Code.Roots.**new_f_frac**(*f_frac*, *z0*, *residues*, *roots*, *val=None*)

Functions that evaluate the f_frac after some roots and their residues are subtracted. This function does NOT check to see if there is division by zero of if the values become too large.

We assume here that the poles are of order 1.

Parameters

- **f_frac** (*function*) – function for which roots will be subtracted.
- **z0** (*complex number*) – point where new_f_frac is evaluated.
- **residues** (*list of complex numbers*) – The corresponding residues to subtract.
- **roots** (*list of complex numbers*) – The corresponding roots to subtract.
- **val** (*optional[complex number]*) – We can impose a value f_frac(z0) if we wish.

Returns The new value of f_frac(z0) once the chosen poles have been subtracted.

Potapov_Code.Roots.**new_f_frac_safe**(*f_frac*, *z0*, *residues*, *roots*, *max_ok*, *val=None*, *verbose=False*)

Functions that evaluate the f_frac after some roots and their residues are subtracted. The safe version checks for large values and division by zero. If the value of f_frac(z0) is too large, subtracting the roots of f becomes numerically unstable. In this case, we approximate the new function f_frac by using the limit function.

We assume here that the poles are of order 1.

Parameters

- **f_frac** (*function*) – function for which roots will be subtracted.
- **z0** (*complex number*) – point where new_f_frac is evaluated.
- **residues** (*list of complex numbers*) – The corresponding residues to subtract.
- **roots** (*list of complex numbers*) – The corresponding roots to subtract.
- **val** (*optional[complex number]*) – We can impose a value f_frac(z0) if we wish.
- **max_ok** (*float) Maximum absolute value of f_frac(z0 to use*) –
- **verbose** (*optional[boolean]*) – print warnings.

Returns The new value of f_frac(z0) once the chosen poles have been subtracted.

Potapov_Code.Roots.**purge**(*lst*, *eps=1e-14*)

Get rid of redundant elements in a list. There is a precision cutoff eps.

Parameters

- **lst** (*list*) – elements.
- **eps** (*optional[float]*) – precision cutoff.

Returns A list without redundant elements.

Potapov_Code.Roots.**residues**(*f_frac*, *roots*)

Finds the resides of $f_{frac} = f'/f$ given the location of some roots of f. The roots of f are the poles of f_frac.

Parameters

- **f_frac** (*function*) – a complex.
- **roots** (*a list of complex numbers*) – the roots of f; poles of f_frac.

Returns A list of residues of f_frac.

2.6 Potapov_Code.Time_Sims module

Created on Mon Mar 2 15:52:32 2015

@author: gil @title: Time_Sims

Potapov_Code.Time_Sims.**f**(*t*, *y*, *A*, *B*, *force_func*, *forcing_port*)

Potapov_Code.Time_Sims.**plot_time**(*time*, *y*, *port_out*, *port_in*, *num=0*, *kind='FP'*, *format='pdf'*)

Potapov_Code.Time_Sims.**stack_func_port**(*force_func*, *forcing_port*, *t*, *max_size*)

Potapov_Code.Time_Sims.**test_stacking**()

Potapov_Code.Time_Sims.**time_sim**(*Example, omega=0.0, t1=150, dt=0.05, freq=None, port_in=0, port_out=[0, 1], kind='FP'*)

takes an example and simulates it up to t1 increments of dt. freq indicates the maximum frequency where we look for modes omega indicates the frequency of driving. omega = 0 is DC. port_in and port_out are where the system is driven.

2.7 Potapov_Code.Time_Sims_nonlin module

Created on Mon Mar 31 2015

@author: gil @title: Time_Sims_Nonlin.py

Potapov_Code.Time_Sims_nonlin.**make_f**(*eq_mot*, *B*, *a_in*)

Equations of motion, including possibly nonlinear internal dynamics.

Parameters

- **eq_mot** (*function*) – The equations of motion, which map $(t, a) \to v$. Here math:*t* is a scalar corresponding to time, a is an array of inputs correpsonding to the internal degrees of freedom, and v is a complex-valued column matrix describing the gradient.
- **B** (*matrix*) – The matrix multiplying the inputs to the system.
- **a_in** (*function*) – The inputs to the system.

Returns A function that maps $(t, a) \to f'(t, a)$, where t is a scalar (time), and a is an array representing the state of the system.

Return type (function)

Potapov_Code.Time_Sims_nonlin.**make_f_lin**(*A*, *B*, *a_in*)

Linear equations of motion

Parameters

- **A** (*matrix*) – The matrix for the linear equations of motion: $\frac{d}{dt}\begin{pmatrix} a \\ a^+ \end{pmatrix} = A\begin{pmatrix} a \\ a^+ \end{pmatrix} + B\breve{a}_{in}(t)$.
- **B** (*matrix*) – The matrix multiplying the inputs to the system.
- **a_in** (*function*) – The inputs to the system $\breve{a}$.

Returns (function); A function that maps $(t, a) \to f'(t, a)$, where t is a scalar (time), and a is an array representing the state of the system.

Potapov_Code.Time_Sims_nonlin.**run_ODE**(*f*, *a_in*, *C*, *D*, *num_of_variables*, *T=10*, *dt=0.01*, *y0=None*)

Run the ODE for the given set of equations and record the outputs.

Parameters

- **f** (*function*) – Evolution of the system.
- **a_in** (*function*) – inputs as a function of time.
- **C,D** (*matrices*) – matrices to use to obtain output from system state and input.
- **num_of_variables** (*int*) – number of variables the system has.
- **T** (*optional[positive float]*) – length of simulation.
- **dt** (*optional[float]*) – time step used by the simulation.

Returns An array Y of outputs.

Return type (array)

2.8 Potapov_Code.functions module

Created on Mon Mar 2 13:59:30 2015

@author: Gil Tabak @title: Potapov

Functions used by other files.

Potapov_Code.functions.**Pade**(*n*, *z*)
Pade pproximation of e^z

Parameters

- **n** (*integer*) – order of approximation
- **z** (*complex number*) – point of evaluation.

Returns

Return type Value of Pade approximation. (float)

Potapov_Code.functions.**Q**(*z*, *n*)
Numerator of Pade approximation of e^z.

Parameters

- **n** (*integer*) – order of approximation.
- **z** (*complex number*) – point of evaluation.

Returns

Return type Value of Numerator of Pade approximation. (float)

Potapov_Code.functions.**der**(*f*, *z*, *eps=1e-05*)
Estimate the derivative of the function f at z

Parameters

- **f** (*function*) – the function to use.
- **z** (*complex number*) – point at which to evaluate derivative.
- **eps** (*optional[complex number]*) – number to perturb z to find derivative.

Returns

Return type Derivative of f. (complex)

Potapov_Code.functions.**double_up**(*M1*, *M2=None*)

Takes a given matrix M1 and an optional matrix M2 and generates a doubled-up matrix to use for simulations when the doubled-up notation is needed. i.e.

$$\begin{pmatrix} M_1 & M_2 \end{pmatrix} \to \begin{pmatrix} M_1 & M_2 \\ M_2^{\#} & M_1^{\#} \end{pmatrix}$$

In the case M2 == None, it becomes replaced by the zero matrix.

Parameters

- **M1** (*matrix*) – matrix to double-up
- **M2** (*matrix*) – optional second matrix to double-up

Returns The doubled-up matrix.

Return type (complex-valued matrix)

Potapov_Code.functions.**factorial**(*n*)

Find the factorial of n.

Parameters **n** (*integer*) –

Returns

Return type factorial of n. (int)

Potapov_Code.functions.**gcd_lst**(*lst*)

Potapov_Code.functions.**inner_product_of_two_modes**(*root1*, *root2*, *v1*, *v2*, *delays*, *eps=1e-07*, *func=<function <lambda>>*)

This function integrates two spatial modes against each other along the various delays of the system. Each delay follows a node in the system.

The frequency is assumed to be the imaginary part of each root.

Parameters

- **root1,root2** (*complex number*) – the two roots.
- **v1,v2** (*column matrices*) – the amplitude of each mode at the various nodes.
- **delays** (*list of floats*) – The duration of each delay following each node in the system.
- **eps** (*optional[float]*) – cutoff for two frequencies being equal
- **func** (*optional[funciton]*) – used to transform the roots. Default value is set to lambda z: z.imag, meaning we take the frequency of each mode.

Returns Sanity check: if root1==root2 and v1==v2, returns real value.

Return type The inner product of the two modes. (complex)

Potapov_Code.functions.**limit**(*f*, *z0*, *N=10*, *eps=0.001*)

Takes possibly matrix-valued function f and its simple pole z0 and returns limit_{z o val} f(z). Estimates the value based on N surrounding points at a distance eps.

Parameters

- **f** (*function*) – the function for which the limit will be found.
- **z0** (*complex number*) – The value at which the limit is evaluated.
- **N** (*int*) – number of points used in the estimate.
- **eps** (*optional[float]*) – distance from z0 at which estimating points are placed.

Returns The estimated value of $limit_{z->z_0} f(z)$.

Return type Limit value (complex)

Potapov_Code.functions.**make_dict_values_to_lists_of_inputs**(*values*, *inputs*)

Make a dictionary mapping value to lists of corresponding inputs.

Parameters

- **values** (*list of floats*) – Values in a list, corresponding to the inputs.
- **inputs** (*list of floats*) – Inputs in a list.

Returns dictionary mapping value to lists of corresponding inputs.

Return type D (dict)

Potapov_Code.functions.**make_nonlinear_interaction**(*natural_freqs*, *modes*, *delays*, *delay_indices*, *start_nonlin*, *length_nonlin*, *plus_or_minus_arr*, *indices_of_refraction=None*, *eps=1e-05*)

This function takes several (say M) natural_freqs and their corresponding modes, as well as the (N) delay lengths of the network, and determines the term we need to add to the Hamiltonian corresponding to the resulting nonlinearity. We assume there is a crystal going from start_nonlin to and has length length_nonlin. The plus_or_minus_arr is an array of length m of 1 or -1 used to determined whether a mode corresponds to a creation (1, a^dag) or annihilation (-1,a) operator. The corresponding electric field integrated will be E^dag for 1 and E for -1.

The k-vectors are computed from the following formula: k = omega / v_p = omega n(omega) / c.

If the indices of refraction n(omega_i) are given, we use them to compute the phase-mismatch delta_k. Otherwise we assume they are all equal to 1.

Parameters

- **natural_freqs** (*list of floats*) – The natural frequencies of the various eigenmodes (i.e. the imaginary component of each root).
- **modes** (*list of column matrices*) – the amplitudes of the modes at various nodes.
- **delays** (*list of floats*) – The duration of each delay following each node in the system.
- **delay_indices** (*int OR list/tuple of ints*) – the index representing the delay line along which the nonlinearity lies. If given a list/tuple then the nonlinearity interacts the N different modes.
- **start_nonlin** (*float OR list/tuple of floats*) – the beginning of the nonlinearity. If a list/tuple then each nonlinearity begins at a different time along its corresponding delay line.
- **length_nonlin** (*float*) – duration of the nonlinearity in terms of length.
- **plus_or_minus_arr** (*array of 1s and -1s*) – Creation/annihilation of a photon in each of the given modes.
- **indices_of_refraction** (*float/int or list/tuple of float/int*) – the indices of refraction corresponding to the various modes. If float or int then all are the same.
- **eps** (*optional[float]*) – cutoff for two frequencies being equal.

Returns strength of nonlinearity.

Return type nonlinear interaction (complex)

Potapov_Code.functions.**make_normalized_inner_product_matrix**(*roots*, *modes*, *delays*, *eps=1e-12*, *func=<function <lambda>>*)

Given a list of roots and a list of vectors representing the electric field at each node of the corresponding nodes, compute the normalized matrix representing the inner products among the various modes.

TODO: add weights for different delays to account for geometry.

Parameters

- **roots** (*list of complex numbers*) – The roots of the various eigenmodes.
- **modes** (*list of column matrices*) – the amplitudes of the modes at various nodes.
- **delays** (*list of floats*) – The duration of each delay following each node in the system.
- **eps** (*optional[float]*) – cutoff for two frequencies being equal.
- **func** (*optional[funciton]*) – used to transform the roots. Default value is set to lambda z: z.imag, meaning we take the frequency of each mode.

Returns A matrix of normalized inner products representing the geometric overlap of the various given modes in the system.

Return type inner product matrix (complex-valued matrix)

Potapov_Code.functions.**pade_approx**(*n*)

Numerator coefficients of symmetric Pade approximation of math:*e^z* of order n.

Parameters **n** (*integer*) –

Returns

Return type Coefficients for Pade approximation numerator. (float)

Potapov_Code.functions.**pade_roots**(*n*)

Extract roots of Pade polynomial.

Parameters **n** (*integer*) –

Returns

Return type Roots of Pade polynomial. (list of complex numbers)

Potapov_Code.functions.**spatial_modes**(*roots*, *M1*, *E*, *delays=None*)

Obtain the spetial mode profile at each node up to a constant. If the delays are provided, the modes will be normalized using the delays. Otherwise, the modes will not be normalized.

Parameters

- **roots** (*list of complex numbers*) – The eigenvalues of the system.
- **M1** (*matrix*) – The connectivity matrix among internal nodes.
- **E** (*matrix-valued function*) – Time-delay matrix.
- **delays** (*optional[list of floats]*) – List of delays in the network.

Returns

Return type A list of spatial eigenvectors. (list of complex-valued column matrices)

Potapov_Code.functions.**timeit**(*method*)

from https://www.andreas-jung.com/contents/a-python-decorator-for-measuring-the-execution-time-of-methods

2.9 Potapov_Code.tests module

2.10 Module contents

Potapov_Code.**contour_plot**(*Mat*)

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

I

L

M

N

O

P

Q

R

S

T