SANDIA REPORT
SAND2021-1212
Printed February, 2021

Sandia National Laboratories

CSPlib - A Software Toolkit for the Analysis of Dynamical Systems and Chemical Kinetic Models

Oscar Diaz-Ibarra, Kyungjoo Kim, Cosmin Safta, Habib N. Najm

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods

ACKNOWLEDGMENT

The authors acknowledge contributions by Fazle Rob, currently at Insight Global, to the development of CSPlib.

This work is supported as part of the Computational Chemical Sciences Program funded by the U.S. Department of Energy, Office of Science, Basic Energy Sciences, Chemical Sciences, Geosciences and Biosciences Division. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

CSPlib is an open source software library for analyzing general ordinary differential equation (ODE) systems and detailed chemical kinetic ODE systems. It relies on the computational singular perturbation (CSP) method for the analysis of these systems. The software provides support for

- General ODE models (gODE model class) for computing source terms and Jacobians for a generic ODE system.
- TChem model (ChemElemODETChem model class) for computing source term, Jacobian, other necessary chemical reaction data, as well as the rates of progress for a homogenous batch reactor using an elementary step detailed chemical kinetic reaction mechanism. This class relies on the TChem [2] library.
- A set of functions to compute essential elements of CSP analysis (Kernel class). This includes computations of the eigensolution of the Jacobian matrix, CSP basis vectors and co-vectors, time scales (reciprocals of the magnitudes of the Jacobian eigenvalues), mode amplitudes, CSP pointers, and the number of exhausted modes. This class relies on the Tines library.
- A set of functions to compute the eigensolution of the Jacobian matrix using Tines library GPU eigensolver.
- A set of functions to compute CSP indices (Index Class). This includes participation indices and both slow and fast importance indices.

1.1. Nomenclature

Notation	Description
$\mathbf{y}$	State vector
$\mathbf{g}$	Source vector
t	Time
$\mathbf{a}_i$	CSP basis vector
$\mathbf{b}^i$	CSP basis co-vector
f^i	Mode amplitude
J_{ij}	Jacobian matrix of the ODE right hand side (RHS)
g_{fast}	ODE RHS component in the fast subspace
g_{slow}	ODE RHS component in the slow subspace
M	Number of fast exhausted modes
$\delta y^i_{\text{error}}$	Error for variable i
$\text{tol}_{\text{relative}}$	Relative error tolerance
$\text{tol}_{\text{absolute}}$	Absolute error tolerance
τ	Time scale

λ	Eigenvalues of Jacobian matrix
N_{spec}	Number of species
N_{reac}	Number of reactions
N_{var}	Number of variables
S	S matrix
$\mathscr{R}_r$	Rate of progress or reaction r
RoP	Rate of progess
CSPpointer$_{ij}$	CSP pointer for mode i with respect to variable j
$(I_r^i)_{\text{slow}}$	Slow importance index of reaction r for variable i
$(I_r^i)_{\text{fast}}$	Fast importance index of reaction r for variable i
P_r^i	Participation index of reaction r for mode i

2. BUILDING CSPLIB

CSPlib requires Tines and Kokkos for the computation of the eigendecomposition on GPU or CPU hardware, and for linear algebra operations. Additionally, CSPlib has an interface to TChem [2].

For convenience, we explain how to build the CSPlib code using the following environment variables that one can modify according to their working environment.

```
/// repositories
export CSP_REPOSITORY_PATH=/where/you/clone/csp/git/repo

/// build directories
export CSP_BUILD_PATH=/where/you/build/csp

/// install directories
export TCHEM_INSTALL_PATH=/where/you/install/tchem
export KOKKOS_INSTALL_PATH=/where/you/install/kokkos
export TINES_INSTALL_PATH=/where/you/install/tines

/// Tines requires OpenBlass
export LIBRARY_PATH=${LIBRARY_PATH}:=/where/you/install/OpenBlas/lib
```

2.1. Download CSPlib

Clone the CSPlib repository. Instructions on how to download and install TChem, Kokkos and Tines are found in the TChem repository [2].

```
```
git clone getz.ca.sandia.gov:/home/gitroot/csp ${CSP_REPOSITORY_PATH};
```
```

2.2. Configuring CSPlib

The following example cmake script compiles CSPlib on the host, linking with Tines.

```
cmake \
    -D CMAKE_INSTALL_PREFIX=${CSP_INSTALL_PATH} \
    -D CMAKE_CXX_COMPILER="${my_cxx}" \
    -D CMAKE_C_COMPILER="${my_cc}" \
    -D KOKKOS_INSTALL_PATH=${KOKKOS_INSTALL_PATH} \
    -D TINES_INSTALL_PATH=${TINES_INSTALL_PATH} \
    ${CSP_REPOSITORY_PATH}/src
make -j install
```

The following cmake example compiles CSPlib with TChem. CSPlib uses TChem to compute source terms, the Jacobian of the source term and the S matrix and the rate of progress. TChem requires Kokkos github pages [1] and Tines. Therefore, these libraries must also be installed.

```
cmake \

    -D CMAKE_INSTALL_PREFIX=${CSP_INSTALL_PATH} \
    -D CMAKE_CXX_COMPILER="${my_cxx}" \
    -D CMAKE_C_COMPILER="${my_cc}" \
    -D CSP_ENABLE_TCHEMPP=ON \
    -D TCHEM_INSTALL_PATH=${TCHEM_INSTALL_PATH}\
    -D KOKKOS_INSTALL_PATH=${KOKKOS_INSTALL_PATH} \
    -D TINES_INSTALL_PATH=${TINES_INSTALL_PATH} \
    ${CSP_REPOSITORY_PATH}/src
make install
```

TChem is designed and implemented using Kokkos (a performance portable parallel programming model), thus, CSPlib can also carry out computations on a GPU. For GPUs, we can use the above cmake script and replace the compiler choice by adding:
`-D CMAKE_CXX_COMPILER="${KOKKOS_INSTALL_PATH}/bin/nvcc_wrapper"`.

3. CSP BASIC CONCEPTS

3.1. Formulation

Consider the autonomous ODE system in $\mathbb{R}^N$:

$$\frac{d\boldsymbol{y}}{dt} = \boldsymbol{g}(\boldsymbol{y}) \tag{3.1.1}$$

With the initial value $\boldsymbol{y}(t=0) = \boldsymbol{y}_0$.

Where $\boldsymbol{y}$ is a vector of state variables. For example, for a chemical kinetic model in a homogeneous gas phase constant pressure system, this can be comprised of the gas temperature and the mass fractions for the gas species. The right hand side (RHS) $\boldsymbol{g}(\boldsymbol{y})$ vector is a function of the state vector $\boldsymbol{y}$.

CSP analysis is primarily useful in the context of stiff dynamical systems exhibiting a wide range of fast/slow time scales. The goal of the analysis is to decouple fast and slow processes, thereby enabling specific dynamical diagnostic capabilities, by rewriting the system RHS using a suitable set of basis vectors [10]. CSP analysis seeks a set of basis vectors $\boldsymbol{a}_i$, $i = 1,\ldots,N$, that linearly expand $\boldsymbol{g}$ [10]:

$$\boldsymbol{g} = \sum_{i=1}^{N} \boldsymbol{a}_i f^i \tag{3.1.2}$$

where f^i is the (signed) “amplitude" of $\boldsymbol{g}$ as projected on the basis vector $\boldsymbol{b}_i$,

$$f^i = \boldsymbol{b}^i \cdot \boldsymbol{g} \tag{3.1.3}$$

and where the $\boldsymbol{b}^i$ vectors are, by construction, orthonormal to the $\boldsymbol{a}_i$ vectors,

$$\boldsymbol{b}^i \cdot \boldsymbol{a}_i = \delta^i_j. \tag{3.1.4}$$

Given the $\boldsymbol{a}_i$ CSP basis vectors, the associated co-vectors $\boldsymbol{b}^i$ are computed using the orthonormality constraint (Eq. 3.1.4), and mode amplitudes f^i (Eq. 3.1.3). CSP provides a refinement procedure to construct the basis vectors $\boldsymbol{a}_i$ [10,19]. Alternatively, the right eigenvectors of the Jacobian $J_{ij} = \frac{\partial g_i}{\partial y_j}$ provide a first order approximation of the ideal CSP $\boldsymbol{a}_i$ basis vectors. For a linear ODE system, the eigensolution perfectly decouples the fast and the slow time scales of $\boldsymbol{g}$. For a nonlinear system it provides only approximate decoupling. This library uses the Jacobian eigenvectors as the CSP basis vectors. Given that we are dealing with real, generally non-symmetric, Jacobian matrices, we can expect that any complex eigenvalues will be complex conjugate pairs, and similarly for the

associated eigenvectors. When a pair of modes are complex conjugates, we do not use the complex eigenvectors as CSP basis vectors, rather we use two real eigenvectors that span the same plane. Thus we always have real CSP basis vectors.

We order the eigenmodes in terms of decreasing eigenvalue magnitude $|\lambda_i|$, Thus in order of increasing time scales $\tau_i = 1/|\lambda_i|$,

$$\tau_1 < \tau_2 < \cdots < \tau_N$$

so that mode 1 is the fastest mode, mode 2 is the next slower mode, etc.

Typically, chemical kinetic ODE models exhibit a number of fast decaying eigenmodes, associated with eigenvalues having large magnitudes (small timescales) with negative real components. These modes exhibit fast decay towards a slow invariant manifold developed from the equilibration of fast exhausted processes. Typical dynamics in systems that evolve towards an equilibrium involve a gradual increase in the number of fast exhausted modes, as successive time scales are exhausted, and the system approach the equilibrium point.

At any point in time, presuming M fast exhausted modes, we split $\boldsymbol{g}$ into slow and fast components:

$$\boldsymbol{g} = \underbrace{\sum_{i=1}^{M} \boldsymbol{a}_i f^i}_{g_{\text{fast}} \approx 0} + \underbrace{\sum_{i=M+1}^{N} \boldsymbol{a}_i f^i}_{g_{\text{slow}}} \tag{3.1.5}$$

where M defines the dimension of the fast subspace. It is computed as the maximum M for which

$$\delta y^i_{\text{fast}} = \left| \sum_{r=1}^{M} a^i_r f^r \frac{e^{\lambda^r_{\text{real}} \tau^\kappa} - 1}{\lambda^r_{\text{real}}} \right| < \delta y^i_{\text{error}} \tag{3.1.6}$$

Where $\kappa = \min(M+1, N)$. Note that $\delta y^i_{\text{error}}$ is critical to calculate M. We estimate $\delta y^i_{\text{error}}$ with employing absolute and relative tolerances,

$$\delta \boldsymbol{y}_{\text{error}} = \text{tol}_{\text{relative}} |\boldsymbol{y}| + \text{tol}_{\text{absolute}} \tag{3.1.7}$$

In equation 3.1.6, $\tau = \frac{1}{|\lambda|}$ is the time scale, and λ is an eigenvalue.

With the CSP basis vectors we can also compute the CSP pointers. The CSP pointers identify the degree of orthogonality between the dimension of each species in the configuration space and the equation of state constraint developed out of the exhaustion of each of the fast modes [10]. The pointer for mode i and species j is defined as:

$$\text{CSPpointer}_{ij} = a_{ij} b_{ij} \tag{3.1.8}$$

The equations presented above outline the basics of CSP. Detailed mathematical derivations and description of the method are presented in [9–12]. Example applications of CSP in combustion and other fields are presented in [3, 5–8, 13–15, 17–22]

3.2. CSP Indices

The following definitions for CSP indices are relevant for an elementary reaction based chemical kinetic mechanism, involving N_s species and N_r reactions. The model is presumed to involve $N = N_s + 1$ state variables, being the temperature T, and the mass fractions of the species.

We start by writing the RHS $\boldsymbol{g}$ as the product of the $N \times \mathfrak{R}$ matrix S, which is the generalized stoichiometric matrix, and the vector $[\mathscr{R}_1, \ldots, \mathscr{R}_{\mathfrak{R}}]$, where $\mathscr{R}_k$ is the rate of progress for elementary reaction k. By construction, we treat each reaction as reversible, thus we have $\mathfrak{R} = 2N_r$ reactions. In this context, an irreversible reaction is assigned a zero-rate in the opposite direction. Thus, we write $\boldsymbol{g}$ as

$$\boldsymbol{g} = \sum_{k=1}^{\mathfrak{R}} S_k \mathscr{R}_k \tag{3.2.1}$$

where S_k is the k-th column of S.

The S matrix is defined by $S = [Q\mathscr{S}, Q\mathscr{S}]$, where $\mathscr{S}$ is the $(N_s \times N_r)$ matrix of stoichiometric coefficients. For a constant pressure, homogeneous batch reactor, the $(N \times N_s)$ matrix Q is defined by:

$$Q = \begin{bmatrix} -\frac{1}{\rho c_p} W_1 h_1 & -\frac{1}{\rho c_p} W_2 h_2 & \cdots & -\frac{1}{\rho c_p} W_{N_s} h_{N_s} \\ \frac{1}{\rho} W_1 & 0 & \cdots & 0 \\ 0 & \frac{1}{\rho} W_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \frac{1}{\rho} W_{N_s} \end{bmatrix}$$

where ρ is density and c_p is specific heat at constant pressure of the gas mixture, h_k is the enthalpy of species k, and W_k is the molar mass of species k.

The rate of progress is defined by $\mathscr{R}_k = [q_{\text{fwd},1}, ..., q_{\text{fwd},N_r}, -q_{\text{rev},1}, ..., -q_{\text{rev},N_r}]$. Where $q_{\text{fwd},k}$ and $q_{\text{rev},k}$ are the forward and reverse rates of progress of reaction k.

With the definition of the amplitude of the $i-th$ mode f_i.

$$f_i = \boldsymbol{b}^i \cdot \boldsymbol{g} = \sum_{k=1}^{\mathfrak{R}} \beta_k^i \mathscr{R}_k \tag{3.2.2}$$

$$\beta_k^i = \boldsymbol{b}^i \cdot S_k \tag{3.2.3}$$

3.2.1. CSP Slow Importance Index

The CSP representation of the source term in the slow subspace is given by

$$g_{\text{slow}} = \sum_{i=M+1}^{N} \boldsymbol{a}_i f^i = \sum_{i=M+1}^{N} \boldsymbol{a}_i \sum_{k=1}^{\mathfrak{R}} \beta_k^i \mathscr{R}_k = \sum_{k=1}^{\mathfrak{R}} \alpha_k \mathscr{R}_k \tag{3.2.4}$$

where

$$\boldsymbol{\alpha}_k = \sum_{i=M+1}^{N} \boldsymbol{a}_i \beta_k^i \tag{3.2.5}$$

and $\boldsymbol{\alpha}_k = (\alpha_k^1, \ldots, \alpha_k^N)$. The slow importance index of reaction k with respect to state variable j is defined as:

$$(I_k^j)_{\text{slow}} = \frac{\alpha_k^j \mathscr{R}_k}{\sum_{r=1}^{\Re} |\alpha_r^j \mathscr{R}_r|} \tag{3.2.6}$$

3.2.2. CSP Fast Importance Index

The CSP representation of the source term in the fast subspace is given by

$$g_{\text{fast}} = \sum_{i=1}^{M} a_i f^i = \sum_{i=1}^{M} \boldsymbol{a}_i \sum_{k=1}^{\Re} \beta_k^i \mathscr{R}_k = \sum_{k=1}^{\Re} \boldsymbol{\gamma}_k \mathscr{R}_k \tag{3.2.7}$$

where

$$\boldsymbol{\gamma}_k = \sum_{i=1}^{M} \boldsymbol{a}_i \beta_k^i \tag{3.2.8}$$

with $\boldsymbol{\gamma}_k = (\gamma_k^1, \ldots, \gamma_k^N)$. The fast importance index of reaction k with respect to state variable j is defined as:

$$(I_k^j)_{\text{fast}} = \frac{\gamma_k^j \mathscr{R}_k}{\sum_{r=1}^{\Re} |\gamma_r^j \mathscr{R}_r|} \tag{3.2.9}$$

3.2.3. CSP Participation Index

The Participation Index of the $k-th$ reaction in the $i-th$ mode is defined as

$$P_k^i = \frac{\beta_k^i \mathscr{R}_k}{\sum_{r=1}^{\Re} |\beta_r^i R_r|} \tag{3.2.10}$$

4. APPLICATION PROGRAMMING INTERFACE

A typical CSPlib analysis involves the following steps:

1. Model class
    a) Compute : Source terms or RHS.
    b) Compute : Jacobian of RHS.
    c) Compute : Rate of progress.
    d) Compute : S matrix.
2. Kernel class
    a) Compute : Eigenvalues and eigenvectors
    b) Sort : Eigenvalues and eigenvaluesvectors.
    c) Set : Matrix whose columns are the CSP basis vectors (right eigenvectors of Jacobian), and its inverse matrix.
    d) Compute : Amplitudes of modes.
    e) Compute : Time scales.
    f) Compute : Number of exhausted modes.
    g) Compute : Compute CSP pointers.
3. Index class
    a) Compute : Participation indices.
    b) Compute : Slow Importance indices.
    c) Compute : Fast Importance indices.

4.1. Model Class

The model class is responsible for computing the source term (RHS) of the system and its Jacobian matrix. Additionally, the model class will compute the S matrix and the reaction rates of progress for a chemical kinetic model. We have two available model classes, the general ODE class (gODE) and the TChem model class.

4.1.1. General ODE Class (gODE)

The general ODE class (`CSP_REPOSITORY_PATH/src/core/gODE.cpp`) can handle any ODE system. This class requires a function for RHS and the Jacobian matrix.

For example, for the Davis-Skodje problem [4, 20] , the RHS and Jacobian functions are:

```
int rhs_Davis_Skodje(const std::vector<double>& state, std::vector<double>& source){
    const double epsilon = 0.01;
    const double y = state[0];
    const double z = state[1];
    source[0] = (-y+z/(1.+z))/epsilon - z/(1.+z)/(1.+z);
    source[1] = -z;
  return(0);
}

int jac_Davis_Skodje(const std::vector<double>& state, std::vector<std::vector<double>>& jac, int
     flag){
  const double epsilon = 0.01;
  const double y = state[0];
  const double z = state[1];

  jac[0][0] = -1./epsilon;
  jac[1][0] = 0;
  jac[0][1] = 2. * z / std::pow( z + 1. , 3.) - 1. / std::pow( z + 1., 2) +
              ( - z / std::pow( z + 1. , 2.) + 1. / ( z + 1. ) ) / epsilon;
  jac[1][1] = -1;
  return(0);
}
```

We pass these two functions to the gODE class.

```
/// Constructor takes two functions.
GeneralODE mDavis_Skodje(
  std::function<int(const std::vector<double>&, std::vector<double>&)> (std::move(
      rhs_Davis_Skodje)),
  std::function<int(const std::vector<double>&, std::vector<std::vector<double>>&, int)> (std::
      move(jac_Davis_Skodje))
);
```

To evaluate the RHS, and Jacobian we do the following:

```
//set state vector
mDavis_Skodje.setStateVector(state);
//eval rhs
mDavis_Skodje.evalSourceVector();
//get g
mDavis_Skodje.getSourceVector(source);
//eval Jacobian
mDavis_Skodje.evalJacMatrix(flag);
// get Jacobian
mDavis_Skodje.getJacMatrix(jac);
```

4.1.2. TChem Model Class

The TChem model class (`CSP_REPOSITORY_PATH/src/core/chem_elem_ODE_TChem.cpp`) computes the source term, the Jacobian matrix, the rate of progress, and the S matrix for an elementary step chemical kinetic reaction mechanism. This class is a collection of function calls to TChem. TChem is designed and implemented with the Kokkos library. Thus, these computations

can be performed in CPUs:

("Kokkos::DefaultHostExecutionSpace")

or in GPUs:

("Kokkos::DefaultExecutionSpace").

The default execution space is configured as OpenMP or Cuda upon its availability. The default host execution spaces is always configured as OpenMP. Besides, this class performs the computation in a batched mode.

To create an instance of this class, we use :

```
/// Constructor takes two input files.
///   [in] mech_gas_file - Chemkin reaction mechanism file
///   [in] thermo_gas_file - Thermo file
ChemElemODETChem( const std::string &mech_gas_file     ,
                  const std::string &thermo_gas_file   )
```

The Chemkin-input files contain all the parameters of the reaction mechanism.

We can use the TChem model class to read an entire solution from the TChem homogenous batch reactors. This reactor resolves gas temperature and mass fraction in a transient batch reactor.

```
//
///   [in] filename - database filename
///   [out] varnames - vector with variable names from TChem solution
 ChemElemODETChem::readIgnitionZeroDDataBaseFromFile(const std::string &filename,
                           std::vector<std::string> &varnames) ;
```

If we choose to run the computation on the device (GPU), the:

`readIgnitionZeroDDataBaseFromFile`

function will copy and move the data to the GPUs. Alternatively, if we want to run the computation on the host space (CPU), we use the function call `model.run_on_host(true)`.

```
[in] run_on_host: true, run on host space , false, run on execution space
ChemElemODETChem::run_on_host(const bool & run_on_host)
```

If we do not have a TChem's database, we need to pass our database to the model class with the following function.

```
/// [in] state_db: database for CSP analysis
ChemElemODETChem::setStateVectorDB(std::vector<std::vector <double> >& state_db)
```

The database is a 2D std::vector where the rows are the solution for each time. The columns correspond to the "time or iteration, density [kg/m^3], pressure [Pascal], Temperature[K], mass fractions".

With the uploaded database, the following function calls compute the source therms, the Jacobian matrix, the S matrix, and the rate of progress.

```
ChemElemODETChem::evalSourceVector();
/// [in] useNumJac: 0 use analytical Jacobian, 1 use numerical jacabian
ChemElemODETChem::evalJacMatrix(unsigned int useJacAnl);
ChemElemODETChem::evalSmatrix();
ChemElemODETChem::evalRoP();
```

The TChem model class copies the computed data to the host space. To obtain the data from this class, we use the following functions:

```
/// [out] state_db :  state vector for the whole database
ChemElemODETChem::getStateVector(std::vector<std::vector <double> >& state_db);

/// [out] source_db: source vector for the whole database
ChemElemODETChem::getSourceVector(std::vector<std::vector <double> >& source_db);

// [out] jac_db : Jacobian matrix for the whole database
ChemElemODETChem::getJacMatrix(std::vector <std::vector
                    <std::vector <double> > >& jac_db);

// [out] RoP: rate of progress for the whole database
ChemElemODETChem::getRoP(std::vector<std::vector <double> >& RoP);

// [out] Smatrixdb: S matrix for the whole database
ChemElemODETChem::getSmatrix(std::vector < std::vector
                        <std::vector <double> > >& Smatrixdb);
```

The state and source vectors have a size of $N = N_s + 1$, involving temperature and mass fractions, the size of the Jacobian matrix is $N \times N$, the size of S matrix and rate of progress vector is $N \times 2N_r$ and $2N_r$ respectively. The rate of progress vector includes the forward and reverse rate of progress.

This class has additional functions to help post-process the CSP data.

```
/// [out] return the number of species
ChemElemODETChem::NumOfSpecies()
/// [out] return the number of reactions
ChemElemODETChem::NumOfReactions()
/// [out] spec_name: name of species in the reaction mechanism
ChemElemODETChem::getSpeciesNames(std::vector<std::string>& spec_name)
/// [in] var_name: variable name, use "Temperature" for temperature
///[out] return index of the variable in the CSP analysis.
ChemElemODETChem::getVarIndex(const std::string & var_name)
/// [out] return number of variables in the CSP analysis
ChemElemODETChem::getNumOfVariables()
/// [out] return  number of elements
ChemElemODETChem::getNumOfElements()
```

4.2. Kernel Class

The second group of steps are implemented in the kernel class:
(`CSP_REPOSITORY_PATH/src/core/kernel.cpp`).
This class computes the eigendecomposition for the Jacobian matrix, the time scales $\tau = \frac{1}{|\lambda|}$, the number of exhausted model (M), the $\boldsymbol{a}$ and $\boldsymbol{b}$ CSP basis vectors, the amplitude of the modes f^i and the CSP pointers.

We initialize this class with the number of variables, the g (source) vector, and the Jacobian matrix(jac).

```
/// The constructor takes four inputs.
///   [in] nvars - number of state variable
///   [in] state_vec - y vector of state vector
///   [in] source_vec - g vector or rhs vector
///   [in] Jmat - Jacobian matrix of g
Kernel(int nvars,
       std::vector<double> &state_vec,
       std::vector<double> &source_vec,
       std::vector< std::vector<double> > &Jmat
      )
```

This class calculates the eigendecomposition of the Jacobian matrix. Next, It sorts the eigenvalues in descending order with respect to their magnitudes. With the sorted eigenvalues and eigenvectors, it sets the right eigenvectors as the $\boldsymbol{a}$ CSP basis vectors, and form the matrix $\boldsymbol{A}$ whose columns are the $\boldsymbol{a}$ vectors. The matrix $\boldsymbol{B}$, whose rows are the $\boldsymbol{b}$ vectors, is the inverse of $\mathbf{A}$ (see equation 3.1.4). The matrix inversion is done by Tines.

```
/// Computation of eigendecomposition
/// This function does not have inputs. The Jacobian matrix is a private member of the kernel
    class.
Kernel::evalEigenValVec();
// sort eigenvalues in descending order, we use new order to sort eigenvectors as well.
Kernel::sortEigValVec();

//Set CSP basis vectors.
Kernel::setCSPVec(); // A = eig_vec_R and B = A^{-1}
//get CSP basis vector csp_vec_R(a) csp_vec_L(b).
Kernel::getCSPVec(csp_vec_L, csp_vec_R);
```

The time scales are computed as $\tau = \frac{1}{|\lambda|}$, where λ is the magnitude of the eigenvalues. The amplitude of the mode f^i is computed with equation 3.1.3.

```
// compute time scale.
Kernel::evalTau();
/// [out]  tauvec - time scales
Kernel::getTau(std::vector<double> &tauvec);
// compute the magnitude of the modes.
Kernel::evalModalAmp( );
/// [out] fvec - magnitud of the modes
Kernel::getModalAmp(std::vector<double> &fvec );
```

The number of exhausted modes M is computed using relative and absolute tolerances (see [equation 3.1.6]) and a state vector. The tolerances are inputs of the analysis. The value of M cannot be bigger than $N - N_{\text{elements}} - 1$, or the number of eigenvalues with negative real component.

```
/// [in] csp_rtolvar - relative tolerance for CSP analysis.
/// [in] csp_atolvar - absolute tolerance for CSP analysis.
Kernel::setCSPerr(double csp_rtolvar, double csp_atolvar);
/// [in] nel- number of elements in the reaction mechanism or system
Kernel::evalM(const int &nElem);
/// [out] number of exhausted (M).
Kernel::getM(int &NofDM);
```

The CSP pointers (equation 3.1.8) for all modes are computed by:

```
Kernel::evalCSPPointers();
To obtain the CSP pointer data from the kernel class we use:
\begin{lstlisting}[language=bash]
/// [out] cspp_ij - csp pointers; row
Kernel::getCSPPointers( std::vector<std::vector<double>> &cspp_ij );
```

We can also use the function:

```
/// [in] modeIndx - mode element position
/// [out] cspp_k - CSP pointer position for mode with element position modeIndx
Kernel::evalAndGetCSPPointers(const int & modeIndx, std::vector<double> &cspp_k)
```

To compute the CSP pointers for one mode.

At this point, the kernel class has computed all CSP data for a basic ODE system. Among this data, the time scales (τ), the amplitude of the modes (f), the CSP basis vectors $\boldsymbol{a}$ and $\boldsymbol{b}$, the eigenvalues and eigenvectors of the system, the number of exhausted modes M, and the CSP pointers.

Additionally, the kernel class has diagnostic tools to test if the CSP data is not corrupted by numerical error.

The numerical rank of the Jacobian is used to check how many of the eigenvalues are reliably computed. The number of valid eigenvalues is equal to the numerical rank. Thus, if a Jacobian is not full rank, the smallest eigenvalues are essentially numerical noise.

```
///[out] return the numerical rank of the Jacobian matrix
Kernel::computeJacobianNumericalRank()
```

We check the eigensolution only for the valid eigenvalues, according to the numerical rank of the Jacobian.

```
//If a residual bigger than 1e-6 is obtained. " ---- High residual --- " will print out.
Kernel::DiagEigValVec();
```

We also check the orthonormality condition for the CSP basis vector.

```
// If a residual bigger than 1e-10 is obtained. ": --- Orthogonality test failed: .." will print
    out.
Kernel::DiagOrthogonalityCSPVec();
```

4.2.1. EigenSolver With Tines

CSPlib has four different interfaces to Tines' eigensolver depending on the execution spaces and the input type. The first interface performs the eigensolution on the GPUs (CUDA, device execution space), and the inputs are in Kokkos-view format allocated in the GPUs memory space. The second interface carries out the computation on the CPUs (OPENMP, host execution spaces) and the inputs also in Kokkos-view format. The third interface uses the GPUs with the inputs in 3D std::vector format. Finally, in the fourth interface, the computation is performed in CPUs, and the inputs are in 3D std::vector format.

The input of these interfaces is a database of Jacobians. The outputs are the real and imaginary part of the eigenvalues and the right eigenvectors for the whole database in 3D std vectors format.

The function to call the GPU's interface with Kokkos-view type is the following:

```
///    [in] jac - database of Jacobians - data is allocated on the GPU
///    [out] eig_val_real_bath - real part eigenvalues of database
///    [out] eig_val_imag - imaginary part eigenvalues of database
///    [out] eig_vec_R - right eigenvectors of database
EigenSolver::evalDevice(const value_type_3d_view& jac,
                        std::vector<std::vector <value_type> >& eig_val_real,
                        std::vector<std::vector <value_type> >& eig_val_imag,
                        std::vector < std::vector<std::vector <value_type> > >& eig_vec_R);
```

The function to call the CPU's interface with Kokkos-view:

```
///    [in] jac - database of Jacobians - data is allocated on the CPU
///    [out] eig_val_real_bath - real part eigenvalues of database
///    [in] eig_val_imag - imaginary part eigenvalues of database
///    [in] eig_vec_R - right eigenvectors of database
EigenSolver::evalHost(const value_type_3d_view_host& jac,
                        std::vector<std::vector <value_type> >& eig_val_real,
                        std::vector<std::vector <value_type> >& eig_val_imag,
                        std::vector < std::vector<std::vector <value_type> > >& eig_vec_R);
```

The function to call the GPU's interface with 3D std::vectors:

```
///   [in] jac - database of Jacobians
///   [out] eig_val_real_bath - real part eigenvalues of database
///   [out] eig_val_imag - imaginary part eigenvalues of database
///   [out] eig_vec_R - right eigenvectors of database
EigenSolver::evalDevice(const std::vector < std::vector<std::vector <value_type> > >& jac,
                        std::vector<std::vector <value_type> >& eig_val_real,
                        std::vector<std::vector <value_type> >& eig_val_imag,
                        std::vector < std::vector<std::vector <value_type> > >& eig_vec_R);
```

The function to call the CPU's interface with 3D std::vectors:

```
///   [in] jac - database of Jacobians - data exists on the host
///   [out] eig_val_real_bath - real part eigenvalues of database
///   [in] eig_val_imag - imaginary part eigenvalues of database
///   [in] eig_vec_R - right eigenvectors of database
EigenSolver::evalHost(const std::vector < std::vector<std::vector <value_type> > >& jac,
                      std::vector<std::vector <value_type> >& eig_val_real,
                      std::vector<std::vector <value_type> >& eig_val_imag,
                      std::vector < std::vector<std::vector <value_type> > >& eig_vec_R);
```

4.3. Index Class

To instantiate the index class, we need nine inputs from the model and kernel classes.

```
/// Constructor takes eight inputs.
///   [in] Nreac - number of reactions
///   [in] Nvar - number of variables
///   [in]  M - number of exhausted modes
///   [in] eig_val_real - eigenvalues real part
///   [in] eig_val_imag - eigenvalues imaginary part
///   [in] A - a CSP basis vector
///   [in] B - b CSP basis vector
///   [in] Smat - S matrix
///   [in] RoP  - rate of progress
CSPIndex(
      int Nreac,
      int Nvar,
      int M,
      std::vector<double> &eig_val_real,
      std::vector<double> &eig_val_imag,
      std::vector<std::vector<double> > &A,
      std::vector<std::vector<double> > &B,
      std::vector<std::vector<double> > &Smat,
      std::vector<double> &RoP
     )
```

The following functions compute the Participation indices (equation 3.2.10), and the slow and fast Importance indices ([equations 3.2.6 and 3.2.9) for all variables and modes for one state vector.

```
CSPIndex::evalParticipationIndex();
CSPIndex::evalImportanceIndexSlow();
CSPIndex::evalImportanceIndexFast();
```

To obtain the data produced by the above function:

```
/// [out] P_ik - Participation index; rows: modes, columns: rate of progress
CSPIndex::getParticipationIndex (std::vector<std::vector<double> > &P_ik  );
/// [out] Islow_jk - Slow importance index; rows: variable, columns: rate of progress
CSPIndex::getImportanceIndexSlow( std::vector<std::vector<double> > &Islow_jk );
/// [out] Ifast_jk- Fast importance index;  rows: variables, columns: rate of progress
CSPIndex::getImportanceIndexFast(std::vector<std::vector<double> > &Ifast_jk );
```

Sometimes, one only wants to compute the index for a few modes/variables. In this case, one can use the following functions:

```
/// [in] modeIndx - index (position) for mode
/// [out] P_k - Participation index for mode with indx modeIndx
CSPIndex::evalAndGetParticipationIndex(const int &modeIndx, std::vector<double> &P_k);
/// [in] varIndx - index (position) for variable
/// [out] Islow_k - Slow importance index for variable with index varIndx
CSPIndex::evalAndGetImportanceIndexSlow(const int & varIndx, std::vector<double> &Islow_k);
/// [in] varIndx - index (position) for variable
/// [out] Ifast_k -  Fast importance index for variable with index varIndx
CSPIndex::evalAndGetImportanceIndexFast(const int & varIndx, std::vector<double> &Ifast_k);
```

The `CSPIndex::getTopIndex` function returns an std::vector<int> with the reaction number (in the the rate of progress vector) for the highest absolute value Participation and slow/fast Importance indices.

```
/// [in] Index - Participation/slow/fast index for one mode or variable
/// [in] Top - only add top absolute values.
/// [in] threshold- only add values bigger than this threshold value.
/// [in/out] IndxList- list of reaction number in the RoP(rate of progress) vector.
CSPIndex::getTopIndex(std::vector<double> &Index,
                const int & Top, const double & threshold,
                std::vector<int> & IndxList );
```

For example, to find out which reactions have the highest contribution in the fastest mode, one can use this function and pass the participation index for mode 0. This participation index (std::vector) is obtained with the function:

`CSPIndex::evalAndGetParticipationIndex`, with `modeIndx=0`.

Alternatively, one can use the `CSPIndex::evalParticipationIndex` function, and get the Participation indices for all modes with `CSPIndex::getParticipationIndex`. The output of the function `CSPIndex::getTopIndex` is `IndxList`, which is a vector containing the reaction number in the rate of progress vector.

5. EXAMPLES

5.1. CSP Analysis For The Davis Skodje Problem Using The General ODE Class

We use the gODE class (`CSP_REPOSITORY_PATH/src/core/gODE.cpp`) to analyze the Davis-Skodje (DS) problem [4,20] to illustrate the utility of CSPlib in analyzing stiff ODE systems.

This is a two dimensional ODE system,

$$\begin{aligned} \frac{dy}{dt} &= \frac{1}{\varepsilon}\left(-y+\frac{z}{1+z}\right)-\frac{z}{(1+z)^2} \\ \frac{dz}{dt} &= -z \end{aligned}$$

With the initial condition $y(0)=y_o, z(0)=z_0$. Where the ε parameter is constant and much smaller than one. In this system, z is the slow variable, and y is the fast variable. The analytical solution of this problem is:

$$\begin{aligned} y(t) &= \left(y_0-\frac{z_0}{1+z_0}\right)e^{-t/\varepsilon}+\frac{z_0e^{-t}}{1+z_0e^{-t}} \\ z(t) &= z_0e^{-t} \end{aligned}$$

The example code for this problem is in:
`CSP_REPOSITORY_PATH/src/example/gODE/driver_gODE_Davis_Skodje.cpp`

Usage information is available:

```
./driver_gODE_Davis_Skodje --help
Usage: ./driver_gODE_Davis_Skodje [options]
  options:
  --atol                          double    absolute tolerance for csp analysis e.g., 1e-8
                                            (default: --atol=1.0e-14)
  --echo-command-line             bool      Echo the command-line but continue as normal
  --help                          bool      Print this help message
  --nPoints                       int       number of points  e.g., 2000
                                            (default: --nPoints=2000)
  --rtol                          double    relative tolerance for csp analysis e.g., 1e-2
                                            (default: --rtol=1.0e-03)
  --tend                          double    time end e.g., 4
                                            (default: --tend=4.0e+00)
  --y0                            double    initial value for y e.g., 2
                                            (default: --y0=2.0)
  --z0                            double    initial value for y e.g., 1e-2
                                            (default: --z0=1.0e-02)
Description:
  This example Number of exhausted and time scale for mDavis Skodje problem
```

The following bash script runs a CSP analysis for the DS problem:

```
exec=$CSP_INSTALL_PATH/example/kernel_class/driver_gODE_Davis_Skodje.exe
rtol=1e-4
atol=1e-14
y0=2.
z0=1.
tend=15.
nPoints=10000
$exec --tend=$tend --y0=$y0 --nPoints=$nPoints --z0=$z0 --rtol=$rtol --atol=$atol
```

The above script and a jupyter-notebook with the below figures is located at:
`CSP_INSTALL_PATH/example/runs/Davis_Skodje`

The inputs from the script are:

- the executable for this example ("driver_gODE_Davis_Skodje"),
- the relative and absolute tolerances for csp analysis ("rtol" and "atol"),
- the initial condition for the problem "y0" and "z0",
- the final time "tend" and the number of points that we want to produce for the analysis "nPoints".

The example has the following structure:

```
 //set the GeneralODE with the mDavis_Skodje rhs and Jacobian

 // make a list of file to save data

//for example
 std::string mNew_file_name = firstname + "_m.dat";
 FILE *fout = fopen ( (mNew_file_name).c_str(), "w" );


 // make a for loop over the nPoints

 for (int sp = 0; sp < nPoints; sp++) {

  // set state vector: from analytical solution

  // compute source terms and Jacobian

  // make an instance of the kernel class

  // compute eigenvalues and eigenvectors

  // set CSP basis vectors

  // sort eigenvalues and eigenvectors

  // compute time scales

  // compute modal amplitude

  // compute M

  // compute csp pointers

 }

 fclose(fout)
```

We save data for each time iteration (nPoints), the data correspond to: the number of exhausted modes M "_m.dat" (nPoints), the time scales ($2\times$ nPoints) "_tau.dat", the numerical rank of the Jacobian ($2\times 2\times$ nPoints) "_jac_numerical_rank.dat", the amplitude of the modes "_magMode.dat"

(2×nPoints), the state vector "_state.dat" (2×nPoints), and time ("_time.dat"). With these data we produce Figures (5-1)–(5-5) below.

The CSP analysis reveals characteristics of the DS problem that could not identify from its source term equation. The analysis shows a fast and a slow time scale (see Figure 5-1), where the slow time scale is 1e-2 s, and the fast time scale is 1 s. From Figure 5-2, we can notice that between time 0 s to 1e-1 s, there are zero exhausted modes (M). In this time interval, τ_{M+1} is equal to the slow time scale (Figure 5-1), which means the process advance at the slow time scale because all processes are active. Between 1e-1 s and 30 s, M changes to 1, in this time interval τ_{M+1} also changes to the fast time scale. At the end of this period, both curves become constant, M is 2, denoting that the system is in equilibrium. Similarly, the mode amplitude curves in the Figures 5-4 and 5-5 display the behavior describe above. The mode amplitude f_0 reaches equilibrium at 1e-1 s and the mode amplitude f_1 reaches equilibrium at 30 s M.

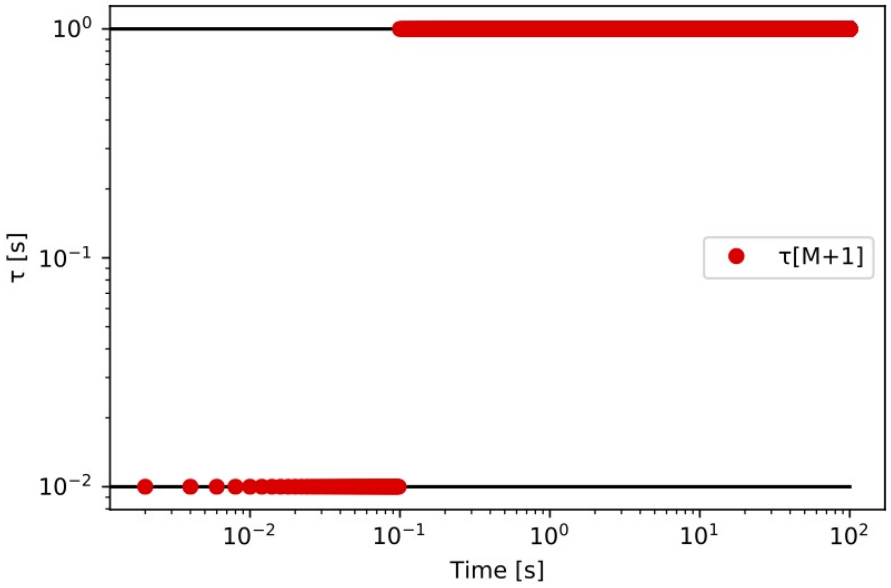

Figure 5-1. Time scales versus time for the DS problem. Red dots correspond to τ_{M+1}, the time scale of the fastest active mode.

5.1.1. CSP Analysis For The Davis Skodje Problem Using Tines (GPU) EigenSolver

The computation of the eigenvalues and eigenvectors is one of the most computational expensive parts of the CSP analysis. Thus, CSPlib offers an interface for the Tines GPU eigensolver(see 4.2.1). We use this interface to compute the eigensolution for DS problem. The example code is in: `CSP_REPOSITORY_PATH/src/example/gODE/driver_gODE_Davis_Skodje_K.cpp`

The structure of the DS example with Tines' eigensolver is:

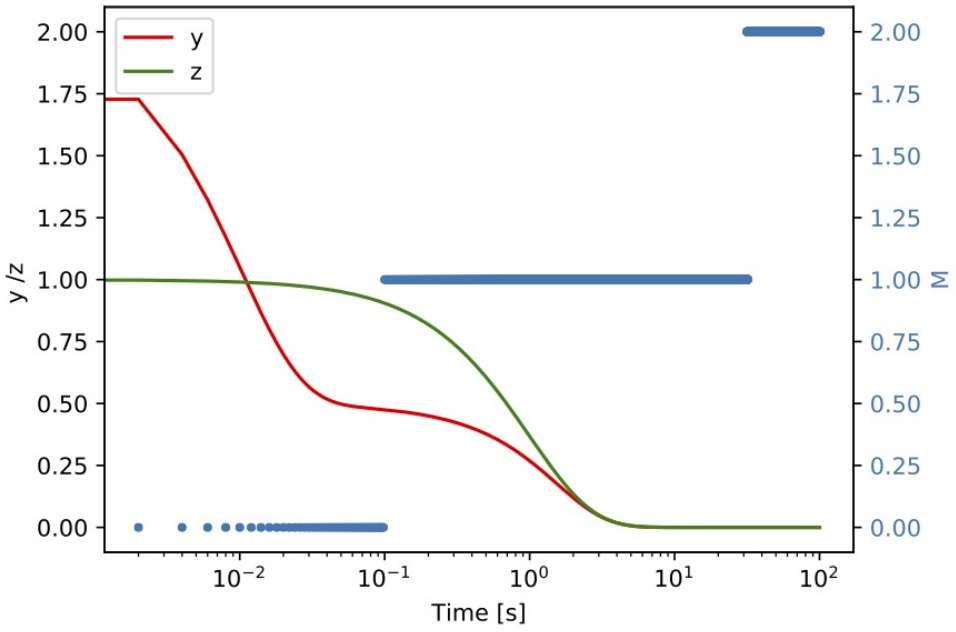

Figure 5-2. A plot of (y,z) (left axis) and M (right axis) versus time for the DS problem.

```
CSP::ScopeGuard guard(argc, argv);

//set the GeneralODE with the mDavis_Skodje rhs and Jacobian

// make a list of file to save data

//for example
std::string mNew_file_name = firstname + "_m.dat";
FILE *fout = fopen ( (mNew_file_name).c_str(), "w" );


// make a for loop over the nPoints

for (int sp = 0; sp < nPoints; sp++) {

 // set state vector: from analytical solution

 // compute source terms and Jacobian

}

 // compute eigenvalues and eigenvectors using Tines GPU EigenSolver

 EigenSolver::evalDevice(jac_db,
                         eig_val_real_bath,
                         eig_val_imag_bath,
                         eig_vec_R_bath);

for (int sp = 0; sp < nPoints; sp++) {

 // make an instance of the kernel class

 // set eigenvalues and eigenvectors

 // sort eigenvalues and eigenvectors
```

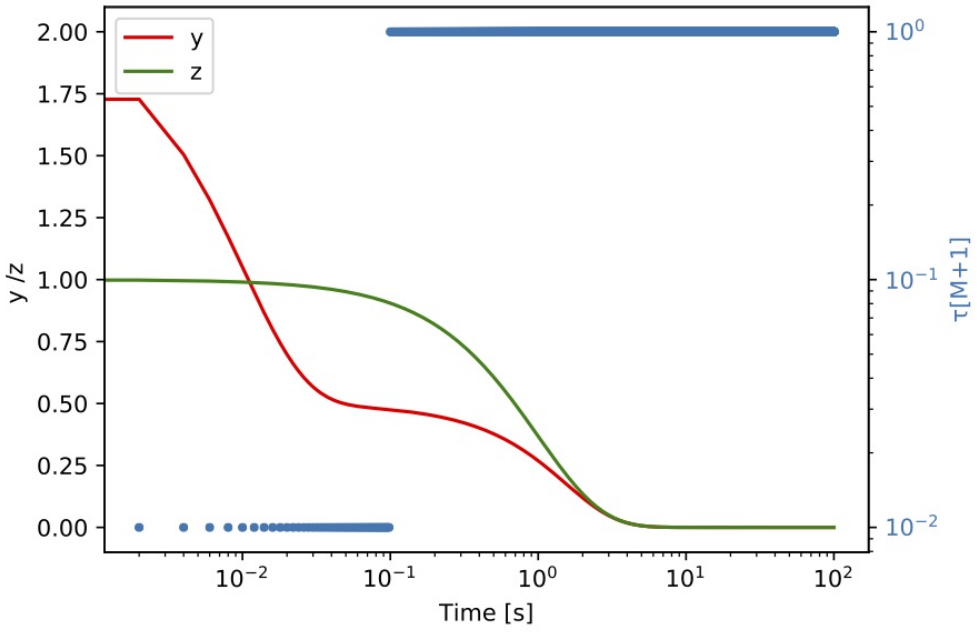

Figure 5-3. A plot of (y,z) (left axis) and τ_{M+1} (right axis) versus time for the DS problem.

```
 // set csp basis vector

 // compute time scales

 // compute modal amplitude

 // compute M

}

fclose(fout)
```

We need to add "ScopeGuard" at the top of the code. This scope guard initializes Kokkos when the program begins and also finalizes Kokkos when the program ends. The ScopeGuard is a simple struct:

```
struct ScopeGuard {
    ScopeGuard(int argc, char** argv) { Kokkos::initialize(argc, argv); }
    ~ScopeGuard() { Kokkos::finalize(); }
  };
```

We split the loop over the whole database into three parts. In the first part, we compute the source and Jacobians, and we store it in 3D std vectors. In the second part, we pass the Jacobians to the Tines' eigensolver.

```
 EigenSolver::evalDevice(jac_db,
                         eig_val_real_bath,
                         eig_val_imag_bath,
                         eig_vec_R_bath);
```

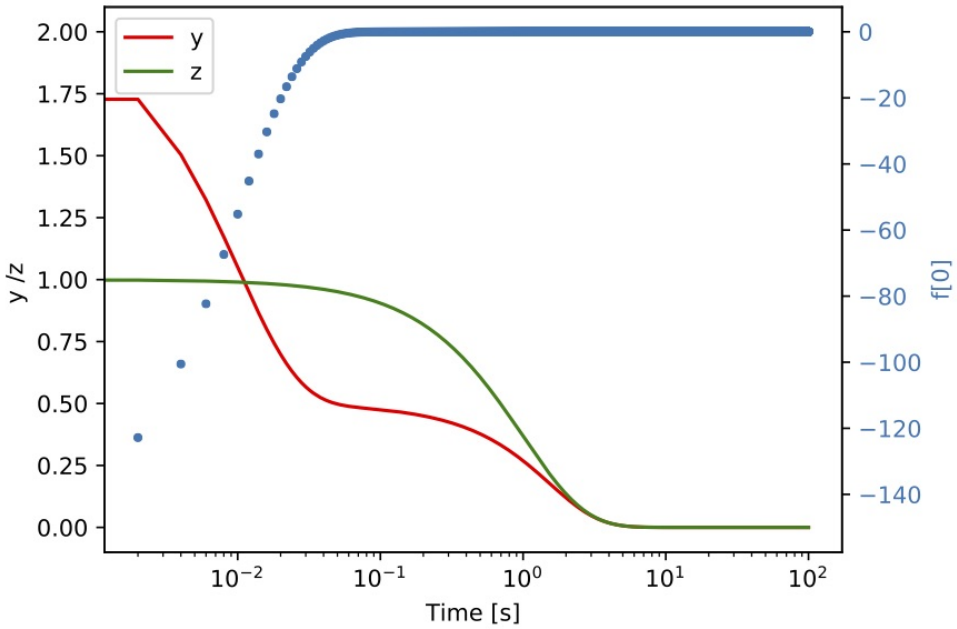

Figure 5-4. A plot of (y,z) (left axis) and f^0 (right axis) versus time for the DS problem.

Finally, in the third part, we feed the eigenvalues and eigenvectors into the kernel class using the function "ker.setEigenValVec", and then we compute the CSP data.

```
ker.setEigenValVec(eig_val_real, eig_val_imag, eig_vec_R);
// Sorting eigen values and vectors
// of, sign(eig_val_real)*Mod(eig_val_real + i * eig_val_imag)
ker.sortEigValVec();
```

The results of this example are exactly the same to the DS example presented above.

5.2. CSP Analysis For a ODE System Using TChem

The source term, Jacobian, S matrix, and rate of progress (RoP) vector computed by the TChem model class corresponds to a homogeneous reactor [2]. This reactor consists of one equation for temperature and N_s equations for the species mass fractions.

The source code for this example is at:
`CSP_REPOSITORY_PATH/src/example/indexODETChem/run_index_ODE_TChem.cpp`
The executable is installed at:
`CSP_INSTALL_PATH/example/index_class/run_index_ODE_TChem.exe`

The inputs are as follows:

```
./run_index_ODE_TChem.exe --help

Usage: ./run_index_ODE_TChem.exe [options]
```

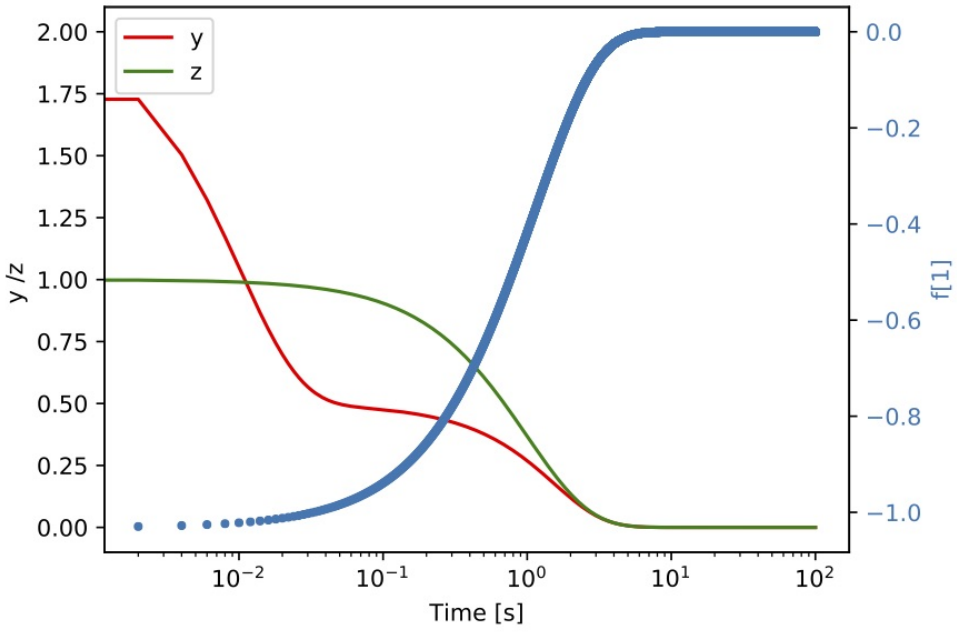

Figure 5-5. A plot of (y,z) (left axis) and f^1 (right axis) versus time for the DS problem.

```
  options:
  --atol                          double    absolute tolerance for csp analysis e.g., 1e-8
                                            (default: --atol=1.0e-08)
  --chemfile                      string    Chem file name e.g., chem.inp
                                            (default: --chemfile=chem.inp)
  --echo-command-line             bool      Echo the command-line but continue as normal
  --help                          bool      Print this help message
  --inputfile                     string    database file name e.g., input.dat
                                            (default: --inputfile=input.dat)
  --prefix                        string    prefix to save output files e.g., pos_
                                            (default: --prefix=)
  --rtol                          double    relative tolerance for csp analysis e.g., 1e-2
                                            (default: --rtol=1.0e-02)
  --thermfile                     string    Therm file name e.g., therm.dat
                                            (default: --thermfile=therm.dat)
  --useTChemSolution              bool      Use a solution produced by TChem e.g., true
                                            (default: --useTChemSolution=true)
Description:
  This example carries out a csp analysis with TChem model class
```

We tested this example with the GRImech3.0 reaction mechanism [16]. We used TChem with the homogeneous reactor to produce a database of state vectors. The script to run this example and a jupyter-notebook for post-processing are located at:

`CSP_INSTALL_PATH/example/runs/GRI3`.

We use the following bash script to run this example:

```
exec=$CSP_INSTALL_PATH/example/index_class/run_index_ODE_TChem.exe
inputs=data/
chemfile=$inputs"chem.inp"
thermfile=$inputs"therm.dat"
inputfile=$inputs"input.dat"
useTChemSolution=true
```

```
prefix=csp_output/
rtol=1e-6
atol=1e-10
$exec --useTChemSolution=$useTChemSolution --chemfile=$chemfile --thermfile=$thermfile --
    inputfile=$inputfile --rtol=$rtol --atol=$atol --prefix=$prefix
```

The inputs are:

- the Chemkin file names “chemfile” for the reaction mechanism,
- the “thermfile” for the thermodynamic data, and
- the database file name “inputfile”.

We can use any filename for the input files. In this case, we use “chem.inp”, “therm.dat”, and “input.dat”, and the files are located in the directory “data”. To use a solution produced by TChem, we set “useTChemSolution=true” (Note that we have to run TChem to produce the database). The “prefix” is for the output files, for example, we used “prefix=csp_output/”. So, CSPlib will save the output files at the “csp_output/” directory (we need to create this directory). If the “prefix” is not specified, CSPlib writes the files in the local directory. Finally, we use “rtol” and “atol” to pass the absolute and relative tolerances for the CSP analysis.

This example is structured as follows:

```
// create a TChem model instantiation.
// read the database
// compute source therm (rhs), Jacobian, S matrix and RoP
// get data from device or host

The TChem class computes the data in batched mode.

// make a list of files to save data
std::string m_file_name = firstname + "_m.dat";
fout = fopen ( (m_file_name).c_str(), "w" );


for (int sp = 0; sp < nSample; sp++) {
 // make an instance of the kernel class

 // compute eigenvalues and eigenvectors

 // sort eigenvalues and eigenvectors

 // set CSP basis vectors

 // compute time scales

 // compute M

 // compute f

// make an instance of the index class

// compute indices

// save data at each iteration
fprintf(fout," %d \n", NofDM);

}

//close files
fclose(fout);
```

5.2.1. GRI 3.0 Results

We ran the script presented above, and CSPLib saved the data at "csp_output/". CSPlib will not write output files. However, in this example, we have collected data to plot results and further analysis. We recommend using this example as a template. Thus, the users can delete or add data depending on their requirements.

5.2.1.1. Model Class

We create an instantiation of the TChem model with two chemical files, as we described in section 4.1.2:

```
ChemElemODETChem  model(chemFile, thermFile);
```

The example can read a database produced by TChem ("useTChemSolution=true") or a database generated by another library.

```
if (useTChemSolution) {
      // read a database from the TChem homogeneous reactor
      std::vector<std::string> var_names;
      model.readIgnitionZeroDDataBaseFromFile(inputFile,var_names);
    } else{
      // read a database that was not produced by TChem
      std::vector<std::vector <double> > state_db_read;
      // Density, pressure, temperature and species mass fraction
      const int numofStateVariables = 3 + model.NumOfSpecies();
      readDataBase(inputFile, state_db_read, numofStateVariables );
      model.setStateVectorDB(state_db_read);
    }
```

The source term, Jacobian, S matrix, and RoP for the entire database is computed by:

```
 //computes RHS
model.evalSourceVector();

//computes Jacobian
model.evalJacMatrix(0);

 //compute Smatrix
 model.evalSmatrix();

 // compute RoP
  model.evalRoP();
```

These computations are done in a batched mode and executed in the CPU or the GPU. To obtain the data from the TChem model class, we use the following functions.

```
/*get data from model class to perform csp analysis*/
 std::vector< std::vector< double> > state_db;
 model.getStateVector(state_db);

 std::vector< std::vector< double> > source_db;
 model.getSourceVector(source_db);

std::vector< std::vector< std::vector< double> > > jac_db;
model.getJacMatrix(jac_db);

std::vector< std::vector< double> > RoP_db;
model.getRoP(RoP_db);
```

```
std::vector< std::vector< std::vector< double> > > Smatrixdb;
model.getSmatrix(Smatrixdb);
```

Additionally, the TChem model class has functions to obtain auxiliary quantities, such as N_{var} (ndiff_var), number of reactions (nReactions) and number of elements(nElem).

```
// get number of variables in the ODE system
 auto ndiff_var = model.getNumOfVariables();

const auto nReactions = model.NumOfReactions();
// we split the net RoP in fwd and rev rate
// if a reaction is irreversible one rate is set to zero
const auto nTotalReactions = 2*nReactions;

const int nElem = model.getNumOfElements();
```

5.2.1.2. Kernel Class

The kernel and index classes do not perform batched computations. Thus, we created a for loop to iterate the database. The kernel class is instantiated with the N_{var}, the state vector, source term, and the Jacobian as we described in section 4.2.

```
for (int i = 0; i < nSample; i++) {
     // data from TChem model class
     source = source_db[i];
     state  = state_db[i];
     jac    = jac_db[i];
     Smat   = Smatrixdb[i];
     RoP    = RoP_db[i];

     // instantiation of kernel class
      Kernel ker(ndiff_var, state, source, jac);
     ....

}
```

The kernel class computes the eigensolution, sorts the eigenvalues and eigenvectors, sets the CSP basis vectors, and calculates the mode amplitude. These computations are the core of the CSP analysis.

```
// Eigen solution:
 ker.evalEigenValVec();

 // Sorting eigen values and vectors in ascending order
 ker.sortEigValVec();

// Setting CSP vectors:
ker.setCSPVec(); // A = eig_vec_R and B = A^{-1}

// Compute mode amplitude
ker.evalModalAmp( );
```

The time scales for the ODE system is computed by `ker.evalTau()` and the data is obtained by `ker.getTau(tau_vec)` for one time step. We saved the time scales at every time step in the file "_tau.dat". In this file, the number of elements is the product of N_{var} and the number of time steps.

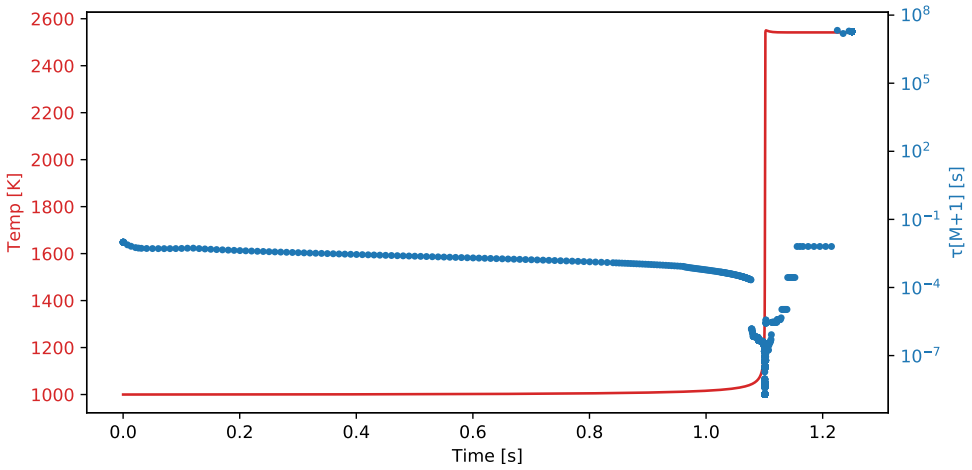

Figure 5-6. τ_{M+1} (blue, right y-axis) and temperature (red, left y-axis) versus time, for the GRI3.0 problem.

The number of exhausted modes M is computed by `ker.evalM(nElem)` and we obtained the data with `ker.getM(NofDM)`. M is saved at every time step in the file "_m.dat". The number of elements in this file is equal to the number of time steps in the database. The time-profile of M is presented in Figure 5-7. We also plot the gas temperature on the left y-axis for reference and easier interpretation of the analysis results.

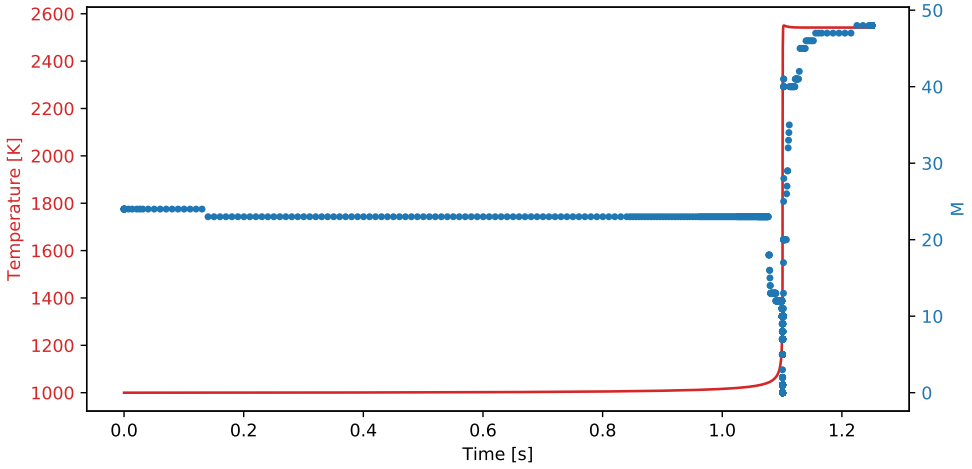

Figure 5-7. The number of exhausted modes M (blue, right y-axis) and temperature (red, left y-axis), plotted versus time, for the GRI3.0 problem.

We plot all time scales against time in Figure 5-6. Note that τ_{rank} is the time scale evaluated at the numerical rank of the Jacobian, the numerical rank is saved in the file "_jac_numerical_rank.dat", and it is computed by
`int jac_rank = ker.computeJacobianNumericalRank()`. We can use the numerical rank to check which eigenvalues are unreliable/invalid. In this case, all the time scales above the green curve are dominated by numerical noise. Therefore, these time scales should not

be considered in the analysis.

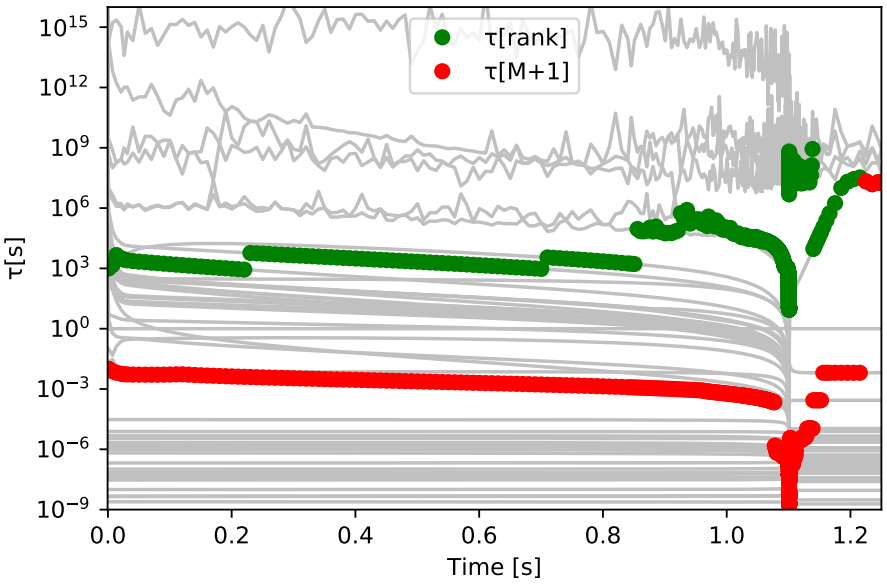

Figure 5-8. Time scales versus time for the GRI3.0 problem.

The CSP pointers for all modes were saved in the file "_cspPointers.dat". The data shows that mode 0 points at NNH, as shown in Figure 5-9. We produce this file with functions:

`ker.evalCSPPointers()` and
`ker.getCSPPointers( cspp_ij )`

as we describe in section 4.2. The amplitude of modes can have negative or positive values. In this figure, we plot the absolute value of the amplitude of mode 0. The absolute amplitude is close to zero in the whole database except near the ignition point. The matrix produced by the `ker.getCSPPointers( cspp_ij )` has a size of $N_{\text{var}} \times N_{\text{var}}$, we saved this matrix in the file for each time step, thus the size of this file is the number of time steps times $N_{\text{var}} \times N_{\text{var}}$. We load this data in the jupyter-notebook and reshape this matrix to further analysis.

```
Pointers = np.loadtxt(firstname +"_cspPointers.dat")
Ptrs = np.reshape(Pointers,[NtimeStep,Nvar,Nvar])
```

To find the element position of the variables that each mode points to, we used the python function `getTopIndex`, which is in the `CSP_INSTALL_PATH/example/runs/scripts/CSPindexHelper.py` script.

We can compute the CSP pointers for a given mode using the function `ker.evalAndGetCSPPointers(cspp_k)` as we described in section 4.2. In this example, we save the CSP pointer data to the file "_Mode0_cspPointers.dat" for the mode 0.

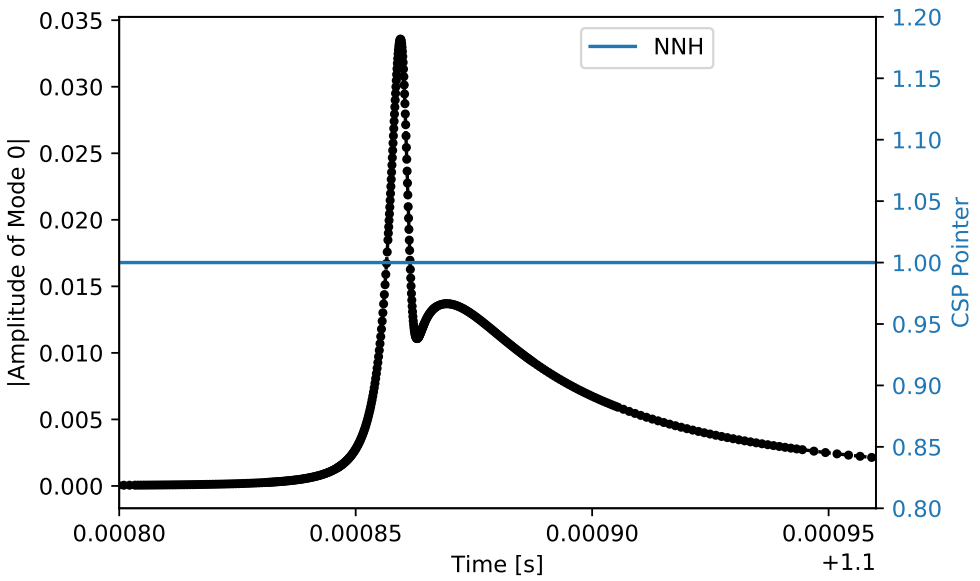

Figure 5-9. CSP pointers for mode 0 (right y-axis) and absolute amplitude of mode 0 versus time (left y-axis) for the GRI3.0 problem. Mode 0 points at NNH**. The time axis includes a short-time-interval around the ignition time.**

5.2.1.3. Index Class

We instantiate the index class with inputs from the TChem model class (nTotalReactions, ndiff_var, Smat, RoP) and from the kernel class NofDM, eig_val_real, eig_val_imag, csp_vec_R_2d, csp_vec_L_2d) as described in section 4.3.

```
// instantiate CSP Index class
 CSPIndex idx(nTotalReactions, ndiff_var,
                  NofDM, eig_val_real, eig_val_imag,
                  csp_vec_R_2d, csp_vec_L_2d, Smat, RoP );
```

We implemented two alternatives to compute the CSP indices. In the first approach, we calculate the index for all variables/modes, and in the second approach, we only calculate one variable/mode.

First approach An example use of the first approach is as in the following.

```
//First approach indices for all variable and mode are computed by
//eval index class data
idx.evalParticipationIndex();
idx.evalImportanceIndexSlow();
idx.evalImportanceIndexFast();
```

```
//get data
idx.getParticipationIndex ( P_ik     );
idx.getImportanceIndexSlow( Islow_jk );
idx.getImportanceIndexFast( Ifast_jk );
```

This last set of functions return the Participation and the slow/fast Importance indices. We save these results in the files "_SlowImportanceIndex.dat" for the slow Importance index (Eq. 3.2.6), "_FastImportanceIndex.dat" for the fast Importance index (Eq. 3.2.9), and "_ParticipationIndex.dat"

for the Participation index (Eq. 3.2.10). These files can be easily read by a python/numpy script, for example:

```
Slowind = np.loadtxt(firstname +"_SlowImportanceIndex.dat")
St = np.reshape(Slowind,[NtimeStep,Nvar,NtotalReactions])

Fastind = np.loadtxt(firstname +"_FastImportanceIndex.dat")
Ft = np.reshape(Fastind,[NtimeStep,Nvar,NtotalReactions])

PIind = np.loadtxt(firstname +"_ParticipationIndex.dat")
Pt = np.reshape(PIind,[NtimeStep,Nvar,NtotalReactions])
```

We plotted the slow/fast Importance indices for temperature and CO in Figures (5-10)–(5-13). The list of reactions in these figures corresponds to the reactions with indices having the first and second highest absolute value. Additionally, we only selected values higher than 1e-2. To obtain these lists of reactions (reaction number in the RoP vector), we used the python function `getTopIndex`, which is in the `CSP_INSTALL_PATH/example/runs/scripts/CSPindexHelper.py` script. We could produce similar plots for all variables for the fast/slow importance indices.

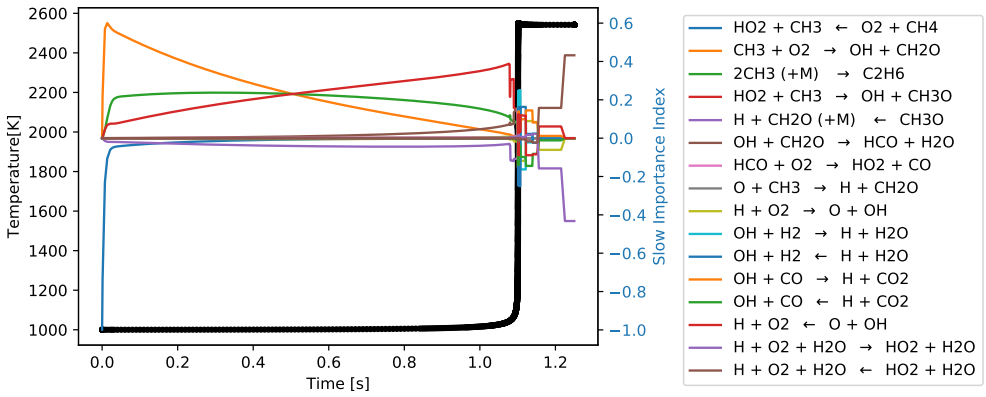

Figure 5-10. Temperature (black, left y-axis), and the slow Importance indices for temperature (right y-axis), versus time, for the GRI3.0 problem. The list of reactions corresponds to the top two reactions for each iteration and with an index (absolute value) bigger than threshold=1e-2.

We plotted the Participation index of mode 0 in Figure 5-14. Note that NNH is involved in three reactions that have a high value of the participation index for mode 0, which is consistent with the CSP pointers for mode 0 that also pointed at this species (see above).

As we can see in Figures (5-10)–(5-14), the values of the Participation and fast/slow Importance indices are in [-1,1] range. The sum over a point in time in these figures is not always one because we only plot the top reactions. However, the sum over indices of all reactions in absolute value is one (see Eqs. 3.2.6, 3.2.9, and 3.2.10). Note that, while the source term and Jacobian of the ODE system evolve smoothly in time, the analysis is always local at each instant in time. Since the eigenvectors are indeterminate up to multiplication by a ($\pm$) constant, and, further, given the step-wise variation of the integer M, the various indices can exhibit step changes as seen in these plots.

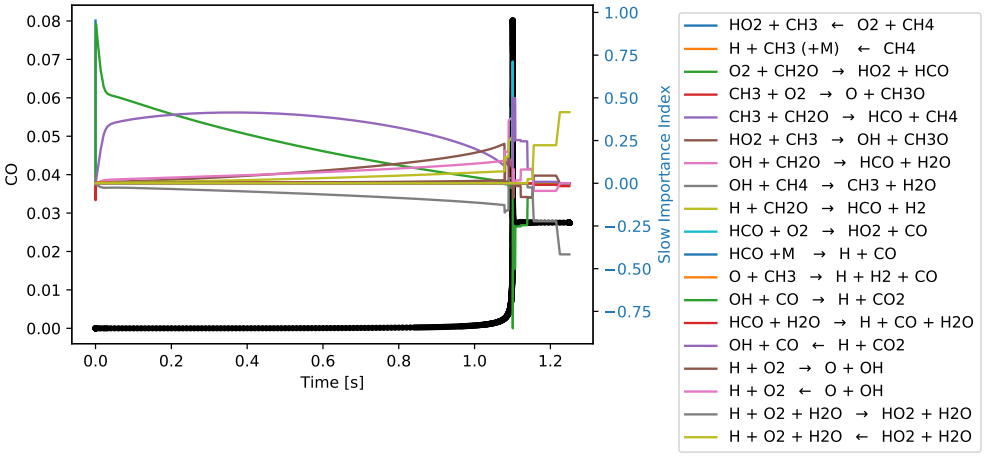

Figure 5-11. Mass fraction of CO (black, left y-axis), and the Slow importance indices for CO (right y-axis), versus time, for the GRI3.0 problem. The list of reactions corresponds to the top two reactions for each iteration and with an index (absolute value) bigger than threshold=1e-2.

Second approach There are cases where we only want to obtain CSP analysis results for a specific variable. Thus, we do not want to compute indices for the whole set of variables, only the variables of interest. To do this, we can use the following functions (see section 4.3):

```
/* eval and get participation index for one mode*/
int modeIndx(0);
idx.evalAndGetParticipationIndex(modeIndx, P_k);

/* eval and get slow importance index for one variable */
idx.evalAndGetImportanceIndexSlow(indxCH4, Islow_k_ch4);

/* eval and get fast importance index for one variable */
idx.evalAndGetImportanceIndexFast(indxCH4, Ifast_k_ch4);
```

In the above functions, we only computed the Participation index of mode 0, and the slow/fast Importance indices of CH_4. We obtained the species index of the CH_4 variable in the state vector using `ChemElemODETChem::getVarIndex(var_name)`. We saved the above indices for all iterations in the files: "_Mode0_ParticipationIndex.dat" for the Participation index of mode 0, "_CH4_FastImportanceIndex.dat" and "_CH4_SlowImportanceIndex.dat" for the slow and Importance indices of CH_4 species. Further, we obtained the top reactions with the function `CSPIndex::getTopIndex` described in section 4.3.

```
example for CH4 and mode 0
/* get top rate of progess */
idx.getTopIndex(P_k, Top_rop, threshold_rop,  IndxListPart );
idx.getTopIndex(Islow_k_ch4, Top_rop, threshold_rop,  IndxListch4 );
idx.getTopIndex(Ifast_k_ch4, Top_rop, threshold_rop,  IndxListFastch4 );
```

The function `CSPIndex::getTopIndex` produces a "std::vector<int>" with the reaction number for reactions that are in the top values(in this case top_rop=2) and with a CSP index with absolute value bigger than 1e-2 (threshold_rop=1e-2). We saved these "std::vector<int>" in the files:

"_Mode0_ParticipationIndexTopElemPosition.dat" for the Participation index of mode 0,

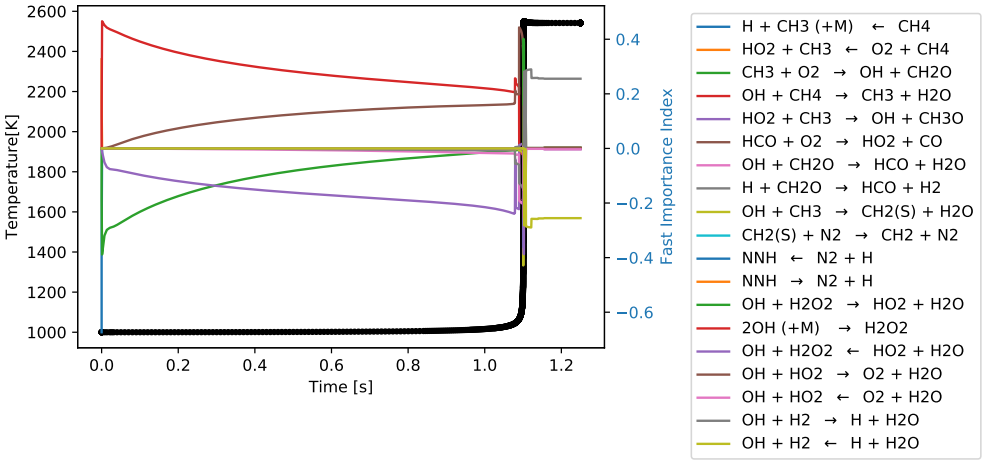

Figure 5-12. Temperature (black, left y-axis), and the Fast importance indices for temperature (right y-axis), versus time, for the GRI3.0 problem. The list of reactions corresponds to the top two reactions for each iteration and with an index (absolute value) bigger than threshold=1e-2.

"_CH4_FastImportanceIndexTopElemPosition.dat" for the fast Importance index of CH_4,
"_CH4_SlowImportanceIndexTopElemPosition.dat" for the slow Importance index of CH_4.
With these files, the plots in Figures (5-16,5-17) are produced.

5.2.2. CSP Analysis Using The Tines EigenSolver

CSPlib has an interface for the Tines eigensolver for GPU computation (see section 4.2.1). We create an example using this solver and the source code is at `CSP_REPOSITORY_PATH/src/example/indexODETChem/run_index_ODE_TChem_K.cpp`. The executable is installed at `CSP_INSTALL_PATH/example/index_class/run_index_ODE_TChem_K.exe`. This example is the same as the above example for ODE except for the computation of eigenvalues and eigenvectors.

To call the Tines eigensolver:

```
std::vector< std::vector< double> >  eig_val_real_bath;
std::vector< std::vector< double> >  eig_val_imag_bath;
std::vector< std::vector< std::vector< double> > > eig_vec_R_bath;

EigenSolver::evalDevice(model._jac,
                        eig_val_real_bath,
                        eig_val_imag_bath,
                        eig_vec_R_bath);
```

The "model._jac" is a public attribute of the TChem model class. This attribute is allocated on the device memory space and represents the Jacobians for the whole database. The outputs of this function are the real and imaginary parts of the eigenvalues and the right eigenvectors.

With the eigenvalues and eigenvector, we loop over the whole database:

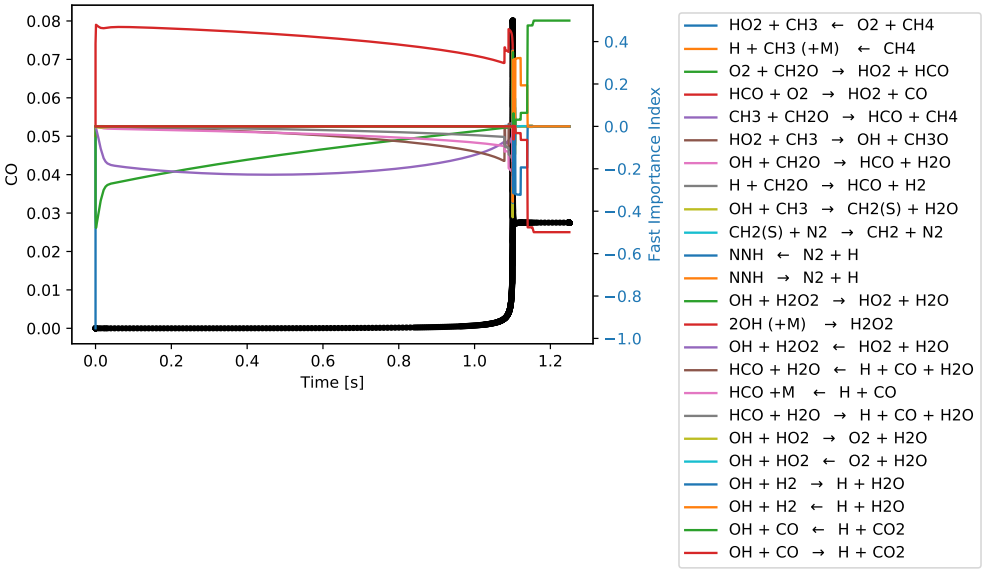

Figure 5-13. Mass fraction of CO (left y-axis), and the Fast importance indices for CO (right y-axis), versus time, for the GRI3.0 problem. The list of reactions corresponds to the top two reactions for each iteration and with an index (absolute value) bigger than threshold=1e-2.

```
for (int i = 0; i < nSample; i++) {
  eig_val_real = eig_val_real_bath[i];
  eig_val_imag = eig_val_imag_bath[i];
  eig_vec_R_2D = eig_vec_R_bath[i];

  // convert 2D to 1D
  int count=0;
  for (size_t k=0; k<ndiff_var; k++) {
    for (size_t j=0; j<ndiff_var; j++) {
      eig_vec_R[count] = eig_vec_R_2D[k][j];
      count++;
    }
  }

  ker.setEigenValVec(eig_val_real, eig_val_imag, eig_vec_R);
  // Sorting eigen values and vectors
  // of, sign(eig_val_real)*Mod(eig_val_real + i * eig_val_imag)
  ker.sortEigValVec();

  ...

}
```

We pass the eigensolution to the kernel class with the function “ker.setEigenValVec”, and then we sort the eigensolution.

The outputs of this example are the same as the above example. However, there are small differences between the results because of discrepancies between the smallest eigenvalues computed by Lapack and the Tines solver.

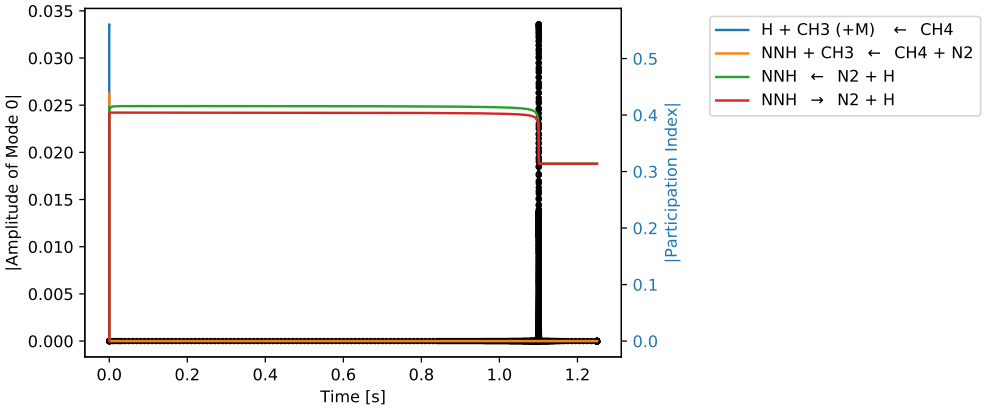

Figure 5-14. Participation index for mode 0 (right y-axis), and amplitude of mode 0 (left y-axis), versus time, for the GRI3.0 problem. The list of reactions corresponds to the top two reactions for each iteration and with an index (absolute value) bigger than threshold=1e-2. Both axes are in absolute value.

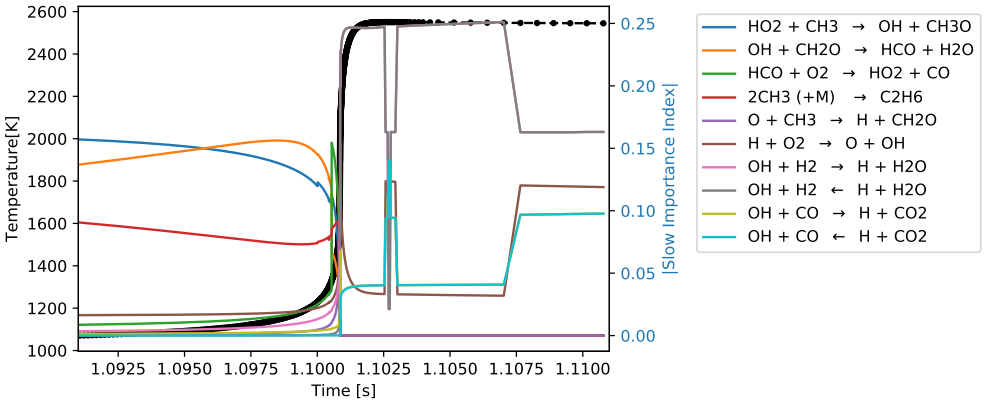

Figure 5-15. Temperature (black, left y-axis), and the absolute value of slow Importance index for temperature (right y-axis), versus time, for the GRI3.0 problem. The list of reactions corresponds to the top two reactions for each iteration and with an index (absolute value) larger than 1e-2. Zoom in around ignition point.

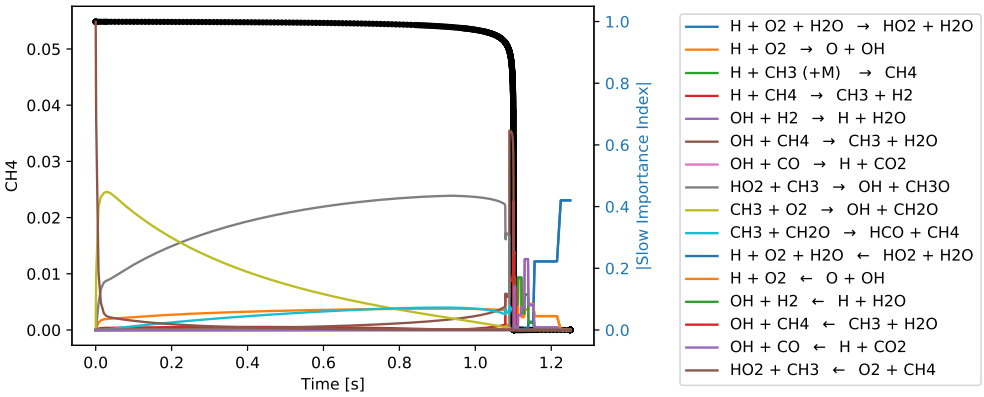

Figure 5-16. Mass fraction of CH_4 (black, left y-axis), and the absolute value of the slow Importance index for CH_4 (right y-axis), versus time, for the GRI3.0 problem. The list of reactions corresponds to the top two reactions for each iteration and with an index (absolute value) larger than 1e-2.

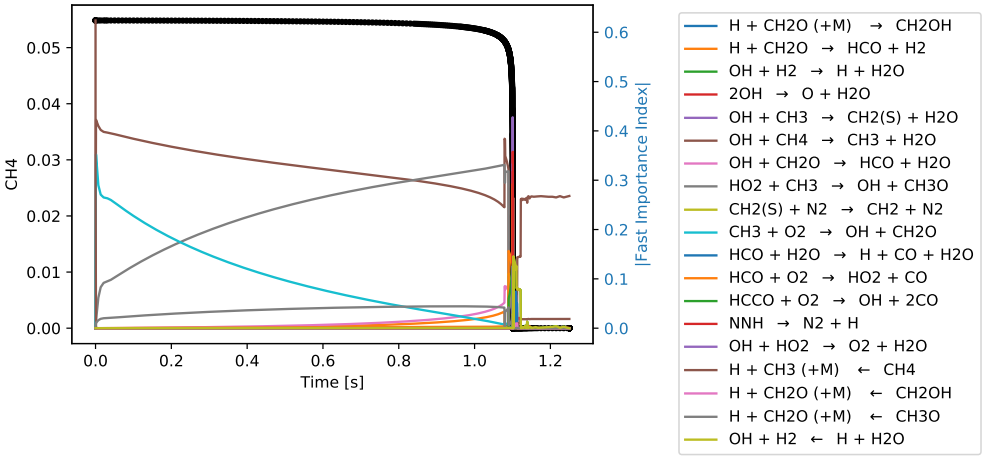

Figure 5-17. Mass fraction of CH_4 (black, left y-axis), and the absolute value of the fast importance index for CH_4 (right y-axis), versus time, for the GRI3.0 problem. The list of reactions corresponds to the top two for each iteration and with an index (absolute value) bigger than 1e-2.

6. SUMMARY

We provided instructions to perform a CSP analysis for a general ODE system and a detailed chemical kinetic ODE system in examples 5.1.1 and 5.2.2. These examples showed how to use the different CSPlib functions. We divided the CSP analysis into three blocks, the model block (see section 4.1), the kernel block (see section 4.2), and the index block (see section 4.3). For each block, we have implemented a class. Additionally, we have an interface for the Tines-GPU eigensolver (see section 4.2.1). We describe the CSPlib functions in the application programming interface section 4. We recommend using these examples as a template and add or delete parts to create an analysis that fits the user's demands.

REFERENCES

[1] Kokkos. `https://github.com/kokkos/kokkos`, 2020. Accessed: 2020-08-26.

[2] tchem. `https://github.com/sandialabs/TChem`, 2020. Accessed: 2020-12-01.

[3] M.D. Ardema. Computational singular perturbation method for dynamic systems. *IFAC Proceedings Volumes*, 23(8, Part 3):409 – 413, 1990. 11th IFAC World Congress on Automatic Control, Tallinn, 1990 - Volume 3, Tallinn, Finland.

[4] Michael J. Davis and Rex T. Skodje. Geometric investigation of low-dimensional manifolds in systems approaching equilibrium. *The Journal of Chemical Physics*, 111(3):859–874, 1999.

[5] D. GOUSSIS, S. LAM, and P. GNOFFO. *Reduced and simplified chemical kinetics for air dissociation using Computational Singular Perturbation*.

[6] D.A. Goussis. On the construction and use of reduced chemical kinetic mechanisms produced on the basis of given algebraic relations. *Journal of Computational Physics*, 128(2):261 – 273, 1996.

[7] D.A. Goussis and S.H. Lam. A study of homogeneous methanol oxidation kinetics using csp. *Symposium (International) on Combustion*, 24(1):113 – 120, 1992. Twenty-Fourth Symposium on Combustion.

[8] Saurabh Gupta, Hong G. Im, and Mauro Valorani. Analysis of n-heptane auto-ignition characteristics using computational singular perturbation. *Proceedings of the Combustion Institute*, 34(1):1125 – 1133, 2013.

[9] S. H. Lam. *Singular Perturbation for Stiff Equations Using Numerical Methods*, pages 3–19. Springer US, Boston, MA, 1985.

[10] S. H. Lam. Using csp to understand complex chemical kinetics. *Combustion Science and Technology*, 89(5-6):375–404, 1993.

[11] S. H. Lam and D. A. Coussis. *Conventional asymptotics and computational singular perturbation for simplified kinetics modelling*, pages 227–242. Springer Berlin Heidelberg, Berlin, Heidelberg, 1991.

[12] S.H. Lam and D.A. Goussis. Understanding complex chemical kinetics with computational singular perturbation. *Symposium (International) on Combustion*, 22(1):931 – 941, 1989.

[13] Riccardo Malpica Galassi, Mauro Valorani, Habib N. Najm, Cosmin Safta, Mohammad Khalil, and Pietro P. Ciottoli. Chemical model reduction under uncertainty. *Combustion and Flame*, 179:242 – 252, 2017.

[14] J. Prager, H.N. Najm, M. Valorani, and D.A. Goussis. Structure of n-heptane/air triple flames in partially-premixed mixing layers. *Combustion and Flame*, 158(11):2128 – 2144, 2011.

[15] Anil Rao and Kenneth Mease. *A new method for solving optimal control problems*.

[16] G. P. Smith, D. M. Golden, M. Frenklach, N. W. Moriarty, B. Eiteneer, M. Goldenberg, C. T. Bowman, R. K. Hanson, S. Song, W. C. Gardiner Jr., V. V. Lissianski, and Z. Qin. GRI-Mech v3.0. `http://www.me.berkeley.edu/gri_mech`, 2011. Accessed: 2020-08-26.

[17] C. Treviño and F. Méndez. Reduced kinetic mechanism for methane ignition. *Symposium (International) on Combustion*, 24(1):121 – 127, 1992. Twenty-Fourth Symposium on Combustion.

[18] Mauro Valorani, Francesco Creta, Dimitris A. Goussis, Jeremiah C. Lee, and Habib N. Najm. An automatic procedure for the simplification of chemical kinetic mechanisms based on csp. *Combustion and Flame*, 146(1):29 – 51, 2006.

[19] Mauro Valorani and Dimitrios A. Goussis. Explicit time-scale splitting algorithm for stiff problems: Auto-ignition of gaseous mixtures behind a steady shock. *Journal of Computational Physics*, 169(1):44 – 79, 2001.

[20] Mauro Valorani, Dimitris A. Goussis, Francesco Creta, and Habib N. Najm. Higher order corrections in the approximation of low-dimensional manifolds and the construction of simplified problems with the csp method. *Journal of Computational Physics*, 209(2):754 – 786, 2005.

[21] Mauro Valorani, Habib N. Najm, and Dimitris A. Goussis. Csp analysis of a transient flame-vortex interaction: time scales and manifolds. *Combustion and Flame*, 134(1):35 – 53, 2003.

[22] Mauro Valorani, Samuel Paolucci, Emanuele Martelli, Temistocle Grenga, and Pietro P. Ciottoli. Dynamical system analysis of ignition phenomena using the tangential stretching rate concept. *Combustion and Flame*, 162(8):2963 – 2990, 2015.

DISTRIBUTION

Hardcopy—External

Number of Copies	Name(s)	Company Name and Company Mailing Address

Hardcopy—Internal

Number of Copies	Name	Org.	Mailstop

Email—Internal (encrypt for OUO)

Name	Org.	Sandia Email Address
Technical Library	01177	libref@sandia.gov

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.