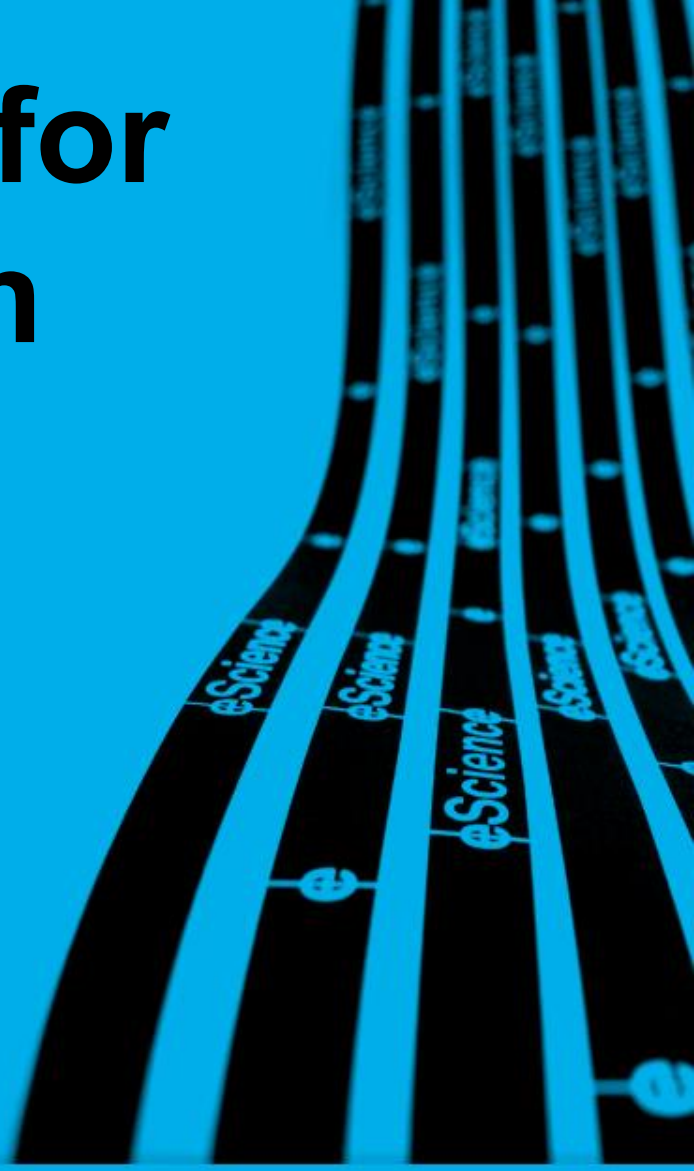
Davidson method for solving large eigen value problems

Applications to diagonalization of Bethe Salpeter Equation



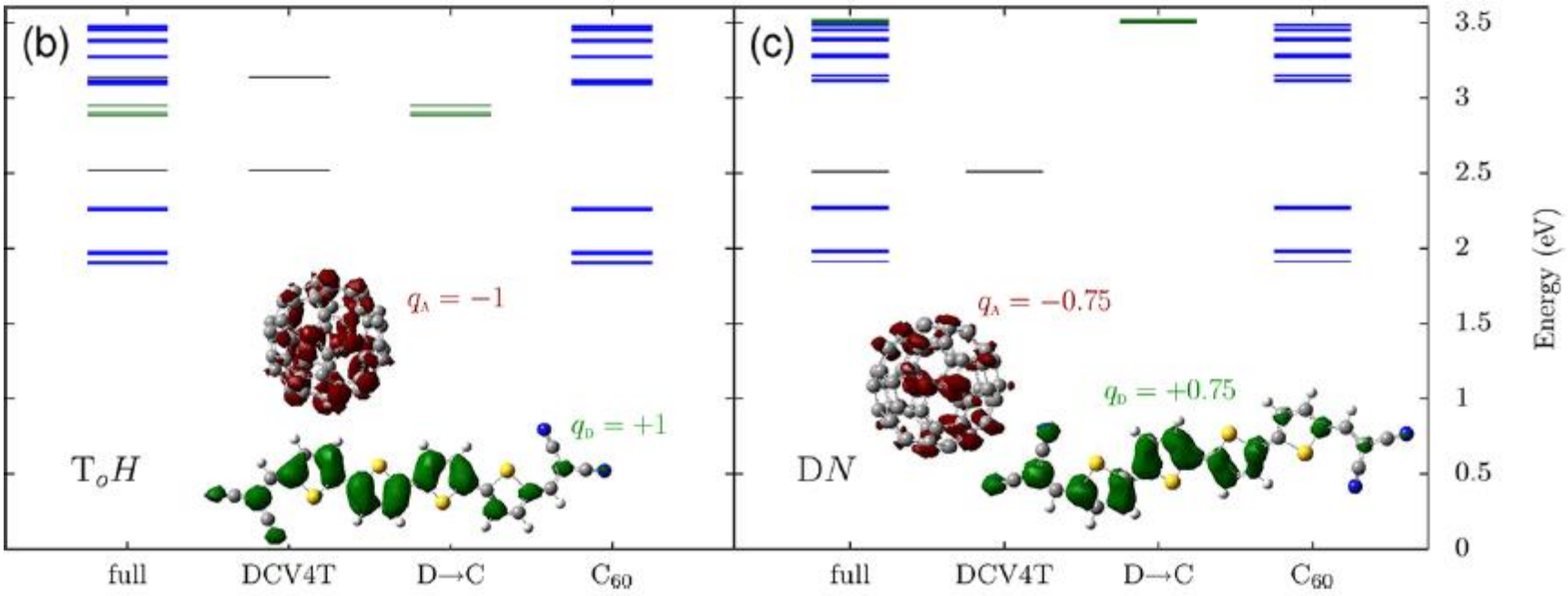
netherlands eScience center

by SURF & NWO

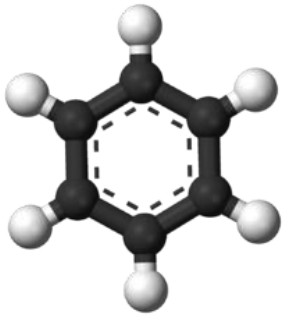
Light absorption by molecules

Multiscale simulations of excitation dynamics in molecular materials for sustainable energy applications (MULTIXMAS)

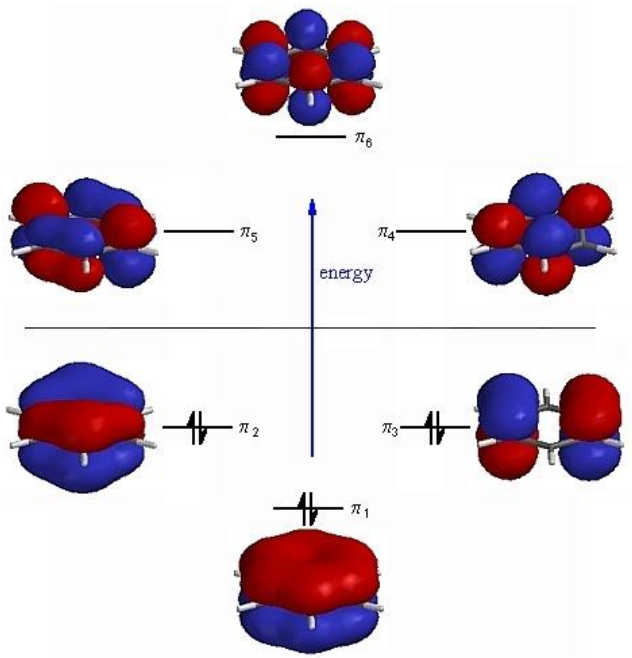
"Understand how molecular systems absorb light and how energy propagates in molecular materials"



Light absorption by molecules



N = 30 orbitals



Bethe - Salpeter Equation

$$\mathcal{H}_{SS'}(\mathbf{q}) = \epsilon_S \delta_{SS'} - f_S K_{SS'}(\mathbf{q})$$

$$\begin{pmatrix} R & C \\ -C^* & -R \end{pmatrix} \begin{pmatrix} A \\ B \end{pmatrix} = \lambda \begin{pmatrix} A \\ B \end{pmatrix}$$

$$R : N^2 \times N^2 \quad !!$$

Light absorption by molecules

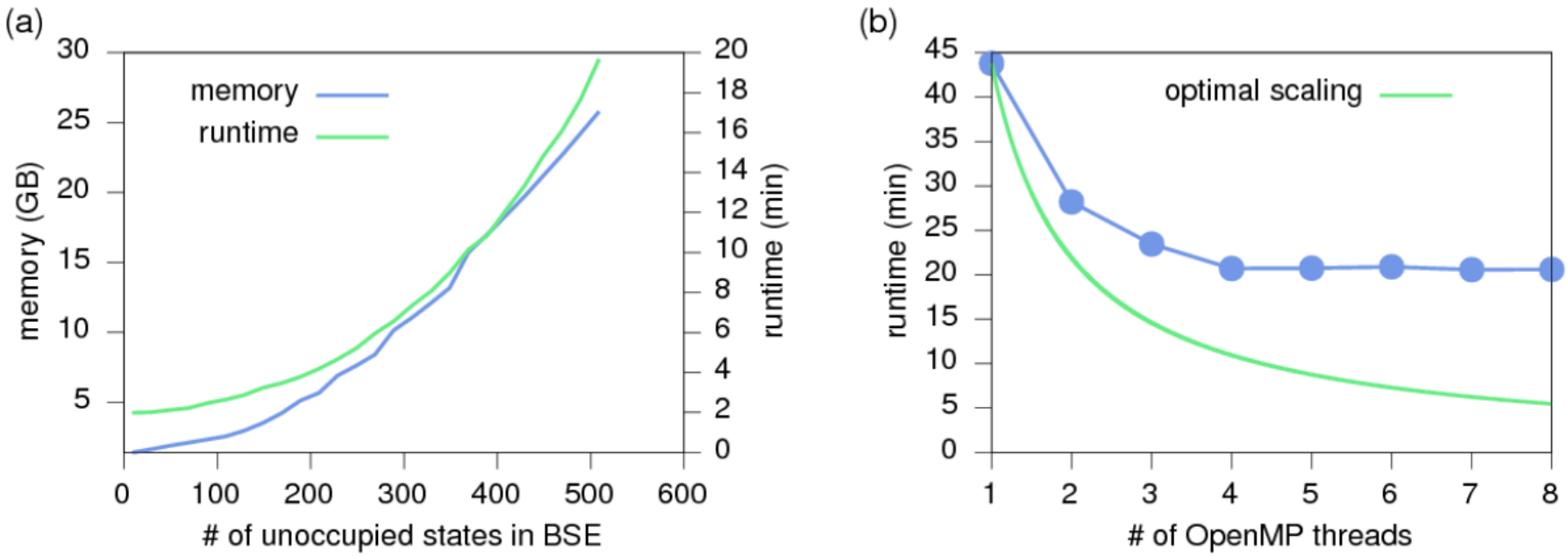
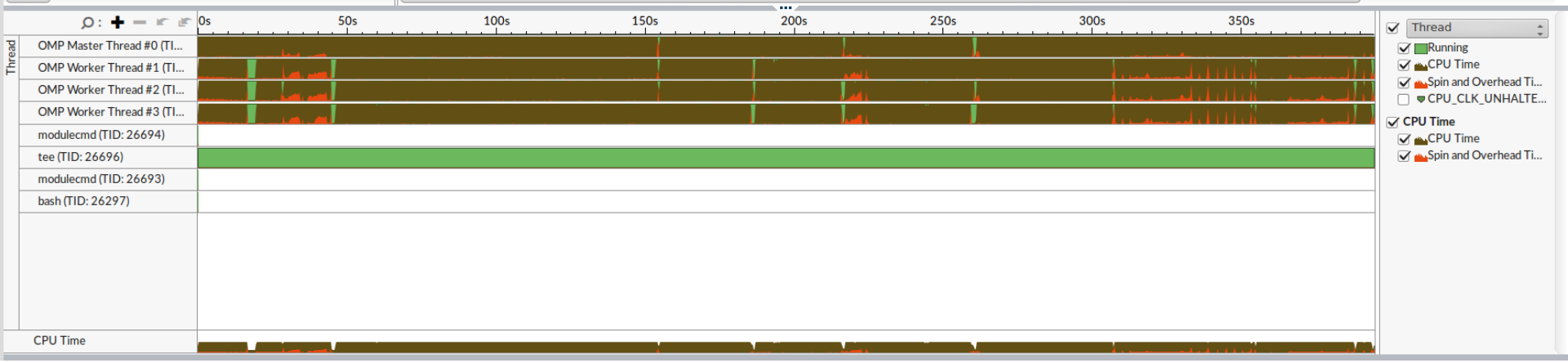


Figure 5: *VOTCA-XTP's current GW-BSE performance, calculating excitations of adenine (a) Scaling of runtime and memory consumption with increasing system size. (b) Parallel scaling with number of OpenMP threads. No improvement in runtime for > 4 due to Intel MKL's linear algebra not using Hyperthreading. Calculations performed on Intel Core i7-4770 CPU @ 3.40GHz.*

Light absorption by molecules

Grouping: Function / Call Stack

Function / Call Stack	CPU Time ▼	Instructions Retired	CPI Rate	CPU Frequency Ratio	Module	Function (Full)	S
[MKL BLAS]@avx2_dgemm_kernel_0	815.657s	4,737,182,800,000	0.365	0.754	libmkl_avx2.so	mkl_blas_avx2_dgemm_kernel_0	
[MKL LAPACK]@dsymv_nb	132.096s	134,108,800,000	2.557	0.925	libmkl_avx2.so	mkl_lapack_ps_avx2_dsymv_nb	
__libm_erfc_ex	53.355s	145,440,400,000	1.085	1.054	libimf.so	__libm_erfc_ex	
votca::xtp::TCMatrix::FillThreeCenterRepBlock	43.246s	201,034,400,000	0.464	0.768	libvotca_xtp.so.5	votca::xtp::TCMatrix::FillThreeCenterRepBlock(boost::multi_array<double, (unsigned lon...	threecen
_INTERNAL_25_____src_kmp_barrier_cpp_71f3cf03::__kmp_wait_te	43.021s	29,209,600,000	3.744	0.905	libiomp5.so	void _INTERNAL_25_____src_kmp_barrier_cpp_71f3cf03::__kmp_wait_template<kmp_f...	kmp_wa
votca::xtp::Sigma::Stabilize	30.461s	151,174,800,000	0.511	0.903	libvotca_xtp.so.5	votca::xtp::Sigma::Stabilize(Eigen::Array<double, (int)-1, (int)1, (int)0, (int)-1, (int)1>&)	sigma.cc
[vmlinux]	21.679s	30,816,800,000	1.607	0.813	vmlinux	[vmlinux]	
votca::xtp::TCMatrix_gwbse::FillBlock	18.007s	50,276,800,000	0.769	0.764	libvotca_xtp.so.5	votca::xtp::TCMatrix_gwbse::FillBlock(std::vector<Eigen::Matrix<double, (int)-1, (int)-1, (...	threecen
Eigen::internal::ploadu<__m256d>	13.844s	20,851,600,000	1.687	0.905	libvotca_xtp.so.5	__m256d Eigen::internal::ploadu<__m256d>(Eigen::internal::unpacket_traits<__m256d>:...	PacketM
[MKL BLAS]@avx2_xdgemv_t	12.228s	24,833,200,000	1.140	0.825	libmkl_avx2.so	mkl_blas_avx2_xdgemv_t	
Eigen::internal::pdiv<__m256d>	11.968s	16,318,400,000	1.873	0.910	libvotca_xtp.so.5	__m256d Eigen::internal::pdiv<__m256d>(__m256d const&, __m256d const&)	PacketM
Eigen::internal::padd<__m256d>	11.038s	23,816,800,000	1.155	0.887	libvotca_xtp.so.5	__m256d Eigen::internal::padd<__m256d>(__m256d const&, __m256d const&)	PacketM
Eigen::internal::ploadu<__m256d>	11.023s	22,920,800,000	1.243	0.920	libvotca_xtp.so.5	__m256d Eigen::internal::ploadu<__m256d>(Eigen::internal::unpacket_traits<__m256d>:...	PacketM
votca::xtp::Sigma::X_offdiagompparallel_for@145	10.804s	8,176,000,000	3.579	0.965	libvotca_xtp.so.5	votca::xtp::Sigma::X_offdiagompparallel_for@145	sigma.cc
__GI___libc_malloc	10.308s	33,457,200,000	0.725	0.838	libc-2.23.so	__GI___libc_malloc	malloc.c
[MKL BLAS]@avx2_dgemm_dcopy_right4_ea	9.873s	8,884,400,000	2.336	0.749	libmkl_avx2.so	mkl_blas_avx2_dgemm_dcopy_right4_ea	
[MKL BLAS]@avx2_dgemm_dcopy_down12_ea	9.528s	4,746,000,000	4.447	0.789	libmkl_avx2.so	mkl_blas_avx2_dgemm_dcopy_down12_ea	
_int_malloc	9.446s	47,012,000,000	0.474	0.841	libc-2.23.so	_int_malloc	malloc.c
_int_free	9.114s	37,052,400,000	0.580	0.839	libc-2.23.so	_int_free	malloc.c
Eigen::internal::pmul<__m256d>	8.336s	15,125,600,000	1.387	0.896	libvotca_xtp.so.5	__m256d Eigen::internal::pmul<__m256d>(__m256d const&, __m256d const&)	PacketM
Eigen::internal::dense_assignment_loop<Eigen::internal::generic_dense	7.286s	46,494,000,000	0.397	0.902	libvotca_xtp.so.5	Eigen::internal::dense_assignment_loop<Eigen::internal::generic_dense_assignment_kern...	AssignEv
Eigen::internal::dense_assignment_loop<Eigen::internal::generic_dense	6.031s	38,900,400,000	0.396	0.909	libvotca_xtp.so.5	Eigen::internal::dense_assignment_loop<Eigen::internal::generic_dense_assignment_kern...	AssignEv
Eigen::internal::pdiv<__m256d>	5.892s	3,054,800,000	4.974	0.918	libvotca_xtp.so.5	__m256d Eigen::internal::pdiv<__m256d>(__m256d const&, __m256d const&)	PacketM
Eigen::internal::redux_impl<Eigen::internal::scalar_sum_op<double, dc	5.887s	4,925,200,000	3.099	0.923	libvotca_xtp.so.5	Eigen::internal::redux_impl<Eigen::internal::scalar_sum_op<double, double>, Eigen::inter...	Redux.h
Eigen::internal::padd<__m256d>	5.799s	22,400,000	664.125	0.914	libvotca_xtp.so.5	__m256d Eigen::internal::padd<__m256d>(__m256d const&, __m256d const&)	PacketM
Eigen::internal::pstore<double, __m256d>	5.533s	18,527,600,000	0.668	0.797	libvotca_xtp.so.5	void Eigen::internal::pstore<double, __m256d>(double*, __m256d const&)	PacketM
Eigen::internal::pmul< m256d>	5.439s	3,060,400,000	4.539	0.909	libvotca_xtp.so.5	m256d Eigen::internal::pmul< m256d>(m256d const&, m256d const&)	PacketM



Davidson Method

- Popular technique to compute a few of the smallest (or largest) eigenvalues of a large (sparse) real symmetric matrix
- It fails if the matrix is diagonal (Davidson paradox). It is therefore mainly used in quantum chemistry where matrices are strongly diagonally dominant.
- Many different version exists. The most popular is the Jacobi-Davidson method that works for non-diagonally dominant matrices
- The method never requires to have the full matrix stored in memory but only the result of its multiplication with a vector. It's a *matrix free* method !

Davidson Method

0: Start with the eigenvalue problem : $H\Psi = \theta\Psi$
1: Define a set of k guess eigenvectors : $U = [u_0, u_1, ..., u_k]$
2: **while** $\epsilon > tol$:
3: Project H on the guess subspace : $T = U^T\ H\ U\ (k \times k)$
4: Solve the small eigenvalue problem : $Tx = \tilde{\lambda}x$
4: $(\tilde{\lambda},\ \tilde{\Psi} = Ux)$ are the approximation of the real eigepair
5: Compute the residue : $r = -(H - \tilde{\lambda})\tilde{\Psi}$ and $\epsilon = ||r||^2$
6: Compute the correction vector δ from r
7: Append t to the guess $U = [u_0, u_1, ..., u_k, t]$
8: Orthogonalize the new basis

Getting the correction (DPR)

The true eigenvector is : $\Psi = \tilde{\Psi} + \delta$

And the true eigenvalue : $\lambda = \tilde{\lambda} + \epsilon$

The eigenvalue problem $H\Psi = \lambda\Psi$

$$H(\tilde{\Psi} + \delta) = (\tilde{\lambda} + \epsilon)(\tilde{\Psi} + \delta)$$

$$\longrightarrow \quad (H - \tilde{\lambda} - \epsilon)\delta = -r + \epsilon\tilde{\Psi}$$

Diagonal Preconditioned Residue $\epsilon \to 0 \qquad H \to D$

$$\delta = -(D - \tilde{\lambda})^{-1} r$$

Python Version

https://github.com/NLESC-JCER/DavidsonPython

Davidson Example

In this small tutorial we will unroll the Davidson method to compute the lowest eigenvalue of a 5 x 5 matrix. The example is taken from the online presentation : http://www.esqc.org/static/lectures/Malmqvist_2B.pdf. The Davidson method can be summarized as:

- Initialize : Define n vectors $b = \{b_1, ...b_n\}$
- Iterate : loop untul convergence
  1. Orthogonalize the **b** vectors
  2. project the matrix on the subspace $A_p = b^T \times A \times b$
  3. Diagonalize the projected matrix : $A_p \times v = \lambda \times v$
  4. Compute the residue vector : $r = A \times b - \lambda \times b$
  5. Compute correction vector : $q = -r / (A_{ii} - \lambda)$
  6. Append the correction vector to b : $b = \{b_1, ..., b_n, q\}$

In [2]:
```
import numpy as np
```

Define the matrix to diagonalize : Let's define the matrix we want to diagonalize as :

In [3]:
```
A = 0.1*np.ones((5,5)) + np.diag([0.9,1.9,2.9,2.9,2.9])
```

In [4]:
```
print(A)
```

```
[[1.  0.1 0.1 0.1 0.1]
 [0.1 2.  0.1 0.1 0.1]
 [0.1 0.1 3.  0.1 0.1]
 [0.1 0.1 0.1 3.  0.1]
 [0.1 0.1 0.1 0.1 3. ]]
```

Python Version

https://github.com/NLESC-JCER/DavidsonPython

```python
def davidson_solver(A, neigen, tol=1E-6, itermax = 1000, jacobi=False):
    """Davidosn solver for eigenvalue problem

    Args :
        A (numpy matrix) : the matrix to diagonalize
        neigen (int)     : the number of eigenvalue requied
        tol (float)      : the rpecision required
        itermax (int)    : the maximum number of iteration
        jacobi (bool)    : do the jacobi correction
    Returns :
        eigenvalues (array) : lowest eigenvalues
        eigenvectors (numpy.array) : eigenvectors
    """
    n = A.shape[0]
    k = 2*neigen            # number of initial guess vectors
    V = np.eye(n,k)         # set of k unit vectors as guess
    I = np.eye(n)           # identity matrix same dimen as A
    Adiag = np.diag(A)

    print('\n'+'='*20)
    print("= Davidson Solver ")
    print('='*20)

    # Begin block Davidson routine
    print("iter size norm (%e)" %tol)
    for i in range(itermax):

        # QR of V t oorthonormalize the V matrix
        # this uses GrahmShmidtd in the back
        V,R = np.linalg.qr(V)

        # form the projected matrix
        T = np.dot(V.T,np.dot(A,V))

        # Diagonalize the projected matrix
        theta,s = np.linalg.eigh(T)
```

```
====================
= Davidson Solver
====================
iter size norm (1.000000e-06)
 000 020 3.155051e-02
 001 030 1.205168e-05
 002 040 5.167015e-09
= Davidson has converged
davidson :  1.2921724319458008  seconds
numpy    :  2.4214675426483154  seconds
0  0.999809   0.999809
1  2.000385   2.000385
2  3.000963   3.000963
3  3.999762   3.999762
4  4.997463   4.997463
```

C++ Version

https://github.com/NLESC-JCER/DavidsonEigen

Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms

NLESC-JCER / DavidsonEigen

Watch 2 | Star 1 | Fork 0

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki | Insights | Settings

Implementation of the (matrix free) Davidson Algorithm using Eigen

Edit

Manage topics

56 commits | 9 branches | 0 releases | 1 contributor | Apache-2.0

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

NicoRenaud diag optional — Latest commit 5b2bd1f 4 hours ago

CMakeModules	refactoring	6 days ago
src	diag optional	4 hours ago
test	put the template definition in header	a day ago
.travis.yml	conda2	a day ago
CMakeLists.txt	c++11	a day ago
CODE_OF_CONDUCT.md	add license and stuff	a day ago
CONTRIBUTING.md	add license and stuff	a day ago
LICENSE	add license and stuff	a day ago
README.md	add badge	a day ago

README.md

build passing

DavidsonEigen

This package contains a C++ implementation of the *Davidson diagonalization algorithms*. The calculation can be performed *matrix free*, i.e. without having to ever store the entire matrix. Different schemas are available to compute the correction.

Available correction methods are:

- DPR: Diagonal-Preconditioned-Residue
- GJD: Generalized Jacobi Davidson

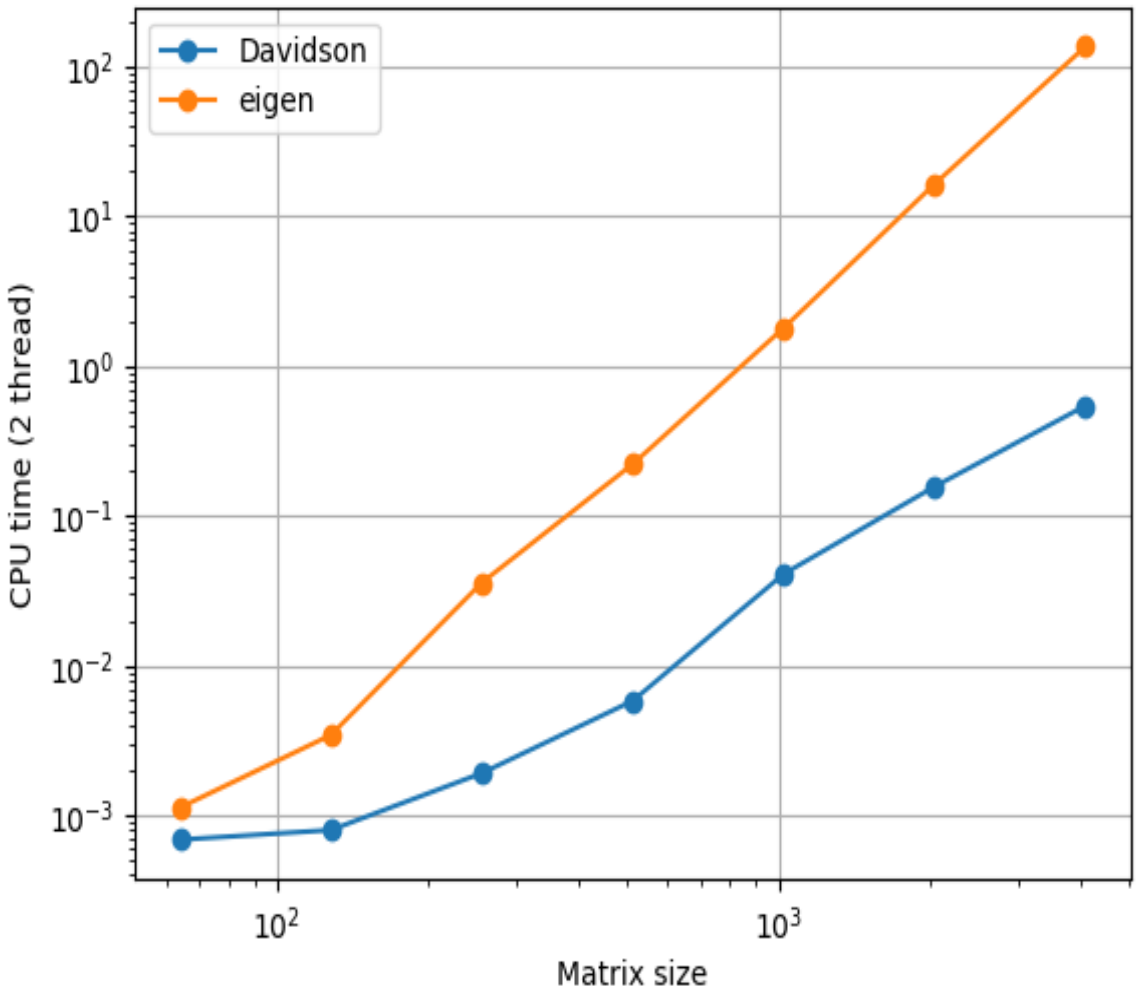
Note:

The Davidson method is suitable for **diagonal-dominant symmetric matrices**, that are quite common in certain scientific problems like electronic structure. The Davidson method could be not practical for other kind of symmetric matrice.

Usage

C++ Version

https://github.com/NLESC-JCER/DavidsonEigen




```
Matrix size : 1000x1000
Num Threads : 2
eps : 0.01

============================
= Davidson (DPR)
============================

iter    Search Space    Norm/1e-06
---------------------------------
   0             10    2.71e-03
   1             15    2.89e-06
   2             20    6.73e-11
---------------------------------
- Davidson converged
- final residue norm 6.73e-11
- final eigenvalue norm 2.26e-12
---------------------------------

Davidson              : 0.0337307 secs
Eigen                 : 1.63471 secs

      Davidson          Eigen
#   0 0.9899555         0.9899555
#   1 1.9993530         1.9993530
#   2 2.9999073         2.9999073
#   3 3.9999770         3.9999770
#   4 4.9999920         4.9999920
```

C++ Version Matrix Free

https://github.com/NLESC-JCER/DavidsonEigen

```cpp
class MatrixFreeOperator : public Eigen::EigenBase<Eigen::MatrixXd>
{
    public:

        typedef double Scalar;
        typedef double RealScalar;
        typedef int StorageIndex;

        enum {
            ColsAtCompileTime = Eigen::Dynamic,
            MaxColsAtCompileTime = Eigen::Dynamic,
            IsRowMajor = false
        };

        Index rows() const {return this-> _size;}
        Index cols() const {return this-> _size;}

        template<typename Vtype>
        Eigen::Product<MatrixFreeOperator,Vtype,Eigen::AliasFreeProduct> operator*(const Eigen::MatrixBase<Vtype>& x) const
            return Eigen::Product<MatrixFreeOperator,Vtype,Eigen::AliasFreeProduct>(*this, x.derived());
        }

        // custom API
        MatrixFreeOperator();

        // convenience function
        Eigen::MatrixXd get_full_mat() const;
        Eigen::VectorXd diagonal() const;
        int get_size() const {return this->_size;}
        void set_size(int N) {this->_size = N;}

        // extract row/col of the operator
        virtual Eigen::VectorXd col(int index) const = 0;
        Eigen::VectorXd diag_el;

    protected:

        int _size;
};
```

C++ Version Matrix Free

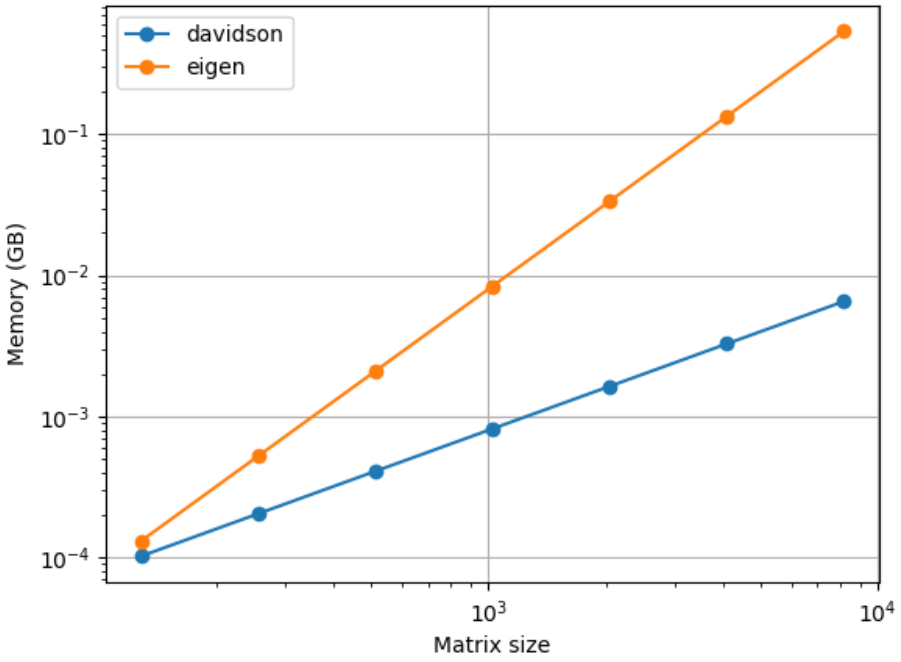
https://github.com/NLESC-JCER/DavidsonEigen

```cpp
class MatrixFreeOperator : public Eigen::EigenBase<Eigen::MatrixXd>
{
    public:

        typedef double Scalar;
        typedef double RealScalar;
        typedef int StorageIndex;

        enum {
            ColsAtCompileTime = Eigen::Dynamic,
            MaxColsAtCompileTime = Eigen::Dynamic,
            IsRowMajor = false
        };

        Index rows() const {return this-> _size;}
        Index cols() const {return this-> _size;}

        template<typename Vtype>
        Eigen::Product<MatrixFreeOperator,Vtype,Eigen::AliasFreeProduct> operator*(const Eigen::MatrixBase<Vtype>& x) const
            return Eigen::Product<MatrixFreeOperator,Vtype,Eigen::AliasFreeProduct>(*this, x.derived());
        }

        // custom API
        MatrixFreeOperator();

        // convenience function
        Eigen::MatrixXd get_full_mat() const;
        Eigen::VectorXd diagonal() const;
        int get_size() const {return this-> _size;}
        void set_size(int N) {this-> _size = N;}

        // extract row/col of the operator
        virtual Eigen::VectorXd col(int index) const = 0;
        Eigen::VectorXd diag_el;

    protected:

        int _size;
};
```

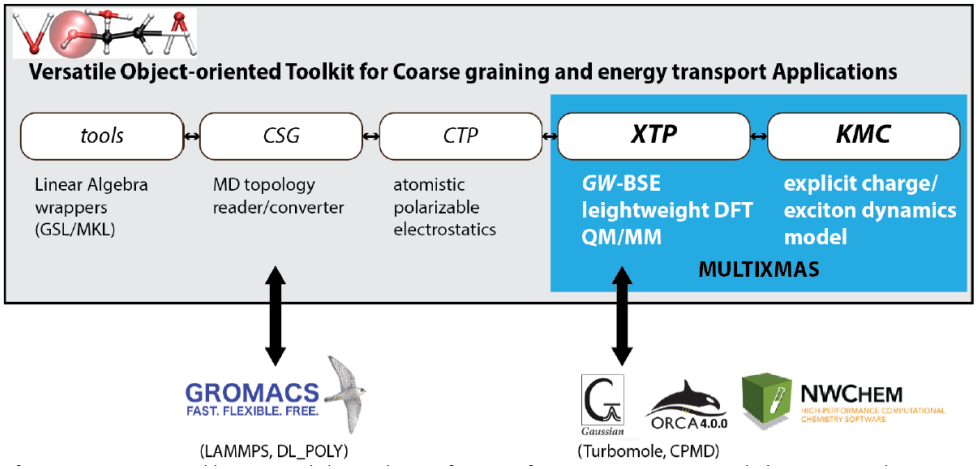
```cpp
class DavidsonOperator : public MatrixFreeOperator
{
    public:

        DavidsonOperator(int n, bool d);
        Eigen::VectorXd col(int index) const;

    private:
        double _sparsity = 0.1;
        bool _odiag = false;
};

#endif
```

C++ Version Matrix Free

https://github.com/NLESC-JCER/DavidsonEigen



Now included in the software of out partner



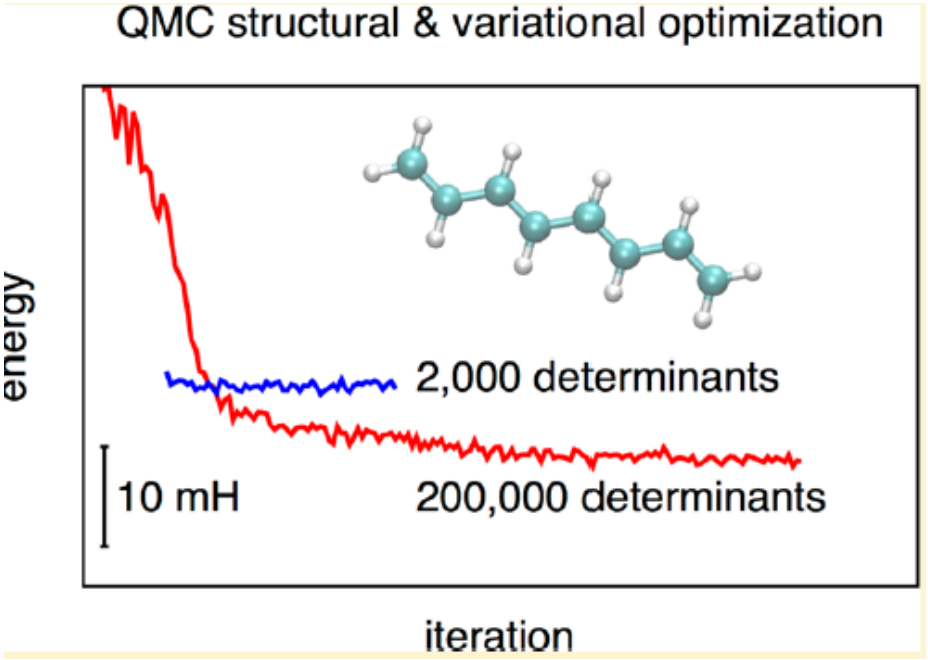
Fortran Version

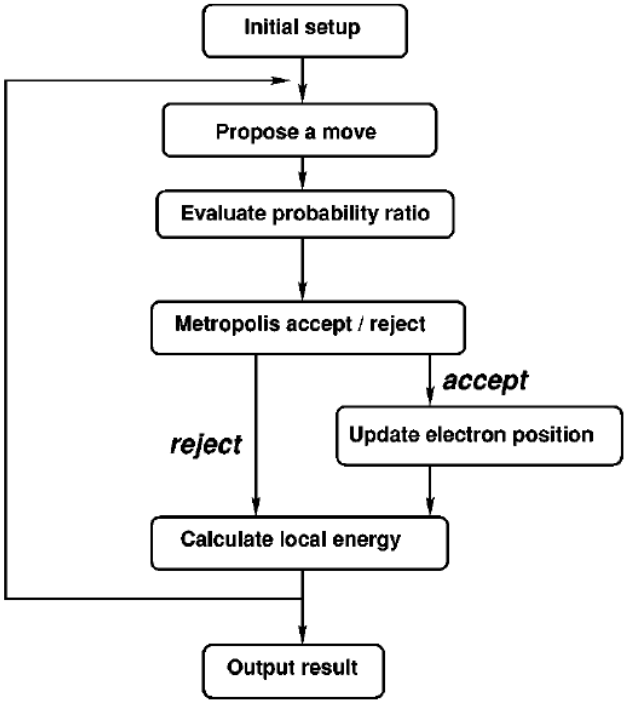
https://github.com/NLESC-JCER/Fortran_Davidson

Dr. Felipe Zapata

eScience Research Engineer

Quantum Monte Carlo simulations of light harvesting materials





Jacobi-Davidson

Look for a correction vector orthogonal to the current approximation

$$H(\tilde{\Psi} + \delta) = \lambda(\tilde{\Psi} + \delta), \quad \tilde{\Psi} \perp \delta$$

$$P_{\Psi_\perp} = \mathbb{I} - \tilde{\Psi}\tilde{\Psi}^T$$
$$P_{\Psi_\perp} H(\tilde{\Psi} + \delta) = P_{\Psi_\perp} \lambda(\tilde{\Psi} + \delta)$$

.....

Jacobi orthogonal component correction

$$P_{\Psi_\perp}(H - \tilde{\lambda})P_{\Psi_\perp}\delta = -(H - \tilde{\lambda})\tilde{\Psi} = -r$$

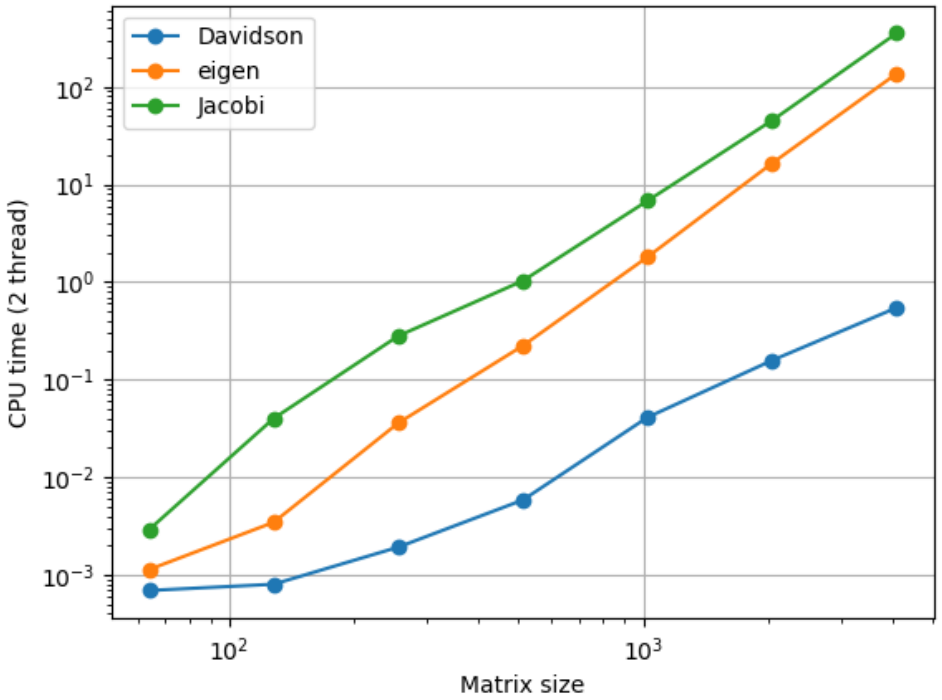
Linear system that can be solved approximatively
(CG, GMRES, BCGStab, …)

Supposed to help convergence for non-diagonally dominant matrices ….

Getting the correction (Jacobi)

C++ :

Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms

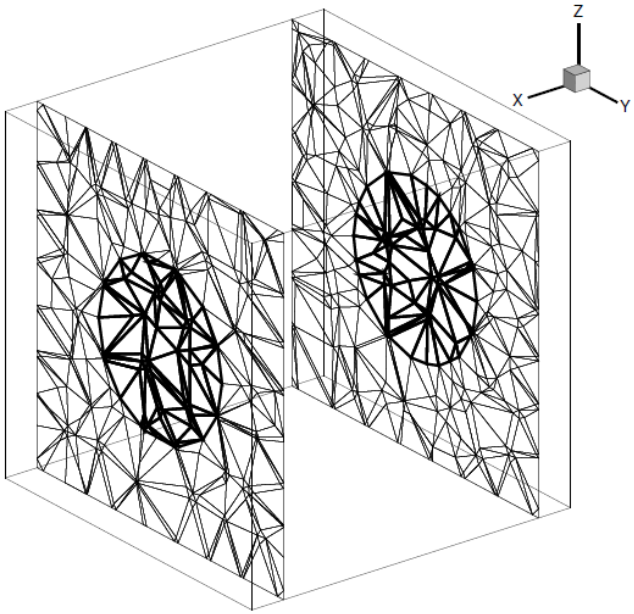


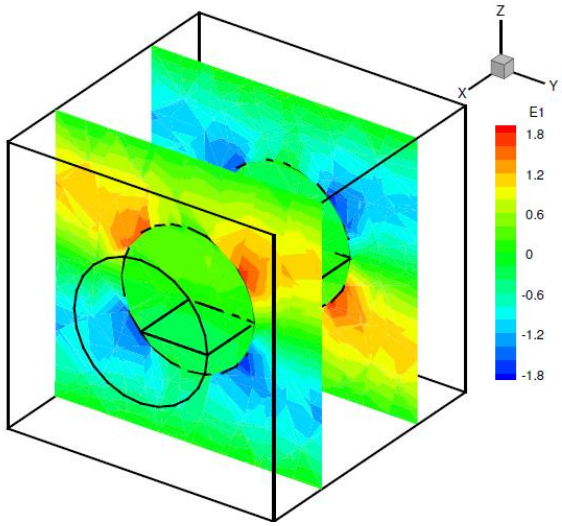
Python : http://pysparse.sourceforge.net

C++MPI and/or CUDA

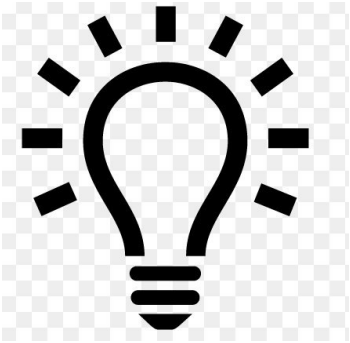
Dr. Adithya Vijaykumar
eScience Research Engineer

Accurate and Efficient Computation of the Optical Properties of Nanostructures for Improved Photovoltaics





Sustainability



Use the generalization budget of the JCER projects to include these solvers in existing libraries (Eigen, Elemental,)

Work in sprint of 3 / 4 people to polish/improve/document …. the code so that we can include them in the libraries