Description of Supplementary Material

OpenFOAM v2.3.x compatible C++ code for the SEULEX solver that was used to generate the results shown in this paper is provided below. Computer files can be provided by the authors on request. This is followed by additional figures (supplementing those provided in the main text) showing the relationships among user-specified error tolerances, global solution accuracy and CPU time. Normalized errors are shown for several different physical quantities at three different instants in time and using two different chemical mechanisms.

```
 //                           seulex.H

#ifndef seulex_H
#define seulex_H
#include "klu.h"
#include "ODESolver.H"
#include "scalarMatrices.H"
#include "labelField.H"

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

namespace Foam
{

/*---------------------------------------------------------------------------*\
                            Class seulex Declaration
\*---------------------------------------------------------------------------*/

class seulex
:
    public ODESolver
{
    // Private data

        // Static constants
```

```
    static const label kMaxx_ = 12;
    static const label iMaxx_ = kMaxx_ + 1;

    static const scalar
        stepFactor1_, stepFactor2_, stepFactor3_,
        stepFactor4_, stepFactor5_,
        kFactor1_, kFactor2_;

// Evaluated constants

    scalar jacRedo_;
    labelField nSeq_;
    scalarField cpu_;
    scalarSquareMatrix coeff_;

// Temporary storage
// held to avoid dynamic memory allocation between calls
// and to transfer internal values between functions

    mutable scalar theta_;
    mutable label kTarg_;
    mutable scalarRectangularMatrix table_;

    mutable scalarField dfdx_;
    mutable scalarSquareMatrix dfdy_;

    // Fields space for "solve" function
    mutable scalarField dxOpt_, temp_;
    mutable scalarField y0_, ySequence_, scale_;

    // Fields used in "seul" function
    mutable scalarField dy_, yTemp_, dydx_;
    mutable klu_symbolic *Symbolic ;
    mutable klu_numeric *Numeric ;
    mutable klu_common Common ;

    scalar NASAP_mintemp;
```

```
            scalar NASAP_maxtemp;

    // Private Member Functions

        //- Computes the j-th line of the extrapolation table
        bool seul
        (
            const scalar x0,
            const scalarField& y0,
            const scalar dxTot,
            const label k,
            scalarField& y,
            const scalarField& scale
        ) const;

        //- Polynomial extrapolation
        void extrapolate
        (
            const label k,
            scalarRectangularMatrix& table,
            scalarField& y
        ) const;

public:

    //- Runtime type information
    TypeName("seulex");

    // Constructors

        //- Construct from ODE
        seulex(const ODESystem& ode, const dictionary& dict);

    // Member Functions

        //- Solve the ODE system and the update the state
        void solve
        (
```

```
            scalar& x,
            scalarField& y,
            stepState& step
        ) const;
};

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

} // End namespace Foam



 //                           seulex.C

include "seulex.H"
#include "addToRunTimeSelectionTable.H"

// * * * * * * * * * * * * * * Static Data Members * * * * * * * * * * * * * //

namespace Foam
{
    defineTypeNameAndDebug(seulex, 0);

    addToRunTimeSelectionTable(ODESolver, seulex, dictionary);

    const scalar
        seulex::stepFactor1_ = 0.6,
        seulex::stepFactor2_ = 0.93,
        seulex::stepFactor3_ = 0.1,
        seulex::stepFactor4_ = 4.0,
        seulex::stepFactor5_ = 0.5,
        seulex::kFactor1_ = 0.7,
        seulex::kFactor2_ = 0.9;
}

// * * * * * * * * * * * * * * * * Constructors  * * * * * * * * * * * * * * //

Foam::seulex::seulex(const ODESystem& ode, const dictionary& dict)
```

```
:
    ODESolver(ode, dict),
    jacRedo_(min(1.0e-4, min(relTol_))),
    nSeq_(iMaxx_, 0),
    cpu_(iMaxx_, 0.0),
    coeff_(iMaxx_, iMaxx_, 0.0),
    theta_(2.0*jacRedo_),
    table_(kMaxx_, n_),
    dfdx_(n_, 0.0),
    dfdy_(n_, n_, 0.0),
    dxOpt_(iMaxx_, 0.0),
    temp_(iMaxx_, 0.0),
    y0_(n_, 0.0),
    ySequence_(n_, 0.0),
    scale_(n_, 0.0),
    dy_(n_, 0.0),
    yTemp_(n_, 0.0),
    dydx_(n_, 0.0),
    NASAP_mintemp(300),
    NASAP_maxtemp(5000)
{
    // The CPU time factors for the major parts of the algorithm
    const scalar cpuFunc = 1.0, cpuJac = 5.0, cpuLU = 1.0, cpuSolve = 1.0;

    nSeq_[0] = 2;
    nSeq_[1] = 3;

    for (int i=2; i<iMaxx_; i++)
    {
        nSeq_[i] = 2*nSeq_[i-2];
    }
    cpu_[0] = cpuJac + cpuLU + nSeq_[0]*(cpuFunc + cpuSolve);

    for (int k=0; k<kMaxx_; k++)
    {
        cpu_[k+1] = cpu_[k] + (nSeq_[k+1]-1)*(cpuFunc + cpuSolve) + cpuLU;
    }
```

```
    // Set the extrapolation coefficients array
    for (int k=0; k<iMaxx_; k++)
    {
        for (int l=0; l<k; l++)
        {
            scalar ratio = scalar(nSeq_[k])/nSeq_[l];
            coeff_[k][l] = 1.0/(ratio - 1.0);
        }
    }
    // Initialize KLU structures
    Symbolic = NULL;
    Numeric = NULL;
    // Set default parameters for KLU
    klu_defaults (&Common);
}

// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

bool Foam::seulex::seul
(
    const scalar x0,
    const scalarField& y0,
    const scalar dxTot,
    const label k,
    scalarField& y,
    const scalarField& scale
) const
{
    label nSteps = nSeq_[k];
    scalar dx = dxTot/nSteps;
    // KLU compressed sparse arrays
    int num = n_;
    int numcol = num + 1;
    int nnzz = num*num;
    double *data, *b;
    int *colptrs, *rowvals;
    // allocate sparse matrix vectors
    data = new double[nnzz];
```

```
rowvals = new int[nnzz];
colptrs = new int[numcol];

int kk=0;
for (int j=0; j<num; j++)
{
    colptrs[j] = kk;
    for (int i=0; i<num; i++)
    {
        scalar a_ = -dfdy_[i][j];
        if(i == j) a_ += 1.0/dx;
        if (a_ != 0.0  )
        {
            data[kk] = a_;
            rowvals[kk++] = i;
        }
     }
 }
colptrs[num] = kk;   //rhs

if( Symbolic != NULL)
   klu_free_symbolic(&Symbolic, &Common);
if( Numeric != NULL)
   klu_free_numeric(&Numeric, &Common);
// KLU factorize
Symbolic = klu_analyze (num, colptrs, rowvals, &Common) ;
Numeric = klu_factor (colptrs, rowvals, data, Symbolic, &Common) ;
// free data, rowvals, colptrs; allocate b
delete[] data;
delete[] rowvals;
delete[] colptrs;
b = new double[num];
// call RHSs
scalar xnew = x0 + dx;
odes_.derivatives(xnew, y0, dy_);

// solve the linear system
for (label i=0; i<num; i++)
```

```
{
    b[i] = dy_[i];
}

klu_solve (Symbolic, Numeric, num, 1, b, &Common) ;

for (label i=0; i<num; i++)
{
    dy_[i] = b[i];
}

yTemp_ = y0;

for (label nn=1; nn<nSteps; nn++)
{
    for (label i=0; i<num; i++)
    {
        yTemp_[i] = yTemp_[i] + dy_[i];
    }
    // check that temperature remains in range
    if (   yTemp_[num-2] <= NASAP_mintemp
        || yTemp_[num-2] > NASAP_maxtemp)
    {
        delete[] b;
        klu_free_symbolic (&Symbolic, &Common) ;
        klu_free_numeric (&Numeric, &Common) ;
        return false;
    }
    xnew += dx;
    if (nn == 1 && k<=1)
    {
        scalar dy1 = 0.0;
        for (label i=0; i<num; i++)
        {
            dy1 += sqr(dy_[i]/scale[i]);
        }
        dy1 = sqrt(dy1);
```

```
    odes_.derivatives(x0 + dx, yTemp_, dydx_);

    for (label i=0; i<num; i++)
    {
        b[i] = dydx_[i] - dy_[i]/dx;
    }
    // solve the linear system
    klu_solve (Symbolic, Numeric, num, 1, b, &Common) ;
    for (label i=0; i<num; i++)
    {
        dy_[i] = b[i];
    }

    scalar dy2 = 0.0;
    for (label i=0; i<n_; i++)
    {
        // check magnitude of change in solution vector, compared to VGREAT
        if (mag(dy_[i]) > sqrt(VGREAT) ){ dy2 = VGREAT; break;}
        dy2 += sqr(dy_[i]/scale[i]);
    }

    dy2 = sqrt(dy2);
    // check convergence of Newton method: apply monotonicity check
    theta_ = dy2/max(1.0, dy1 + SMALL);

    if (theta_ > 1.0 )
    {
        delete[] b;
        klu_free_symbolic (&Symbolic, &Common) ;
        klu_free_numeric (&Numeric, &Common) ;
        return false;
    }
}
odes_.derivatives(xnew, yTemp_, dy_);
// solve the linear system
for (label i=0; i<num; i++)
{
    b[i] = dy_[i];
```

```
        }
        klu_solve (Symbolic, Numeric, num, 1, b, &Common) ;
        for (label i=0; i<num; i++)
        {
            dy_[i] = b[i];
        }
    }

    for (label i=0; i<num; i++)
    {
        y[i] = yTemp_[i] + dy_[i];
    }

    delete[] b;
    klu_free_symbolic (&Symbolic, &Common) ;
    klu_free_numeric (&Numeric, &Common) ;

    return true;
}

void Foam::seulex::extrapolate
(
    const label k,
    scalarRectangularMatrix& table,
    scalarField& y
) const
{
    for (int j=k-1; j>0; j--)
    {
        for (label i=0; i<n_; i++)
        {
            table[j-1][i] =
                table[j][i] + coeff_[k][j]*(table[j][i] - table[j-1][i]);
        }
    }

    for (int i=0; i<n_; i++)
    {
```

```
        y[i] = table[0][i] + coeff_[k][0]*(table[0][i] - y[i]);
    }
}

void Foam::seulex::solve
(
    scalar& x,
    scalarField& y,
    stepState& step
) const
{
    temp_[0] = GREAT;
    scalar dx = step.dxTry;
    y0_ = y;
    dxOpt_[0] = mag(0.1*dx);
    if (step.first || step.prevReject)
    {
        theta_ = 2.0*jacRedo_;
    }

    if (step.first)
    {
        // NOTE: the first element of relTol_ and absTol_ are used here.
        scalar logTol = -log10(relTol_[0] + absTol_[0])*0.6 + 0.5;
        kTarg_ = max(1, min(kMaxx_ - 1, int(logTol)));
    }
    forAll(y, i)
    {
        scale_[i] = absTol_[i] + relTol_[i]*mag(y[i]);
    }

    bool jacUpdated = false;
    if (theta_ > jacRedo_)
    {
        odes_.jacobian(x, y, dfdx_, dfdy_);
        jacUpdated = true;
    }
```

```
int k;
scalar dxNew = mag(dx);
bool firstk = true;

while (firstk || step.reject)
{
    dx = step.forward ? dxNew : -dxNew;
    firstk = false;
    step.reject = false;

    if (mag(dx) <= mag(x)*SMALL)
    {
         WarningIn("seulex::solve(scalar& x, scalarField& y, stepState&")
                << "step size underflow :"  << dx << endl;
    }

    scalar errOld = 0;
    for (k=0; k<=kTarg_+1; k++)
    {
        bool success = seul(x, y0_, dx, k, ySequence_, scale_);
        if (!success)
        {
            step.reject = true;
            dxNew = mag(dx)*stepFactor5_;
            break;
        }

        if (k == 0)
        {
            forAll(y,i)
            {
               y[i] = ySequence_[i];
             }
        }
        else
        {
            forAll(ySequence_, i)
            {
```

```
        table_[k-1][i] = ySequence_[i];
    }
}
if (k != 0)
{
    extrapolate(k, table_, y);
    scalar err = 0.0;
    forAll(y, i)
    {
        scale_[i] = absTol_[i] + relTol_[i]*mag(y0_[i]);
        err += sqr((y[i] - table_[0][i])/scale_[i]);
    }
    err = sqrt(err/n_);
    if (err > 1.0/SMALL || (k > 1 && err >= errOld))
    {
        step.reject = true;
        dxNew = mag(dx)*stepFactor5_;
        break;
    }
    // tentative timestep estimation
    errOld = max(4*err, 1);
    scalar expo = 1.0/(k + 1);
    scalar facmin = pow(stepFactor3_, expo);
    scalar fac;
    if (err == 0.0)
    {
        fac = 1.0/facmin;
    }
    else
    {
        fac = stepFactor2_/pow(err/stepFactor1_, expo);
        fac = max(facmin/stepFactor4_, min(1.0/facmin, fac));
    }
    dxOpt_[k] = mag(dx*fac);
    temp_[k] = cpu_[k]/dxOpt_[k];

    if ((step.first || step.last) && err <= 1.0)
    {
```

```
    break;
}

if
(
    k == kTarg_ - 1
 && !step.prevReject
 && !step.first && !step.last
)
{
    if (err <= 1.0)
    {
        break;
    }
    else if (err > nSeq_[kTarg_]*nSeq_[kTarg_ + 1]*4.0)
    {
        step.reject = true;
        kTarg_ = k;
        if (kTarg_>1 && temp_[k-1] < kFactor1_*temp_[k])
        {
            kTarg_--;
        }
        dxNew = dxOpt_[kTarg_];
        break;
    }
}

if (k == kTarg_)
{
    if (err <= 1.0)
    {
        break;
    }
    else if (err > nSeq_[k + 1]*2.0)
    {
        step.reject = true;
        if (kTarg_>1 && temp_[k-1] < kFactor1_*temp_[k])
        {
```

```
                    kTarg_--;
                }
                dxNew = dxOpt_[kTarg_];
                break;
            }
        }

        if (k == kTarg_+1)
        {
            if (err > 1.0)
            {
                step.reject = true;
                if
                (
                    kTarg_ > 1
                 && temp_[kTarg_-1] < kFactor1_*temp_[kTarg_]
                )
                {
                    kTarg_--;
                }
                dxNew = dxOpt_[kTarg_];
            }
            break;
        }
    }
}
if (step.reject)
{
    step.prevReject = true;
    if (!jacUpdated)
    {
        theta_ = 2.0*jacRedo_;

        if (theta_ > jacRedo_ && !jacUpdated)
        {
            odes_.jacobian(x, y, dfdx_, dfdy_);
            jacUpdated = true;
        }
```

```
        }
    }
}

jacUpdated = false;

step.dxDid = dx;
x += dx;

label kopt;
if (k == 1)
{
    kopt = 2;
}
else if (k <= kTarg_)
{
    kopt=k;
    if (temp_[k-1] < kFactor1_*temp_[k])
    {
        kopt = k - 1;
    }
    else if (temp_[k] < kFactor2_*temp_[k - 1])
    {
        kopt = min(k + 1, kMaxx_ - 1);
    }
}
else
{
    kopt = k - 1;
    if (k > 2 && temp_[k-2] < kFactor1_*temp_[k - 1])
    {
        kopt = k - 2;
    }
    if (temp_[k] < kFactor2_*temp_[kopt])
    {
        kopt = min(k, kMaxx_ - 1);
    }
}
```

```
    if (step.prevReject)
    {
        kTarg_ = min(kopt, k);
        dxNew = min(mag(dx), dxOpt_[kTarg_]);
        step.prevReject = false;
    }
    else
    {
        if (kopt <= k)
        {
            dxNew = dxOpt_[kopt];
        }
        else
        {
            if (k < kTarg_ && temp_[k] < kFactor2_*temp_[k - 1])
            {
                dxNew = dxOpt_[k]*cpu_[kopt + 1]/cpu_[k];
            }
            else
            {
                dxNew = dxOpt_[k]*cpu_[kopt]/cpu_[k];
            }
        }
        kTarg_ = kopt;
    }

    step.dxTry = step.forward ? dxNew : -dxNew;
}
```

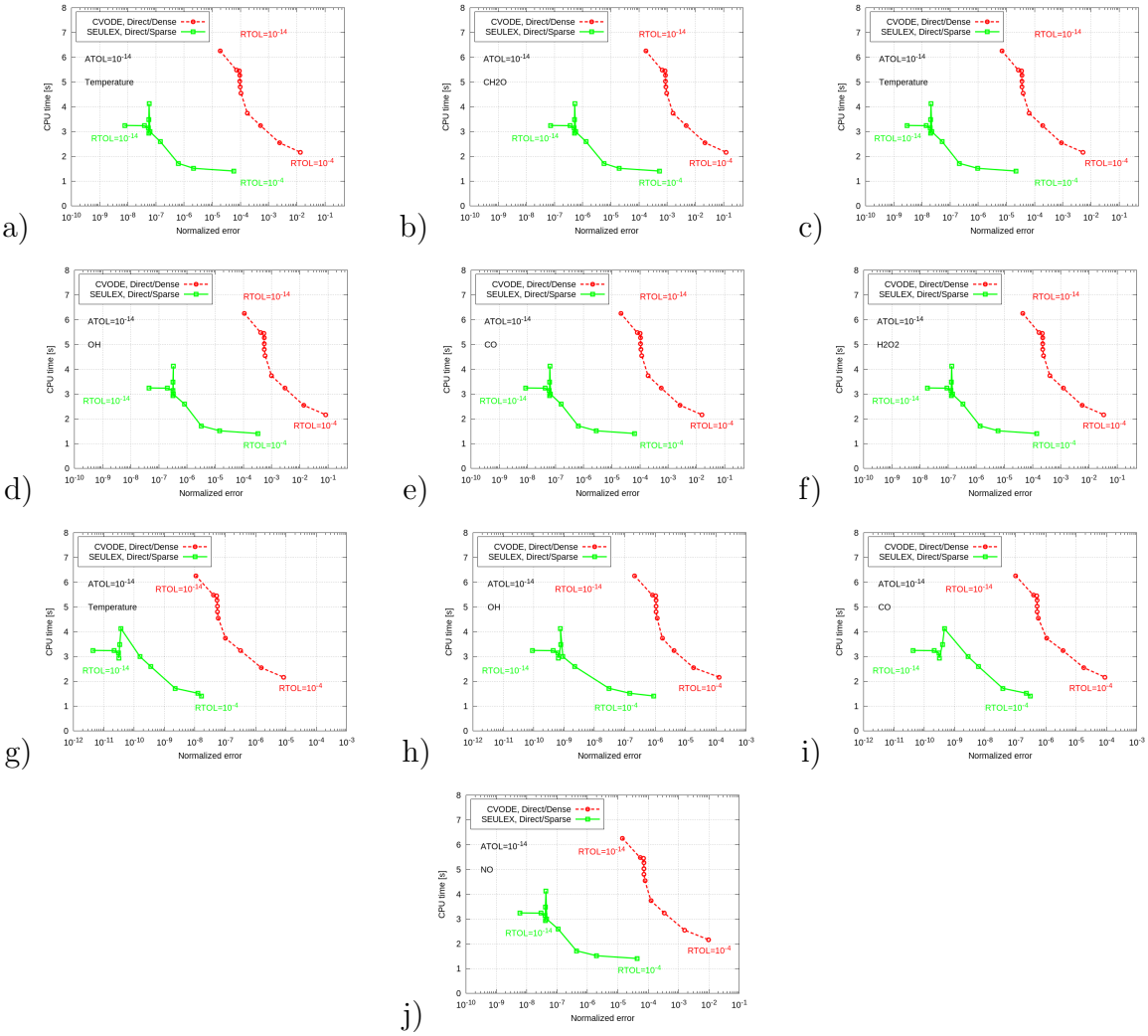

Figure 1: Relationships among user-specified error tolerances, global solution accuracy and CPU time for a homogeneous constant-pressure adiabatic system with reinitialization once every 10^{-6} s, for a BDF solver (CVODE) and an extrapolation solver (SEULEX). The stoichiometric n-heptane/air reactants are initially at 50 atm, 800 K. and a 42-species mechanism (including thermal NO) has been used. a) CPU time versus normalized error in computed temperature during first-stage ignition ($t = 0.00033$ s). b) CPU time versus normalized error in computed CH_2O mass fraction during first-stage ignition ($t = 0.00033$ s). c) CPU time versus normalized error in computed temperature during main ignition ($t = 0.00051$ s). d) CPU time versus normalized error in computed OH mass fraction during main ignition ($t = 0.00051$ s). e) CPU time versus normalized error in computed CO mass fraction during main ignition ($t = 0.00051$ s). f) CPU time versus normalized error in computed H_2O_2 mass fraction during main ignition ($t = 0.00051$ s). g) CPU time versus normalized error in computed temperature shortly after the end of main ignition ($t = 0.0006$ s). h) CPU time versus normalized error in computed OH mass fraction shortly after the end of main ignition ($t = 0.0006$ s). i) CPU time versus normalized error in computed CO mass fraction shortly after the end of main ignition ($t = 0.0006$ s). j) CPU time versus normalized error in computed NO mass fraction shortly after the end of main ignition ($t = 0.0006$ s).

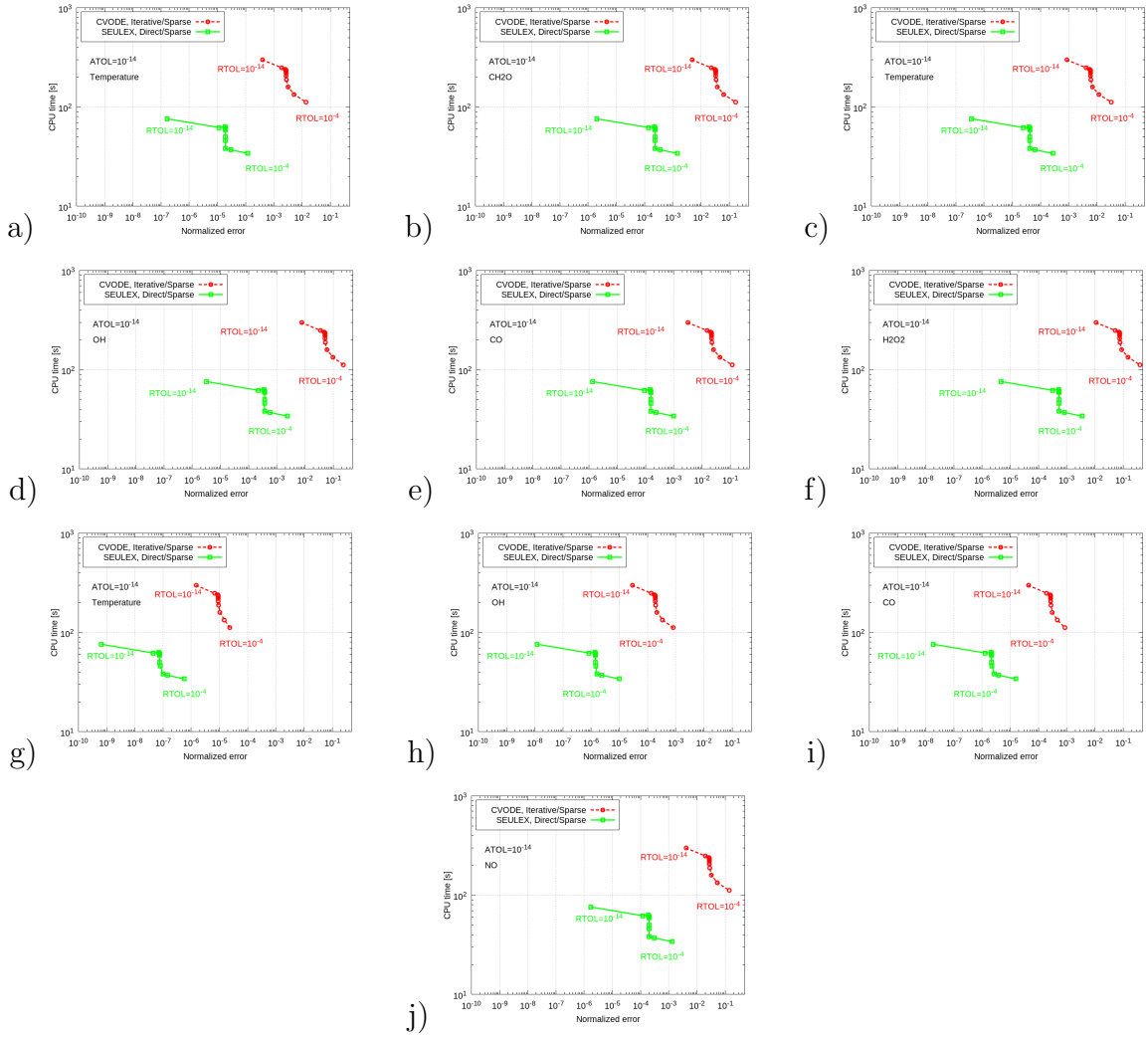

Figure 2: Relationships among user-specified error tolerances, global solution accuracy and CPU time for a homogeneous constant-pressure adiabatic system with reinitialization once every 10^{-6} s, for a BDF solver (CVODE) and an extrapolation solver (SEULEX). The stoichiometric n-heptane/air reactants are initially at 50 atm, 800 K. and a 658-species mechanism (including thermal NO) has been used. a) CPU time versus normalized error in computed temperature during first-stage ignition ($t = 0.00052$ s). b) CPU time versus normalized error in computed CH_2O mass fraction during first-stage ignition ($t = 0.00052$ s). c) CPU time versus normalized error in computed temperature during main ignition ($t = 0.000675$ s). d) CPU time versus normalized error in computed OH mass fraction during main ignition ($t = 0.000675$ s). e) CPU time versus normalized error in computed CO mass fraction during main ignition ($t = 0.000675$ s). f) CPU time versus normalized error in computed H_2O_2 mass fraction during main ignition ($t = 0.000675$ s). g) CPU time versus normalized error in computed temperature shortly after the end of main ignition ($t = 0.00071$ s). h) CPU time versus normalized error in computed OH mass fraction shortly after the end of main ignition ($t = 0.00071$ s). i) CPU time versus normalized error in computed CO mass fraction shortly after the end of main ignition ($t = 0.00071$ s). j) CPU time versus normalized error in computed NO mass fraction shortly after the end of main ignition ($t = 0.00071$ s).