SANDIA REPORT

SAND2003-8276
Unlimited Release
Printed May 2003

Cvode Component User Guidelines

Kylene Smith, Jaideep Ray and Benjamin Allan

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

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

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@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2003-8276
Unlimited Release
Printed May 2003

Cvode Component User Guidelines

Kylene Smith, Jaideep Ray and Benjamin Allan
High Performance Computing and Networking
Sandia National Laboratories
Livermore, CA 94551

Abstract

This report describes the wrapping of *cvode*, a serial library of BDF-based solvers for stiff ODE systems, into a CCA component. It also gives examples of how one loads in the Cvode Component into the CCA framework, (Sandia's dccafe) as well as how the interface to the component (called CvodePort) is used. The report concludes with some timing results whereby we empirically show that componentization results in a maximum 2% performance degradation on a single CPU.

The component can be obtained from Jaideep Ray (jairay@ca.sandia.gov, 925-294-3638).

Acknowledgements

We acknowledge the help of Sandia's ERI program in funding Kylene Smith for the summer of 2001 when the bulk of the work was done. Thanks are also due to Robert Armstrong for his help in recruiting Ms. Smith.

Table of Contents

1. Introduction 6
2. Inputs to the Cvode Component 8
3. The CvodePort : Description and Implementation 11
4. Details of the Cvode Component Implementation 16
    a. Example of Usage 23
5. Performance Testing and Results 26

Distribution 30

1. Introduction

The Cvode Component is a wrapper around LLNL's Cvode library. The Cvode library package is provided with the distribution. Find it at `Cvode_Home/tars/Cvode.tar.gz`. It is capable of solving systems of stiff ordinary differential equations. It contains a dense solver, banded solver, diagonal solver, and SPGMR (Generalized Minimal Residual (GMRES) method with scaling and preconditioning) solver. This component distribution contains a work horse (Cvode Component) and several examples of components that use the Cvode Component. The Cvode Component exports a simple interface known as a CvodePort and also requires the user to provide an additional port corresponding to the solver type which provides the right-hand-side function and Jacobian if necessary. For more about Cvode's capabilities see its user guide.

This project created the Cvode Component. Also provided with the distribution are multiple examples of components that work as the Driver Component or the Functions Component in the figure above. A Driver Component is a component that provides the initializations, configurations and calls. The Functions Component calculates the Y' and Jacobian. The ports, the smaller rectangles, show the naming conventions used. To use the Cvode Component, connect the driving component's CvodePort to the CvodeComponent's THE_CVODE port, and then connect the CvodeComponent's solver port which matches the matrix data structure of the FunctionsComponent. These connections are done in the `CCAFE_RC_FILE`. There are numerous examples in `Cvode_Home/Tests/`. For this example the script might look like this:

```
repository get CvodeComponent
repository get DriverComponent
repository get FunctionsComponent
create CvodeComponent CvodeSolver
create DriverComponent Driver
create FunctionsComponent Functions
```

```
connect Driver CV CvodeSolver THE_CVODE
connect CvodeSolver BAND Functions Band
```

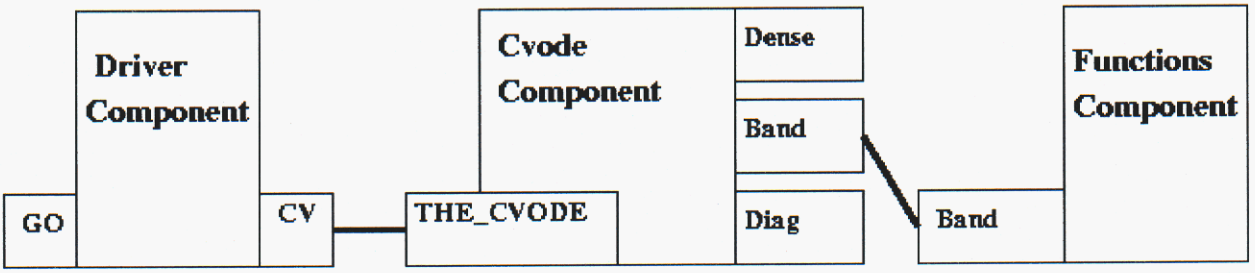

Figure 1. Schematic layout of an application code using Cvode Component

2. Inputs to the Cvode Component

The Cvode library requires a number of inputs to solve a given problem according to user specifications. The Cvode Component offers a few ways to configure these inputs:

1. Cvode Component defaults
2. Cvode Component's CONFIG port
3. Property Object

Any combination of these is acceptable.

To use **Method 1**, do nothing and the Cvode Component will enter its default values which have been set to the most common settings. This is only valid for some inputs, you should should refer to Table 1 for more information.

Method 2 is done in the CCAFE_RC_FILE through the CONFIG port that Cvode Component provides. The general form is:

```
configure CvodeComponentName CONFIG InputName value
```

Example:

```
repository get CvodeComponent
create CvodeComponent CvodeSolver
configure CvodeSolver CONFIG itol SV
```

For the input names and appropriate values see Table 1. More information about the inputs can be found in the Cvode user guide (names were preserved). This port is a very convenient way to set almost all of the inputs. The few exceptions come when vectors or arrays are involved. The input "itol" can be set to "SS" or "SV" where the first letter indicates a type for relative tolerances and the second for absolute tolerances (S= Scalar, V=Vector). If itol = SV the list of absolute tolerance values should be specified later. Also use another method for "iopt" and "ropt" array values. For examples of using a script to configure inputs look at the files in `Cvode_Home/Tests/`.

Method 3, uses the object that holds all of the inputs called a **PropertyObject**. It can be obtained through the CvodePort and queried to set an input. Again, see Table 1 for names

and values of inputs that can be set this way. The general format, after obtaining the property object from the CvodePort is:

```
PropertyObjectName->setProp("InputName", value);
```

Example (assuming you have already obtained the CvodePort and called it Port):

```
PropObj* PO;
PO = Port->get_PropObj();
PO->setProp("lmm", "BDF");
```

One exception is that the "iopt" and "ropt" inputs take an index between the name and value, also they only take effect when "optIn" is set to true.

Example:

```
PO -> setProp("iopt", "MXSTEP", 1000);
```

There are examples of using the Property Object in `Cvode_Home/examples/Driver/BandDense/src/BD.cxx` and `Cvode_Home/examples/Driver/BandDense/src/DB.cxx`.

Table 1. Cvode Component inputs. Entries marked with ** are used only if optIn is set to TRUE.

Input Name	Meaning	Type	Default	Valid Values	Ways to initialize
N	N. of equations	int		> 0	Script, PropertyObject
lmm	Linear Multistep Method	string	BDF	Adams, BDF	Script, PropertyObject
iter	Internal Iteration Type	string	NEWTON	FUNCTIONAL, NEWTON	Script, PropertyObject
itol	Tolerance Type	string	SV	SS,SV	Script, PropertyObject
errfp	Error file	string	stdout	Any filename	Script PropertyObject
t0	t(initial)	double	0.0	Any	Script, PropertyObject
reltol	Relative Tolerance	double	1e-6	>=0	Script, PropertyObject
abstol	Absolute	double	1e-6	>=0	Script,

	Tolerance				PropertyObject
optIn	Optional Inputs	bool	FALSE	True, False	Script, PropertyObject
mupper	Upper Bandwidth	int	1	>0	Script, PropertyObject
mlower	Lower Bandwidth	int	1	>0	Script, PropertyObject
maxl	Maximum Krylov dimension	int	0	>=0	Script, PropertyObject
pretype	Preconditioner type	string	NONE	NONE, LEFT, RIGHT, BOTH	Script, PropertyObject
gs_type	Gram-Schmidt Orthogonalization	string		MODIFIED_GS, CLASSICAL_GS	Script, PropertyObject
delta	Used to get linear tolerance	double	0.0	>=0	Script, PropertyObject
UseDefaultJac	Use Cvode's default Jacobian	bool	FALSE	True, False	Script
PrintStats	Print final stats upon free()	bool	FALSE	True, False	Script
LinearSolver	Linear Solver Type	string	BAND	DENSE, BAND, DIAG, SPGMR	Script, PropertyObject
iopt**	Integer Optional Inputs	array	Next three	entries describe	valid indices
MAXORD	Maximum lmm order	int	5	1 to 5	PropertyObject
MXSTEP	Max. no. of internal steps	int	500		PropertyObject
MXHNIL	Max no. of warning messages	int	10		PropertyObject
ropt**	Real Optional Inputs	array	The next	three entries	show valid indices
H0	Initial step size	double			PropertyObject
HMAX	Max. absolute step size	double	infinity		PropertyObject
HMIN	Min. absolute step size	double	0		PropertyObject

3. The CvodePort : Description and Implementation

The CvodePort interface is the main way CvodeComponent interacts with its users. The main Cvode calls have been simplified down to three requiring a fraction of the inputs. The other inputs have been taken care of through other input methods (see the Input section above). The function "setup" with its inputs will do required memory allocation and variable initialization for the ODE problem. Iterate on the "solve" function to integrate the problem as you wish. Call the "free" function to clean up. The only other functions the CvodePort interface provides is a derivative function which corresponds to the Cvode library's CVodeDky should you wish to use it and access methods to retrieve output and set input. The following is a more detailed description of this implementation of the port. In examples provided for clarity it is assumed that the port is already obtained and named Port.

```
class CvodePort: public virtual gov::cca::Port
{
 public:
    CvodePort() : gov::cca::Port() {}
    virtual ~CvodePort() {}

    virtual int setup (double* y, double* abs=NULL,
                       void* f_data=NULL, void* jac_data=NULL)=0;

    virtual int solve (double t_desired)=0;

    virtual void free ()=0;

    virtual int D_k(double t, int k)=0;

    virtual PropObj* get_PropObj() = 0;

    virtual void get_result(double*& data, int& size)=0;
```

```
    virtual double get_result_by_index(int index)=0;

    virtual double get_max()=0;

    virtual double get_t()=0;

    virtual void get_dky(double*& data, int& size)=0;

    virtual double get_dky_by_index(int index) = 0;
};
```

These methods are documented below:

1) **int CvodePort::setup(double* y, double* abs=NULL, void* f_data=NULL, void* jac_data=NULL) :** The setup function is the equivalent of calling CVodeMalloc with all its inputs and then calling the appropriate CVxxx where xxx is the type of linear solver that has been specified. For this function you must specify an array of values that are the initial y values. If using Vector absolute tolerances, you must also provide the array of those values as the second argument. If using Scalar absolute tolerances and have already set the value, you do not need the second argument. Unless, you plan to use the last two then simply pass an explicit NULL. If you have not set the value, pass a pointer to it. The last two arguments are pointers to data structures you have allocated and filled up for use in your functions that Cvode will call. They are optional. They are passed around as void* so if you wish to use them in your functions both the driver and functions components that interact with CvodeComponent must know the data structure definition (your responsibility) and you will have to do the casting back to this type in your function. "jac_data" is the "p_data" argument in the case of the SPGMR solver. This function returns 0 upon successful completion or -1 for an error. Examples:

   a. `double y_values[]={1, 0, 0};  Port->setup(y_values);`
   b. `double atol_vec[]={1e-6, 1e-14, 1e-8};  Port->setup(y_values, atol_vec);`
   c. `double atol_scl = 1e-4;  Port->setup(y_values, &atol_scl);`

```
d. my_Struct f_data, jac_data;   Port->setup (y_values,
   &atol_scl, f_data, jac_data);
e. Port->setup (y_values, NULL, f_data, jac_data);
f. Port->setup (y_values, NULL, NULL, jac_data);
```

2) **int CvodePort::solve(double t_desired) :** The solve function is equivalent to the call CVode. All configuration inputs have already been established at this point, simply pass the time to which you want to integrate. The function returns a 0 for success and a negative number for an error. The error codes are: -1 = CVODE_NO_MEM, -2 = ILL_INPUT, -3 = TOO_MUCH_WORK, -4 = TOO_MUCH_ACC, -5 = ERR_FAILURE, -6 = CONV_FAILURE, -7 = SETUP_FAILURE, -8 = SOLVE_FAILURE, -9 = A setting in the properties object was invalid. Example:

```
a. double t=.1;    Port->solve(t);
```

3) **void CvodePort::free() :** The free function tells the Cvode Component that you are done solving this particular problem and frees up the associated Vector and Cvode memory. No inputs are needed. If you have set PrintStats to true this will cause those outputs to be displayed at the standard output. Example: `Port->free();`

4) **int CvodePort::D_k(double t, int k) :** This is equivalent to CVodeDky. The function D_k can be used to compute a derivative. Inputs are the time at which you want the derivative and which derivative you want. The value of t must be within the range of the last internal step of the most recent call to setup. The function returns 0 for a success and a negative value for and error. The corresponding error codes are: -1 = BAD_K, -2 = BAD_T, -3 = BAD_DKY, -4 = DKY_NO_MEM.. Example:

```
a. double t=.1;    Port->solve(t); Port-> D_k (t, 2);
```

5) **PropObj* CvodePort::get_PropObj() :** To use **Method 3** for input as described above you need to use this accessor function. This method is the only way to change a setting after execution has begun. It takes no inputs but will return a pointer to the Property Object so that you can then use its member functions to get set the properties accordingly. Example:

```
a.  PropObj * PO;    PO = Port->get_PropObj();
```

6) **void CvodePort::get_result(double*& data, int& size)** : This function combines the functionality of N_VData and N_VLength invoked on the results from the most recent call to CVode. In order to access the results from a solve step, give get_result a pointer to a double and an integer. You do not need to allocate any memory for the array, but you do for the integer. CvodeComponent will take the addresses of these inputs and give you a copy of the size at the location of the integer and a pointer to the data. This data will be written over by the next solve step so make sure to use it now or copy it somewhere if you want to access it later. Example:

```
a. double t=.1, *Y_data; int Y_size;  Port->solve(t); Port-
   >get_result(Y_data, Y_size);
```

7) **double CvodePort::get_result_by_index(int index) :** Similar to N_VIth, this function gives access to just one element of the result. Pass an index between 0 and N-1 (see Table 1 for the definition of N) the corresponding result from the most recent solve step is returned. Example:

```
a. double t=.1, y1;  Port->solve(t);
   y1 = Port->get_result_by_index(1);
```

8) **double CvodePort::get_max() :** Sometimes, you may only be interested in the largest result (in y) from a solve step. This function is like N_VMax, it will return the maximum value found amongst the results. Example:

```
a. double t=.1, y_max;  Port->solve(t);
   y_max = Port->get_max();
```

9) **double CvodePort::get_t() :** Use this function to get the exact value of time actually reached for the most recent solve step. Call with no inputs and the value is returned. Example:

```
a. double t_wanted=.1, t_obtained;  Port->solve(t_wanted);
  t_obtained = Port->get_t();
```

10) **void CvodePort::get_dky(double*& data, int& size) :** This function provides similar functionality to get_result. It takes the same types of arguments but the values come from a derivative step obtained with D_k in this instance. Pass a pointer to double (no memory allocation) and integer (allocated). The function gives you a pointer to the data and the number of elements. Example:

```
a. double t=.1, *d_data; int d_size;  Port->solve(t);
   Port->D_k(t, 1); Port->get_dky(d_data, size);
```

11) **double CvodePort::get_dky_by_index(int index) :** Also similar to its counterpart get_result_by_index, you supply the index between 0 and N-1, the function returns the corresponding value returned from the latest D_k step. Example:

```
a. double t=.1, d_1;  Port->solve(t);  Port->D_k(t, 1);
    d_1 = Port->get_dky_by_index(1);
```

4. Details of the Cvode Component Implementation

This section is dedicated to detailing every inner working of all methods both public and private for the Cvode Component implementation should anyone need to fully understand how or why things are being done or need to understand for maintenance purposes. It is probably not interesting nor necessary for the user to know (i.e. the point of CCA). The source code for the Cvode Component is found in five files split up according to their purposes. `CC.cxx` is the main file where general top level stuff is found. Here lies the constructor, destructor, and setServices (to make it a legal component). `CC_cvode.cxx` has the functions that make the calls to the library. So setup, solve, free and D_k are here. The `CC_access.cxx` file has all the methods for the user to access inputs and outputs. The method to get a Property Object and all return functions for retrieving outputs from integration are found here. The last file with CvodeComponent members in it is `CC_priv.cxx`. The file has all the private methods that do behind the scenes work. Finally, `CC_myfunc.cxx` has generic right-hand-side, Jacobian, and Preconditioning functions which are always passed to the library's functions and then they in turn call the user's actual functions. This was done because the library requires a static function and it cuts down on the number of arguments the user needs to deal with since many of the original arguments on these functions were there only for the default case anyway. The Cvode Component header file follows this order and we use it as a guide to detailing the functions.

```
class CvodeComponent: public virtual gov::cca::Component,
                      public virtual CvodePort
{
 public:
    CvodeComponent();
    virtual ~CvodeComponent();
    virtual void setServices (gov::cca::Services *svc);
```

```
    virtual int setup (double* y, double* abs=NULL,
                       void* f_data=NULL, void* jac_data=NULL);
    virtual int solve (double t_desired);
    virtual void free ();
    virtual int D_k(double t, int k);

    virtual PropObj* get_PropObj();
    virtual void get_result(double*& data, int& size);
    virtual double get_result_by_index(int index);
    virtual double get_max();
    virtual double get_t();

    virtual void get_dky(double*& data, int& size);
    virtual double get_dky_by_index(int index);

private:
    void init();
    void allocate();
    void setupParameters();
    void PrintFinalStats();
    void PrintErrorStats();
    void set_initial(double* val);
    int set_abstol (double* val);

    IntParameter    *N, *maxl, *mupper, *mlower;
    DoubleParameter *t0, *abstol, *reltol, *delta;
    BoolParameter   *optIn, *PrintStats, *UseDefaultJac;
    StringParameter *errfp, *LinearSolver, *lmm, *iter,
                        *itol, *itask, *pretype, *gs_type;

    ConfigurableParameterPort *pp;
    gov::cca::Services *psvc;
    DensePort* Dense;
    BandPort* Band;
```

```
    DiagPort* Diag;
    SpgmrPort* Spgmr;

    void *cvode_mem, *MachEnv;
    PropObj*  PO;
    myStruct *my_data_f, *my_data_jac;
    double T, ATOL, RTOL;
    N_Vector y0, Y, ATOL_V, dky;

    bool isInitialized, unusable, isMall, isAlloc;
    int SetupCount;
};
void my_f (int N, double t, N_Vector y, N_Vector ydot,
           void* f_data);

void my_Dense_Jac(int N, DenseMat J, RhsFn f, void *f_data,
                  double t, N_Vector y, N_Vector fy,
                  N_Vector ewt, double h, double uround,
                  void *jac_data, long int *nfePtr,
                  N_Vector vtemp1, N_Vector vtemp2,
                  N_Vector vtemp3);


void my_Band_Jac(int N, int mupper, int mlower, BandMat J,
                 RhsFn f, void *f_data, double t, N_Vector y,
                      N_Vector fy, N_Vector ewt, double h,
                 double uround, void *jac_data, long int *nfePtr,
                 N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3);

int my_Precond(int N, double t, N_Vector y, N_Vector fy,
               bool jok, bool * jcurPtr, double gamma,
               N_Vector ewt, double h, double uround,
               long int *nfePtr, void *P_data, N_Vector vtemp1,
                    N_Vector vtemp2, N_Vector vtemp3);
```

```
int my_Psolve(int N, double t, N_Vector y, N_Vector fy,
              N_Vector vtemp, double gamma, N_Vector ewt,
              double delta, long int *nfePtr, N_Vector r,
              int lr, void* P_data, N_Vector z);
```

1) **CvodeComponent::CvodeComponent() :** The constructor takes no arguments. It initializes the pointers, zeros the counters, sets the logic variables, and ends by displaying a line to tell the user it was instantiated properly. The bulk of its work in the middle is memory allocation. Data structures that will carry the user's f_data and jac_data as well as information about how to contact the appropriate functions are allocated. PO stands for PropertyObject. The constructor allocates one such object to hold all the inputs for Cvode. Initial settings will come from parameters configured in the script, or default values if nothing is set, but can later be changed by the user. This can happen up until the setup function is called or after the function free is called. The rest of the allocations are Parameters. To use the parameters set them using the script. See the Inputs section or examples for more details. Briefly this is what they represent. "LinearSolver" is what type of solver to use the default is a BandSolver. "N" is the number of equations and is the magic number for how big all arrays and vectors need to be. This is characterisic of the ODE to be solved must be set by the parameter or via the PropertyObject for the problem to work. "mupper" and "mlower" are the upper and lower bandwidths respectively in a BandMatrix. Their default is 1. "lmm" stands for Linear Multistep Method. It is an input to CVodeMalloc called in setup. The default value is BDF, the only other valid value is ADAMS. The Cvode user guide suggests using BDF for stiff problems. "iter," also a CVodeMalloc input, is the type of internal iteration. Default is NEWTON which the user guide recommends for stiff problems. FUNCTIONAL is also valid. "itol" is the type of tolerances for the problem. It has two valid values SV, default, and SS. S stands for Scalar, V for Vector. The first letter is for they type of relative tolerances and then second for the type of absolute tolerances. "itask" is an input for CVode and describes the solvers job. NORMAL, the default, iterates via internal steps until the time out desired is reached or surpassed. ONE_STEP is also legitimate and results in only one internal step. "t0" is the time initial. It defaults to 0.0 but you may start

your integration with any time you need. The next two parameters are the values of the tolerances. "reltol" is for relative tolerances. It is always a scalar value greater than or equal to 0. The default is 1e^-6. "abstol" is for absolute tolerances. The value can only be set this way if you are using scalar tolerances (i.e. itol has been set to SS). The value can be anything greater than 0; the default is 1e^-6. "optIn" needs to be set to true if you wish to set any of the iopt or ropt values to something other than their defaults. Its default is false. "errfp" is the name of a file you wish to have the Cvode library drop its error messages in. Any messages generated by the Component will still be sent to the standard output, which is also the default for the library and this parameter. The next four parameters only apply to a SPGMR solver. "delta" is the factor that the nonlinear iteration tolerance is multiplied by to get the linear tolerance value. the default value for the parameter is 0 which invokes Cvode's default of 0.05. "pretype" is the type of preconditioning. Default is NONE, other values are RIGHT, LEFT, or BOTH. "gs_type" is the Gram-Schmidt orthogonalization type. Values are CLASSICAL_GS or MODIFIED_GS there is no logical default. "maxl" is the maximum Krylov dimension like delta a 0, which the Component defaults to, invokes Cvode's default of 5. The final two parameters are for the Component's use. "PrintStats," default = false, will cause final statistics from iopt and ropt to be printed upon a call to free when set to true. "UseDefaultJac," also defaulted to false, indicates whether to use Cvode's default Jacobian (for Dense or Band solvers) or the user defined one. True means use the default.

2) **CvodeComponent::~CvodeComponent() :** This destructor frees all the memory from the parameters and the PropertyObject. If there is still memory allocated for the Vectors or Cvode memory. That is also freed. The logic variables are cleared and any ports that the Component is holding are released. A final note is displayed to the user to indicate destruction.

3) **CvodeComponent::setServices(gov::cca::Services *svc) :** This is a function that all CCA Component's must define. The first part links the component up with CCA services and ends if this is not accomplished. This is standard on all components.

The next bunch of commands setup the Component's ability to support parameters. The last thing this function does is tell the rest of CCA that it may need to use a DensePort, BandPort, DiagPort, and SpgmrPort; and that it will provide a CvodePort and ConfigurableParameterPort. It also gives these ports a name which it should be called for connecting and configuring in the `CCAFE_RC_FILE`.

4) **int CvodeComponent::setup(double* y, double* abs=NULL, void* f_data, void* jac_data=NULL) :** The setup function itself calls for initialization and allocation if that has not been done already. These functions will be detailed later but they read in the Parameter inputs and allocate the Y and tolerance vectors respectively. The data you passed to this function is then filled into the proper vectors with set_initial and set_abstol. Retrieve a bunch of arguments from the Property Object needed by CVodeMalloc shortly. The switch statement connects to the appropriate port and tells the data structure how to find the port. The rest of the Data Structure is filled in with size, solver type and user data. CVodeMalloc is called next with the appropriate arguments depending on absolute tolerance type. The next switch statement calls the appropriate `CVxxx` function with the necessary arguments. This is also where CvodeComponent handles whether to use the default Jacobian or not. Unless Cvode memory is not able to be allocated the function returns a 0.

5) **int CvodeComponent::solve(double t_desired) :** The solve function returns error code -9 if the Component has been marked unusable for some reason. This can happen initialization. Next CVode is called. If it returns a negative value, error statistics from iopt and ropt are displayed on the standard output. Finally, the value returned by CVode is returned to the user.

6) **void CvodeComponent::free() :** If the "PrintStats" parameter was activated. Final Statistics from iopt and ropt are printed. Memory allocated with CVodeMalloc, provided it was successful, is freed. Memory allocated for vectors is freed. The logic variables are reset and any ports obtained are released.

7) **int CvodeComponent::D_k(double t, int k) :** If the memory to hold derivative results has not been allocated do that first. Then call the CVodeDky function and return whatever value it returns (O for success, negative for an error) to the user. All of the functions in `CC_access.cxx` and `CC_priv.cxx` are small and self explanatory.

8) **void my_f (int N, double t, N_Vector y, N_Vector ydot, void* f_data) :** This function provides a static function to CVodeMalloc. It allows us to hide the details of a N_Vector from the user. This function is an argument to CVodeMalloc and called within the CVode library in calls to CVodeMalloc and CVode. The data passed here in f_data is in the form of myStruct filled up in setup. Here it is taken apart so we have the problem size, type, and user's original f_data. The data from "y" and "ydot" is transferred into a cca_vec so that it is usable by the users defined "f" provided by an appropriate solver port. The switch statement sends the problem off with the new arguments to the appropriate port depending on type and the pointer from the data structure.

9) **void my_Dense_Jac (int N, DenseMat J, RhsFn f, void *f_data, double t, N_Vector y, N_Vector fy, N_Vector ewt, double h, double uround, void *jac_data, long int *nfePtr, N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3) :** This function comes out of the same theory as my_f. Since it is only used for dense problems, it just disassembles the data structure for jac_data and size, converts to cca_vec and cca_mat from their CVode forms where appropriate and passes it on to the user defined Jacobian. The argument list has been simplified to only include those needed by the user. The others are a part of the prototype only for the default purpose.

10) **void my_Band_Jac (int N, int mupper, int mlower, BandMat J, RhsFn f, void *f_data, double t, N_Vector y, N_Vector fy, N_Vector ewt, double h, double uround, void *jac_data, long int *nfePtr, N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3) :** This function is very similar to its Dense counterpart. The

matrices are handled slightly differently. Since BandMatrices are stored in a very specific way, it is necessary for the user code to continue to handle them in the same fashion. The Band Port contains the structure. The Jacobian matrix is passed on as a BandMat_cca which is defined on the BandPort. The vectors are changed as usual to cca_vec and data is extracted the same way from the data structure.

11) **int my_Precond(int N, double t, N_Vector y, N_Vector fy, bool jok, bool * jcurPtr, double gamma, N_Vector ewt, double h, double uround, long int *nfePtr, void *P_data, N_Vector vtemp1, N_Vector vtemp2, N_Vector vtemp3) :** This is the equivalent of the Jacobian functions for a SPGMR solver. Vectors are converted to cca_vec. Data extracted from P_data and control passed onto the user's function. The return value is equivalent to whatever the user returns.

12) **int my_Psolve(int N, double t, N_Vector y, N_Vector fy, N_Vector vtemp, double gamma, N_Vector ewt, double delta, long int *nfePtr, N_Vector r, int lr, void* P_data, N_Vector z) :** This function is also for the SPGMR solver. Data is extracted, vectors changed, and control passed off to the user's function. Return value is whatever the user returns.

4.1 Example of Usage

One place the CvodeComponent might be used is to compute Chemistry reactions. Within this distribution, there is one such example provided. In the directory, `Cvode_Home/examples/Drivers/ChemDriver/` there is a DriverComponent and in the directory, `Cvode_Home/examples/Functions/ChemFunc/` there is a Functions Component. The problem is run with the script `Cvode_Home/Tests/ChemTest.scr` Both components have the usual constructor, destructor, and setServices functions. The interesting work though is done in the go function on the DriverComponent and the f_band function of the FunctionsComponent. Here's how they work.

1) **void Driver::init() :** This function reads in the Drivers parameters, obtains the CvodePort connection and initializes some data. The Driver provides three parameters. The first and most important is called DT it should be set to the time you want to integrate to. Since this is just a test problem the only way to configure this parameter is use the default (0.1) or set it to your own value in the CCAFE__RC_FILE. The other parameters deal with how you want to solve the problem. The normal way, which is fastest and what the program defaults to, is to solve all the cells at once. Each cells consists of temperature + 8 species (make sure N on the CvodePort is set to a multiple of 9). This initialization function which will be called by the go function sets up the problem size, filling in a data structure that will tell f how many variables and cells the problem contains. The data is then filled into an array of appropriate size depending on the number of cells (N/9) that you have decided on. If you would rather solve on a cell by cell basis (N should be 9) you need to configure the other two parameters. They are "All" which should be set to false. and Num_Cells which should be set to the number of cells you want. The variables are set up for this case two except cells is set to one because the Functions side will have no idea that there are more than one since each problem is separate. The array is still initialized for all the cells at this time though.

2) **int Driver::go() :** This function is how the user starts the problem. Depending on which type of solving you have chosen (cells done all at once or one at a time), this function calls the appropriate private function after initialization. If you are using the Cvode Component as part of a larger simulation this type of function could be provided by any type of port that has the capability to connect to another component and call it appropriately.

3) **void Driver::all_at_once() :** If the parameter "All" is true (default) this function is invoked. It calls the setup function. It breaks the "DT" up into smaller parts and loops on solve until it has reached your destination. It retrieves the data and puts it into the original locations and prints it out on the screen.

4) **void Driver::one_by_one() :** This function is invoked to do things one cell at a time. This is very poor memory management as it entails numerous CVodeMalloc and Free calls which tend to be slow and is not recommended. It makes loops around the cells making its own copy of a cell so that the arrays are the correct size. Then it goes through the usual calls of setup, solve, free. The data is displayed as it is obtained. If PrintStats is true you will receive statistics on a cell by cell basis as well.

5) **void Functions::f_band (int N, double t, cca_vec* y, cca_vec* ydot, void* f_data) :** This function puts the data into the format needed by the mechanism and calls the mechanism. It loops over the cells and the mechanism is called on each cell individually. After the mechanism finishes, this function puts the data into the proper location of the ydot vector.

6) **void Functions::Jac(int N, int mupper, int mlower, void* J, void* f_data, double t, cca_vec* y, cca_vec* ewt, void* jac_data) :** This function contains no operations because we have opted to use Cvode's default Jacobian function by setting UseDefaultJac to true in the CCAFE_RC_FILE.

7) **void Functions::ckwyp (double* P, double* T, double* Y, double* WDOT) :** This function is the real work horse. It computes the chemical reaction for a given cell that it is sent given the f function.

In the script there are a number of examples of using the script to configure settings for Cvode as well as the instantiations of the components within the demo and the connections are made.

5. Performance Testing and Results

As a second part of this project, after the development of the new component, we were testing the performance hit do to CCA. In the tests, the same problem was run on the library version of Cvode and the component version. A problem consists of solving a system on a given number of identical cells. The number of cells varies on the x-axis. Figures 3-7 document the results of runs on a Chemistry problem for different time steps. The number of f evaluations rose from 58 to 920, and the number of Jac evaluations varied between 1 and 22. Figure 2 is from the dense problem supplied with the Cvode library adapted to run on a Band Port so we could vary the number of cells as before. As the graphs show. For a problem of at least 100 cells there is no more than a 2% performance hit. This was an encouraging finding.

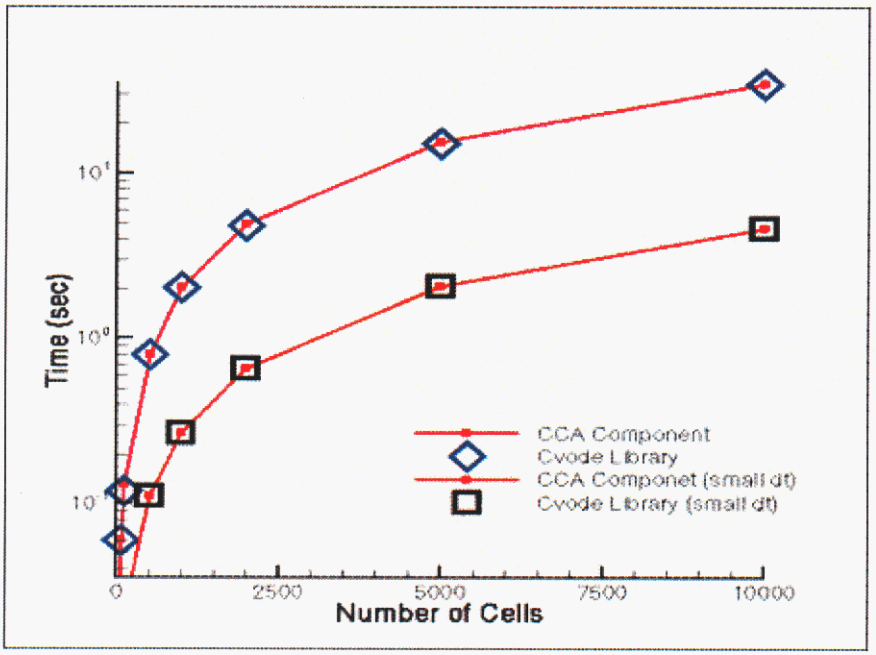

Figure 2. Time taken as a function of the size of the problem.

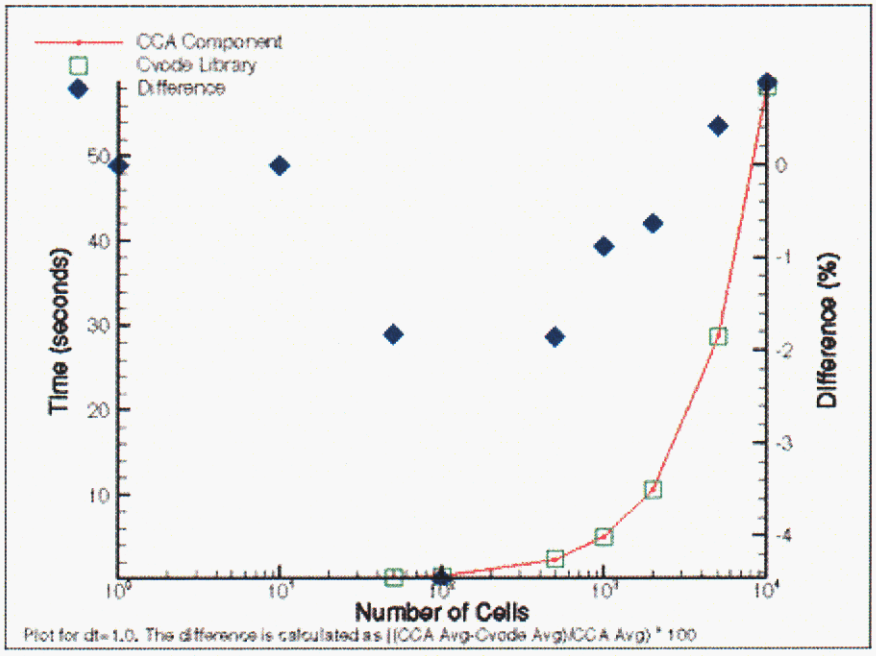

Figure 3. Run time as a function of problem size for the chemistry problem. dt = 1.0. Runs were done with the library the component form of Cvode. The blue dots show difference as percentage (plotted on the right Y-axis).

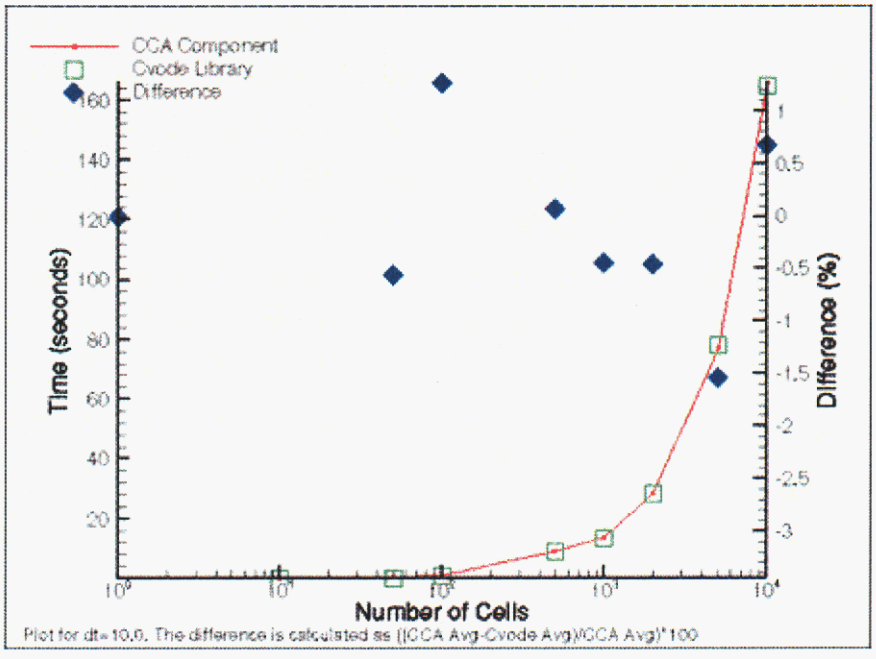

Figure 4. Run times for the chemistry problem, but for dt = 10.0. The runs were done with the library and component form of Cvode.

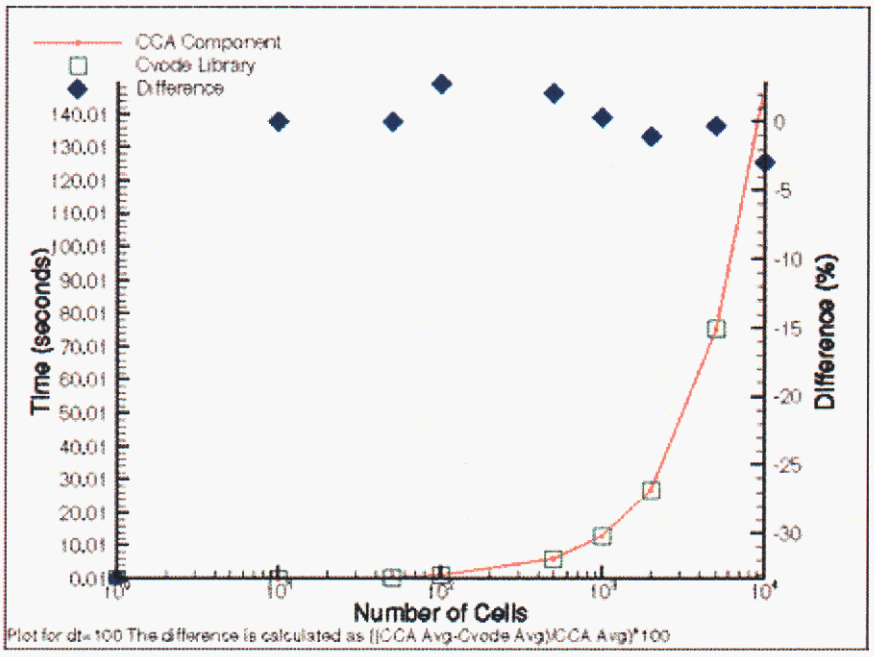

Figure 5. Run times for the chemistry problem with dt = 100.00

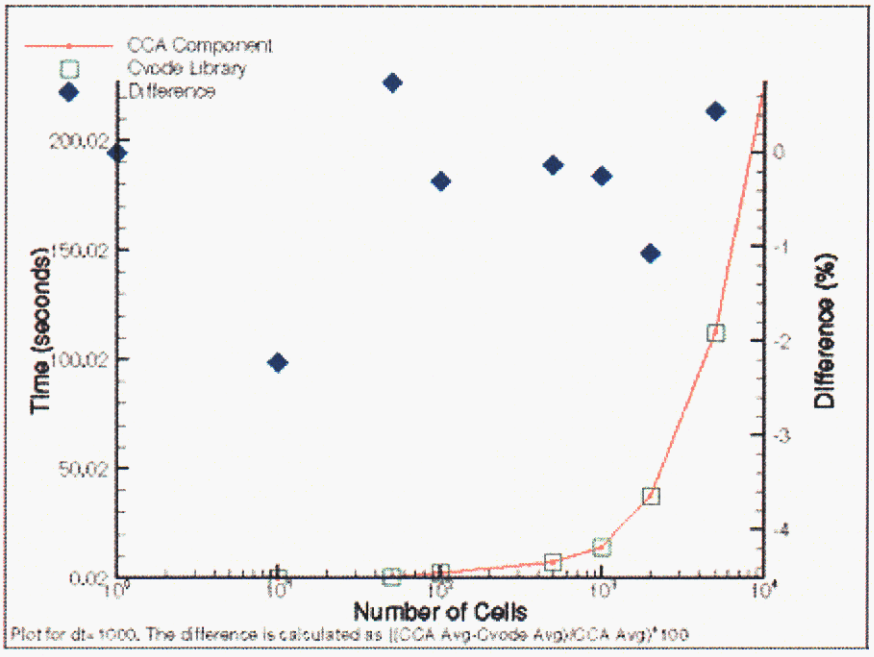

Figure 6. Run times for the chemistry problem with dt = 1000.0

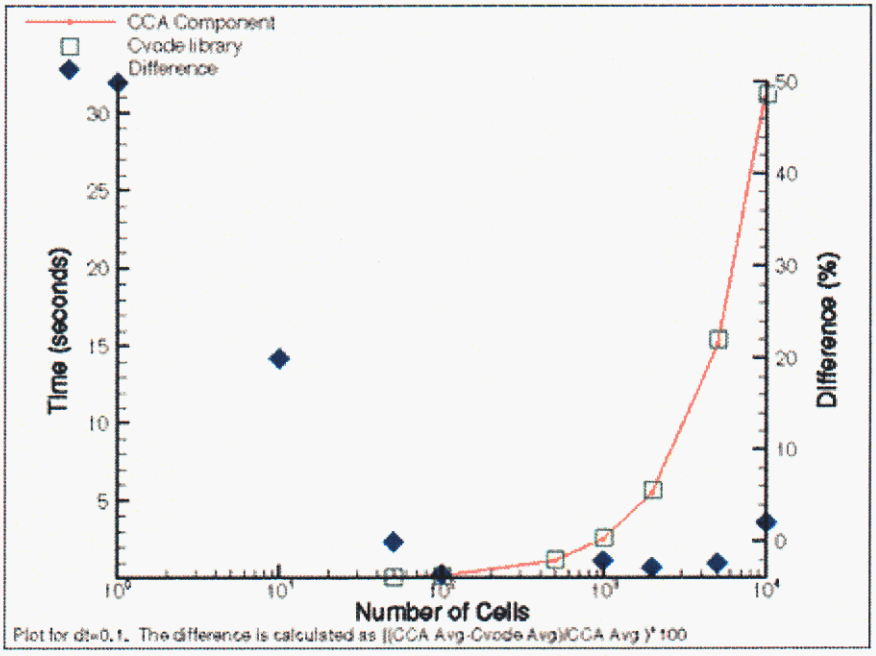

Figure 7. Run times for the component and library form of Cvode for the chemistry problem for dt = 0.1.

Distribution

Internal Distribution

1	MS 9051	Jaideep Ray, 8961
1	MS 9915	Benjamin Allan, 8961
3	MS 9018	Central Technical Files, 8945-1
1	MS 0899	Technical Library, 9616
1	MS 9021	Classification Office, 8511 for Technical Library, MS 0899, 9616
		DOE/OSTI via URL