PySUNDIALS Documentation

Release 2.3.0-rc1

James Dominy

December 29, 2008

CONTENTS

Contents:

CHAPTER

ONE

Introduction

PySUNDIALS is a python package providing python bindings for the SUNDIALS suite of solvers. It is being developed by the triple ‘J’ group at Stellenbosch University, South Africa. While python bindings for SUNDIALS will hopefully be generally useful in the computational scientific community, they are being developed with the specific aim of providing a robust underlying numerical solver capable of implementing models conforming to the Systems Biology Markup Language specification (version 2), including triggers, events, and delays. As such the development process is partially driven by the continuing parallel development of PySCeS.

This documentation serves to introduce and act as a reference for PySUNDIALS. As such it focuses on the differences in behaviour between PySUNDIALS and SUNDIALS. If you wish to know more about the underlying mathematical considerations, or wish a more in depth discussion of how to go about using SUNDIALS in general, the SUNDIALS documentation is the place to start. We assume a basic knowledge of initial value problems and their computational solutions on the part of the reader.

CHAPTER

TWO

Installation

2.1 Prerequisites

PySUNDIALS requires the following

- a working copy of SUNDIALS installed with shared libraries compiled
- Python v2.4 with the `ctypes` module, OR Python v2.5 or higher

The python packages `numpy` and `scipy` are recommended but not necessary.

2.2 From Source

The latest release version of PySUNDIALS can be download at http://sourceforge.net/projects/pysundials, or alternatively, the very latest (potentially non-functional) version via anonymous svn using the command:

```
$ svn co https://pysundials.svn.sourceforge.net/svnroot/pysundials pysundials
```

2.2.1 On Linux/BSD or other POSIX

- Download and untar the complete SUNDIALS suite.
- `$ ./configure -enable-shared`
- `$ make && make install`
- Download and untar PySUNDIALS
- Change to the directory where you unpacked PySUNDIALS
- `$ python setup.py install`

2.2.2 On Windows using MSys/MinGW

- Download and untar the complete SUNDIALS suite somewhere inside MSys.
- Do *not* run aclocal, autoconf, or autoheader, or the makefiles will break!
- `$ ./configure -enable-shared`

- `$ make && make install`
- Download and untar PySUNDIALS
- Change to the directory where you unpacked PySUNDIALS
- `$ python setup.py -c mingw32 install`

If you receive a compile time error when executing the final command, which complains about missing `sundials_conf.h`, or other missing `.h` files, set the `CPATH` environment variable to point to the directory containing the sundials include directories, for example:

```
$ CPATH=/local/include python setup.py -c mingw32 install
```

Note that using MSys presents unique problems being a hybrid environment. We recommend using MSys only to compile and install (Py)SUNDIALS, not for use as an environment in which to run PySUNDIALS. Having followed the above instructions, the SUNDIALS shared libs will be in `%MSYSROOT%/usr/local/lib/`, and end with `-<digit>.exe`. On older versions of MSys/MinGW, the `.exe` extension may be left off.

2.3 Binary Installation on Windows

2.4 Configuration

PySUNDIALS uses `ctypes` to link the required SUNDIALS libraries directly into the running python process. In order to do this, it needs to know where to find those shared libraries. On Linux/BSD systems, this is usually auto detected, as library locations are standard, however on windows systems, the final location and even naming convention of the shared library files is compiler dependent. If PySUNDIALS cannot find the SUNDIALS libraries, please locate them yourself, and specify their locations in a file named `~/.pysundials/config` (Linux/BSD), or `%HOMEPATH%\pysundials\config` (Windows). If PySUNDIALS cannot find this configuration file in your home directory, it will seek it in the same directory it was installed in, i.e. `$PYTHONROOT/site-packages/pysundials/config`.

In the 'config' file, anything following a hash (including the hash itself) is considered a comment. Each line specifies the location of a required library in the form:

```
library = path
```

Where *library* is one of `c`, `aux`, `nvecserial`, `nvecparallel`, `cvode`, `cvodes`, `ida`, or `kinsol`, and *path* is the complete path of the appropriate library file. `c`, `aux`, and `nvecparallel` are optional; the first two being generally autodetected, and the second only required if you will be using PySUNDIALS in parallel with MPI.

You can find a sample config file for both posix and mingw32 systems in the doc subdirectory of the PySUNDIALS distribution.

CHAPTER
THREE

Using PySUNDIALS

3.1 Importing PySUNDIALS modules

There are five PySUNDIALS modules available for general use; One for each of the SUNDIALS modules, namely `cvode`, `cvodes`, `ida`, and `kinsol`. The fifth is the `nvecserial` module, which you may wish to use in conjuction with CVODES for it's convenience functions for dealing with senstivity analysis data structures. To import one of these modules use:

```
from pysundials import module_name
```

where *module_name* is the name of the module you wish to use.

3.2 NVectors

The fundamental data type of SUNDIALS, and hence PySUNDIALS is the NVector. PySUNDIALS implements the NVector class in a manner that closely resembles a python list. Creating an NVector is a simple case of class instantiation:

```
>>> from pysundials import nvecserial
>>> v = nvecserial.NVector([1, 2, 3])
>>> v
[1.0, 2.0, 3.0]
```

When instantiating an NVector, simply pass a sequence (tuple, list, numpy array, another NVector, etc...) to the constuctor. The contents of the sequence determine the length of the NVector (which remains immutable for its lifetime) and the initial value of the NVector. NVector objects can be subscripted or sliced like normal python lists.:

```
>>> v[1]
2.0
>>> v[2] = 4
>>> v
[1.0, 2.0, 4.0]
>>> v[0:2]
[1.0, 2.0]
>>> v[0:2] = (-1, -2)
>>> v
[-1.0, -2.0, 4.0]
```

Operators act intuitively on NVectors too,

- + and − perform scalar or vector addition or subtraction respectively, depending on operands.:

  ```
  >>> w = nvecserial.NVector([1,2,-4])
  >>> v+1
  [0.0, -1.0, 5.0]
  >>> v+w
  [0.0, 0.0, 0.0]
  >>> v-w
  [-2.0, -4.0, 8.0]
  ```

- * performs scalar or element-wise multiplication depending on operands.:

  ```
  >>> v*2
  [-2.0, -4.0, 8.0]
  >>> v*w
  [-1.0, -4.0, -16.0]
  ```

- / performs scalar or element-wise division depending on opreands.:

  ```
  >>> v/2
  [-0.5, -1.0, 2.0]
  >>> v/w
  [-1.0, -1.0, -1.0]
  >>> v/v
  [1.0, 1.0, 1.0]
  ```

- and a variety of object methods perform more complex operations including dot product and various norms. See the reference section for a complete list

An NVector object can be used as a numpy array by using its `asarray` method. Note how changes to the array affect the NVector and *vice versa*.:

```
>>> import numpy
>>> a = v.asarray()
>>> a
array([-1., -2.,  4.])
>>> a[0] = 0
>>> a
array([ 0., -2.,  4.])
>>> v
[0.0, -2.0, 4.0]
>>> v[1] = 0
>>> a
array([ 0.,  0.,  4.])
```

The NVector class is exported to each of the main PySUNDIALS modules, so there is rarely a need to import `nvecserial`.:

```
>>> from pysundials import cvode
>>> v = cvode.NVector([1,2,3])
>>> v
[1.0, 2.0, 3.0]
```

3.3 CVODE

Programs using CVODE will generally conform to a certain skeleton layout. The example used here serves to illustrate this skeleton layout, and is neither complete, nor representative of SUNDIALS/PySUNDIALS complete set of capabilities. Please see the function reference or SUNDIALS documentation for more information.

1. Import the cvode and ctypes modules.:

   ```
   from pysundials import cvode
   import ctypes
   ```

2. Define your right-hand side function. This function must take exactly four parameters. The first parameter will be the current value of the indepedent variable (usually time). The second paramter will be an NVector containing the current values of the dependent variables. The third parameter is an NVector whose elements must be filled with the new values of the dependent variables. The fourth parameter is a pointer to any arbitrary user data you may have specified, otherwise None. This function essentially defines your ODE system. For example, a simple problem consisting of three variables and having the following ODES:

   - `v1 = r2 - r1`
   - `v2 = r1 - r2`
   - `v3 = r1 - r3 - r4`

   (where r *i* is a function of the independent variable and the current values of the dependent variables) would have the following RHS function:

   ```
   def f(t, y, ydot, f_data):
       ydot[0] = r2(t,y) - r1(t,y)
       ydot[1] = r1(t,y) - r2(t,y)
       ydot[2] = r1(t,y) - r3(t,y) - r4(t,y)
       return 0
   ```

3. Define any optional functions such as a Jacobian approximation, error weight and/or root finding functions. See function reference for details on parameters and returns.:

   ```
   def rootfind(t, y, gout, g_data):
       gout[0] = y[0] - 0.5
       gout[1] = y[1] - 0.5
       return 0
   ```

4. Initialise an NVector with the initial conditions.:

   ```
   y = cvode.NVector([0.7, 0.3, 0.0])
   ```

5. Create a CVODE object.:

   ```
   cvode_mem = cvode.CVodeCreate(lmm, iter)
   ```

   (where *lmm* is on of `cvode.CV_ADAMS` or `cvode.CV_BDF`, and *iter* is one of `cvode.CV_NEWTON` or `cvode.CV_FUNCTIONAL`)

6. Allocate integrator memory, set the initial value of the independent variable, and set tolerances. Absolute tolerances may be an NVector of the same size as `y` in which case `cvode.CV_SV` should be used, or a scalar value applying to all (`cvode.CV_SS`).:

```
abstol = cvode.NVector([1.0e-8, 1.0e-14, 1.0e-6])
reltol = cvode.realtype(1.0e-4)
cvode.CVodeMalloc(cvode_mem, f, 0.0, y, cvode.CV_SV, reltol, abstol)
```

7. Set any optional inputs using `CVSet*()`.

8. Choose a linear solver and set the problem size, i.e. number of variables. The available linear solvers are `CVDense`, `CVBand`, `CVDiag`, `CVSpgmr`, `CVSpbcg`, and `CVSptfqmr`.:

   ```
   cvode.CVDense(cvode_mem, 3)
   ```

9. Set any optional linear solver inputs using `cvode.CV<solver>Set*`.

10. Optionally initialise root finding passing the CVODE object, the number of roots to find, a vector of size equal to the number of roots, and a pointer to any optional user data you want available in your root finding function. The root finding function should populate a vector of root values, generally using implicit algebraic equations. If any of those values are zero the integrator pauses, returning `cvode.CV_ROOT_RETURN` to indicate that at least one root has been found.:

    ```
    cvode.CVodeRootInit(cvode_mem, 2, rootfind, None)
    ```

11. Advance the solution in time, calling `cvode.CVode` for each desired output time step. Each call to `cvode.CVode` specifies the desired time for the next stop (`tout`) and the current conditions (`y`). On return, `y` will contain the new conditions, and `t` will contain the time at which the integrator stopped. `t`, which must be of type `realtype` and passed into `cvode.CVode` by reference, can be different from `tout` if roots are found, or errors encountered. The last parameter specifies how CVODE should step. See the SUNDIALS documentation for more details.:

    ```
    t = cvode.realtype(0)
    tout = 0.4
    while tout < 0.4*(10**12):
        flag = cvode.CVode(cvode_mem, tout, y, ctypes.byref(t), cvode.CV_NORMAL)
        print (t, y)
        if flag == cvode.CV_ROOT_RETURN:
            rootsfound = cvode.CVodeGetRootInfo(cvode_mem, 2)
            print rootsfound
        elseif flag == cvode.CV_SUCCESS:
            tout *= 10
        else:
            break
    ```

3.4 CVODES

Programs using CVODES will generally conform to a certain skeleton layout very similar to that of CVODE. Our layout here provides an example for simple calculation of sensitivities using forward sensitivity analysis. CVODES is capable of adjoint sensitivity analysis to. See the function reference or the SUNDIALS documentation for information of how to uses these alternative methods.

1. Import the cvodes module, the nvecserial module, and the ctypes module:

   ```
   from pysundials import cvodes
   import nvecserial
   import ctypes
   ```

2. Define a structure to hold your parameters for which you wish to calculate sensitivites as well as any optional user data.:

   ```
   class UserData(ctypes.Structure):
       _fields_ = [
           ('p', cvodes.realtype*4)
       ]
   PUserData = ctypes.POINTER(UserData)
   ```

3. Define your right-hand side function. This function must take exactly four parameters. The first parameter will be the current value of the indepedent variable (usually time). The second paramter will be an NVector containing the current values of the dependent variables. The third parameter is an NVector whose elements must be filled with the new values of the dependent variables. The fourth parameter is a pointer to any arbitrary user data you may have specified, otherwise None. This function essentially defines your ODE system. For example, a simple problem consisting of three variables and having the following ODES:

   - `v1 = r2 - r1`
   - `v2 = r1 - r2`
   - `v3 = r1 - r3 - r4`

   (where r *i* is a function of the independent variable, the current values of the dependent variables and the parameter set) would have the following RHS function.:

   ```
   def f(t, y, ydot, f_data):
       data = ctypes.cast(f_data, PUserData).contents
       ydot[0] = r2(t,y,data.p) - r1(t,y,data.p)
       ydot[1] = r1(t,y,data.p) - r2(t,y,data.p)
       ydot[2] = r1(t,y,data.p) - r3(t,y,data.p) - r4(t,y,data.p)
       return 0
   ```

4. Define any optional functions such as a Jacobian approximation, error weight and/or root finding functions. See function reference for details on parameters and returns.:

   ```
   def rootfind(t, y, gout, g_data):
       gout[0] = y[0] - 0.5
       gout[1] = y[1] - 0.5
       return 0
   ```

5. Initialise an NVector with the initial conditions.:

   ```
   y = cvodes.NVector([0.7, 0.3, 0.0])
   ```

6. Create a CVODE object.:

   ```
   cvode_mem = cvodes.CVodeCreate(lmm, iter)
   ```

   (where *lmm* is on of `cvodes.CV_ADAMS` or `cvodes.CV_BDF`, and *iter* is one of `cvodes.CV_NEWTON` or `cvodes.CV_FUNCTIONAL`)

7. Allocate integrator memory, set the initial value of the independent variable, and set tolerances. Absolute tolerances may be an NVector of the same size as `y` in which case `cvodes.CV_SV` should be used, or a scalar value applying to all (`cvodes.CV_SS`).:

   ```
   abstol = cvodes.NVector([1.0e-8, 1.0e-14, 1.0e-6])
   reltol = cvodes.realtype(1.0e-4)
   cvodes.CVodeMalloc(cvode_mem, f, 0.0, y, cvodes.CV_SV, reltol, abstol)
   ```

8. Set any optional inputs using `CVSet*()`.:

   ```
   cvodes.CVodeSetFdata(cvode_mem, ctypes.pointer(data))
   ```

9. Choose a linear solver and set the problem size, i.e. number of variables. The available linear solvers are `CVDense`, `CVBand`, `CVDiag`, `CVSpgmr`, `CVSpbcg`, and `CVSptfqmr`.:

   ```
   cvodes.CVDense(cvode_mem, 3)
   ```

10. Set sensitivity system options by first creating an NVectorArray of dimensions *v* by *p*, where *v* is the number of variables, and *p* is the number of paramters for which sensitivities will be calculated.:

    ```
    yS = nvecserial.NVectorArray([([0]*2)]*4)
    ```

    Next call `cvodes.CVodeSensMalloc` to allocate and initialise required memory for sensitivity analysis, passing the CVODE object, the number of parameters, the desired method (`cvodes.CV_SIMULTANEOUS`, `cvodes.CV_STAGGERED`, or `cvodes.CV_STAGGERED1`), and the NVectorArray.:

    ```
    cvodes.CVodeSensMalloc(cvodes_mem, 4, cvodes.CV_SIMULTANEOUS, yS)
    ```

    Next we have to inform CVODES which parameters are going to be used for sensitivity calculations by calling `cvodes.CVodeSetSensParams`, which expects four parameters (for more detail see p. 111 of the CVODES user guide).

    (a) the cvodes memory object

    (b) a pointer to the array of parameter values which MUST be passed through the user data structure (so CVODES knows where the values are and can peturb them, presumably)

    (c) an array (i.e. list) of scaling factors, one for each parameter for which sensitivies are to be determined

    (d) an array of integers (either 1 or 0), where a 1 indicates the respective paramter value should be used in estimating sensistivities

    for example:

    ```
    cvodes.CVodeSetSensParams(cvodes_mem,
        data.p, #we have four system parameters (The four VMax's)
        [1]*4, #they should all be scaled by 1, i.e. unscaled,
        [1]*4 #they all contribute to the estimation of sensitivities
    )
    ```

11. Set any optional linear solver inputs using `cvodes.CV<solver>Set*`.

12. Optionally initialise root finding passing the CVODE object, the number of roots to find, a vector of size equal to the number of roots, and a pointer to any optional user data you want available in your root finding function. The root finding function should populate a vector of root values, generally using implicit algebraic equations. If any of those values are zero the integrator pauses, returning `cvodes.CV_ROOT_RETURN` to indicate that at least one root has been found.:

    ```
    cvodes.CVodeRootInit(cvode_mem, 2, g, None)
    ```

13. Advance the solution in time, calling `cvodes.CVode` for each desired output time step. Each call to `cvodes.CVode` specifies the desired time for the next stop (`tout`) and the current conditions (`y`). On return, `y` will contain the new conditions, and `t` will contain the time at which the integrator stopped. `t`, which must be of type `realtype` and passed into `cvodes.CVode` by reference, can be different from `tout` if roots are found, or errors encountered. The last parameter specifies how CVODE should step. See the SUNDIALS documentation for more details.:

```
t = cvodes.realtype(0)
tout = 0.4
while tout < 0.4*(10**12):
    flag = cvodes.CVode(cvode_mem, tout, y, ctypes.byref(t), cvodes.CV_NORMAL)
    cvodes.CVodeGetSens(cvodes_mem, t, yS)
    print (t, y, yS)
    if flag == cvodes.CV_ROOT_RETURN:
        rootsfound = cvodes.CVodeGetRootInfo(cvode_mem, 2)
        print rootsfound
    elseif flag == cvodes.CV_SUCCESS:
        tout *= 10
    else:
        break
```

3.5 IDA

Programs using IDA will generally conform to a certain skeleton layout. The example used here serves to illustrate this skeleton layout, and is neither complete, nor representative of SUNDIALS/PySUNDIALS complete set of capabilities. Please see the function reference or SUNDIALS documentation for more information.

1. Import the ida and ctypes modules.:

   ```
   from pysundials import ida
   import ctypes
   ```

2. Define your right-hand side function. This function must take exactly five parameters. The first parameter will be the current value of the indepedent variable (usually time). The second paramter will be an NVector containing the current values of the dependent variables. The third parameter will be an NVector containing *dy/dt*. The fourth parameter is an NVector whose elements must be filled with the new values of the dependent variables. The fifth parameter is a pointer to any arbitrary user data you may have specified, otherwise None. This function essentially defines your ODE system, and must do so in implicit form for both differential and algebraic equations. Additonally, those variables determined by algrebraic realtions should appear strictly after those determined by differential equations in the dependent variable vector. For example, a simple problem consisting of three variables and having the following ODES (not the rearrangment to order differential and algebraic equations correctly compared to previous examples):

   - `v1 = r1 - r3 - r4`
   - `v2 = r2 - r1`
   - `v3 = r1 - r2`

   (where r *i* is a function of the independent variable and the current values of the dependent variables) would have the following RHS function:

   ```
   def f(t, yy, yp, rr, data):
       rr[0] = r1(yy)-r3(yy)-r4(yy)-yp[0]
       rr[1] = r2(yy)-r1(yy)-yp[1]
       rr[2] = yy[1]+yy[2]-1
       return 0
   ```

3. Define any optional functions such as a Jacobian approximation, error weight and/or root finding functions. See function reference for details on parameters and returns.:

```
def rootfind(t, y, gout, g_data):
    gout[0] = y[0] - 0.5
    gout[1] = y[1] - 0.5
    return 0
```

4. Initialise an NVector with the initial conditions.:

   ```
   yy = ida.NVector([0.7, 0.3, 0.0])
   ```

5. Initialise another NVector with the initial derivative conditions.:

   ```
   yp = ida.NVector([r1(yy)-r3(yy)-r4(yy), r2(yy)-r1(yy), 1-yy[1]])
   ```

6. Create an IDA object.:

   ```
   ida_mem = ida.IDACreate()
   ```

7. Allocate integrator memory, set the initial value of the independent variable, and set tolerances. Absolute tolerances may be an NVector of the same size as `y` in which case `ida.IDA_SV` should be used, or a scalar value applying to all (`ida.IDA_SS`).:

   ```
   abstol = ida.NVector([1.0e-8, 1.0e-14, 1.0e-6])
   reltol = ida.realtype(1.0e-4)
   ida.IDAMalloc(ida_mem, f, 0.0, yy, yp, ida.IDA_SV, reltol, abstol)
   ```

8. Set any optional inputs using `IDASet*()`.

9. Choose a linear solver and set the problem size, i.e. number of variables. The available linear solvers are `IDADense`, `IDABand`, `IDASpgmr`, `IDASpbcg`, and `IDASptfqmr`.:

   ```
   ida.IDADense(ida_mem, 3)
   ```

10. Set any optional linear solver inputs using `ida.IDA<solver>Set*`.

11. Optionally initialise root finding passing the IDA object, the number of roots to find, a vector of size equal to the number of roots, and a pointer to any optional user data you want available in your root finding function. The root finding function should populate a vector of root values, generally using implicit algebraic equations. If any of those values are zero the integrator pauses, returning `ida.IDA_ROOT_RETURN` to indicate that at least one root has been found.:

    ```
    ida.IDARootInit(ida_mem, 2, rootfind, None)
    ```

12. Advance the solution in time, calling `ida.IDASolve` for each desired output time step. Each call to `ida.IDASolve` specifies the desired time for the next stop (`tout`) and the current conditions (`y`). On return, `y` will contain the new conditions, and `t` will contain the time at which the integrator stopped. `t`, which must be of type `realtype` and passed into `ida.IDASolve` by reference, can be different from `tout` if roots are found, or errors encountered. The last parameter specifies how IDA should step. See the SUNDIALS documentation for more details.:

    ```
    t = ida.realtype(0)
    tout = 0.4
    while tout < 0.4*(10**12):
        flag = ida.IDASolve(ida_mem, tout, ctypes.byref(t), yy, yp, ida.IDA_NORMAL)
        print (t, yy)
        if flag == ida.IDA_ROOT_RETURN:
    ```

```
        rootsfound = ida.IDAGetRootInfo(ida_mem, 2)
        print rootsfound
    elseif flag == ida.IDA_SUCCESS:
        tout *= 10
    else:
        break
```

3.6 KINSOL

Programs using KINSOL will generally conform to a certain skeleton layout. The example used here serves to illustrate this skeleton layout, and is neither complete, nor representative of SUNDIALS/PySUNDIALS complete set of capabilities. Please see the function reference or SUNDIALS documentation for more information.

1. Import the kinsol and ctypes modules.:

   ```
   from pysundials import kinsol
   import ctypes
   ```

2. Define your right-hand side function. This function must take exactly three parameters. The first paramter will be an NVector containing the current values of the dependent variables. The second parameter is an NVector whose elements must be filled with the new values of the dependent variables. The third parameter is a pointer to any arbitrary user data you may have specified, otherwise None. This function essentially defines your ODE system and must do so using strictly linearly independent equations. For example, a simple problem consisting of three variables and having the following ODES:

   - `v1 = r1 - r3 - r4`
   - `v2 = r2 - r1`
   - `v3 = r1 - r2`

   (where r *i* is a function of the independent variable and the current values of the dependent variables) would have the following RHS function, ignoring *v3* beacuse it is linearly dependent with *v2*:

   ```
   def f(u, fval, f_data):
       fval[S2] = R2(u) - R1(u)
       fval[S1] = R1(u) - R3(u) - R4(u)
       return 0
   ```

3. Define any optional functions such as a Jacobian approximation, and/or error weight functions. See function reference for details on parameters and returns.

4. Initialise an NVector with an initial guess.:

   ```
   u = kinsol.NVector([1.0, 0.7])
   ```

5. Initialise a template NVector of the same size as your dependent variable vector.:

   ```
   template = kinsol.NVector([0, 0])
   ```

6. Initialise a scaling vector or vectors as necessary. See function reference for `kinsol.KINSol` for more details:

   ```
   s = kinsol.NVector([1, 1])
   ```

7. Create a KINSOL object.:

   ```
   kin_mem = kinsol.KINCreate()
   ```

8. Allocate solver memory, and set the RHS function and size of the system using the template vector.:

```
kinsol.KINMalloc(kin_mem, f, template)
```

9. Set any optional inputs using `KINSet*()`.
10. Choose a linear solver and set the problem size, i.e. number of variables. The available linear solvers are `KINDense`, `KINBand`, `KINSpgmr`, `KINSpbcg`, and `KINSptfqmr`.:

    ```
    kinsol.KINDense(kin_mem, 2)
    ```

11. Set any optional linear solver inputs using `kinsol.KIN<solver>Set*`.

#. Solve the problem by calling `kinsol.KINSol`, passing the KINSOL memory object, the vector with the initial guess, the globalisation strategy (one of `kinsol.KIN_NONE` or `kinsol.KIN_LINESEARCH`), and two scaling vectors, *u_scale* and *f_scale*. In our case no scaling is applied via the repeated use of the scaling vector `s` (*[1,1]*):

```
kinsol.KINSol(kin_mem, u, kinsol.KIN_LINESEARCH, s, s)
```

CHAPTER

FOUR

Indices and tables

- *Index*
- *Module Index*
- *Search Page*