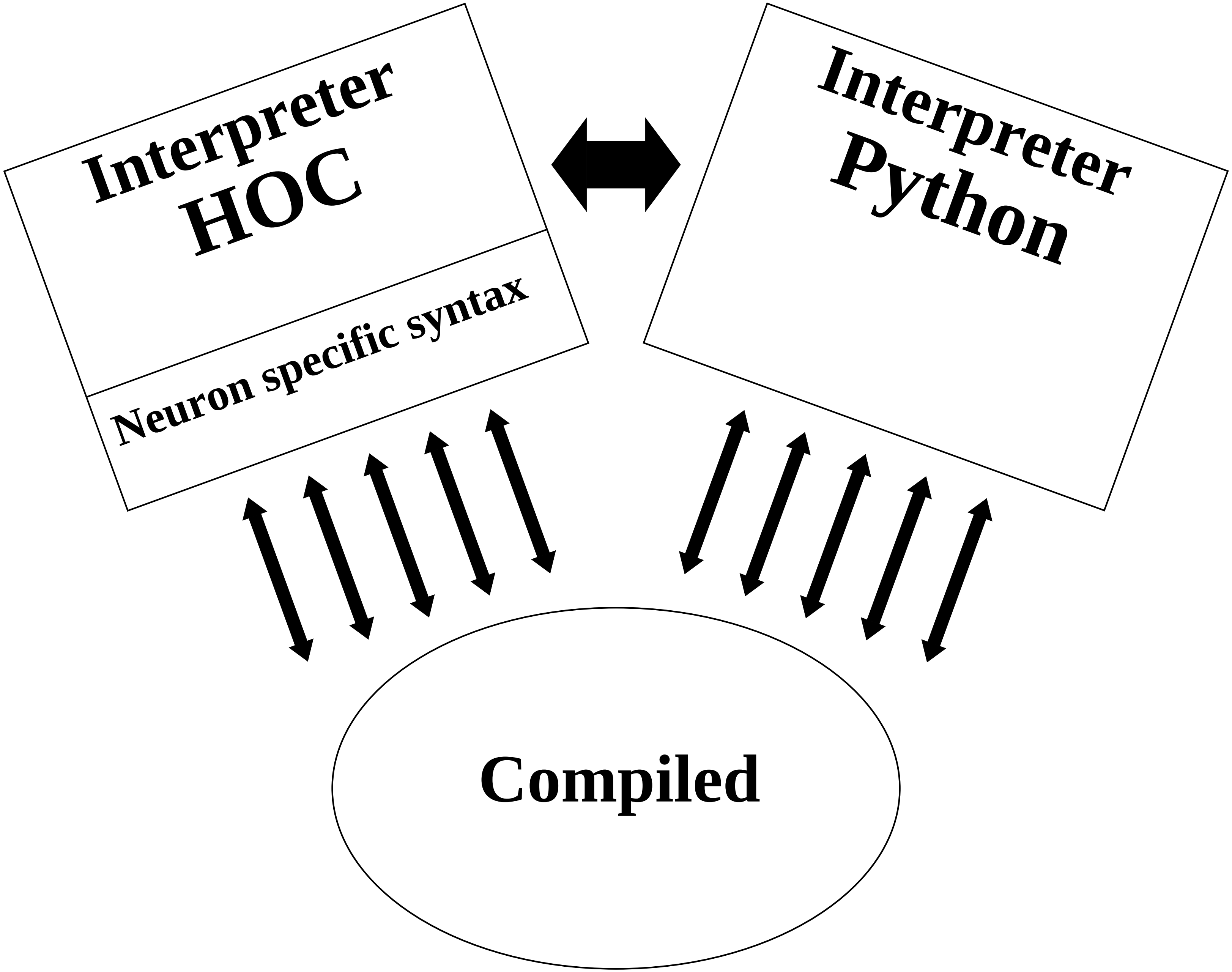
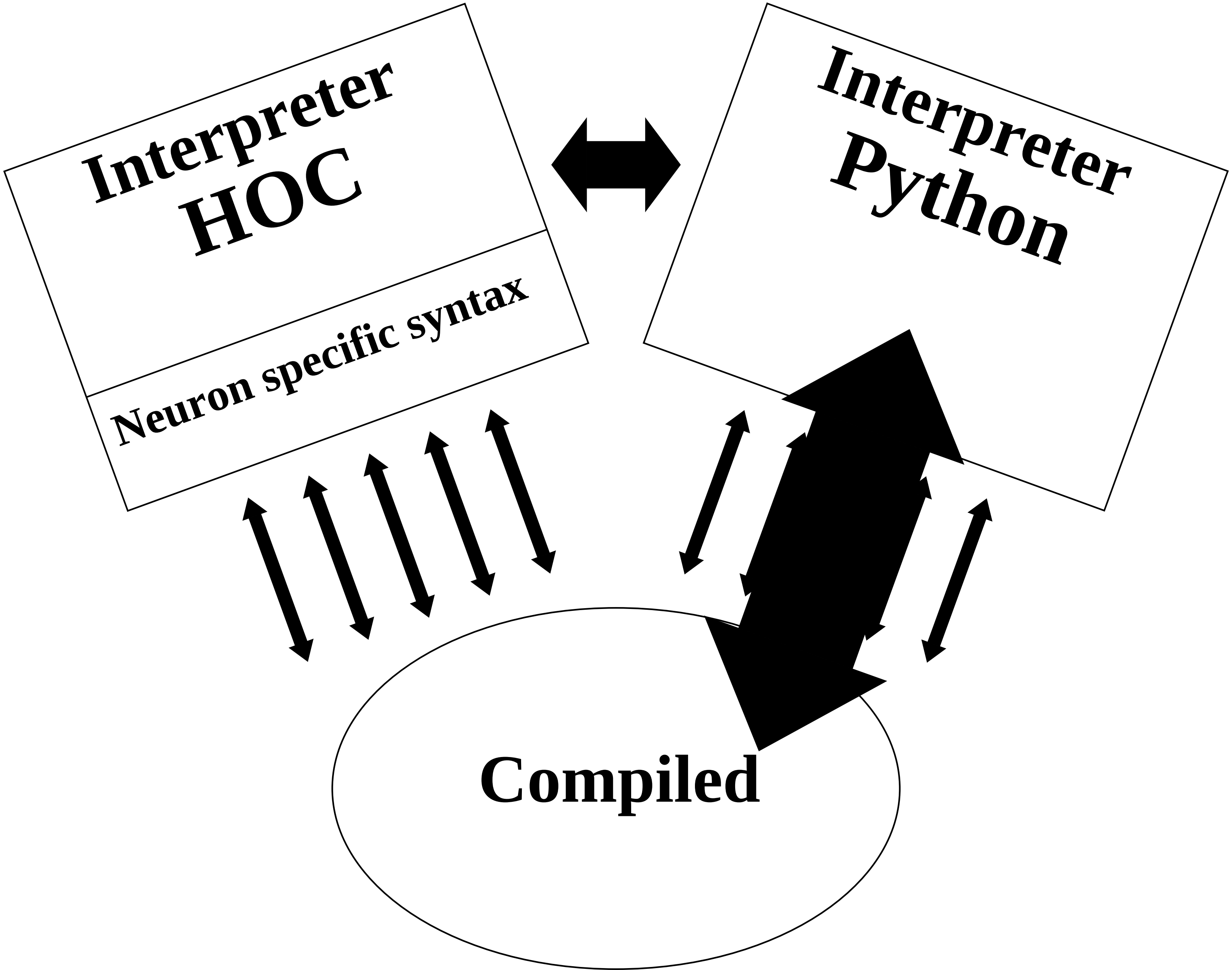
NEURON + Python

Michael Hines

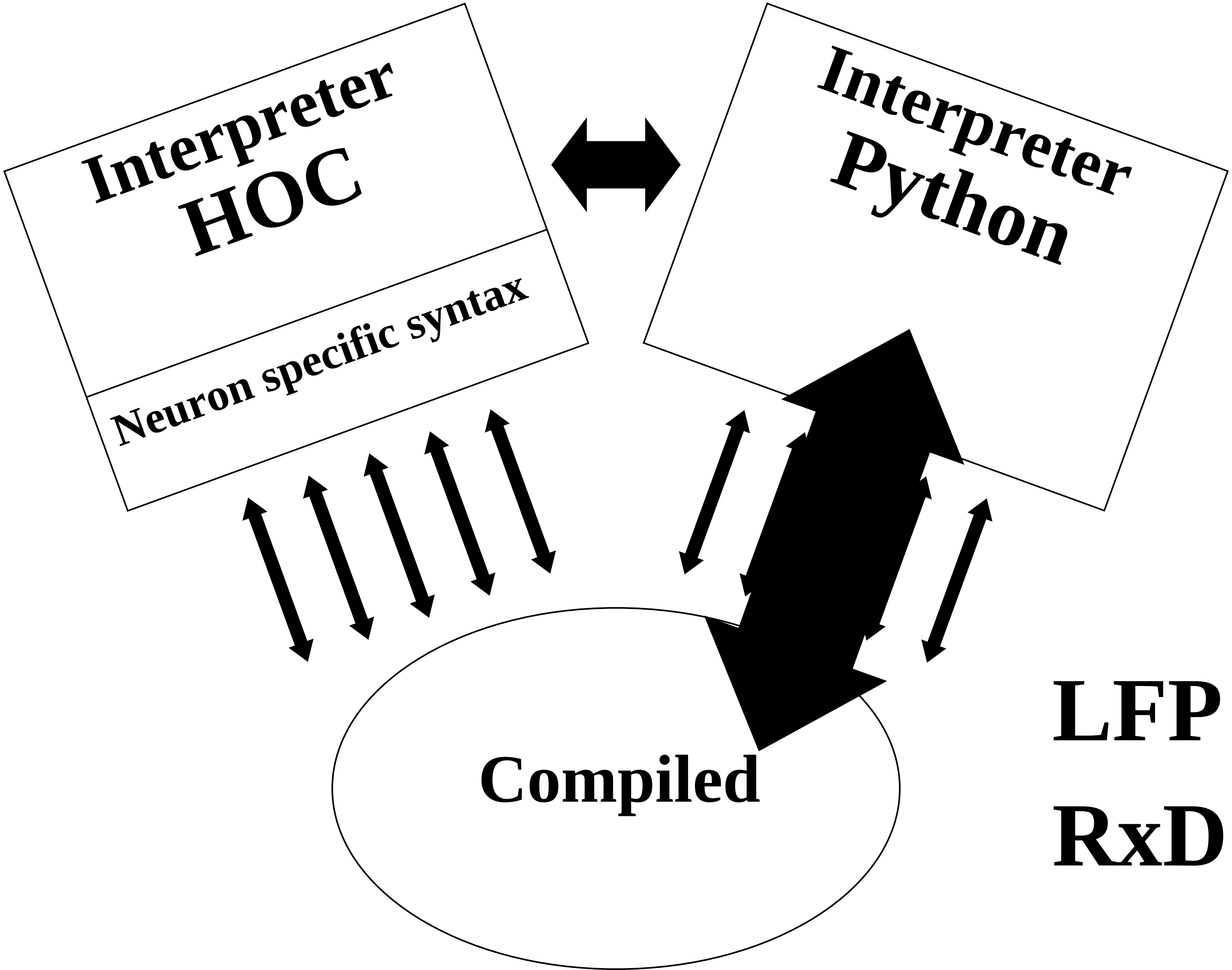
HBP CodeJam Workshop #7
Manchester 2016

NINDS

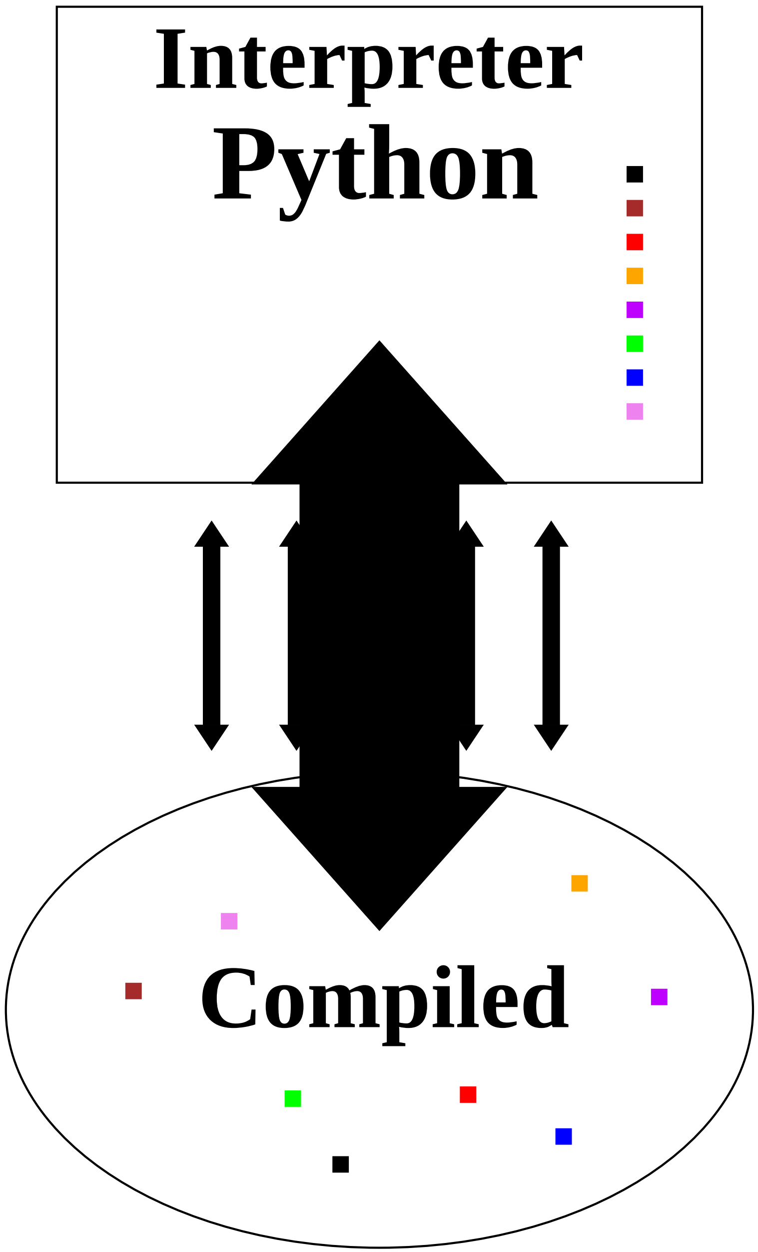

Interpreter
HOC
Neuron specific syntax
Interpreter
Python
Compiled



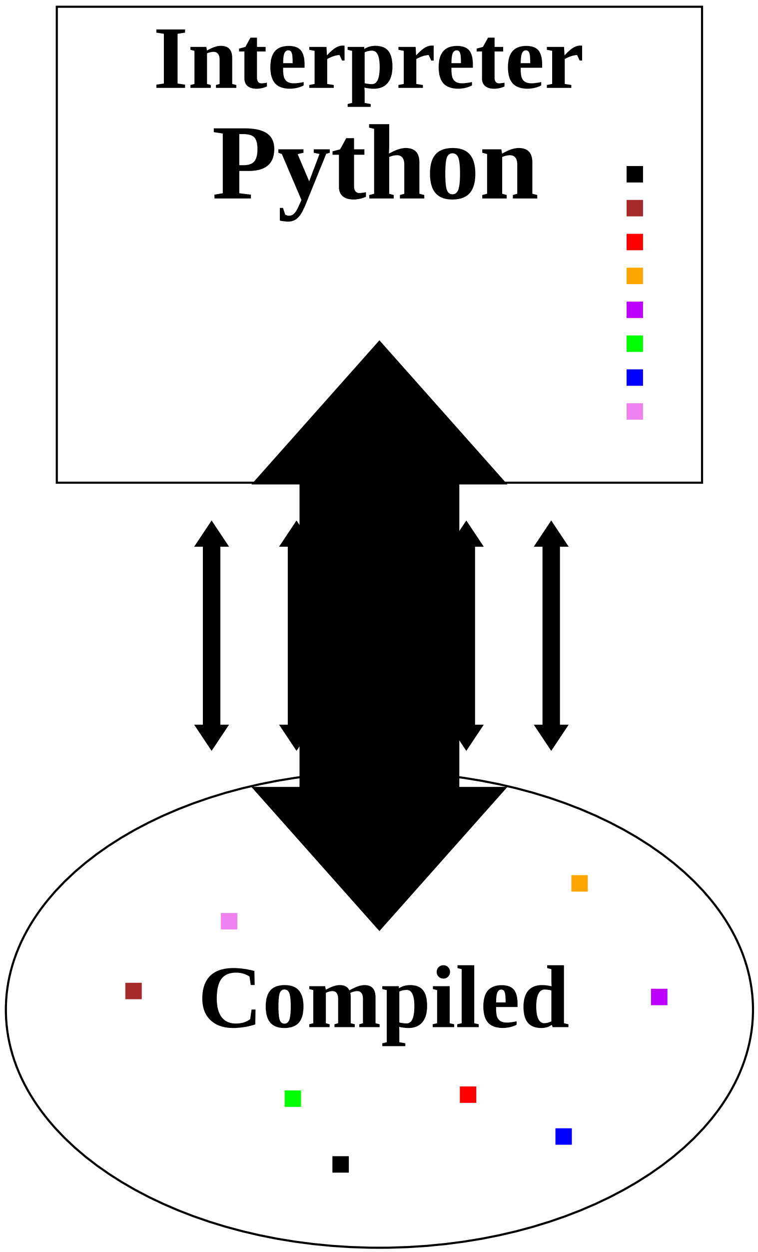
Run: compute model



Run: compute model

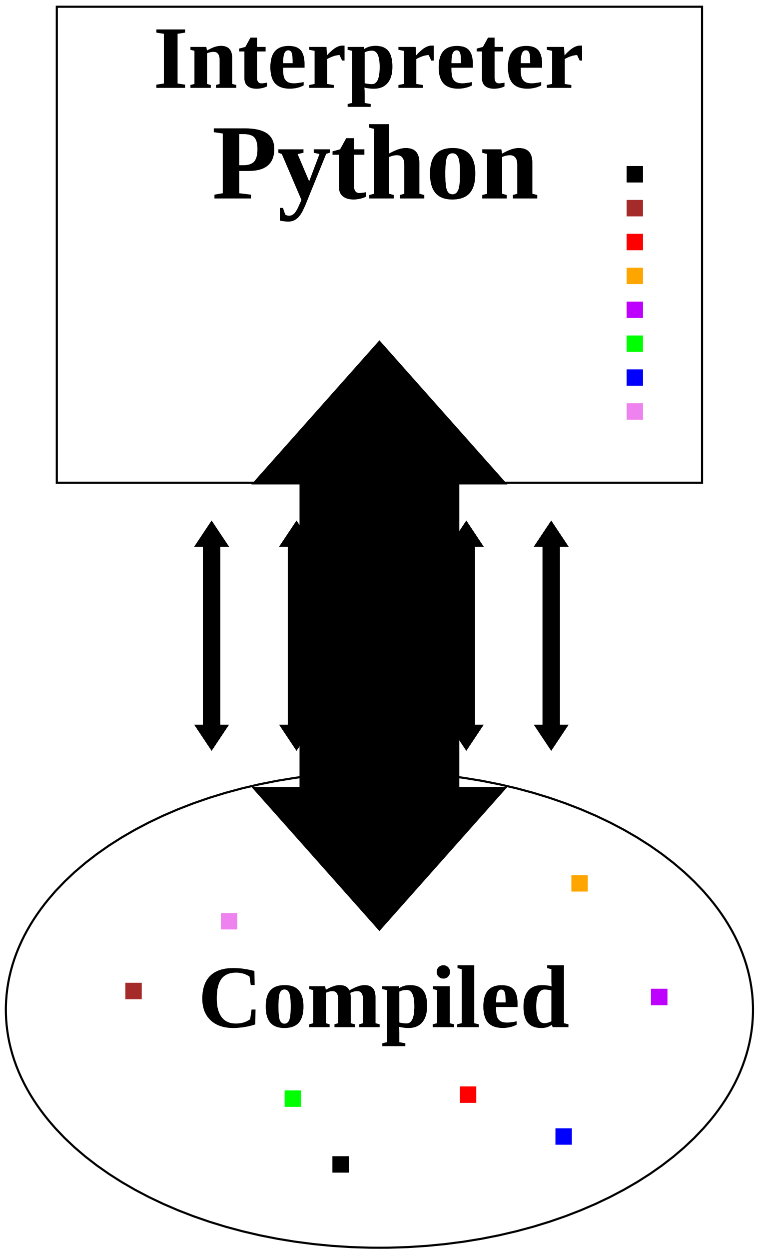


Gather values into numpy array.



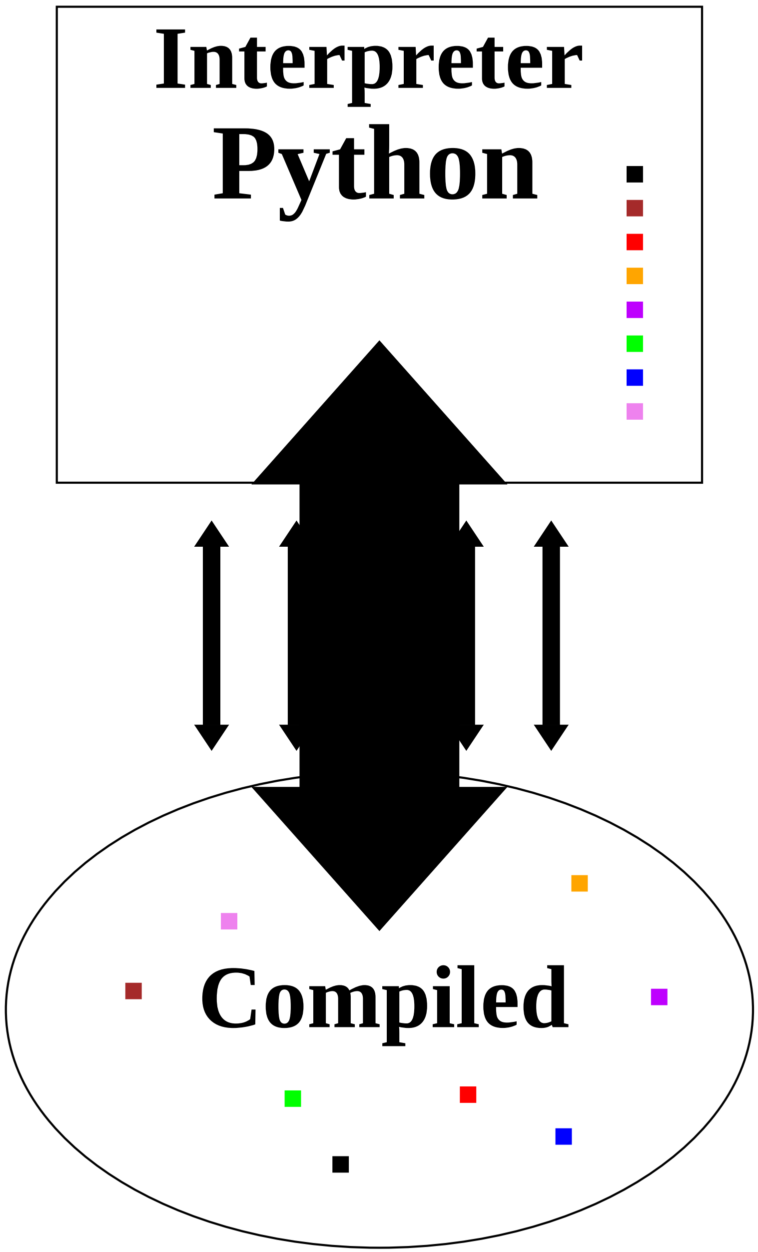
Gather values into numpy array.

```
from neuron import h
import numpy
```



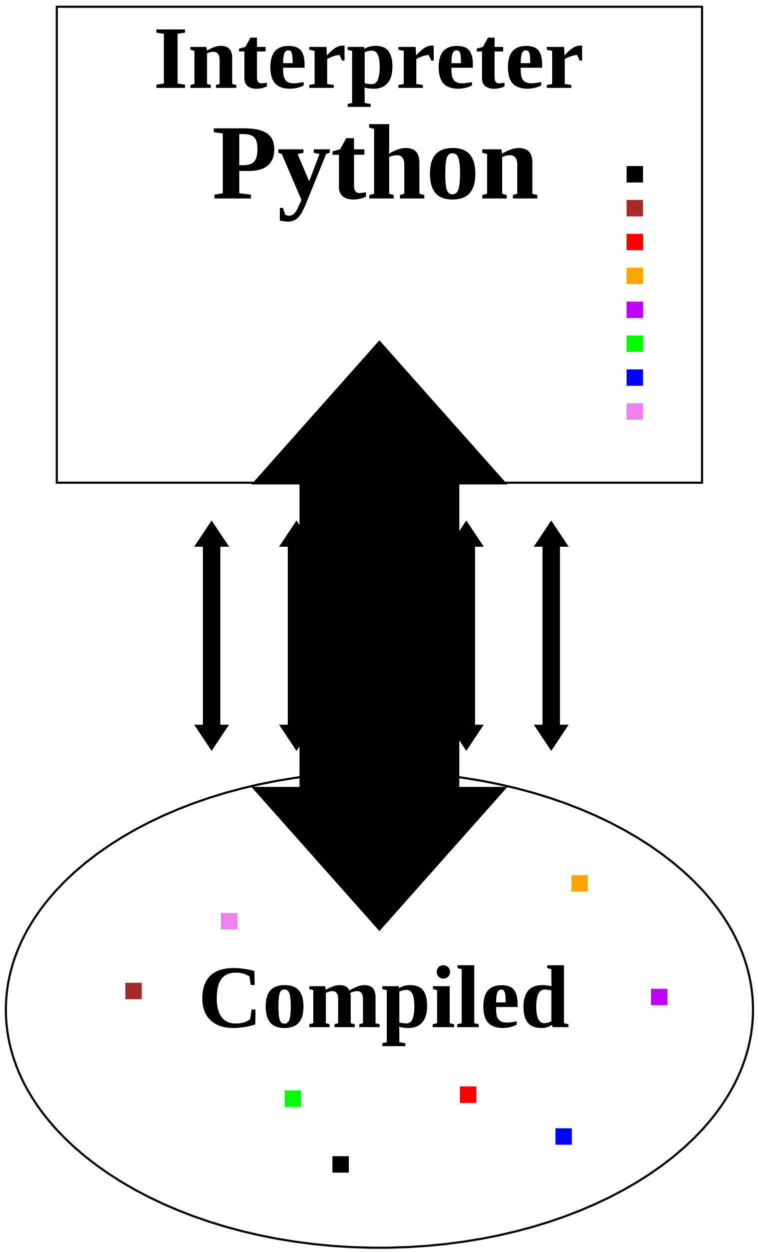
Gather values into numpy array.

```
from neuron import h
import numpy

hv = h.Vector(size)
v = hv.as_numpy()
```



Gather values into numpy array.

```
from neuron import h
import numpy

hv = h.Vector(size)
v = hv.as_numpy()

pv = h.PtrVector(size)
pv.pset(i, _ref_hocvar)
pv.gather(hv)
```



Gather values into numpy array.

```
from neuron import h
import numpy

hv = h.Vector(size)
v = hv.as_numpy()

pv = h.PtrVector(size)
pv.pset(i, _ref_hocvar)
pv.gather(hv)

i=0
for sec in h.allsec():
  for seg in sec:
    pv.pset(i, seg._ref_v)
    i += 1
```

Python API for NEURON solvers

Python API for NEURON solvers

nonvint_block_supervisor

$dy/dt = f(y, v, t)$ y can affect channel conductance

Python API for NEURON solvers

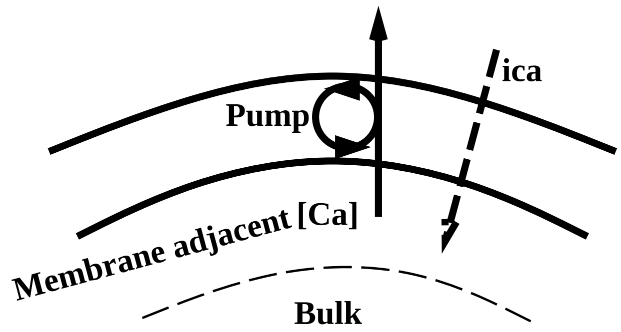
nonvint_block_supervisor

$dy/dt = f(y, v, t)$ y can affect channel conductance

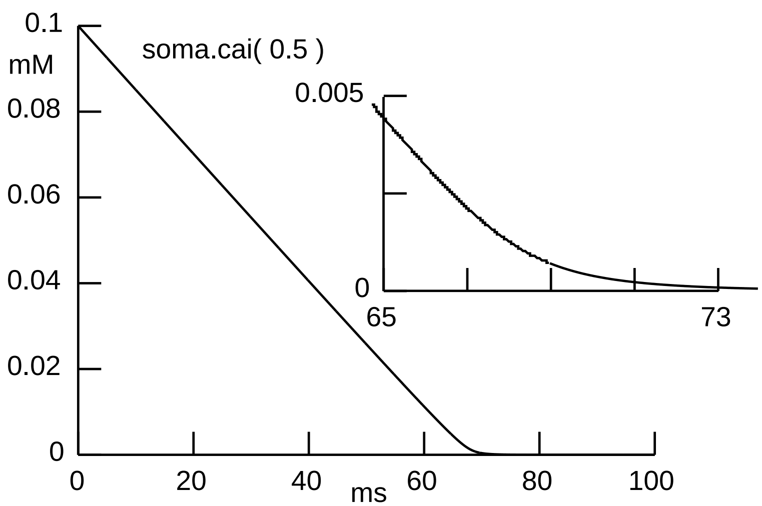
High performance evaluation of $M*x = b$

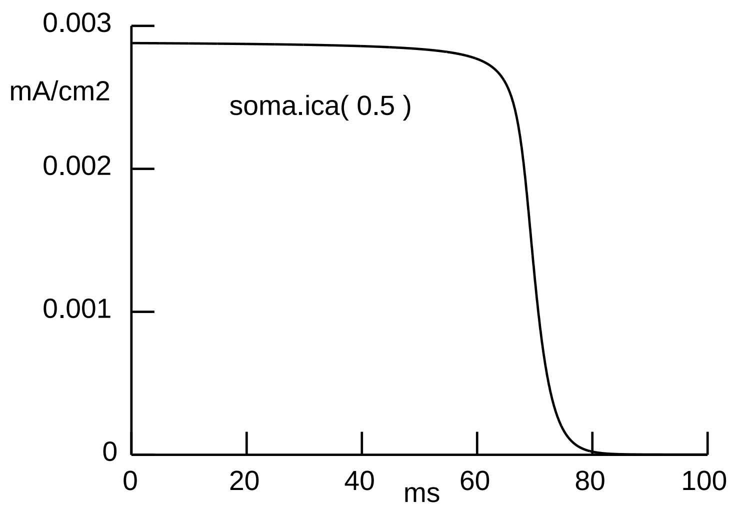
$M \sim 1 - dt*\partial f/\partial y$

NMODL

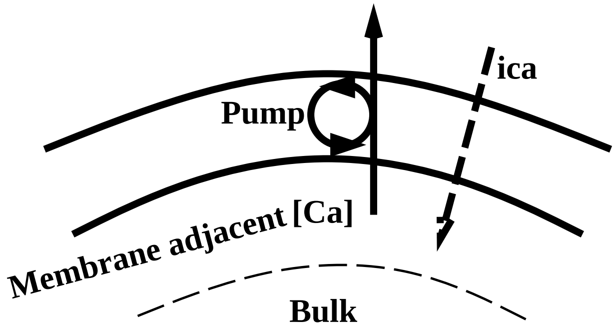



```
KINETIC pmp {
  ~ cabulk <-> cai          (1/tau, 1/tau)
  ~ cai + pump <-> capump  (k1, k2)
  ~ capump <-> cao + pump  (k3, k4)
  ica_pmp = 2*FARADAY*(f_flux - b_flux)

  ~ cai << -(ica) : there is a problem here

  COMPARTMENT width {cai}     : volume has dimensions of (um)
  COMPARTMENT 1 {pump capump} : area is dimensionless
  COMPARTMENT 1(m) {cao cabulk}
}
```

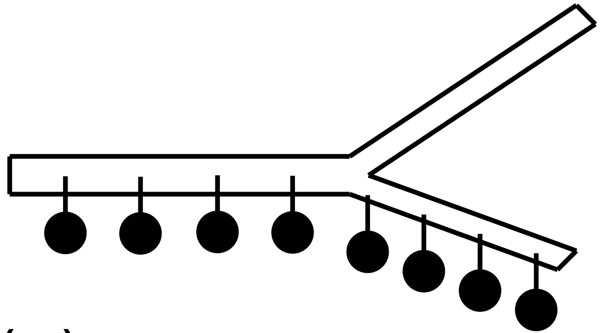


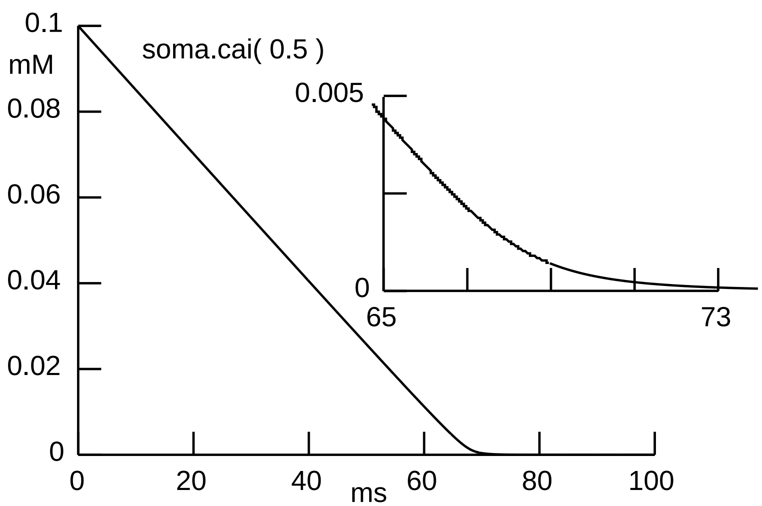


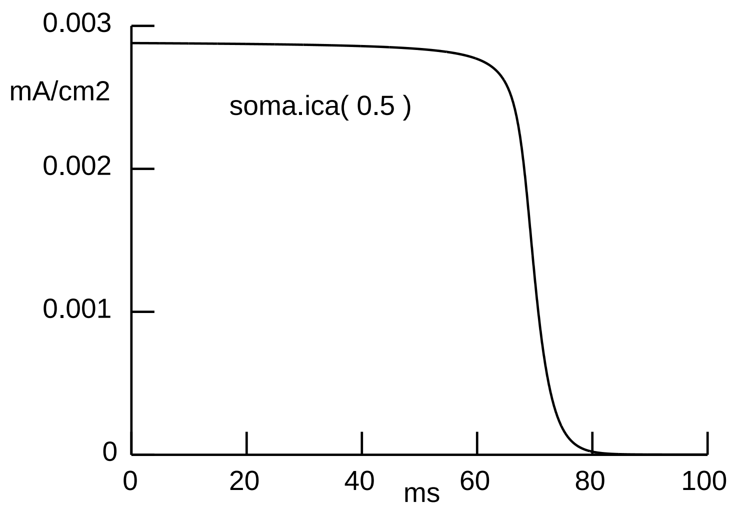
NMODL




```
KINETIC pmp {
  ~ cabulk <-> cai         (1/tau, 1/tau)
  ~ cai + pump <-> capump  (k1, k2)
  ~ capump <-> cao + pump  (k3, k4)
  ica_pmp = 2*FARADAY*(f_flux - b_flux)

  ~ cai << -(ica) : there is a problem here

  COMPARTMENT width {cai}     : volume has dimensions of (um)
  COMPARTMENT 1 {pump capump} : area is dimensionless
  COMPARTMENT 1(m) {cao cabulk}
}
```







```
from neuron import nonvint_block_supervisor as nbs

callables = [
  setup, initialize,
  current, conductance,
  fixed_step_solve,
  count, reinit, fun, msolve,
  jacobian, abs_tolerance
]

nbs.register(callables)
```

```
from neuron import nonvint_block_supervisor as nbs

callables = [
  setup, initialize,
  current, conductance,
  fixed_step_solve,
  count, reinit, fun, msolve,
  jacobian, abs_tolerance
]

nbs.register(callables)
```

```
class ExtraEquations():

  def __init__(self):
    self.callables = [...]
    nbs.register(self.callables)
    ...

  def __del__(self):
    nbs.unregister(self.callables)

  ...
```

```
from neuron import nonvint_block_supervisor as nbs

callables = [
  setup, initialize,
  current, conductance,
  fixed_step_solve,
  count, reinit, fun, msolve,
  jacobian, abs_tolerance
]

nbs.register(callables)
```

Contribution to current balance eqns

```
from neuron import nonvint_block_supervisor as nbs

callables = [
  setup, initialize,
  current, conductance,
  fixed_step_solve,
  count, reinit, fun, msolve,
  jacobian, abs_tolerance
]

nbs.register(callables)
```

$$\mathbf{y(t) \rightarrow y(t + dt)}$$

```
from neuron import nonvint_block_supervisor as nbs

callables = [
  setup, initialize,
  current, conductance,
  fixed_step_solve,
  count, reinit, fun, msolve,
  jacobian, abs_tolerance
]

nbs.register(callables)
```

Additional equations for CVODE

Contribution to current balance eqns.

Subtract current from rhs numpy array.

```
def current(self, rhs):
  self.pv.gather(self.hv)
  n = self.n
  self.g = self.gkbar*n*n*n*n
  i = self.g*(self.v + 77.)
  rhs[self.nodeindices] −= i

def conductance(self, d):
  d[self.nodeindices] += self.g
```

Add conductance to matrix diagonal numpy array.

Contribution to current balance eqns.

```
def current(self, rhs):
  self.pv.gather(self.hv)
  n = self.n
  self.g = self.gkbar*n*n*n*n
  i = self.g*(self.v + 77.)
  rhs[self.nodeindices] −= i

def conductance(self, d):
  d[self.nodeindices] += self.g
```

Copy seg.node_index() during setup.

Contribution to current balance eqns.

```
def current(self, rhs):
  self.pv.gather(self.hv)
  n = self.n
  self.g = self.gkbar*n*n*n*n
  i = self.g*(self.v + 77.)
  rhs[self.nodeindices] −= i

def conductance(self, d):
  d[self.nodeindices] += self.g
```

Voltage needed ...

... to compute current

Contribution to current balance eqns.

```
def current(self, rhs):
  self.pv.gather(self.hv)
  n = self.n
  self.g = self.gkbar*n*n*n*n
  i = self.g*(self.v + 77.)
  rhs[self.nodeindices] −= i

def conductance(self, d):
  d[self.nodeindices] += self.g
```

Instance gating states...
... needed for current as well

CVODE interface

Where our portion of the CVODE state vector begins.

```
def count(self, offset):
  self.offset = offset
  return len(self.v)

def fun(self, t, y, ydot):
  last = self.offset + len(self.n)
  self.pv.gather(self.hv)
  self.n = y[self.offset:last]
  if ydot == None:
    return
  ninf, nrate = self.ninftau(self.v)
  ydot[self.offset:last] = (ninf − self.n)*nrate

def msolve(self, dt, t, b, y):
  last = self.offset + len(self.n)
  self.pv.gather(self.hv)
  x,nrate = self.ninftau(self.v)
  b[self.offset:last] /= 1. + dt*nrate
```

CVODE interface

```
def count(self, offset):
  self.offset = offset
  return len(self.v)

def fun(self, t, y, ydot):
  last = self.offset + len(self.n)
  self.pv.gather(self.hv)
  self.n = y[self.offset:last]
  if ydot == None:
    return
  ninf, nrate = self.ninftau(self.v)
  ydot[self.offset:last] = (ninf − self.n)*nrate

def msolve(self, dt, t, b, y):
  last = self.offset + len(self.n)
  self.pv.gather(self.hv)
  x,nrate = self.ninftau(self.v)
  b[self.offset:last] /= 1. + dt*nrate
```

y’, y are numpy arrays

Our portion of y

Our portion of y’ = f(y, t)

CVODE interface

```
def count(self, offset):
  self.offset = offset
  return len(self.v)

def fun(self, t, y, ydot):
  last = self.offset + len(self.n)
  self.pv.gather(self.hv)
  self.n = y[self.offset:last]
  if ydot == None:
    return
  ninf, nrate = self.ninftau(self.v)
  ydot[self.offset:last] = (ninf − self.n)*nrate

def msolve(self, dt, t, b, y):
  last = self.offset + len(self.n)
  self.pv.gather(self.hv)
  x,nrate = self.ninftau(self.v)
  b[self.offset:last] /= 1. + dt*nrate
```

Solve M*x = b   x−>b
y is rarely used

M is implicitly 1 + dt/ntau(v)