Rolling Your Own Circuit Simulator

with Eigen and Boost.ODEInt

Jeff Trull
edaskel@att.net

September 9, 2014

Rolling Your Own Circuit Simulator

Motivation

Open-source libraries for C++ have grown increasingly sophisticated in areas of interest to scientists and engineers. Techniques formerly found only in ancient academic code or proprietary commercial solutions are becoming available as building blocks for independent coders.

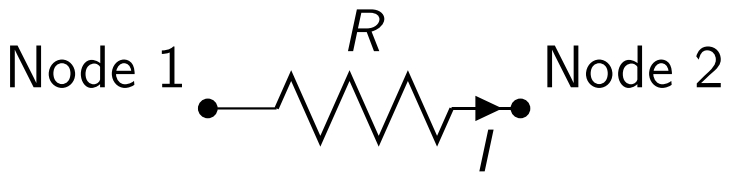
In this talk I hope to demonstrate how you can leverage two of these libraries to achieve some powerful results, by attacking a problem from my own field of electrical engineering.

Circuit Analysis
For Coders

You can think of a circuit as a kind of graph.

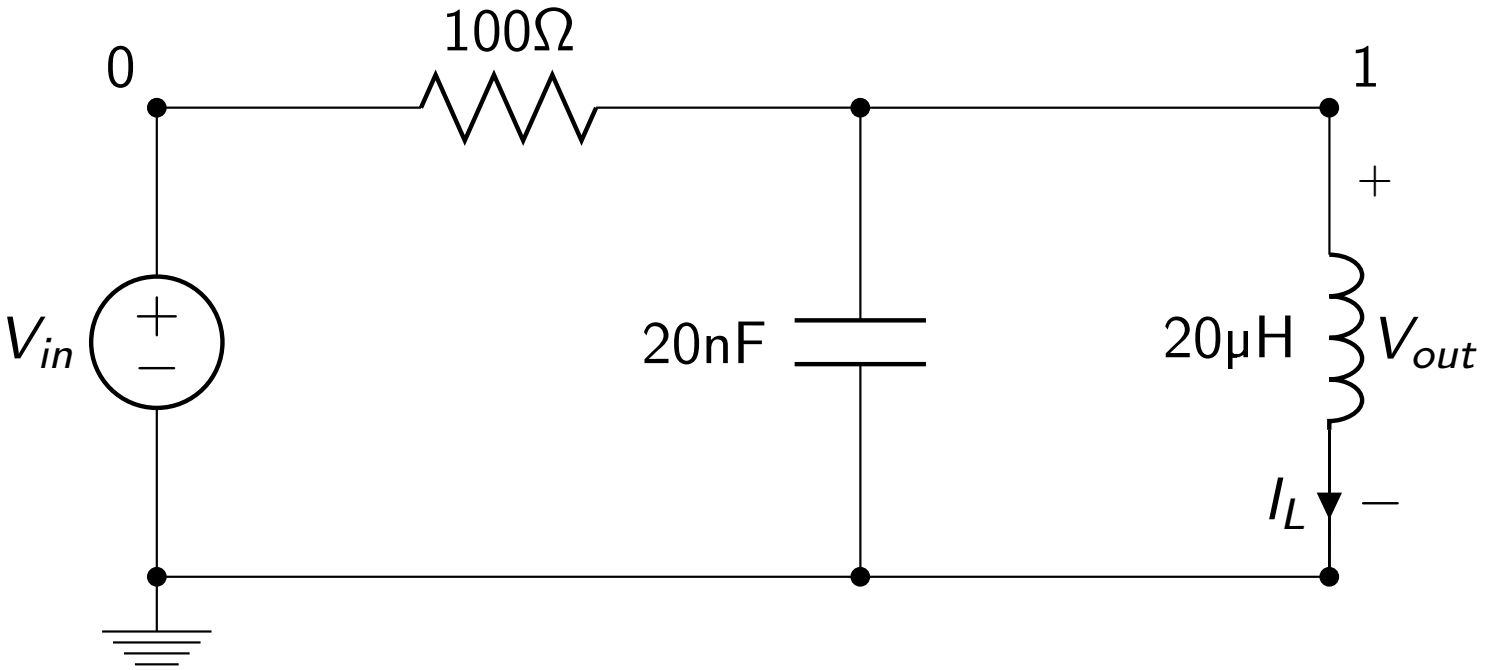
Edges are called *branches* and consist of a single component with a current that depends on its type, value, and the voltages of the nodes it connects

Nodes are simply connection points where currents flow between components



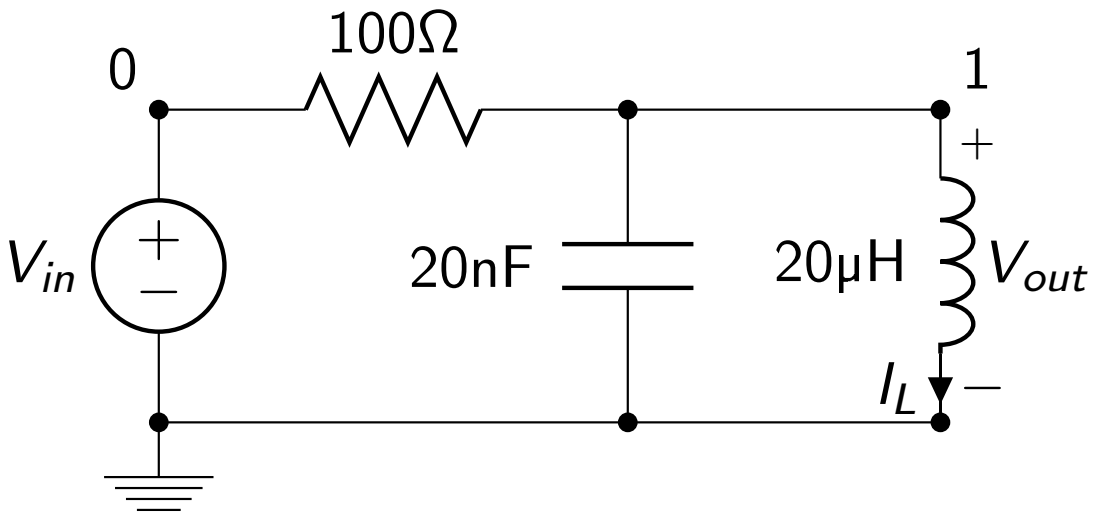
Here $I = (V_{Node1} - V_{Node2})/R$.

Example Circuit



Example Circuit

Equations



$$\begin{aligned} \frac{V_{out}-V_{in}}{R} + C * \frac{dV_{out}}{dt} + I_L &= 0, && \text{currents at node 1} \\ V_{out} &= L * \frac{dI_L}{dt} && \text{from inductor} \end{aligned}$$

For which the solution (given $V_{in}(t)$ is a step function) is:

$V_{out} = \frac{1}{\mu RC} e^{\lambda t} sin(\mu t)$ where $\lambda = \frac{-1}{2RC}$ and $\mu = \sqrt{\frac{1}{LC} - \lambda^2}$

Direct Implementation
Circuit Definition

From this equation we can directly calculate the output voltage at any given time:

```
struct circuit {
  circuit(double r, double l, double c)
  : r_(r), c_(c) {
    lambda_ = -1.0 / (2.0 * r * c);
    mu_     = sqrt((1.0 / (l * c)) -
                   lambda_ * lambda_);
  }
  double operator()(double t) {
    return (1.0 / (mu_ * r_ * c_)) *
           exp(lambda_ * t) * sin(mu_ * t);
  }
private:
  double r_, c_, lambda_, mu_;
};
```

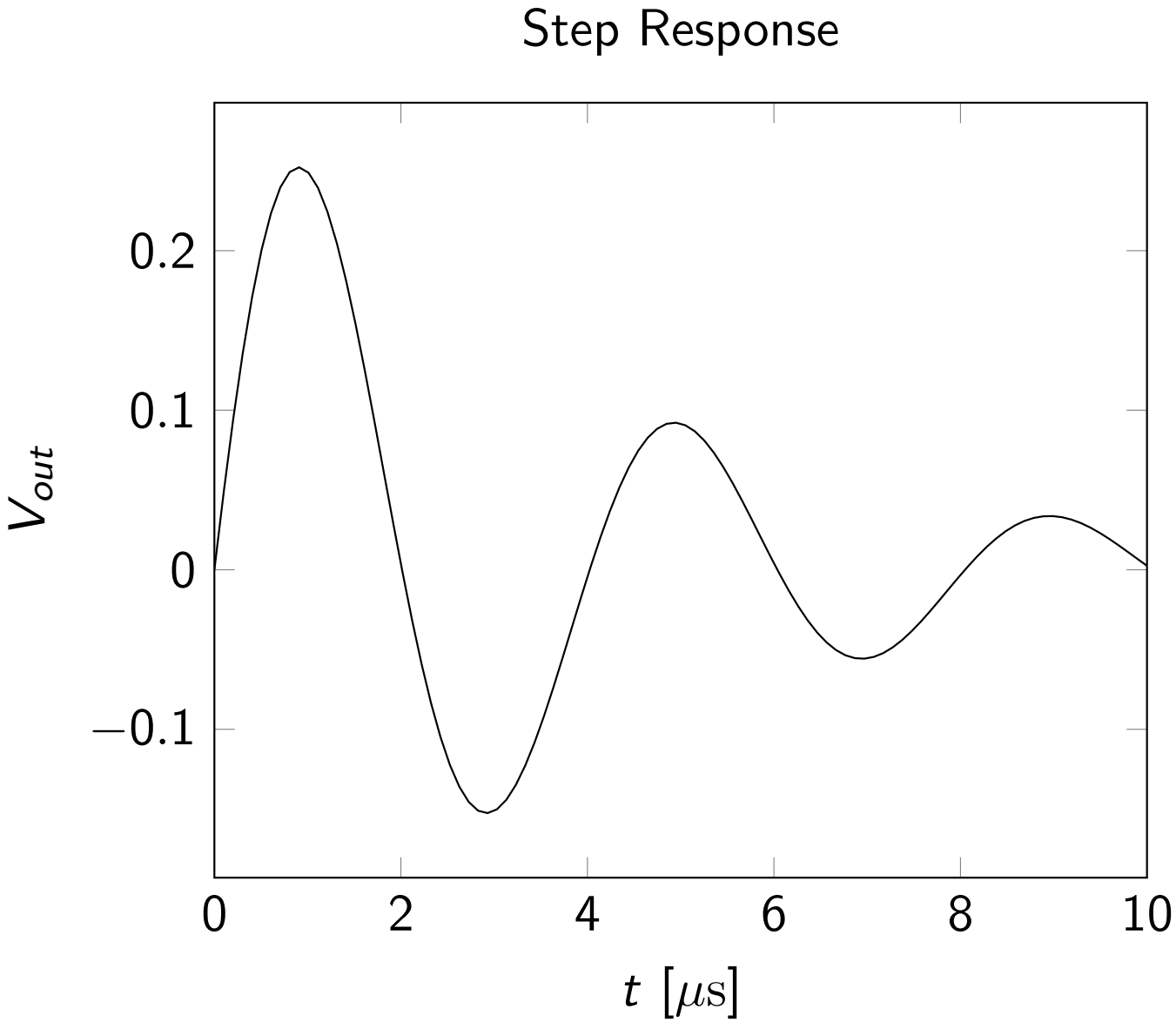
Direct Implementation
Simulation

. . . and plot the values as desired:

```
int main() {
  circuit ckt(100.0, 20e-6, 20e-9);
  for (double t = 0; t < 10e-6; t += 0.1e-6) {
    std::cout << t << " " << ckt(t) << std::endl;
  }
}
```

Direct Implementation

Results



Another Way

This approach is simple to code but has some disadvantages:

- It cannot be automated (requires manually solving differential equations)
- It is not general (some categories of equations cannot be solved in closed form)

By using numeric integration we can overcome both of those problems.

The Boost.ODEInt library

This library, accepted into Boost in September 2012, provides a simple interface for numeric integration. Users need only provide

- A state definition (container, number of states, type of data)
- A callable object that calculates the change in the state vector $\frac{dX(t)}{dt} = f(X(t), t)$
- Initial conditions
- A callable object to serve as a sink for the values at each time point as they are calculated

Let's see how that could work for our example circuit . . .

Numeric Integration with ODEInt

Circuit Definition

The circuit state will be the two independent variables in the equations we previously defined: the inductor current I_L, and the output voltage V_{out}.

```
typedef std::array<double, 2> state_t;
struct circuit {
  circuit(double r, double l, double c)
  : r_(r), l_(l), c_(c) {}
  void operator()(state_t const& x,
                  state_t& dxdt,
                  double t) {
    // calculate state derivatives from current state
    dxdt[0] = ((1 - x[0]) / r_ - x[1]) / c_;
    dxdt[1] = x[0] / l_;
  }
  private:
  double r_, l_, c_;  // ckt params, not ``state''
};
```

Numeric Integration with ODEInt

Simulation

```
int main() {
  using namespace boost::numeric::odeint;
  circuit ckt(100.0, 20e-6, 20e-9);
  state_t x{0.0, 0.0};  // initial conditions
  integrate( ckt, x,
             0.0, 10e-6, 0.1e-6,  // time range
             // sink for values:
             [](state_t const& x, double t) {
               std::cout << t << " " << x[0] <<
                            std::endl;
             });
}
```

Systematizing the Process

This is a good start, but we still had to manually rearrange the circuit equations to be in the required form

$$\frac{dX(t)}{dt} = f(X(t), t)$$

We want to handle arbitrary linear circuits supplied by the user. We need a *canonical* form for representing circuits, that we can calculate with.

Systematizing the Process
Modified Nodal Analysis

MNA is the best-known way of representing circuits mathematically. A circuit is turned into a set of matrices with this relationship[1] :

$$C * \frac{dX}{dt} = -G * X + u$$

where

- X is the state vector (node voltages, inductor currents, and input currents)
- C contains time-derivative circuit values (capacitors and inductors)
- G contains time-invariant circuit element values (everything else)
- u is the input vector

[1] Slightly simplified for this presentation

Modified Nodal Analysis

Example Circuit

$$\overbrace{\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & C & 0 & 0 \\ 0 & 0 & L & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}}^{C} * \overbrace{\begin{bmatrix} \dot{V_0} \\ \dot{V_1} \\ \dot{I_L} \\ \dot{I_{in}} \end{bmatrix}}^{\frac{dX}{dt}} =$$

$$- \overbrace{\begin{bmatrix} \frac{1}{R} & -\frac{1}{R} & 0 & 1 \\ -\frac{1}{R} & \frac{1}{R} & 1 & 0 \\ 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{bmatrix}}^{G} * \overbrace{\begin{bmatrix} V_0 \\ V_1 \\ I_L \\ I_{in} \end{bmatrix}}^{X} + \overbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ -V_{in}(t) \end{bmatrix}}^{u}$$

Values for each of the components are stored in the matrices in a conventional way (a *stamp*) based on their values and connecting nodes.

Modified Nodal Analysis

We Need a Matrix Library

If we take the MNA equation and solve for $\frac{dX(t)}{dt}$ we get the form required for **Boost.ODEInt**:

$$\frac{dX}{dt} = -C^{-1} * G * X + C^{-1} * u$$

To represent the circuit and perform these calculations we need a matrix library[2].

[2] Because we're not going to write our own matrix class. Right? Right?

Eigen

Eigen is a mature expression template based matrix library with a lot of powerful features such as

Sparse Matrix handling - compact storage for matrices with many zero elements

Algorithms such as common factorizations, eigenvalue calculation, etc.

Acceleration lazy evaluation, SIMD parallelization

Eigen
Implementing MNA "stamps"

Resistor example:

```
template<int sz>
void stamp_r(Eigen::Matrix<double, sz, sz>& G,
             Eigen::Matrix<double, sz, sz> const&,
             int node1, int node2, double r) {
  G(node1, node1) += 1.0/r;   // first node current
  G(node1, node2) -= 1.0/r;
  G(node2, node2) += 1.0/r;   // second node current
  G(node2, node1) -= 1.0/r;
}
```

and usage:

```
        stamp_r(G, C, 0, 1, r);
```

Eigen
A small problem

The matrix **C** is *singular* :

$$\overbrace{\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & C & 0 & 0 \\ 0 & 0 & L & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}}^{C} * \overbrace{\begin{bmatrix} \dot{V}_0 \\ \dot{V}_1 \\ \dot{I}_L \\ \dot{I}_{in} \end{bmatrix}}^{\frac{dX}{dt}} =$$

$$- \overbrace{\begin{bmatrix} \frac{1}{R} & -\frac{1}{R} & 0 & 1 \\ -\frac{1}{R} & \frac{1}{R} & 1 & 0 \\ 0 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{bmatrix}}^{G} * \overbrace{\begin{bmatrix} V_0 \\ V_1 \\ I_L \\ I_{in} \end{bmatrix}}^{X} + \overbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ -V_{in}(t) \end{bmatrix}}^{u}$$

But we must invert **C** to put this equation in standard form for **ODEInt**. **Eigen** can help us fix this.

Regularization
The fix

We can reduce the equations to a 2x2, non-singular form by:

1. Moving non-zero elements of C to the upper left corner via Gaussian elimination with pivoting
2. Taking advantage of the all-zero rows of C to restate two state variables in terms of the other two

Regularization
Factorization

Eigen implements Gaussian elimination via the LU Factorization:

```
auto lu = C.fullPivLu();
matrix4_t Cprime = lu.matrixLU().
                   template triangularView<Upper>();
// apply lu to G and u...
```

$$\overbrace{\begin{bmatrix} L & 0 & 0 & 0 \\ 0 & C & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}}^{C} * \overbrace{\begin{bmatrix} \dot{I}_L \\ \dot{V}_1 \\ \dot{V}_0 \\ \dot{I}_{in} \end{bmatrix}}^{\frac{dX}{dt}} =$$

$$- \overbrace{\begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & \frac{1}{R} & -\frac{1}{R} & 0 \\ 0 & -\frac{1}{R} & \frac{1}{R} & 1 \\ 0 & 0 & -1 & 0 \end{bmatrix}}^{G} * \overbrace{\begin{bmatrix} I_L \\ V_1 \\ V_0 \\ I_{in} \end{bmatrix}}^{X} + \overbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ -V_{in}(t) \end{bmatrix}}^{u}$$

Regularization
solving for the reduced state

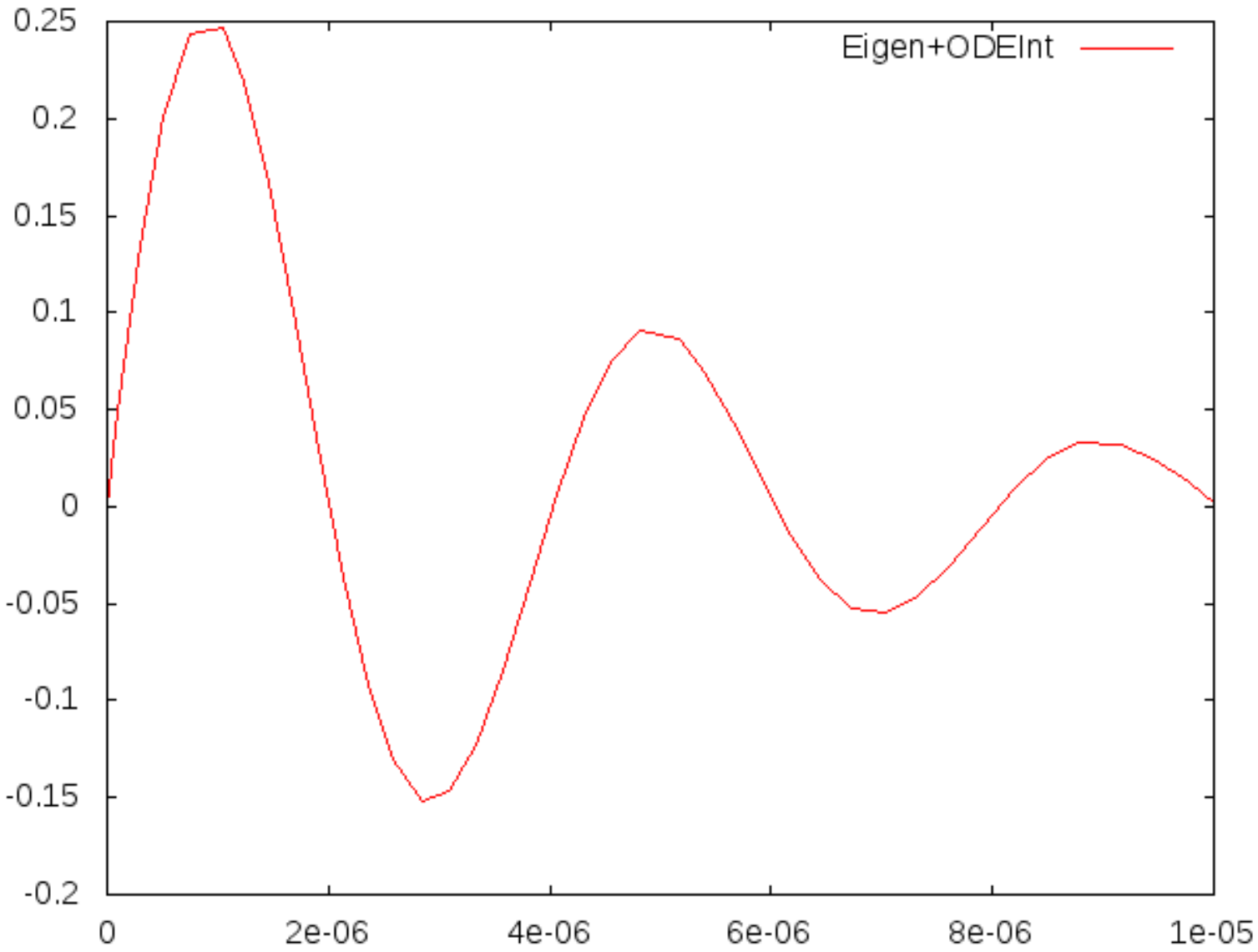
Eliminate V_0 and I_{in} with submatrix algebra:

```
matrix2_t Cnew = Cprime.topLeftCorner(2, 2);
matrix2_t G11 = Gprime.topLeftCorner(2, 2);
// etc.
matrix2_t Gnew = G11 - G12 * G22.inv() * G21;
```

$$\overbrace{\begin{bmatrix} L & 0 \\ 0 & C \end{bmatrix}}^{Cnew} * \begin{bmatrix} \dot{I_L} \\ \dot{V_1} \end{bmatrix} = - \overbrace{\begin{bmatrix} 0 & -1 \\ 1 & \frac{1}{R} \end{bmatrix}}^{Gnew} * \begin{bmatrix} I_L \\ V_1 \end{bmatrix} + \overbrace{\begin{bmatrix} 0 & 0 \\ 0 & \frac{1}{R} \end{bmatrix}}^{Bnew} * \begin{bmatrix} 0 \\ V_{in}(t) \end{bmatrix}$$

```
// now we can calculate the state change:
dxdt = -Cnew.inv() * (Gnew * x + Bnew * u);
```

Results


Eigen+ODEInt
0.25
0.2
0.15
0.1
0.05
0
-0.05
-0.1
-0.15
-0.2
0
2e-06
4e-06
6e-06
8e-06
1e-05

Resources

Eigen	`http://eigen.tuxfamily.org`
ODEInt	`http://www.odeint.com`
Code	`http://github.com/jefftrull/CktSimLightningTalk`