Modeling Physical Systems with Modern Object Oriented Perl YAPC::NA 2012

Joel Berger

University of Illinois at Chicago

June 15, 2012

Physical Simulations

Differential Equations

A set of rules that define how variables change with some parameter

$$x(t_2) = x(t_1) + dt * \frac{dx}{dt}$$

Example: Exponential Growth

$$\frac{dM}{dt} = rM \quad \Longrightarrow \quad M(t) = P\exp(rt)$$

Physical Simulations

Differential Equations

A set of rules that define how variables change with some parameter

$$x(t_2) = x(t_1) + dt * \frac{dx}{dt}(x, f(t1, t2))$$

Example: Exponential Growth

$$\frac{dM}{dt} = rM \quad \Longrightarrow \quad M(t) = P\exp(rt)$$

Physical Simulations

Differential Equations

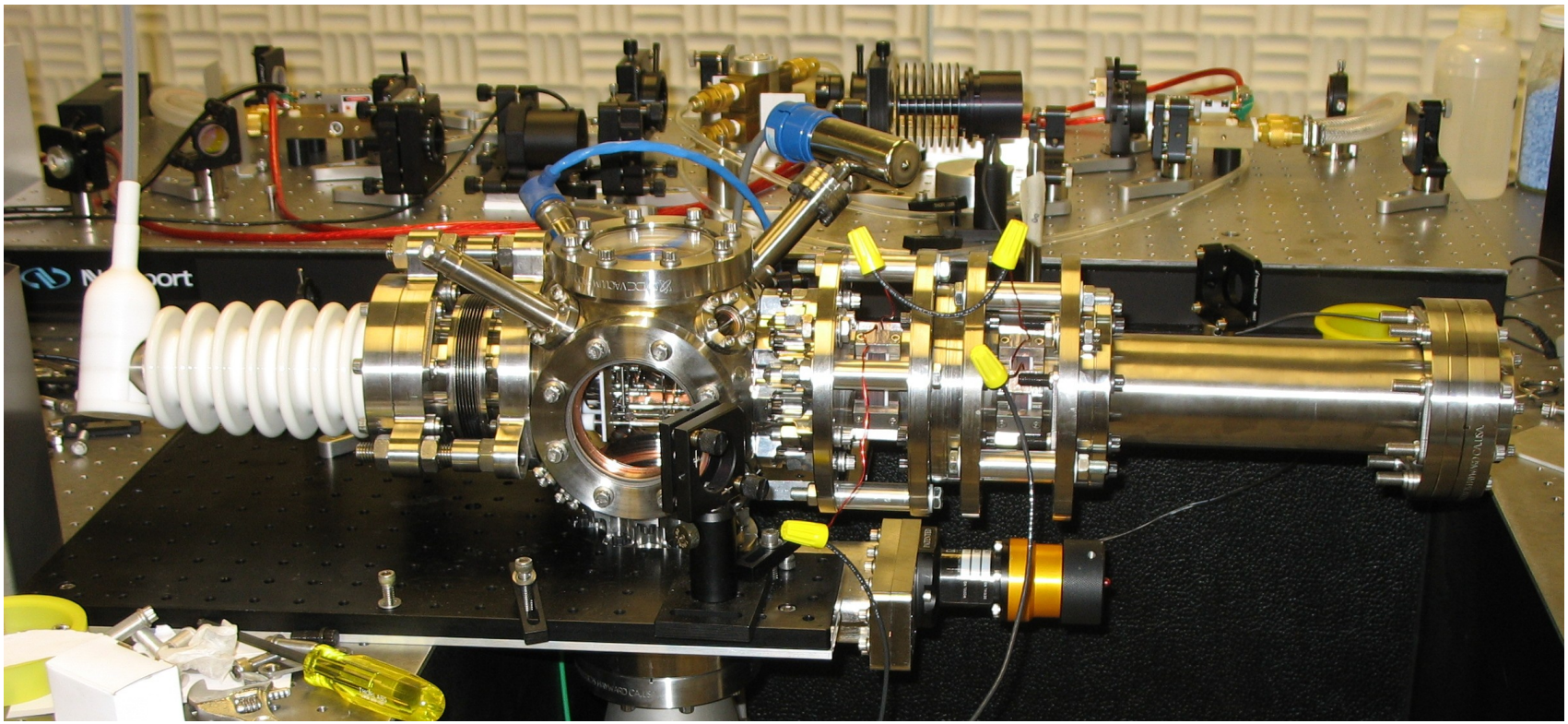
A set of rules that define how variables change with some parameter

$$x(t_2) = x(t_1) + dt * \frac{dx}{dt}(x, f(t1, t2))$$

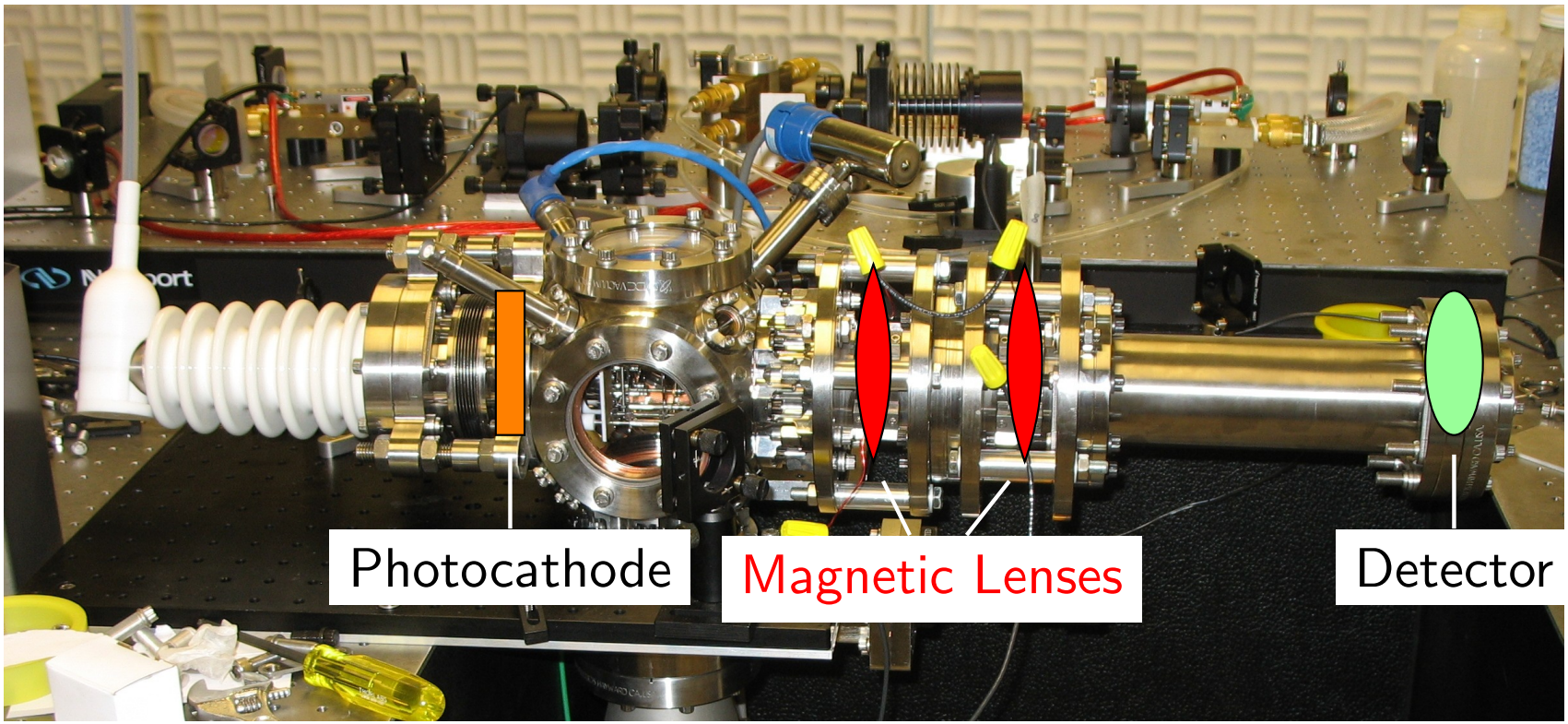
Example: Exponential Growth

$$\frac{dM}{dt} = rM \quad \Longrightarrow \quad M(t) = P\exp(rt)$$

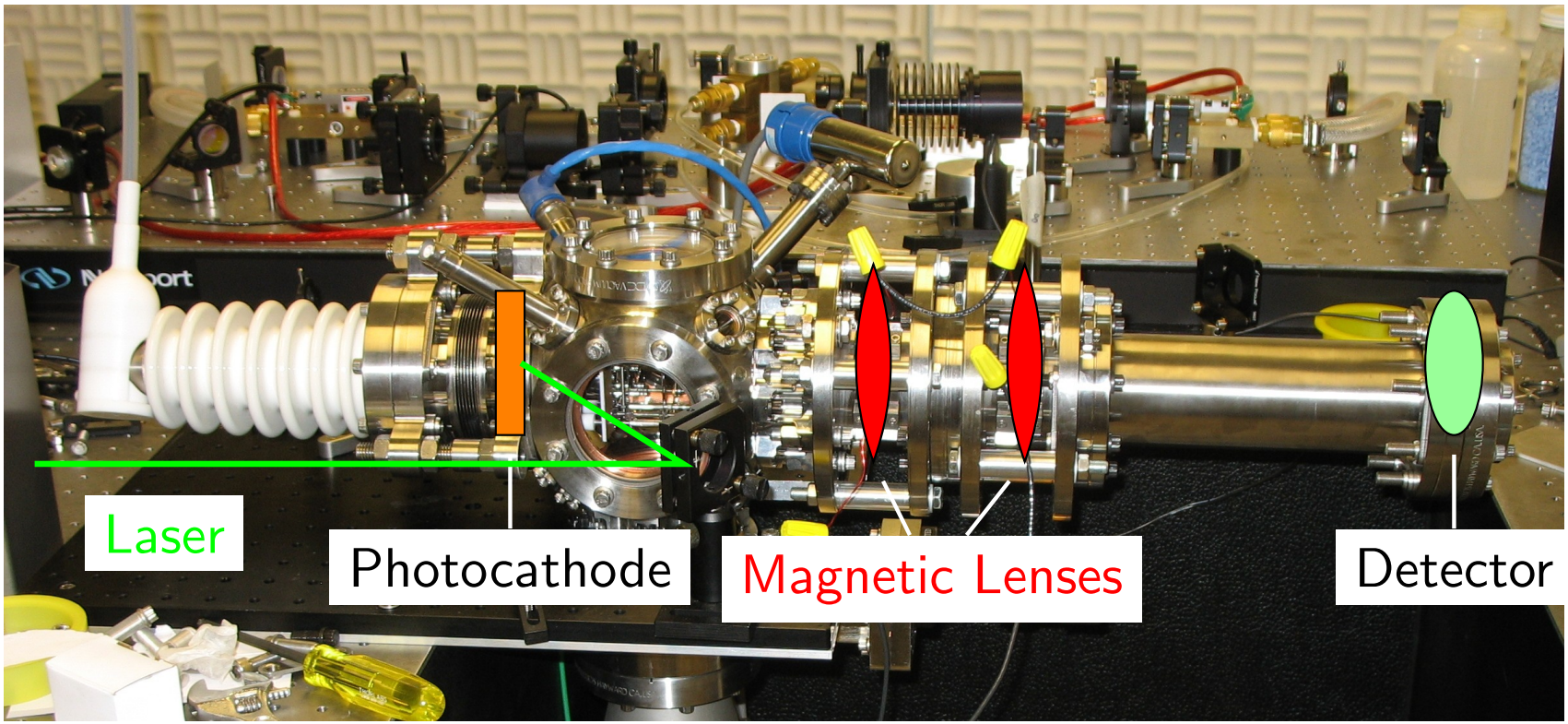
My Research: Ultrafast Electron Microscopy



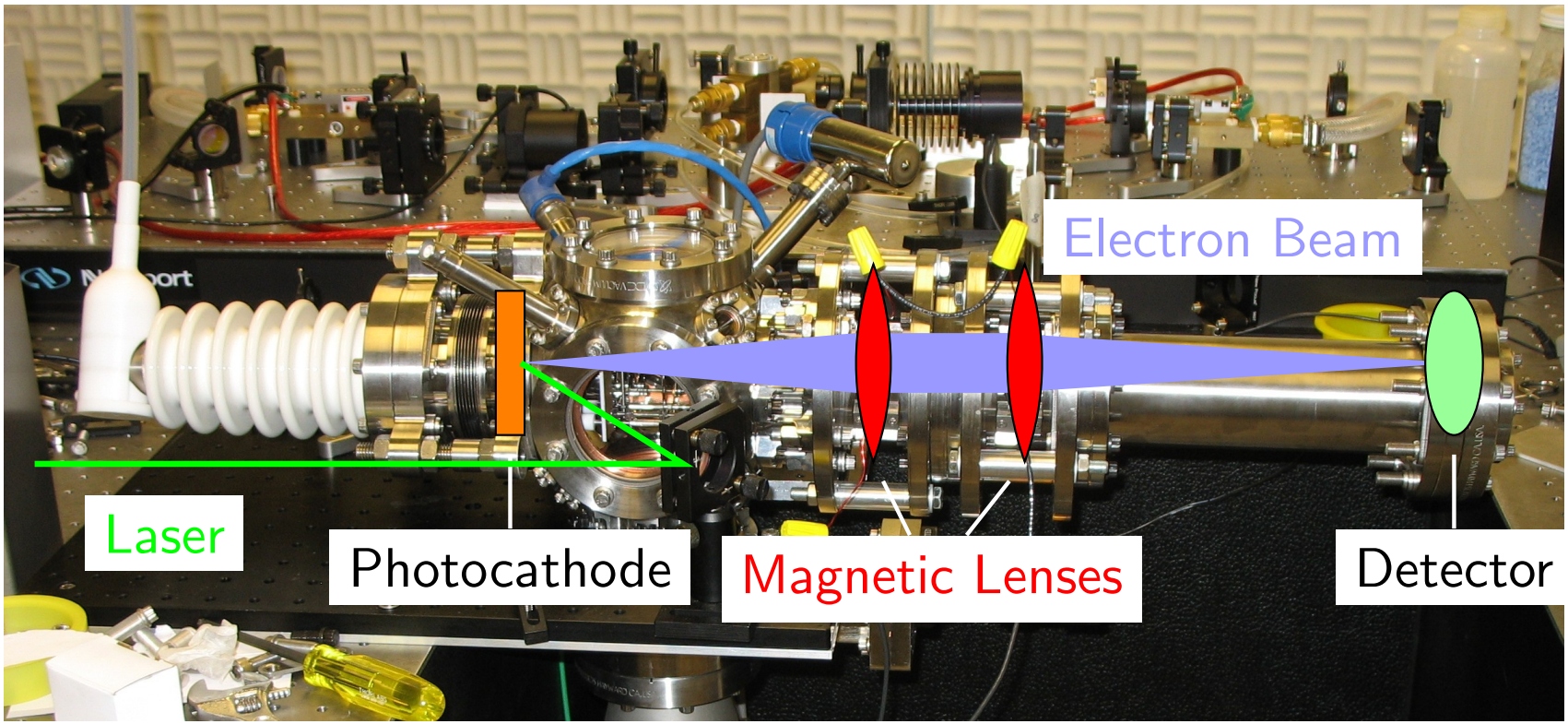
My Research: Ultrafast Electron Microscopy



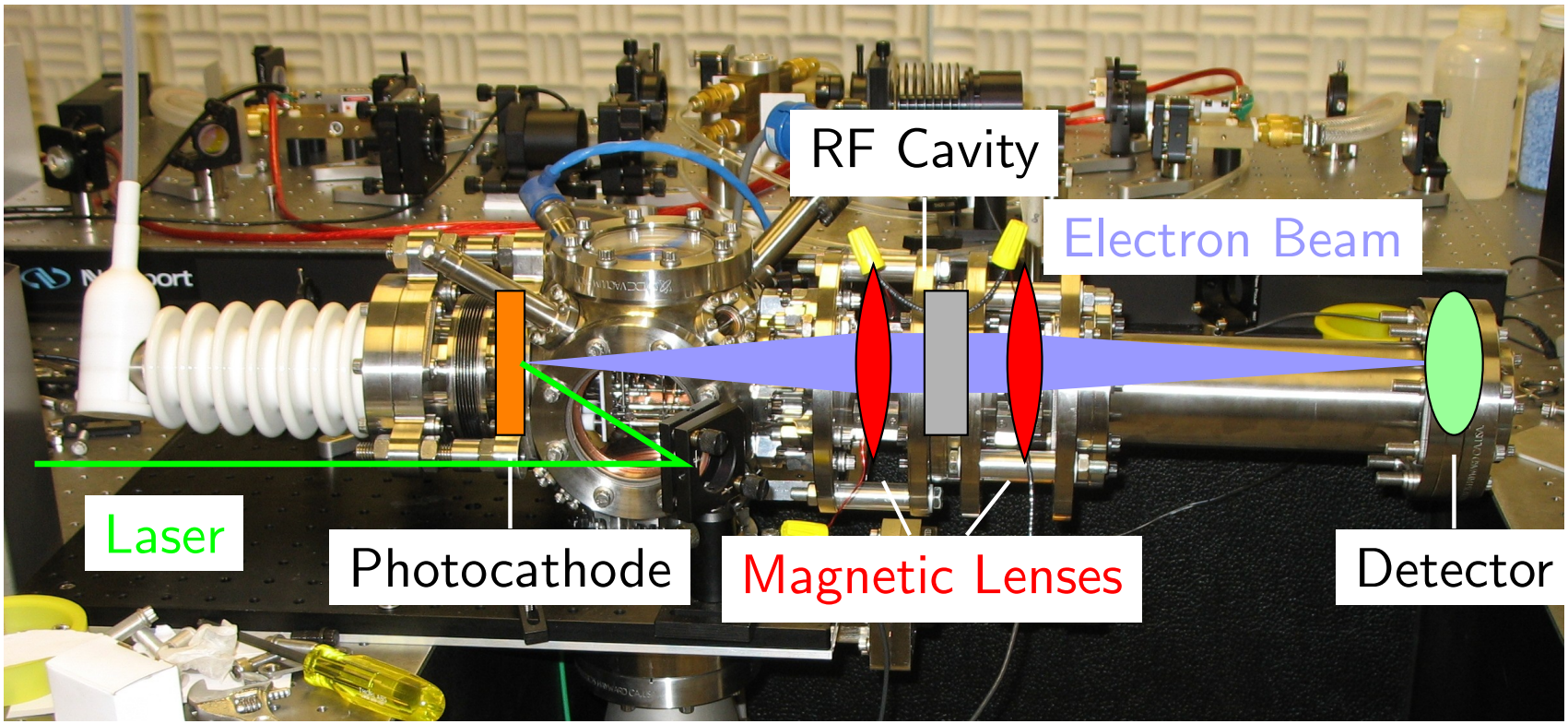
My Research: Ultrafast Electron Microscopy



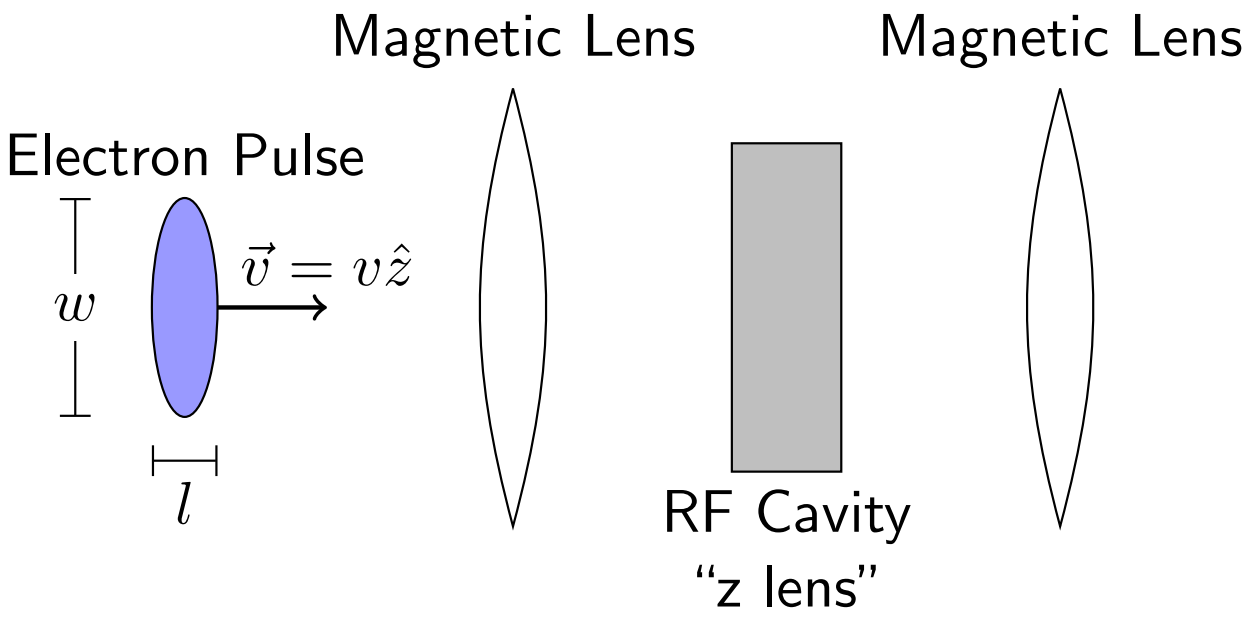
My Research: Ultrafast Electron Microscopy



My Research: Ultrafast Electron Microscopy

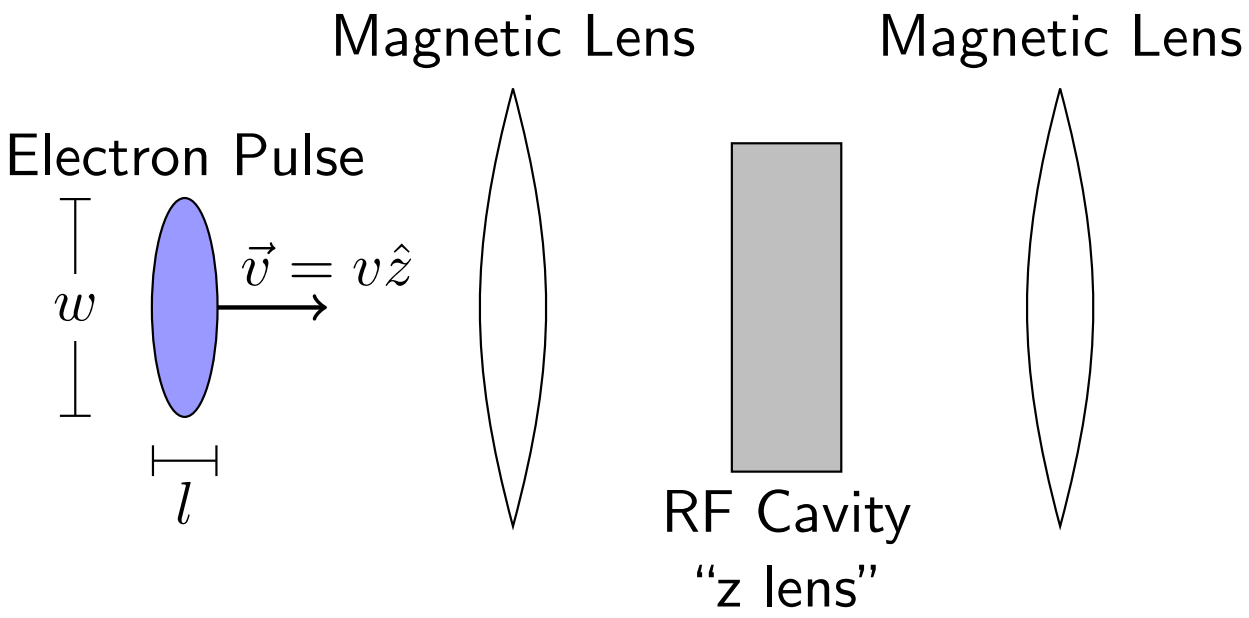


The Challenge: A Flexible Interface to a Complex Model



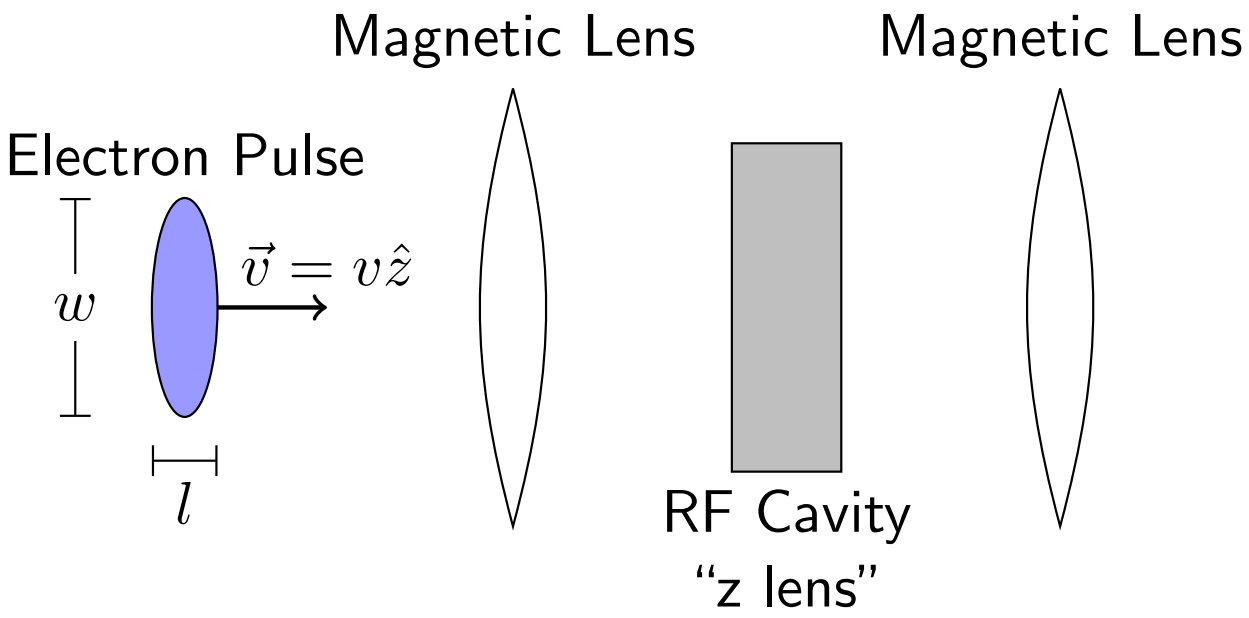
- Need: Compute dynamics of electron pulse $(w(t), l(t))$
- Note: Generation and optical elements add terms to DE
- Want: Representitive OO user-level interface

The Challenge: A Flexible Interface to a Complex Model



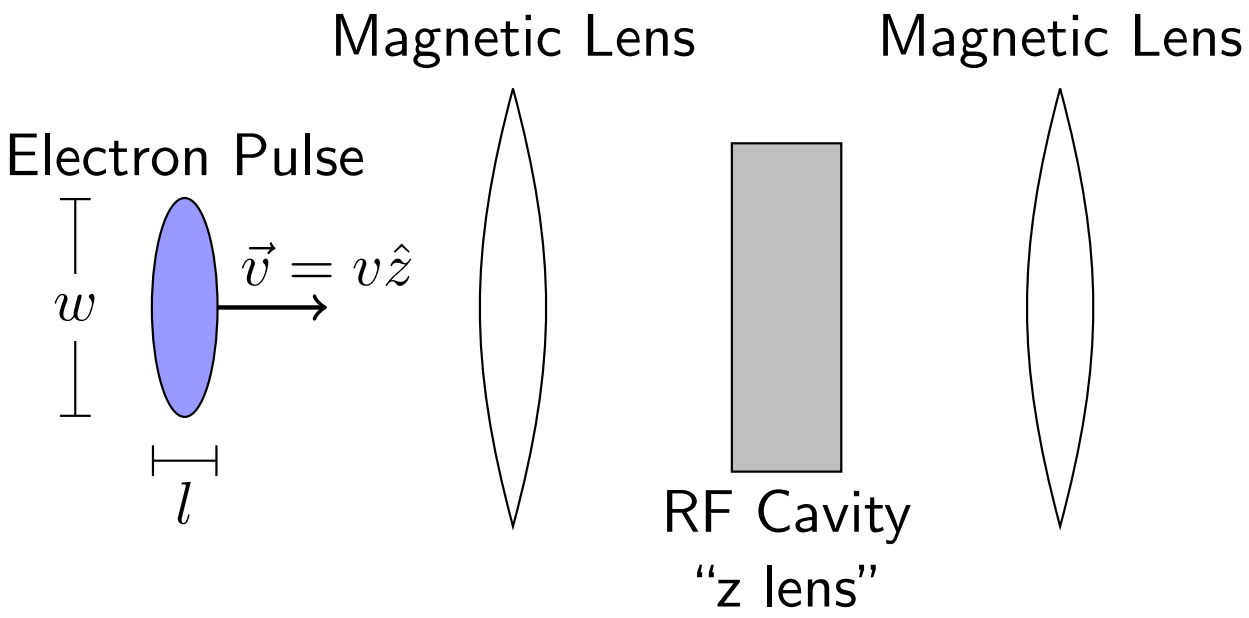
- Need: Compute dynamics of electron pulse ($w(t)$, $l(t)$)
- Note: Generation and optical elements add terms to DE
- Want: Representitive OO user-level interface

The Challenge: A Flexible Interface to a Complex Model



- Need: Compute dynamics of electron pulse $(w(t), l(t))$
- Note: Generation and optical elements add terms to DE
- Want: Representitive OO user-level interface

The Challenge: A Flexible Interface to a Complex Model



- Need: Compute dynamics of electron pulse ($w(t)$, $l(t)$)
- Note: Generation and optical elements add terms to DE
- Want: Representitive OO user-level interface

The "State of the Art"

Old codes are

- lacking full 6D dynamics
- optimized for performance vs usablilty
- hard to customize
- near impossible to comprehend

`http://laacg.lanl.gov/laacg/services/download_sf.phtml`

Getting Started with Poisson Superfish

. . . We do not recommend trying to build an input file "from scratch." Instead, find an example file that is similar to the problem you are trying to solve. Make a copy of the file and then make any necessary modifications to the geometry and options.

The "State of the Art"

Old codes are

- lacking full 6D dynamics
- optimized for performance vs usablilty
- hard to customize
- near impossible to comprehend

`http://laacg.lanl.gov/laacg/services/download_sf.phtml`

Getting Started with Poisson Superfish

. . . We do not recommend trying to build an input file "from scratch." Instead, find an example file that is similar to the problem you are trying to solve. Make a copy of the file and then make any necessary modifications to the geometry and options.

Other Attempts

Mathematica:

Pros:

- Can solve dynamics
- Pretty-printing of math for readability

Cons:

- Closed-source and expensive!
- No OO and no key-value datatypes
- Still rather slow $\sim$2mins/sim

Modelica:

Pros:

- Open-source, but behind close-source variants
- Unique OO language for physical simulation
- Classes have DEs as properties

Cons:

- Lacks "has-a" relationship
- Composing DEs not trivial
- User-facing object instantiation not trivial
- Some numerical problems (?)

Other Attempts

Mathematica:
Pros:

- Can solve dynamics
- Pretty-printing of math for readability

Cons:

- Closed-source and expensive!
- No OO and no key-value datatypes
- Still rather slow $\sim$2mins/sim

Modelica:
Pros:

- Open-source, but behind close-source variants
- Unique OO language for physical simulation
- Classes have DEs as properties

Cons:

- Lacks "has-a" relationship
- Composing DEs not trivial
- User-facing object instantiation not trivial
- Some numerical problems (?)

Other Attempts

Mathematica:
Pros:

- Can solve dynamics
- Pretty-printing of math for readability

Cons:

- Closed-source and expensive!
- No OO and no key-value datatypes
- Still rather slow $\sim$2mins/sim

Modelica:
Pros:

- Open-source, but behind close-source variants
- Unique OO language for physical simulation
- Classes have DEs as properties

Cons:

- Lacks "has-a" relationship
- Composing DEs not trivial
- User-facing object instantiation not trivial
- Some numerical problems (?)

Other Attempts

Mathematica:
Pros:

- Can solve dynamics
- Pretty-printing of math for readability

Cons:

- Closed-source and expensive!
- No OO and no key-value datatypes
- Still rather slow $\sim$2mins/sim

Modelica:
Pros:

- Open-source, but behind close-source variants
- Unique OO language for physical simulation
- Classes have DEs as properties

Cons:

- Lacks "has-a" relationship
- Composing DEs not trivial
- User-facing object instantiation not trivial
- Some numerical problems (?)

Other Attempts

Mathematica:
Pros:

- Can solve dynamics
- Pretty-printing of math for readability

Cons:

- Closed-source and expensive!
- No OO and no key-value datatypes
- Still rather slow $\sim$2mins/sim

Modelica:
Pros:

- Open-source, but behind close-source variants
- Unique OO language for physical simulation
- Classes have DEs as properties

Cons:

- Lacks "has-a" relationship
- Composing DEs not trivial
- User-facing object instantiation not trivial
- Some numerical problems (?)

. . . But First, Some Bookkeeping

If an object knows where it is, can it remember where it came from?

```
use MooseX::Declare;
use Method::Signatures::Modifiers;
use MooseX::RememberHistory;

class MyClass {
  has 'x' => (
    traits => ['RememberHistory'],
    isa => 'Num', is => 'rw',
    default => 0
  );
}

my $obj = MyClass->new;
$obj->x( 1 );
$obj->x( 2 );

print join ', ', @{ $obj->x_history };
# 0, 1, 2
```

- use objects as data storage
  - current data during simulation
  - all data afterwards
- works for fixed width solvers
- adaptive solvers call functions repeatedly (i.e. `PerlGSL::DiffEq`)

. . . But First, Some Bookkeeping

If an object knows where it is, can it remember where it came from?

```
use MooseX::Declare;
use Method::Signatures::Modifiers;
use MooseX::RememberHistory;

class MyClass {
  has 'x' => (
    traits => ['RememberHistory'],
    isa => 'Num', is => 'rw',
    default => 0
  );
}

my $obj = MyClass->new;
$obj->x( 1 );
$obj->x( 2 );

print join ', ', @{ $obj->x_history };
# 0, 1, 2
```

- use objects as data storage
  - current data during simulation
  - all data afterwards
- works for fixed width solvers
- adaptive solvers call functions repeatedly (i.e. `PerlGSL::DiffEq`)

. . . But First, Some Bookkeeping

If an object knows where it is, can it remember where it came from?

```
use MooseX::Declare;
use Method::Signatures::Modifiers;
use MooseX::RememberHistory;

class MyClass {
  has 'x' => (
    traits => ['RememberHistory'],
    isa => 'Num', is => 'rw',
    default => 0
  );
}

my $obj = MyClass->new;
$obj->x( 1 );
$obj->x( 2 );

print join ', ', @{ $obj->x_history };
# 0, 1, 2
```

- use objects as data storage
  - current data during simulation
  - all data afterwards
- works for fixed width solvers
- adaptive solvers call functions repeatedly (i.e. `PerlGSL::DiffEq`)

. . . But First, Some Bookkeeping

If an object knows where it is, can it remember where it came from?

```
use MooseX::Declare;
use Method::Signatures::Modifiers;
use MooseX::RememberHistory;

class MyClass {
  has 'x' => (
    traits => ['RememberHistory'],
    isa => 'Num', is => 'rw',
    default => 0
  );
}

my $obj = MyClass->new;
$obj->x( 1 );
$obj->x( 2 );

print join ', ', @{ $obj->x_history };
# 0, 1, 2
```

- use objects as data storage
  - current data during simulation
  - all data afterwards
- works for fixed width solvers
- adaptive solvers call functions repeatedly (i.e. `PerlGSL::DiffEq`)

. . . But First, Some Bookkeeping

If an object knows where it is, can it remember where it came from?

```
use MooseX::Declare;
use Method::Signatures::Modifiers;
use MooseX::RememberHistory;

class MyClass {
  has 'x' => (
    traits => ['RememberHistory'],
    isa => 'Num', is => 'rw',
    default => 0
  );
}

my $obj = MyClass->new;
$obj->x( 1 );
$obj->x( 2 );

print join ', ', @{ $obj->x_history };
# 0, 1, 2
```

- use objects as data storage
  - current data during simulation
  - all data afterwards
- works for fixed width solvers
- adaptive solvers call functions repeatedly (i.e. `PerlGSL::DiffEq`)

. . . But First, Some Bookkeeping

If an object knows where it is, can it remember where it came from?

```
use MooseX::Declare;
use Method::Signatures::Modifiers;
use MooseX::RememberHistory;

class MyClass {
  has 'x' => (
    traits => ['RememberHistory'],
    isa => 'Num', is => 'rw',
    default => 0
  );
}

my $obj = MyClass->new;
$obj->x( 1 );
$obj->x( 2 );

print join ', ', @{ $obj->x_history };
# 0, 1, 2
```

- use objects as data storage
  - current data during simulation
  - all data afterwards
- works for fixed width solvers
- adaptive solvers call functions repeatedly (i.e. `PerlGSL::DiffEq`)

Physical Classes

```
use MooseX::RememberHistory;
use MooseX::Declare;
use Method::Signatures::Modifiers;

class MyForce {
  has 'strength' => ( isa => 'Num',  is => 'rw', default => 0 );
  has 'affect'   => ( isa => 'CodeRef', is => 'ro', required => 1 );
}

class MyThing {
  has 'mass' => ( isa => 'Num', is => 'ro', required => 1 );
  has 'x'    => ( traits => [ 'RememberHistory' ], isa => 'Num',
                  is => 'rw', default => 0 );
  has 'vx'   => ( traits => [ 'RememberHistory' ], isa => 'Num',
                  is => 'rw', default => 0 );
}
```

The Solver: Attributes

```
class MySim {
  use List::Util 'sum';

  has 'start'  => ( isa => 'Num', is => 'ro', default => 0 );
  has 'end'    => ( isa => 'Num', is => 'ro', default => 1 );
  has 'steps'  => ( isa => 'Num', is => 'ro', default => 100 );

  has 'step'   => ( isa => 'Num', is => 'ro', lazy => 1, builder => 'init_step');
  has 'time'   => ( traits => [ 'RememberHistory' ], isa => 'Num', is => 'rw',
                    lazy => 1, builder => 'init_time' );

  has 'things' => ( isa => 'ArrayRef[MyThing]', is => 'rw', default => sub{[]} );
  has 'forces' => ( isa => 'ArrayRef[MyForce]', is => 'rw', default => sub{[]} );

  method init_step () {
    my $step = ($self->end - $self->start) / $self->steps;
    return $step;
  }

  method init_time () { return $self->start }
```

The Solver: Methods

```
  method evolve ( MyThing $thing ) {
    my $dt = $self->step;
    my $vx = $thing->vx;

    my $force = sum map { $_->affect->($_, $thing) } @{ $self->forces };
    my $acc = $force / ($thing->mass);

    $thing->vx( $vx + $acc * $dt );
    $thing->x( $thing->x + $vx * $dt );
  }

  method run () {
    while ($self->time < $self->end) {
      $self->evolve( $_ ) for @{ $self->things };
      $self->time( $self->time + $self->step );
    }
  }
} # end of class MySim
```

The Script

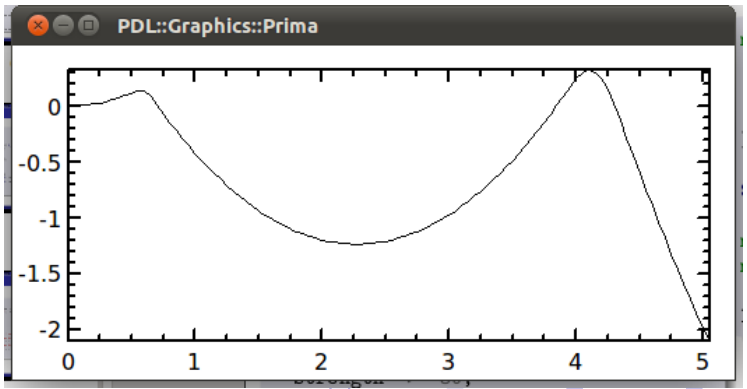
```perl
#!/usr/bin/env perl
use strict; use warnings;
use MySim;
use PDL;
use PDL::Graphics::Prima::Simple;

my $thing = MyThing->new( mass => 2 );

my $acc = MyForce->new(
  strength => 2,
  affect => sub {shift->strength},
);

my $dec = MyForce->new(
  strength => -30,
  affect => sub {
    my ($self, $thing) = @_;
    return 0 if ($thing->x < 0.1);
    return $self->strength;
  },
);
```

```perl
my $sim = MySim->new(
  end => 5,
  things => [ $thing ],
  forces => [ $acc, $dec ],
);

$sim->run;

my $time = pdl $sim->time_history;
my $x    = pdl $thing->x_history;

line_plot($time, $x);
```

The Script

```perl
#!/usr/bin/env perl
use strict; use warnings;
use MySim;
use PDL;
use PDL::Graphics::Prima::Simple;

my $thing = MyThing->new( mass => 2 );

my $acc = MyForce->new(
  strength => 2,
  affect => sub {shift->strength},
);

my $dec = MyForce->new(
  strength => -30,
  affect => sub {
    my ($self, $thing) = @_;
    return 0 if ($thing->x < 0.1);
    return $self->strength;
  },
);
```

```perl
my $sim = MySim->new(
  end => 5,
  things => [ $thing ],
  forces => [ $acc, $dec ],
);

$sim->run;

my $time = pdl $sim->time_history;
my $x    = pdl $thing->x_history;

line_plot($time, $x);
```



Unit Handling (The Implied Covenant)

Mars Surveyor '98 Orbiter

- Software used force in Newtons
- Users entered force in Foot-Pounds

The Covenant:

- Between programmer and user
- "Use the same units!"
- Unexpected and possibly undocumented action at a distance
- With Perl and Moose we can do better . . .

Unit Handling (The Implied Covenant)

Mars Surveyor '98 Orbiter (Sorry Larry!)

- Software used force in Newtons
- Users entered force in Foot-Pounds

The Covenant:

- Between programmer and user
- "Use the same units!"
- Unexpected and possibly undocumented action at a distance
- With Perl and Moose we can do better . . .

Unit Handling (The Implied Covenant)

Mars Surveyor '98 Orbiter (Sorry Larry!)

- Software used force in Newtons
- Users entered force in Foot-Pounds

The Covenant:

- Between programmer and user
- "Use the same units!"
- Unexpected and possibly undocumented action at a distance
- With Perl and Moose we can do better . . .

Unit Handling (The Implied Covenant)

Mars Surveyor '98 Orbiter (Sorry Larry!)

- Software used force in Newtons
- Users entered force in Foot-Pounds

The Covenant:

- Between programmer and user
- "Use the same units!"
- Unexpected and possibly undocumented action at a distance
- With Perl and Moose we can do better . . .

Unit Handling (The Implied Covenant)

Mars Surveyor '98 Orbiter (Sorry Larry!)

- Software used force in Newtons
- Users entered force in Foot-Pounds

The Covenant:

- Between programmer and user
- "Use the same units!"
- Unexpected and possibly undocumented action at a distance
- With Perl and Moose we can do better . . .

Unit Handling (The Implied Covenant)

Mars Surveyor '98 Orbiter (Sorry Larry!)

- Software used force in Newtons
- Users entered force in Foot-Pounds

The Covenant:

- Between programmer and user
- "Use the same units!"
- Unexpected and possibly undocumented action at a distance
- With Perl and Moose we can do better . . .

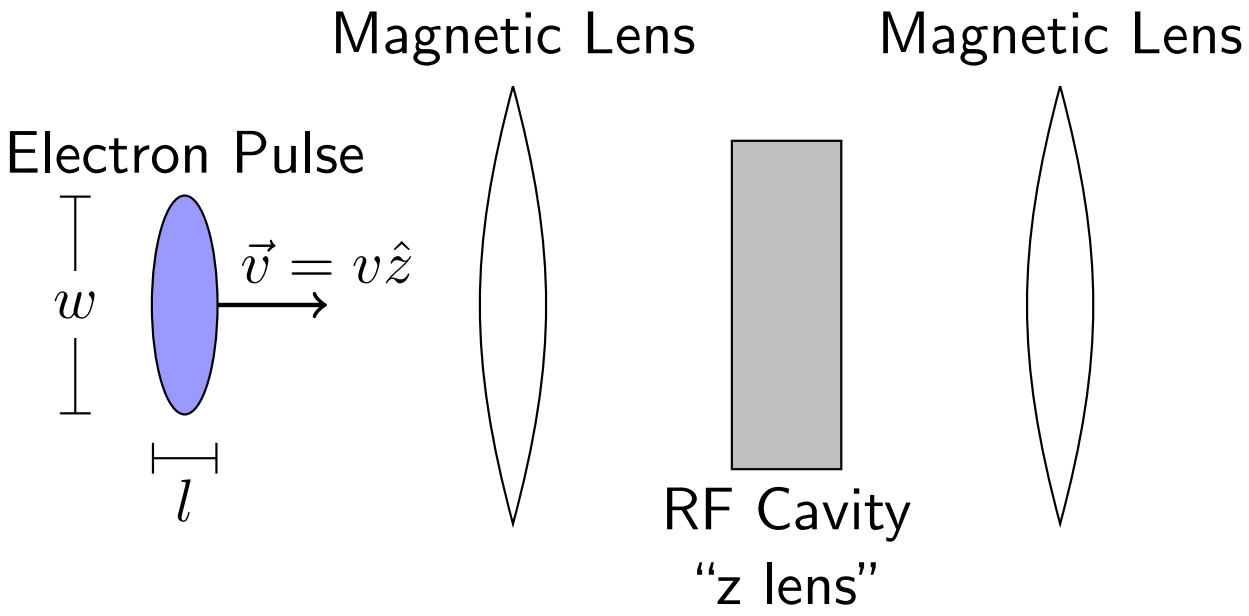
MooseX::Types::NumUnit

- Str to Num coercions
- convert unit if needed

```
use MooseX::Declare;
use Method::Signatures::Modifiers;

class SphericalCow {
  use MooseX::Types::NumUnit qw/num_of_unit/;
  has 'radius'   => ( isa => num_of_unit('m'),   is => 'rw', default => 1 );
  has 'velocity' => ( isa => num_of_unit('m/s'), is => 'rw', default => 0 );
}

use strict; use warnings;

my $cow = SphericalCow->new(
  radius => '1 ft',
);

print $cow->radius . "\n"; # 0.3048
```

Example of `Physics::UEMColumn`

Back to Electron Column Modeling



- As yet unreleased `Physics::UEMColumn`
  - `https://github.com/jberger/Physics-UEMColumn`
- Uses: `PerlGSL::DiffEq` on CPAN
  - C-level solver of Perl-level DE closures

Acknowledgements

UIC UNIVERSITY OF ILLINOIS AT CHICAGO

- Graduate College Dean's Fellowship (Major Student Funding)
- LAS Ph.D. Travel Award (Conference Funding)

- Department of Energy #DE-FG52-09NA29451 (UEM Research Grant)

`https://github.com/jberger/YAPCNA2012`