An adjoint-enabled simulation framework for cardiac electrophysiology

Marie E. Rognes

Center for Biomedical Computing

Simula Research Laboratory

with Patrick Farrell, Simon Funke, Johan Hake, Molly Maleckar

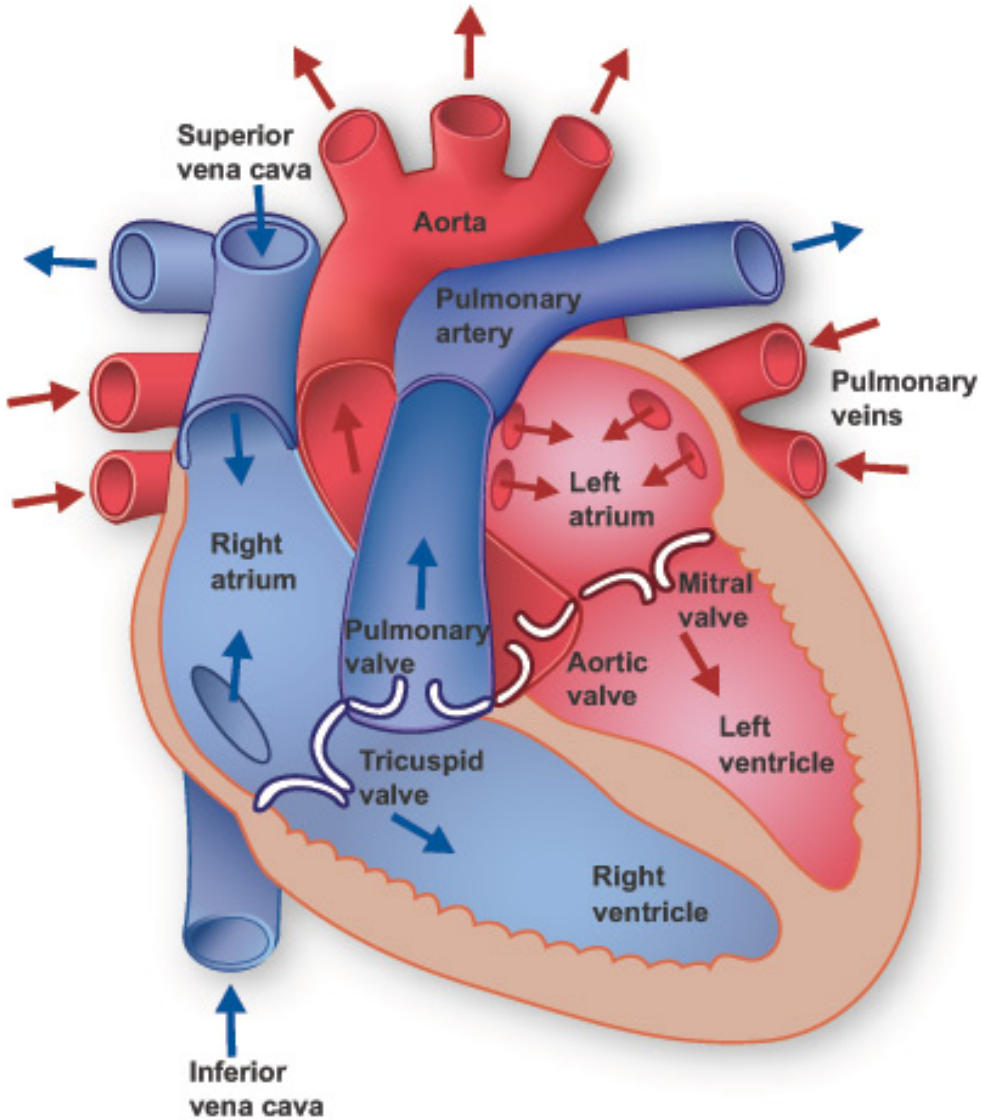
Outline

1. Inverting the heart
2. Point integrals
3. Multistage schemes and their adjoints

Center for Biomedical Computing

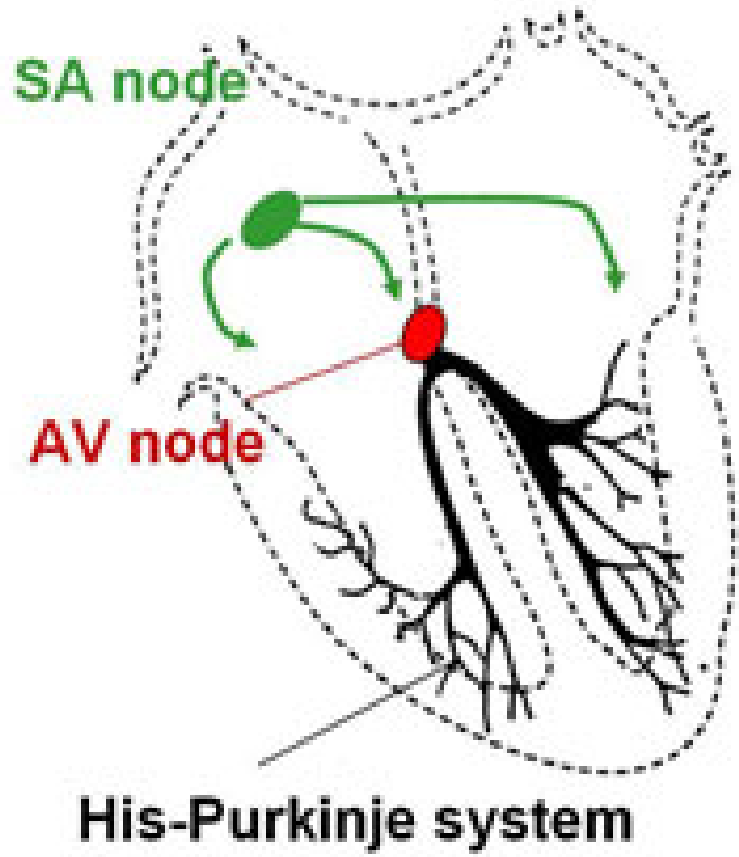
[simula . research laboratory]

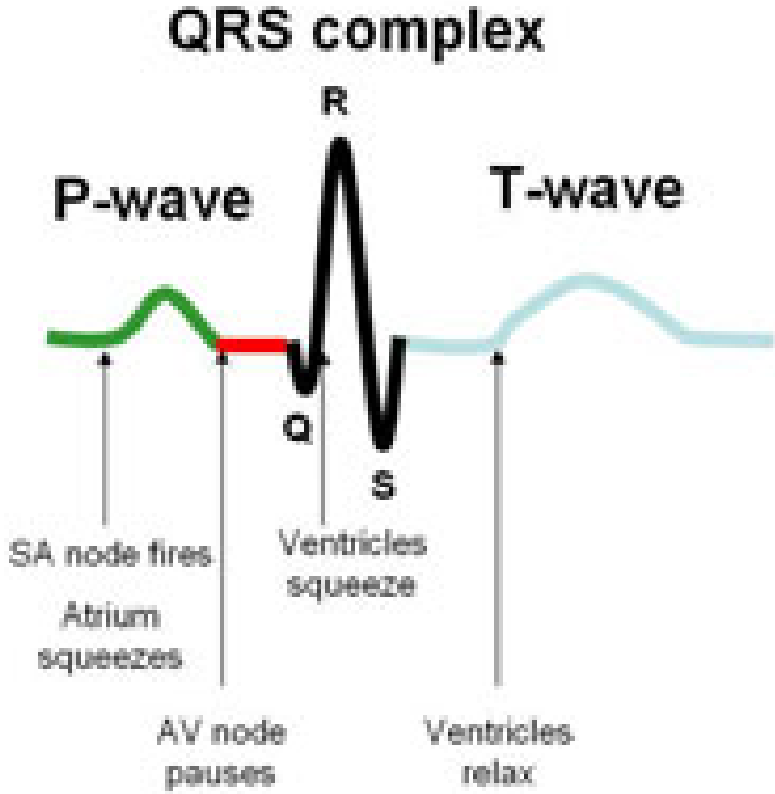
Heart disease is the leading cause of death in the world



[http://health-advisors.org]

The beating of the heart is driven by the electrical signalling of heart cells



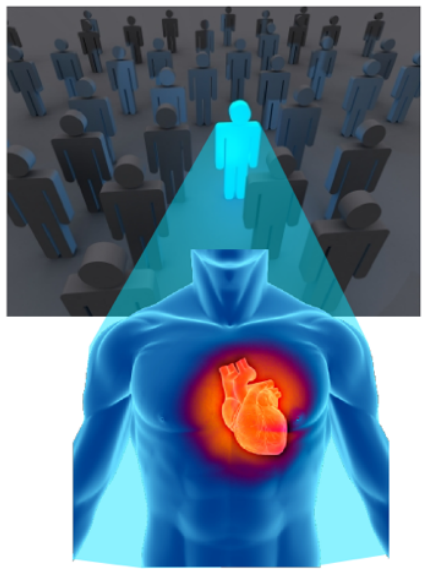


[http://www.bostonscientific.com]

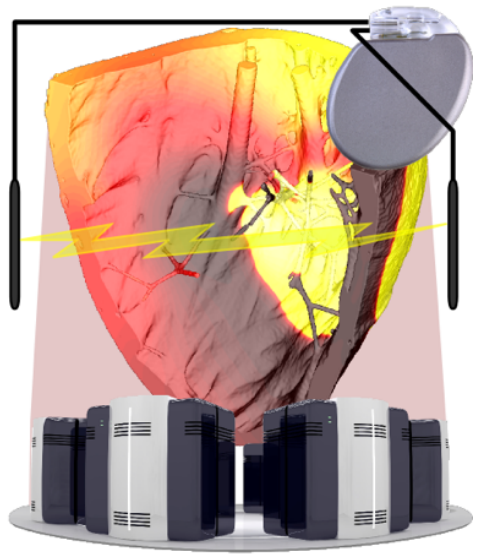
Research aims at commercially and clinically driven advances in cardiac diagnostics and treatments

Combine data from Ultrasound and ECG

Build patient-specific, electromechanical models of the heart



Use models to prescribe and optimize patient treatments



[Edvardsen, Maleckar, Wall et al]

Adjoints are ubiquitous

Constrained optimal control

$$\max_m J(u,m) \quad \text{while} \quad F(u,m) = 0$$

Gradient-based optimization algorithms require the gradient of J with respect to m.

$$\frac{\mathrm{d}J}{\mathrm{d}m} = J_u \frac{\partial u}{\partial m} + J_m$$

Define the adjoint solution z

$$F_u^* z = J_u$$

Then, the derivative computation only involves one forward solve for u and one backward solve for z independent of $\#m$:

$$\frac{\mathrm{d}J}{\mathrm{d}m} = -F_m z + J_m$$

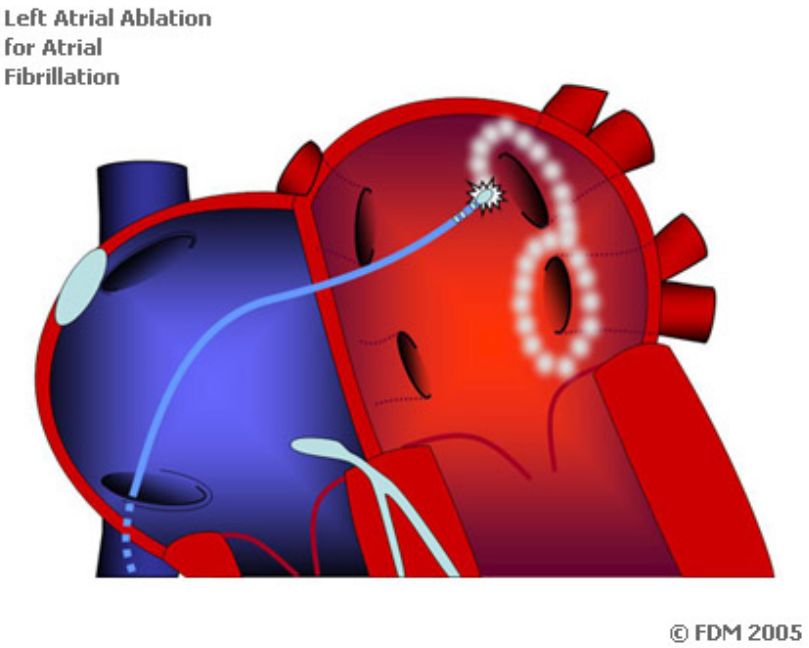
Other applications

Sensitivity analysis, data assimilation, error control, generalized stability theory, ...

Treating abnormal cardiac acitivity: How to find the optimal region to treat atrial fibrillation by ablation?

Find the optimal ablation region m to achieve defibrillation

$$\min_m J(u,m) \quad s.t. \quad F(u,m) = 0$$



[http://www.londonarrhythmiacentre.co.uk/]

For defibrillation, one may consider:

$$J(u,m) = \|u - u_{ideal}\|^2_{L^2(0,T;L^2(\Omega_{obs}))} + \alpha\mathcal{R}(m)$$

[Nagaiah et al, 2011]

The significant gap in maturity between forward and reverse cardiac modelling motivates a new adjoint-enabled simulation framework

Obtaining

```
$ hg clone ssh://hg@bitbucket.org/meg/adjoint-beat
$ cd adjoint-beat
$ python setup.py install --prefix=/home/meg/local
```

Usage

```
from beatadjoint import *
```

The governing equations: the bidomain model

Find the transmembrane potential $v = v(x,t)$, the extracellular potential $u_e(x,t)$ and the ionic current(s) $s = s(x,t)$ such that for almost all $t \in (0,T]$:

$$\begin{aligned} v_t - \operatorname{div}(M_i \nabla v + M_i \nabla u_e) &= -I_{\text{ion}}(v,s) + I_s, \\ \operatorname{div}\left(M_i \nabla v + (M_i + M_e) \nabla u_e\right) &= g, \\ s_t &= F(v,s), \end{aligned}$$

with boundary conditions

$$(M_i \nabla v + M_i \nabla u_e) \cdot n = 0, \quad (M_i \nabla v + (M_i + M_e) \nabla u_e) \cdot n = 0$$

and

$$\int_\Omega u_e = 0.$$

[Tung, 1978]

The typical discretization approach is based on operator splitting and iterations between an ODE and a PDE solve

1. With v^n and s^n as initial conditions at t_n, find v^* and s^* solving

$$v_t^* = -I_{\text{ion}}(v^*, s^*),$$
$$s_t^* = F(v^*, s^*)$$

   on $(t_n, t_n + \theta\kappa]$.
2. With v^* as initial condition, find $v^\dagger$ and u_e^{n+1} such that

$$v^\dagger{}_t - \operatorname{div}(M_i \nabla v^\dagger + M_i \nabla u_e^{n+1}) = I_s,$$
$$\operatorname{div}\left(M_i \nabla v^\dagger + (M_i + M_e) \nabla u_e^{n+1}\right) = g,$$

   on $I_n = (t_n, t_{n+1}]$.
3. If $\theta < 1$: with $v^\dagger$ and s^* as initial conditions at $t_{n+\theta\kappa}$, find v^{n+1} and s^{n+1} solving

$$v_t^{n+1} = -I_{\text{ion}}(v^{n+1}, s^{n+1}),$$
$$s_t^{n+1} = F(v^{n+1}, s^{n+1})$$

   on $(t_n + \theta\kappa, t_{n+1}]$.

[Sundnes et al, 2006]

The specific forms of the ODEs are known as cell models, and greatly vary in complexity

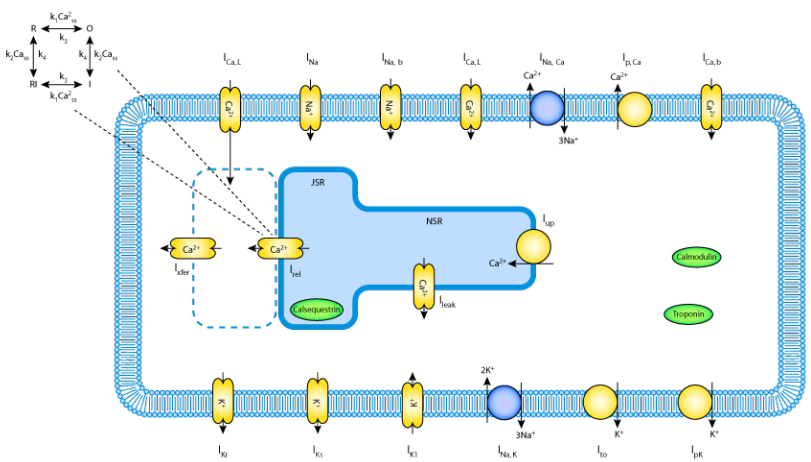
[Fitzhugh, 1961; Rodgers & McCulloch, 1994]

$$v_t = \frac{c_1}{v_a^2}(v - v_r)(v - v_{th})(v_p - v) - \frac{c_2}{v_a}(v - v_r)s$$

$$s_t = b(v - v_{\text{rest}} - c\,s)$$

[ten Tusscher & Panfilov, 2006, www.cellml.org]

```
</units>
<component name="environment">
   <variable name="time" units="millisecond" public_interface="out"/>
</component>
<component name="membrane">
   <variable name="V" units="millivolt" initial_value="-85.23" public_interface="out"/>
   <variable name="R" units="joule_per_mole_kelvin" initial_value="8314.472" public_interface=
   <variable name="T" units="kelvin" initial_value="310" public_interface="out"/>
   <variable name="F" units="coulomb_per_millimole" initial_value="96485.3415" public_interfac
   <variable name="Cm" units="microF" initial_value="0.185" public_interface="out"/>
   <variable name="V_c" units="micrometre3" initial_value="0.016404" public_interface="out"/>
   <variable name="time" units="millisecond" public_interface="in"/>
   <variable name="i_K1" units="picoA_per_picoF" public_interface="in"/>
   <variable name="i_to" units="picoA_per_picoF" public_interface="in"/>
   <variable name="i_Kr" units="picoA_per_picoF" public_interface="in"/>
   <variable name="i_Ks" units="picoA_per_picoF" public_interface="in"/>
   <variable name="i_CaL" units="picoA_per_picoF" public_interface="in"/>
   <variable name="i_NaK" units="picoA_per_picoF" public_interface="in"/>
   <variable name="i_Na" units="picoA_per_picoF" public_interface="in"/>
   <variable name="i_b_Na" units="picoA_per_picoF" public_interface="in"/>
   <variable name="i_NaCa" units="picoA_per_picoF" public_interface="in"/>
   <variable name="i_b_Ca" units="picoA_per_picoF" public_interface="in"/>
   <variable name="i_p_K" units="picoA_per_picoF" public_interface="in"/>
   <variable name="i_p_Ca" units="picoA_per_picoF" public_interface="in"/>
   <variable name="i_Stim" units="picoA_per_picoF" public_interface="out"/>
```



ODE discretizations via multistage schemes

Let $w = (v, s)$ and $G = (-I_{\text{ion}}, F)$:

$$w_t(x,t) = G(x,t,w(x,t)),$$
$$w(0) = w_0.$$

c_1	a_{11}	a_{12}	$\dots$	a_{1s}
c_2	a_{21}	a_{22}	$\dots$	a_{2s}
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
c_s	a_{s1}	a_{s2}	$\dots$	a_{ss}
	b_1	b_2	$\dots$	b_s

For an s-stage scheme with time step κ_n and given w_n, solve

$$k_i(x) = \kappa_n G(x, t_n + c_i\kappa_n, w_n(x) + \sum_{j=1}^{s} a_{ij}k_j(x)), \quad i = 1, \dots, s$$

$$w(x)_{n+1} = w(x)_n + \sum_{i=1}^{s} b_i k_i(x).$$

The three types of solve have different requirements in the FEniCS context

Implicit non-linear solve ($a_{ij} \neq 0$ for all $j \geqslant i$)

$$k_i(x) - \kappa_n G(x, t_n + c_i\kappa_n, w_n(x) + \sum_{j=1}^{s} a_{ij}k_j(x)) = 0$$

Explicit via function evaluation ($a_{ij} = 0$ for all $j \geqslant i$)

$$k_i(x) = \kappa_n G(x, t_n + c_i\kappa_n, w_n(x) + \sum_{j=1}^{i-1} a_{ij}k_j(x))$$

Explicit via assignment

$$w(x)_{n+1} = w(x)_n + \sum_{i=1}^{s} b_i k_i(x).$$

[→ dolfin/multistage/*, site-packages/dolfin/multistage/*, demo/undocumented/multi-stage-solver/*]

Implementation of collocation methods motivated introducing the point measure

Definition

Let $\mathcal{X}$ be a collection of points associated with the domain Ω. We define the *point measure* $\mathrm{d}P$ relative to $\mathcal{X}$ by

$$\int_\Omega I \,\mathrm{d}P = \sum_{x \in \mathcal{X}} I(x) = \sum_{x \in \mathcal{X}} \int_\Omega I \delta_x \,\mathrm{d}x.$$

Example

```
V = FiniteElement("CG", tetrahedron, 1)
v = TestFunction(V)
f = Coefficient(V)
L = f*v*dP
```

[ufl/measure.py, ffc/quadrature/*]

FFC generated code for `f*v*dP`

```
// Array of quadrature weights.
static const double W1 = 1.0;
// Quadrature points on the UFC reference element: ()

// Values of basis functions at quadrature points.
// Reset values in the element tensor.
for (unsigned int r = 0; r < 3; r++)
{
  A[r] = 0.0;
} // end loop over 'r'
// Number of operations to compute geometry constants: 3.
double G[3];
G[0] = W1*w[0][0];
G[1] = W1*w[0][1];
G[2] = W1*w[0][2];

// Compute element tensor using UFL quadrature representation
// Optimisations: ('eliminate zeros', True), ('ignore ones', True), ('ignore zero tables'
switch (vertex)
{
case 0:
  {
    // Total number of operations to compute element tensor (from this point): 1

  // Loop quadrature points for integral.
  // Number of operations to compute element tensor for following IP loop = 1
  for (unsigned int ip = 0; ip < 1; ip++)
  {

    // Number of operations for primary indices: 1
    // Number of operations to compute entry: 1
    A[0] += G[0];
  } // end loop over 'ip'
```

For vector-valued Lagrange elements, the point measure allows for specifying and solving ODEs as variational forms.

Consider the system: find $u \in V$ such that

$$\int_\Omega I_a(u, v)\,\mathrm{d}P = \int_\Omega I_L(v)\,\mathrm{d}P$$

for all $v \in V$.

Let $\mathcal{X}$ be the collection of vertices. Let $V = \mathcal{M}_1^N$.

For each $x_k \in \mathcal{X}$, find $\{u_j\}_{J_k}$ such that

$$\sum_{j \in J_k} I_a(\phi_j, \phi_i)(x_k) u_j = I_L(\phi_i)(x_k)$$

for $i \in J_k$ where J_k is the index set of basis functions that are non-zero at x_k, $|J_k| = N$.

The point measures can be used to define multi-stage schemes for solving collections of ODEs

Example case: Explicit via function evaluation

For each vertex x_k, evaluate

$$k_i(x_k) = \kappa_n G(x_k, t_n + c_i\kappa_n, w_n(x_k) + \sum_{j=1}^{i-1} a_{ij}k_j(x_k))$$

Equivalent FEniCS code

```
# Assume that G is an UFL Expr.
V = VectorElement("CG", T, 1, N)
v = TestFunction(V)
kappa = Constant(T)
rhs = kappa*inner(G, v)*dP
```

Outline of the PointIntegralSolver algorithm

```
def step(G, k_i):

  for x_k in vertices(mesh):
      # Identify one cell and local vertex number
      (cell, i) = cell_and_local_vertex(x_k)

      # Restrict any coefficients in G to this cell
      G.coefficients.restrict(w, cell)

      # Evaluate right hand side
      G.integrals[0].tabulate_tensor(b, w, cell, i)

      # Extract subset of active local dofs
      J_k = find_active_dofs(i)

      # Reduce size of b
      b = b[J_k]

      # Compute the corresponding global dofs
      dofs = ki.tabulate_dofs(cell)[J_k]

      # Update ki
      ki.vector().add(b, dofs)
```

The block structure of a forward multistage solution step

For simplicity of presentation, consider the case where

$$G(\cdot, \cdot, w) = Cw$$

Forward structure (s = 2)

$$\begin{pmatrix} I & 0 & 0 & 0 \\ -\kappa_n C(\cdot) & I - \kappa_n a_{11} C(\cdot) & -\kappa_n a_{12} C(\cdot) & 0 \\ -\kappa_n C(\cdot) & -\kappa_n a_{21} C(\cdot) & I - \kappa_n a_{22} C(\cdot) & 0 \\ -I & -b_1 & -b_2 & I \end{pmatrix} \begin{pmatrix} w_n \\ k_1 \\ k_2 \\ w_{n+1} \end{pmatrix} = \begin{pmatrix} w_0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

The block structure of an adjoint multistage solution step

Adjoint structure

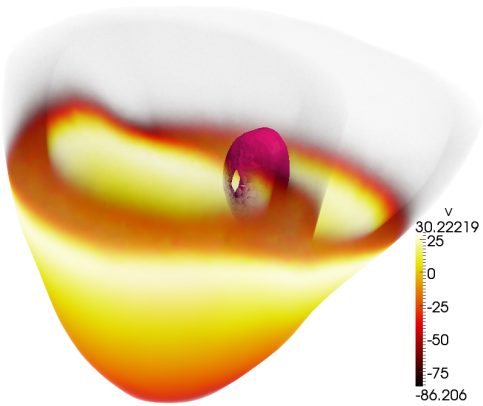
$$\begin{pmatrix} I & -\kappa_n C^*(\cdot) & -\kappa_n C^*(\cdot) & -I \\ 0 & (I - \kappa_n a_{11} C(\cdot))^* & -\kappa_n a_{21} C^*(\cdot) & -b_1 \\ 0 & -\kappa_n a_{12} C^*(\cdot) & (I - \kappa_n a_{22} C(\cdot))^* & -b_2 \\ 0 & 0 & 0 & I \end{pmatrix} \begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} = \frac{\partial J}{\partial w}$$

[dolfin/site-packages/multistage, dolfin-adjoint/dolfin_adjoint/pointintegralsolver.py]

Cardiac wave propagation with abnormal tissue conductivities as a basic example

```
# Set-up simulation scenario
cell = FitzhughNagumo()
heart = CardiacModel(mesh, time, M_i, M_e, cell, I_s)
solver = SplittingSolver(heart)

# Solve as you go along
solutions = solver.solve((0, T), k_n)
for (timestep, fields) in solutions:
    # Do something with solution fields
```



$$v_t - \operatorname{div}(M_i \nabla v + M_i \nabla u_e) - I_{\text{ion}}(v, s) = I_{\text{s}}$$

$$\operatorname{div}(M_i \nabla v + (M_i + M_e) \nabla u_e) = 0$$

$$s_t = b(v - v_{\text{rest}} - c\, s)$$

$$M_{\text{i|e}} = A G_{\text{i|e}} A^T, \quad G_{\text{i|e}} = \operatorname{diag}(g_{\text{i|el}}, g_{\text{i|et}}, g_{\text{i|et}})$$

[Thanks to Sjur Gjerald and Johan Hake for patient-specific mesh (generated from ultrasound), fibers and sheets]

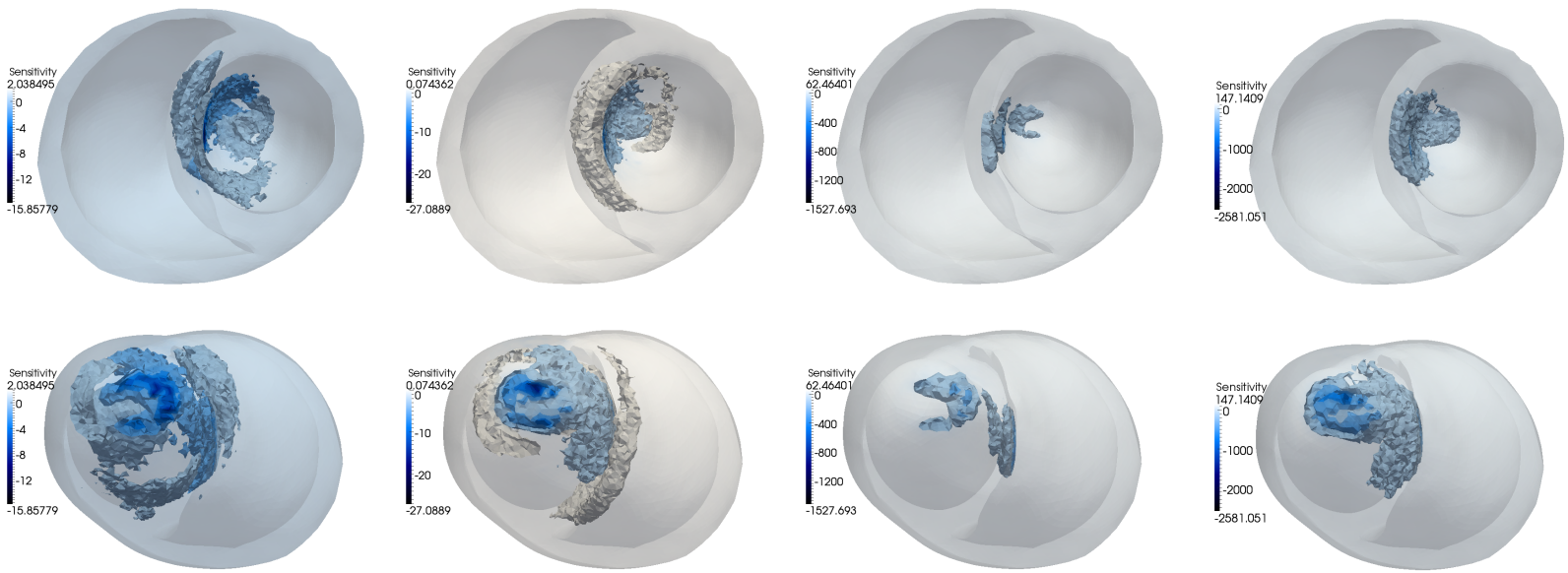
http://youtu.be/r6tLfMARKA0

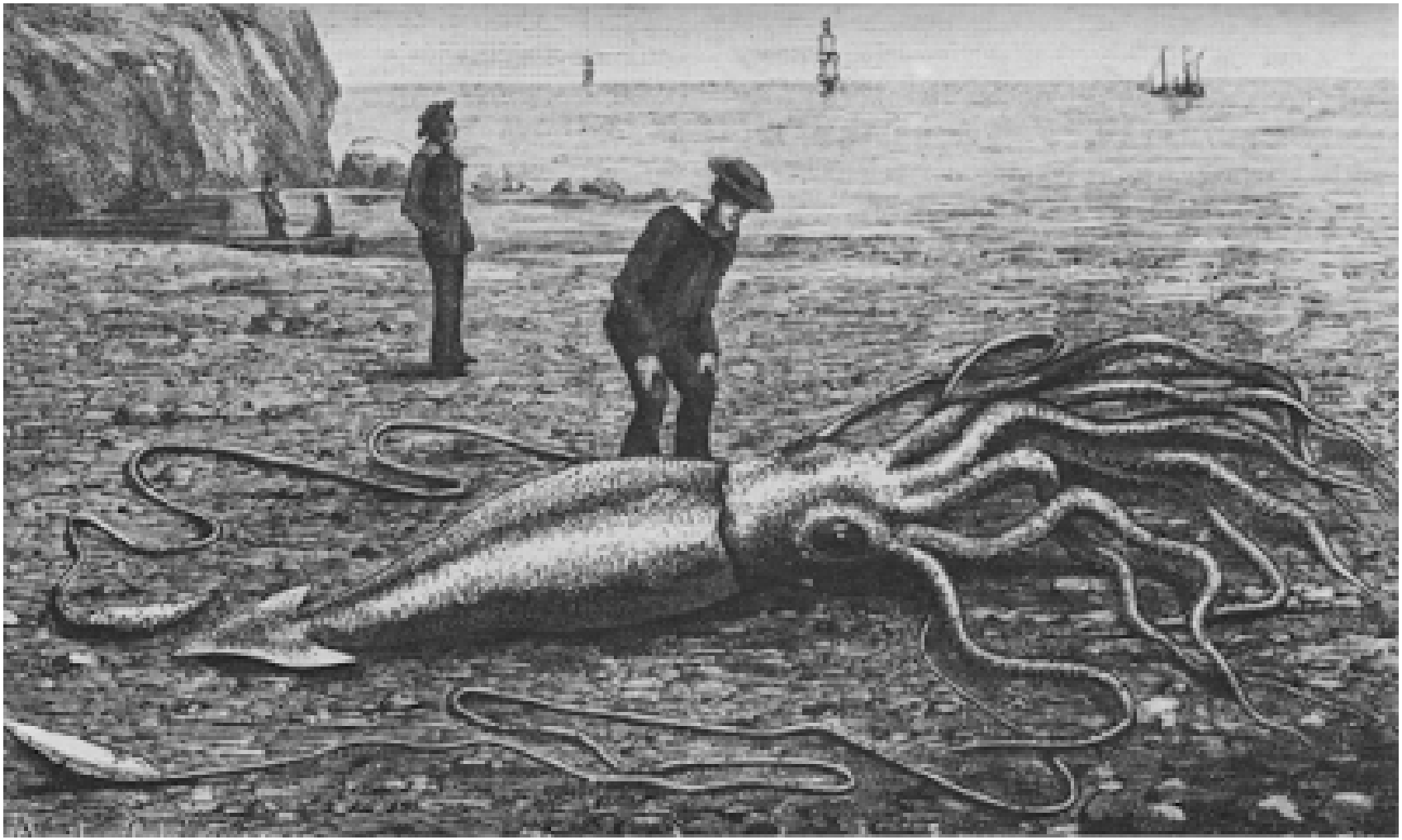
What is the sensitivity of the abnormal wave propagation to the local tissue conductivities?

The wave propagation abnormality at a given time T:

$$J(v, s, u) = \|v(T) - v_{\text{obs}}(T)\|^2, \qquad \frac{\partial J}{\partial g_{\text{e|i|l|t}}} = ?$$

```
v_obs = Function(V, "healthy_obs_200.xml.gz")
J = Functional(inner(v - v_obs, v - v_obs)*dx*dt[T])
dJdg_s = compute_gradient(J, gs)
```





[Wikimedia Commons]