Introduction to Cardioid

Rob Blake

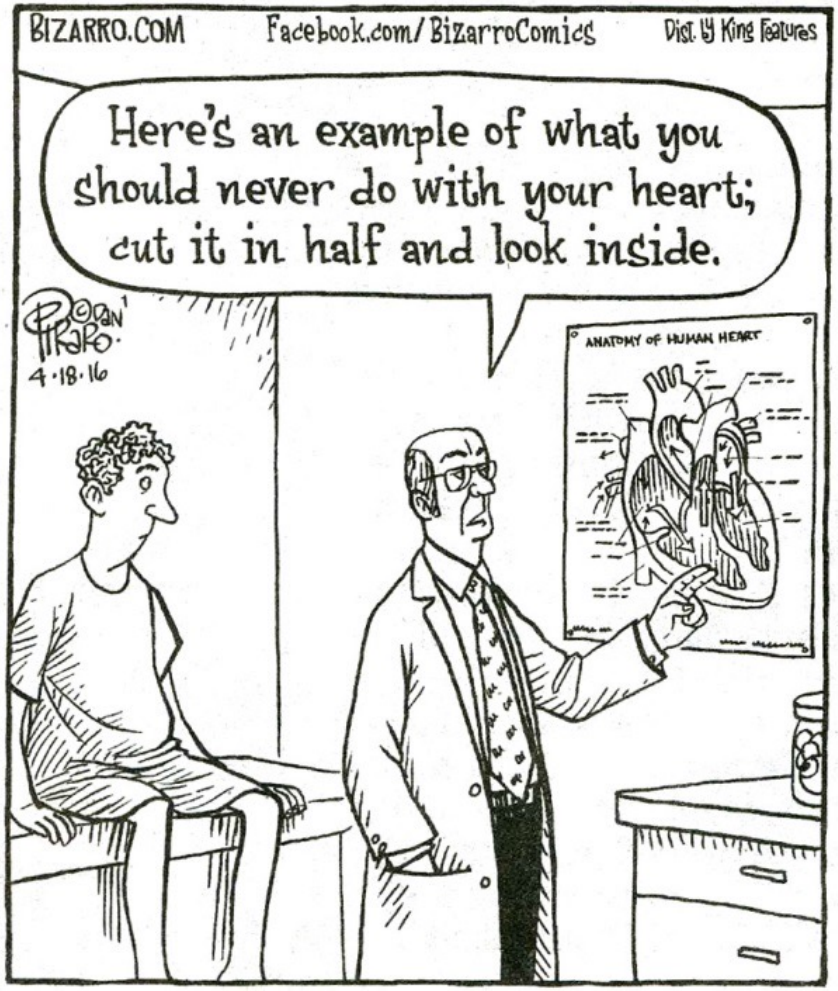
July 26, 2021

LLNL-PRES-824898

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

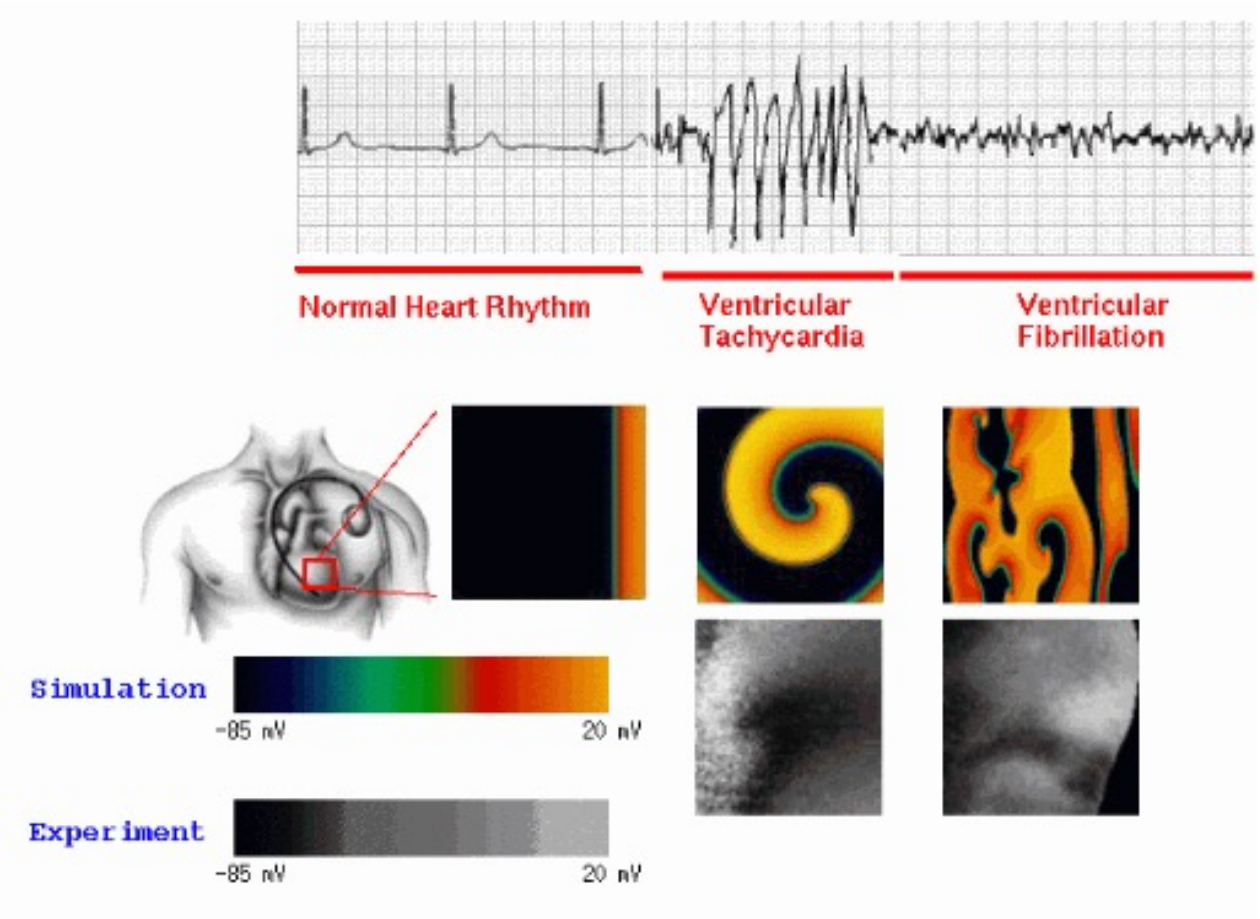
Lawrence Livermore National Laboratory

Why do cardiac modeling?

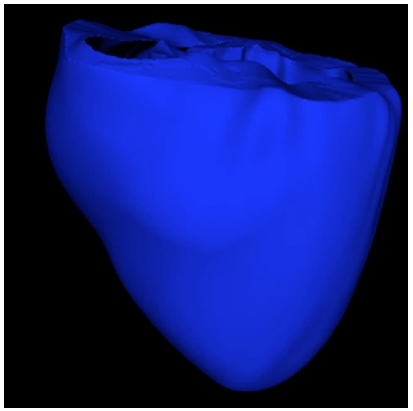


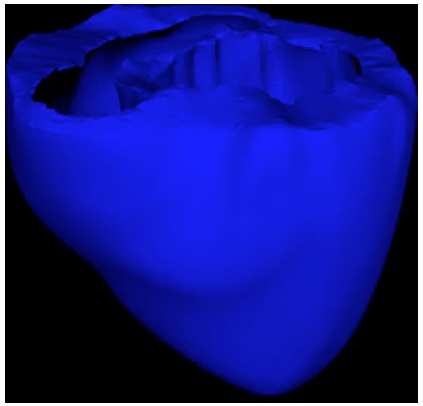
- The heart is difficult to observe
- Interventions are life threatening
- Simulate before you intervene
  - Basic Science
  - Device Design
  - Drug Development
  - Surgery planning
  - Risk stratification

The heart is an electrical/mechanical organ



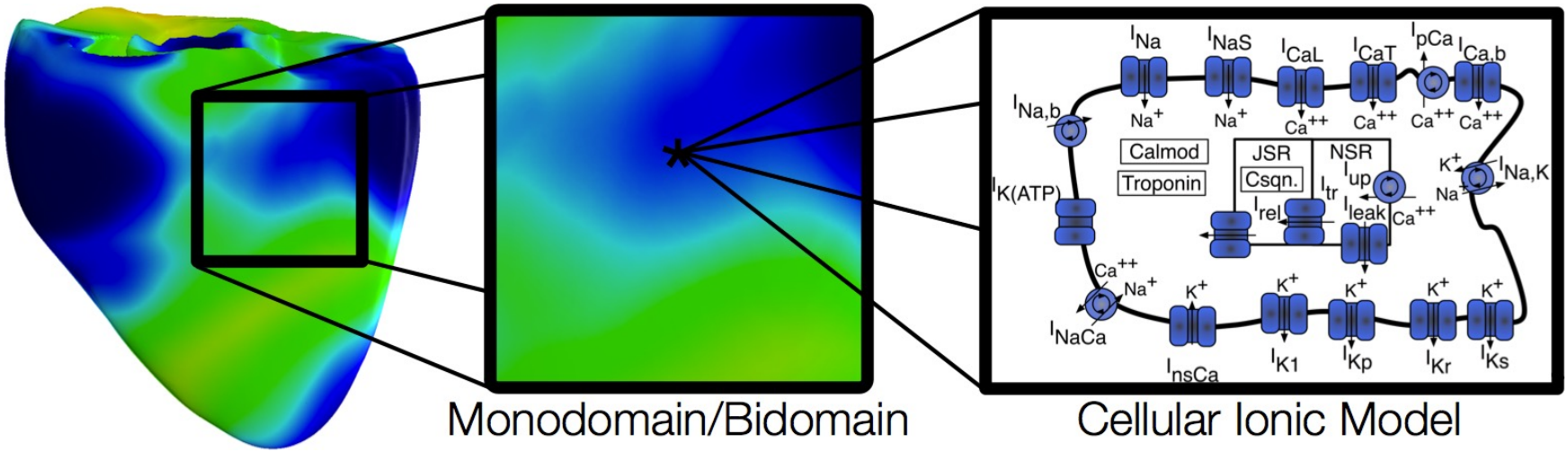
Flavio Fenton, http://thevirtualheart.org





Gurev, Trayanova

The heart requires multiscale modeling

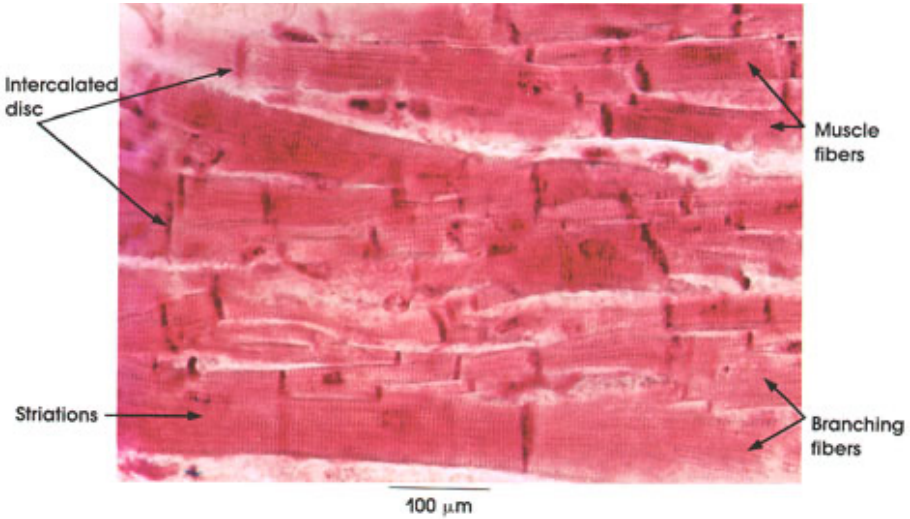


- Space: [10cm – 100um]
- Time: [10s – 10us]
- ~400 million cells
- ~8 billion dofs

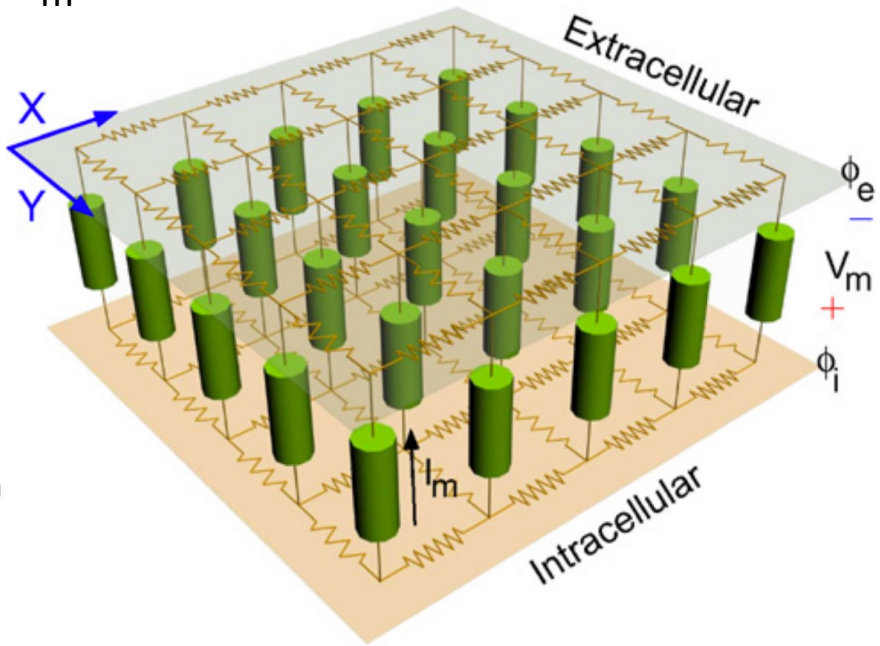
Bidomain model

$$\nabla \cdot \sigma_e \nabla \Phi_e = -I_m$$
$$\nabla \cdot \sigma_i \nabla \Phi_i = I_m$$

- Φ_e - Extracellular potential
- Φ_i - Intracellular potential
- I_m - Membrane current



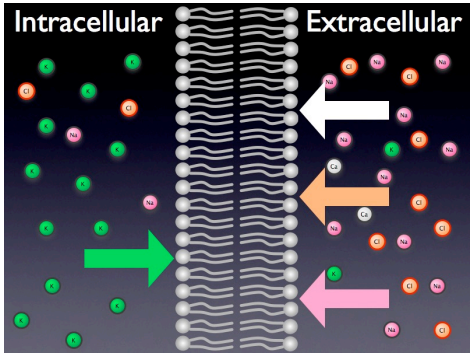
R. Bergman, http://www.anatomyatlases.org/

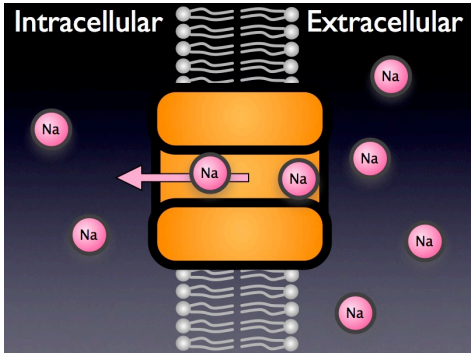


G. Plank, http://carp.medunigraz.at/

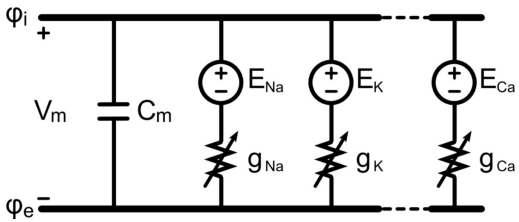
Bidomain model

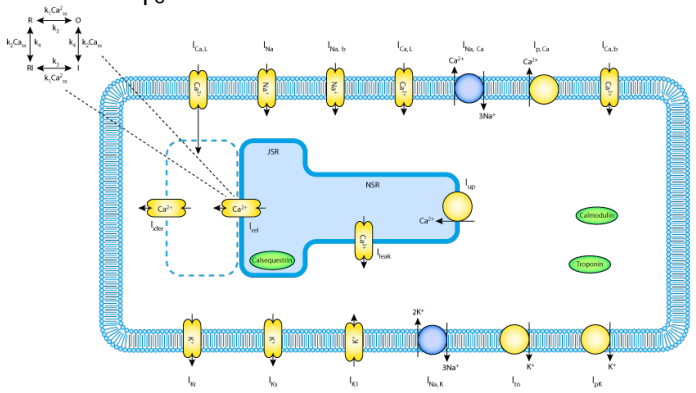
$$
\begin{aligned}
I_m &= \beta_m \left[C_m V_m' - I_{ion} \right] \\
V_m &= \Phi_i - \Phi_e \\
I_{ion} &= I_{ion}(V_m, \overrightarrow{s}) \\
\overrightarrow{s}' &= g(V_m, \overrightarrow{s})
\end{aligned}
$$



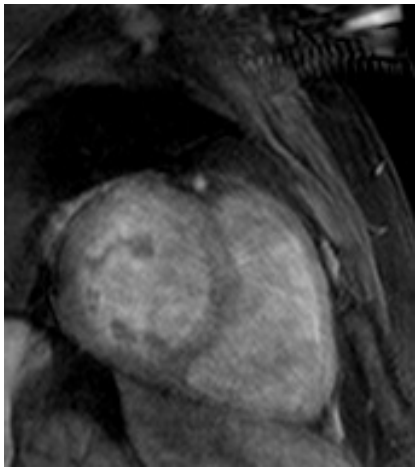


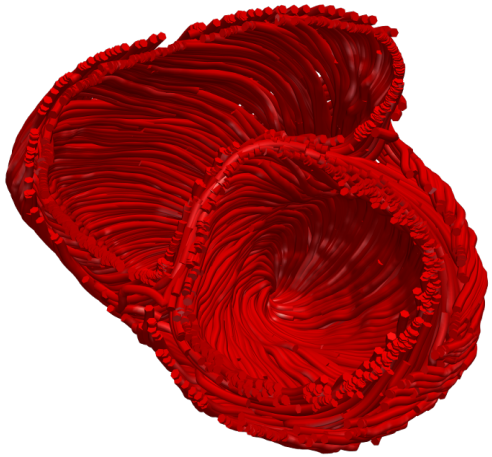
- C_m - Membrane capacitance
- V_m - Transmembrane voltage
- I_{ion} - Ionic current
- s - Ionic state

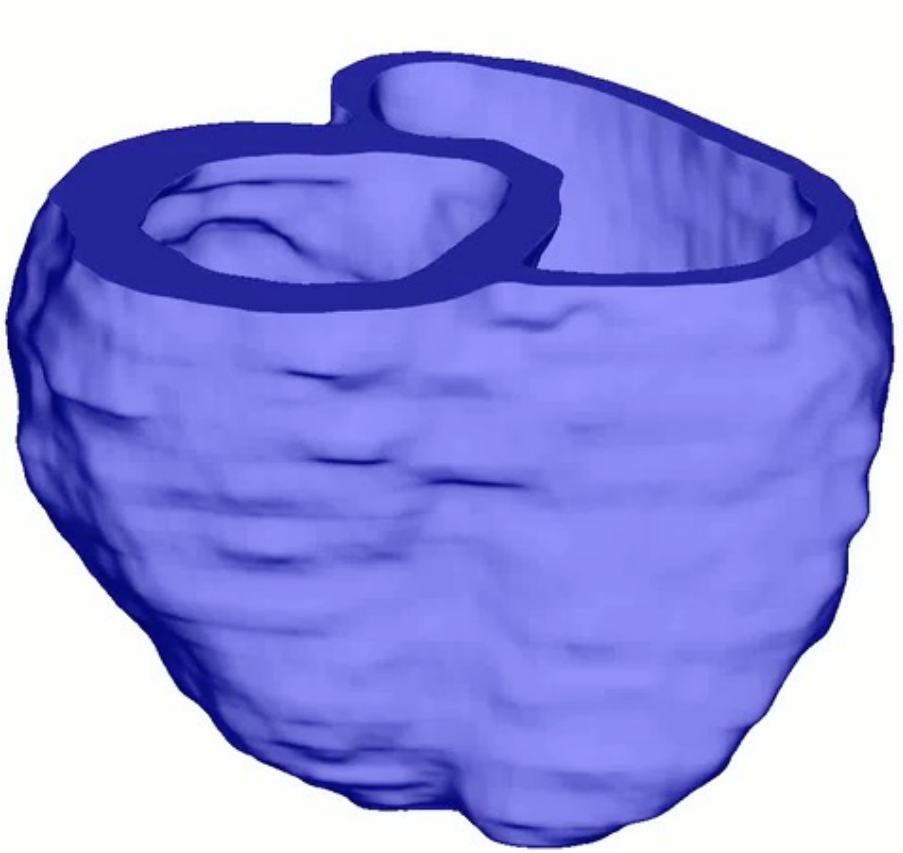




How to simulate your own heart







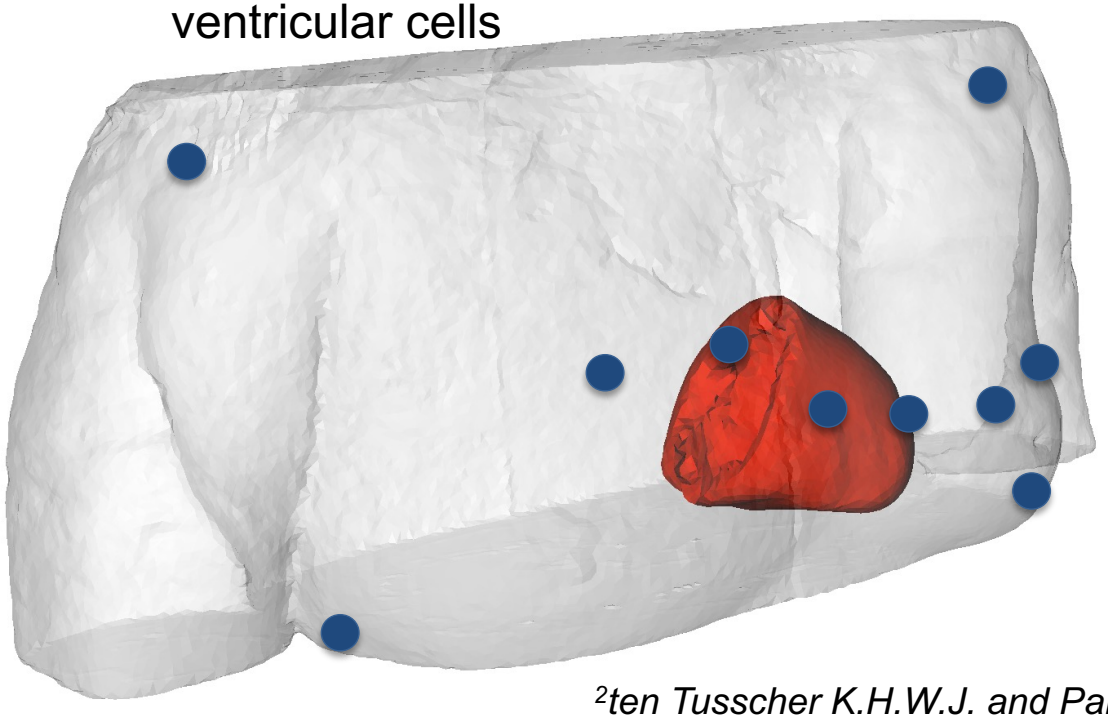
- Clinical image -> simulations in ~2 hours

Modeling surface potentials: ECG

Monodomain Model of Heart[1]

$$C_m \frac{\partial V_m}{\partial t} = \frac{1}{\beta} \nabla \cdot (D \nabla (V_m)) - I_{\text{ion}} + I_{\text{stim}}$$

I_{ion} computed from ten Tusscher *et al*., 2006[2], model of action potential in ventricular cells



Calculation of Torso Potentials[3]

$$\Phi_e = \frac{1}{4\pi\sigma_b} \int_\Omega \frac{\beta I_m}{\|\mathbf{r}\|} \, d\Omega$$

σ_b includes 11 different types of material torso including muscle, bone, and fat

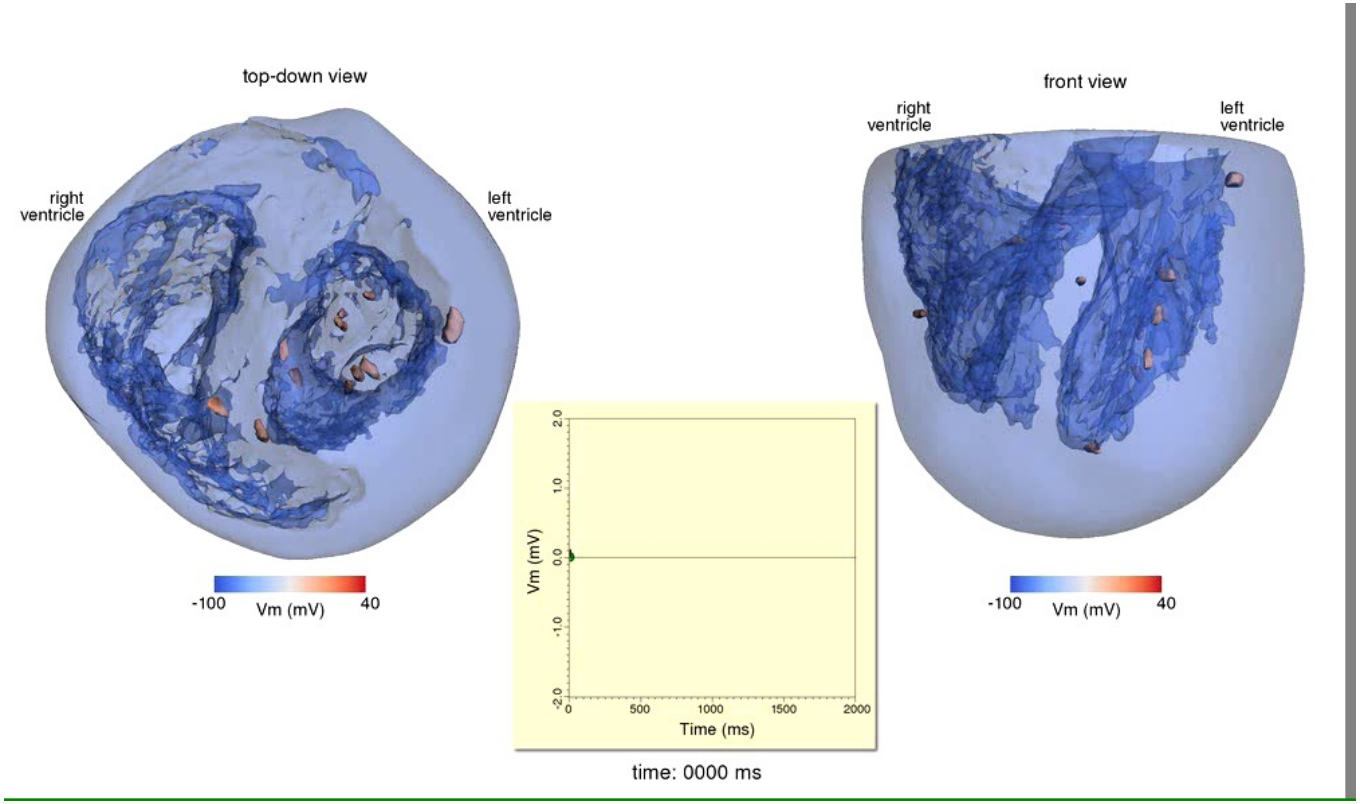
[1]*Mirin, A.A. et al (2012)* 10.1109/SC.2012.108

[2]*ten Tusscher K.H.W.J. and Panfilov A. V. (2006) Am J Physiol, 291, 1088-1100*

[3]*Bishop M.J. and Plank G. (2011) IEEE T Bio-Med Eng, 58, 2297-2307*

Simulation of drug-induced ECG abnormality

e.g. sotalol, a β-adrenergic receptor and K^+ channel blocker



Richards *et al.* Computer Methods in Biomechanics and Biomedical Engineering (2013)

Technical details

Cardioid was a Gordon Bell finalist

- Each human heart has
  - 400 million elements
  - 8 billion DOF
- 1 second of simulation requires
  - 50k-100k timesteps
  - Each timestep requires ~20 billion math function evaluations
- Cardioid scales to all of Sequoia
  - 60s of simulation in 67s wall time!!

How Cardioid strong scaled to all of Sequoia

- Only ~200 elements per thread
- Replace all math functions with rational polynomials
- Hard coded vector intrinsics in every critical loop
- Bare metal coding to the machine
  - SPI usage
  - Thread barriers based on L2 cache access
- My job when I got to the lab: Port all of this to Sierra/GPUs
  - ...and make sure it's fast!

Relevant performance characteristics

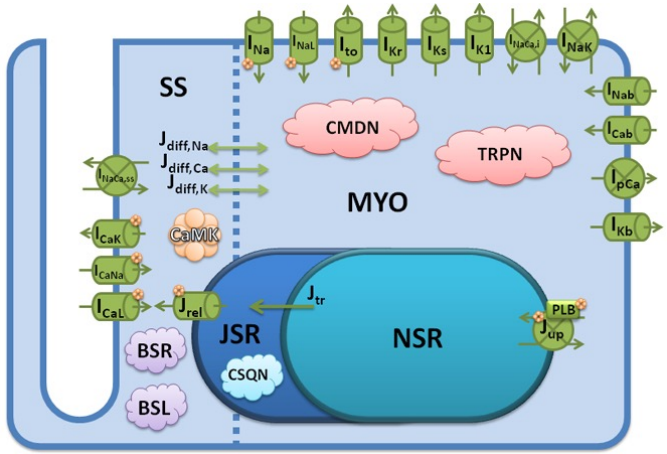
- Cardioid uses a finite volume discretization on a regular grid.
- Two main modes during computation
  - Spatial diffusion – 50%
    - Stencil computation
    - Optimized for CPUs or GPUs
    - Requires MPI communication
    - Parallelism limited by network latency
    - GPU performance limited by memory bandwidth
  - Reaction ODEs – 50%
    - ODE computation
    - Embarrasingly parallel
    - GPU/CPU performance limited by computation FLOPS
- Different drivers implement different computational loops for CPU, GPU

Stencil computation

- Network communication interleaved with computation
- 19 point stencil
- Heavily optimized
  - CPU: use openmp, vector intrinsics for faster computation
  - GPU: use shared memory blocks, polyhedral loop optimization for fast code
    - Currently fastest version uses Volta-specific instructions

Reaction ODE models

- Always embarrassingly parallel
- Compute bound
  - 10-60 differential variables
  - 100-2000 equations
  - 50-400 math function invocations
- Under constant refinement
  - Drug effects
  - Subcellular processes
- Extremely hard to validate



Melodee goal: Port ODEs to GPUs

- Melodee is a language for ODEs
- Scientists use matlab-like syntax
- Melodee uses JIT+NVRTC compilation to optimize code to architecture
  - Vectorization
  - Rational polynomial replacement
  - Automatic differentiation

```
subsystem i_Ks_current {
  shared Xs {1};
  subsystem Xs_gate {
    provides diffvar Xs;
    alpha_xs = 1400/sqrt((1+exp((5-V)/6)));
    beta_xs = 1/(1+exp((V-35)/15));
    xs_inf = 1/(1+exp((-5-V)/14));
    tau_xs = (1*alpha_xs*beta_xs+80);
    Xs.init = 0.0087;
    Xs.diff = (xs_inf-Xs)/tau_xs;
  }
  provides param g_Ks = 0.392;
  P_kna = 0.03;
  E_Ks =
R*T/F*log((K_o+P_kna*Na_o)/(K_i+P_kna*Na_i));
  i_Ks = g_Ks*Xs^2*(V-E_Ks);
  provides accum i_Kitot += i_Ks;
}
```

Reaction model optimizations

- **Rational polynomials** – replace expensive function evaluations with faster functions
- **Kernel fission vs fusion** – separate the ODE into multiple functions or one function
- **Replace exp/log** – variants based on floating point binary representation
- **Intrinsics** –use the compiler to vectorize or do it ourselves
- **SoA vs AoS** – How do we lay out our data structures?

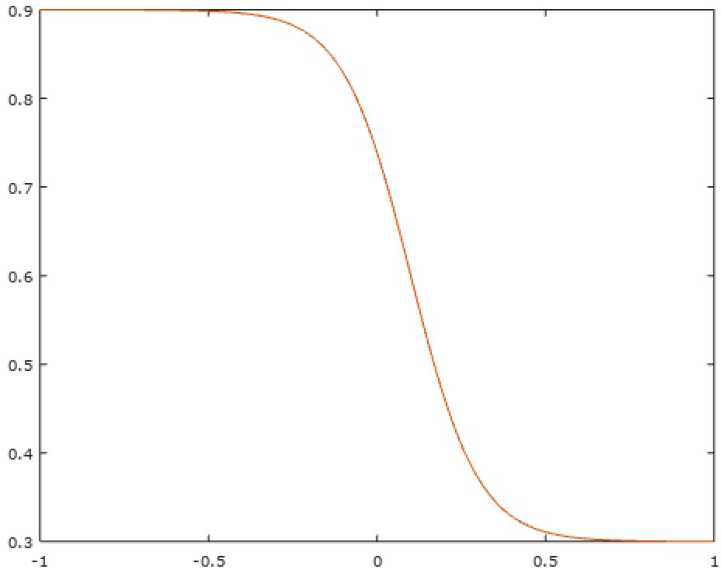
Optimization	BGQ	P100	KNL
Rational polynomials	yes	yes	no
Kernel fission vs fusion	fission	fusion	fusion
Replace exp/log	yes	no	no
Explicit vectorization with intrinsics	yes	no	yes
SoA vs AoS	SoA	SoA	AoS

Rational polynomials can replace expensive functions

```
double Afcaf = 0.3+0.6/(1.0+exp((v-10.0)/10.0));
```

becomes

```
double Afcaf;
{
  double numerCoeff[]={-9.52275328672 ... };
  double denomCoeff[]={2.18001528726e ... };
  double numerator=_numerCoeff[0];
  for (int jj=1; jj<8; jj++)
    _numerator = numerCoeff[jj] + v*numerator;
  double _denominator=denomCoeff[0];
  for (int jj=1; jj<6; jj++)
    _denominator = _denomCoeff[jj] + v*denominator;
  Afcaf = numerator/denominator;
}
```



GPU: Embedding the coefficients is much faster

```
poly(double *in,
     int np, double *p,
     double *out)
{
  int ii = blockIdx.x*blockDim.x + threadIdx.x;
```

np=60
in[1e6]
out[1e6]

Memcpy:
30.940us

```
out[ii]=in[ii];
```

Naïve:
202.15us

```
double z = 0;
for (int k=np-1; k>=0; k--)
   z = p[k] + z*in[ii];
out[ii] = z;
```

```
/* 0x2b8 */
{ IADD32I R3, R3, -0x1;
  LDG.E.64 R10, [R6]; }
  ISETP.GT.AND P0, PT, R3, RZ, PT;
  IADD32I R6.CC, R6, -0x8;
  IADD32I.X R7, R7, -0x1;
  DFMA R4, R8, R4, R10;
 @P0 BRA 0x2b8;
```

Embedded:
40.760us

```
double *my_p[] = {...};
double z = 0;
for (int k=np-1; k>=0; k--)
   z = my_p[k] + z*in[ii];
out[ii] = z;
```

```
DFMA R8,R2,R8,c[0x2][0x68];
DFMA R8,R2.reuse,R8,c[0x2][0x60];
DFMA R8,R2.reuse,R8,c[0x2][0x58];
...
```

Unrolling with a duff's device

Unrolled

```
__constant__ double c_p[];
...
double z = 0;
switch (np) {
   case 8: c_p[7] + z*in[ii];
   case 7: c_p[6] + z*in[ii];
   case 6: c_p[5] + z*in[ii];
   case 5: c_p[4] + z*in[ii];
   case 4: c_p[3] + z*in[ii];
   case 3: c_p[2] + z*in[ii];
   case 2: c_p[1] + z*in[ii];
   case 1: c_p[0] + z*in[ii];
   default:
}
out[ii] = z;
```

- On CPUs, this is
  - just as fast as embedding
  - uses run-time coefficients
- On GPUs
  - c_p must be constant memory
  - c_p must be a constexpr
  - ptxas doesn't emit indirect branches
    - Still have to pay for performance
    - Memcpy: 30us
    - Embedded: 40us
    - Unrolled: 46us

Embedded coefficients are faster and simpler on GPUs

Intrinsics

- BGQ, Haswell, KNL
  - Compilers will NOT auto-vectorize this code
  - Must generate vector intrinsics specific to platform
- GPU
  - No intrinsics necessary

Data layout

Structure of Arrays

```
struct state {
  double x[n];
  double y[n];
}
```

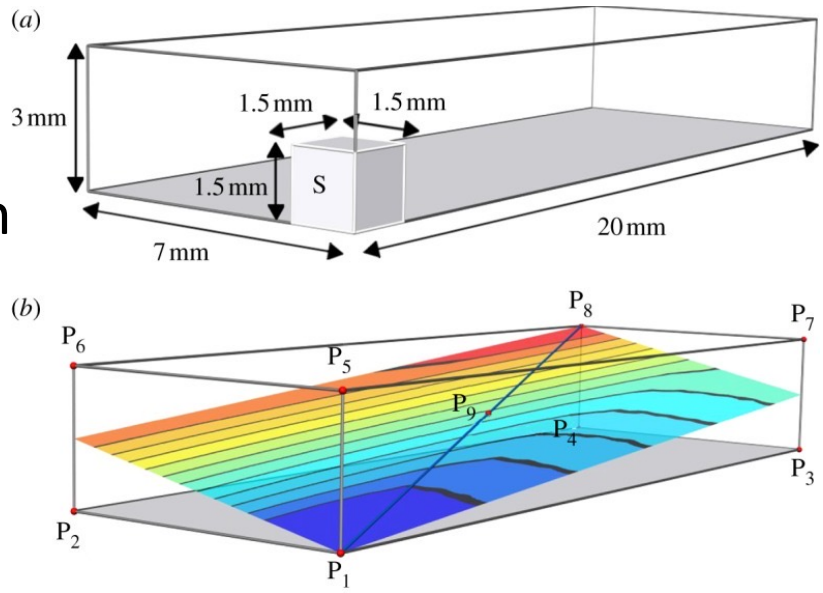
Array of Structures of Vectors

```
struct stateVec {
  double x[vwidth];
  double y[vwidth];
}
stateVec state[n/vwidth];
```

- Haswell, KNL: AoSoV is faster
- BGQ, GPU: SoA is faster

Competition: Niederer benchmark

- Benchmark for cardiac simulations
- 3mm x 7mm x 20mm tissue slab
- Defined stimulus, fiber orientation
- Problem can be scaled to accommodate large problem sizes



Goal: Run a convergence study

- Real medical simulations depend on conduction velocity
  - == How fast the wavefront propagates
- Investigate what resolution is required to get accurate conduction velocity simulations

Live Demo

Lawrence Livermore
National Laboratory