Bernus membrane model

Version 1.62 – November 22, 2008

1. Introduction. This document is part of the large-scale heart modeling project at the Institut de Génie Biomédical, Université de Montréal. It implements a membrane model for human ventricular cells developed by Bernus et al. [bernus02]. The functions defined here are intended to be used by `propag`, our large-scale heart simulation program, as well as by other programs, such as the Matlab interface defined in `mmex.web`. Therefore both generality and efficiency must be taken care of.

This is a second implementation of Bernus' model for `propag`. It is intended as specification and example for a family of membrane models. A single C source file, `bernus.c`, is created; it implements several functions. In the order in which they are typically applied, these are the following.

- *bernus_info* receives the parameters and returns the information needed by a main program to use this membrane model, such as the number of status variables for which memory must be allocated. There is no need to call it more than once.
- *bernus_ccode* returns the cell type code corresponding to the given cell type name. In `propag`, this is used during initialization, to translate the cell type names that the user types in the configuration file into the corresponding codes.
- *bernus_init* is used during initialization of the program; it computes several constants and tabulated functions that are used by *bernus_int_current* to speed up the computations. It must be repeated when the time step changes.
- *bernus_infinite* sets the status variables to the infinite values corresponding to the given membrane potential
- *bernus_int_current* integrates the status variables, computes the currents, and returns the total membrane ionic current
- *bernus_get_currents* returns all ionic currents individually for the given cell. This is not always needed in an application; usually only the sum of the currents is used.

2. Overview of the Bernus membrane model. The purpose of this model is to compute the total ionic current through the cell membrane at a given time. This current depends both on the actual membrane potential and on the history of the membrane.

The history-dependence of the membrane current is implemented using so-called "gating variables," a concept introduced by Hodgin and Huxley [hh52], and subsequently used by many others. Each gating variable y is a dimensionless quantity whose value is governed by a differential equation

$$\frac{d\,y(t)}{dt} = \frac{y_\infty - y(t)}{\tau_y}$$

with solution

$$y(t+dt) = y_\infty - (y_\infty - y(t))\,\exp(-dt/\tau_y)$$

where y_∞ is the steady-state value of y and τ_y is a time constant. The parameters y_∞ and τ_y characterize a "gate," while y characterizes the state of this gate. If they were constants, the solution would be exact for any dt. However, y_∞ and τ_y depend on the membrane potential so that they are implicitly time-dependent. This implies that the integration of y should be done with the same small dt as the other computations in the model, which yield V_m.

The governing equation can also be written in the form of a rate equation:

$$\frac{d\,y(t)}{dt} = \alpha_y(1-y) - \beta_y y$$

where α_y is the likelihood that a closed y-gate opens, and β_y the likelihood that an open y-gate closes. The parameters are related as follows:

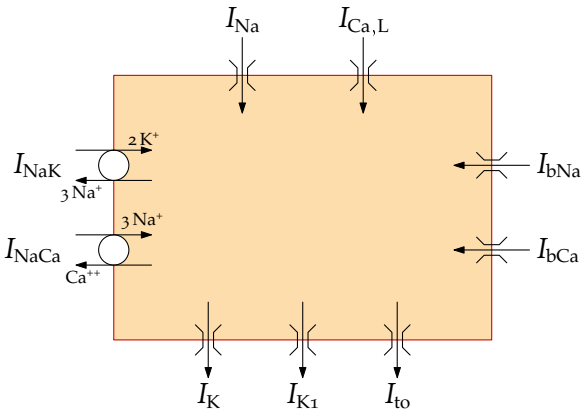
$$y_\infty = \alpha_y/(\alpha_y + \beta_y)$$

$$\tau_y = 1/(\alpha_y + \beta_y)$$

α_y and β_y depend on the membrane potential, and in some cases on the concentration of specific ions. The Bernus model has 5 gating variables: m, v, f, to, and X, as well as three extra steady-state gating variables: d_∞, r_∞, $K1_\infty$. For three of these variables, the expressions given above for y_∞ and τ_y do not apply: the computation of to_∞ and τ_{to} implies extra parameters p and V_shift, and for the gating variables v and X there are no α and β variables at all. The gating variables m, f and to were copied from the P-B model; v and X were introduced by Bernus et al. [bernus02]. The presence of steady-state gating variables, i.e. gating variables $y = y_\infty$, is due to modifications to improve efficiency made by Bernus et al.

3. This model works with 9 ionic currents; the ionic membrane current is the sum of these. The computation of these currents involves the membrane potential V_m, the 5 gating variables, 3 steady-state gating variables, and 2 factors discussed above, as well as several ion-specific membrane conductances g_i and equilibrium potentials E_i, and four special factors: f_{Ca}, f_{NaK}, f'_{NaK}, and f_{NaCa}. The conductances and equilibrium potentials are discussed below. The ionic currents are:

1. Fast Na^+ current	$I_{Na} = g_{Na}\, m^3\, v^2\, (V_m - E_{Na})$	1 Na^+ in
2. Slow Ca^{2+} current	$I_{Ca} = g_{Ca}\, d_\infty\, f\, f_{Ca}\, (V_m - E_{Ca})$	1 Na^+ in
3. Transient outward current	$I_{to} = g_{to}\, r_\infty\, to\, (V_m - E_{to})$	1 K^+ out
4. Delayed rectifier K^+ current	$I_K = g_K\, X^2\, (V_m - E_K)$	1 K^+ out
5. Inward rectifier K^+ current	$I_{K1} = g_{K1}\, K1_\infty\, (V_m - E_K)$	1 K^+ out
6. Ca^{2+} background current	$I_{Ca,b} = g_{Ca,b}\, (V_m - E_{Ca})$	1 Ca^+
7. Na^+ background current	$I_{Na,b} = g_{Na,b}\, (V_m - E_{Na})$	1 Na^+
8. Na^+–K^+ pump	$I_{NaK} = g_{NaK}\, f_{NaK}\, f'_{NaK}$	3 Na^+ out; 2 K^+ in
9. Na^+–Ca^{2+} exchanger	$I_{NaCa} = g_{NaCa}\, f_{NaCa}$	3 Na^+ in; 1 Ca^{2+} out



Only the first 4 currents are explicitly time-dependent; the others have no gating variables, and their dynamic behaviour depends only on the membrane potential. The ion transports that are involved are are also mentioned, although these play no role in this model, which has static ion concentrations.

4. The conductances g_i have fixed values. The table below lists these values for the Bernus model [bernus02] as well as for its ancestor, the Priebe-Beuckelmann model [priebe98]. Note that not all conductances are defined for both models, since they use slightly different subsets. Conductances are given in nS/pF.

	P-B	Bernus	variable
g_{Na}	16.0	16.0	*bernus.g_Na*
g_{Ca}	0.064	0.064	*bernus.g_Ca*
g_{to}	0.3	0.4	*bernus.g_go*[ ]
g_K		0.019	*bernus.g_K*[ ]
g_{Kr}	0.015		
g_{Ks}	0.02		
g_{K1}	2.5	3.9	*bernus.g_K1*
$g_{Na,b}$	0.001	0.001	*bernus.g_Nab*
$g_{Ca,b}$	0.00085	0.00085	*bernus.g_Cab*
g_{NaK}	1.3	1.3	*bernus.g_NaK*
g_{NaCa}	1000.0	1000.0	*bernus.g_NaCa*

Two conductances were taken different for M-cells and endocardial cells: $g_{to} = 0.35$ and 0.13; $g_K = 0.013$ and 0.019, respectively. The values in the table are for epicardial cells, wherever differences apply. *bernus.g_K* and *bernus.g_to* are defined for each cell type separately.

5. The four equilibrium potentials depend on the ratio of the intracellular and extracellular concentrations of several ions. These concentrations are fixed in the Bernus model, while internal $[Ca^{2+}]$, $[Na^+]$, and $[K^+]$ were variable in the P-B model. According to Bernus and coworkers, variable $[K^+]$ and $[Na^+]$ suffer from long-term drift; they don't mention problems with $[Ca^{2+}]$, but keep it fixed anyway.

$$E_{\mathrm{Na}} = \frac{RT}{F} \ln\left(\frac{[\mathrm{Na}^+]_{\mathrm{e}}}{[\mathrm{Na}^+]_{\mathrm{i}}}\right)$$

$$E_{\mathrm{Ca}} = \frac{RT}{2F} \ln\left(\frac{[\mathrm{Ca}^{2+}]_{\mathrm{e}}}{[\mathrm{Ca}^{2+}]_{\mathrm{i}}}\right)$$

$$E_{\mathrm{to}} = \frac{RT}{F} \ln\left(\frac{0.043[\mathrm{Na}^+]_{\mathrm{e}} + [\mathrm{K}^+]_{\mathrm{e}}}{0.043[\mathrm{Na}^+]_{\mathrm{i}} + [\mathrm{K}^+]_{\mathrm{i}}}\right)$$

$$E_{\mathrm{K}} = \frac{RT}{F} \ln\left(\frac{[\mathrm{K}^+]_{\mathrm{e}}}{[\mathrm{K}^+]_{\mathrm{i}}}\right)$$

where R is the universal gas constant, $T = 37 + 273.15 = 310.15$ the absolute temperature, and F the Faraday constant.

6. The intracellular and extracellular ion concentrations (mM) for several models are given in the following table. In this implementation, the concentrations are allowed to be different for each cell type. This allows a rough implementation of concentration differences due to an abnormal current that is not equally present in all cell types. These concentrations are now program parameters, given in the variables *bpar→ci_Ca* etc. as indicated in the table; for their declaration and defalt values see the file `propag.prm`

	P-B	Bernus	variable
$[Ca^{2+}]_{\mathrm{i}}$	var.	0.0004	*bpar→ci_Ca*
$[Ca^{2+}]_{\mathrm{e}}$	2	4	*bpar→ce_Ca*
$[Na^+]_{\mathrm{i}}$	var.	10	*bpar→ci_Na*
$[Na^+]_{\mathrm{e}}$	138	138	*bpar→ce_Na*
$[K^+]_{\mathrm{i}}$	var.	140	*bpar→ci_K*
$[K^+]_{\mathrm{e}}$	4	4	*bpar→ce_K*

7. Thus we arrive at the following fixed equilibrium potentials. These too can be different for each cell type (endo, epi, ...).

```
⟨ calculate equilibrium potentials 7 ⟩ ≡
  {
    int t;
    printf("ce_Na = %f %f ...\n", bpar→ce_Na[0], bpar→ce_Na[1]);
    for (t = 0; t < BERNUS_NTYPES; t++) {
      E_Na[t] = RTonF * log(bpar→ce_Na[t]/bpar→ci_Na[t]);
      E_Ca[t] = 0.5 * RTonF * log(bpar→ce_Ca[t]/bpar→ci_Ca[t]);      /* make this dynamic later? */
      E_to[t] = RTonF * log((0.043 * bpar→ce_Na[t] + bpar→ce_K[t])/(0.043 * bpar→ci_Na[t] + bpar→ci_K[t]));
      E_Kk[t] = RTonF * log(bpar→ce_K[t]/bpar→ci_K[t]);
      printf("cell type %d: E_na=%.2f  E_Ca=%.2f  E_to=%.2f E_K=%.2f\n", t, E_Na[t], E_Ca[t],
          E_to[t], E_Kk[t]);
    }
  }
```

This code is cited in section 20.

This code is used in section 16.

8. The computation of the parameters α_y and β_y (where y stands for any of the gating variables) and the f factors is complicated and numerically intensive. Since they all depend only on the membrane potential $V_{\rm m}$ it is possible to tabulate them for several $V_{\rm m}$ values, before the loop, so that a lookup and a simple table interpolation suffice during the loop ($\sim$ 12 million iterations per time step). The formulas will be given together with the implementation in section ⟨ calculate the table 35 ⟩, below.

9. It should be noted that Bernus et al. used fixed concentrations for all ions, both internal and external. However, the values of these concentrations occur in the equation, and it is straightforward to re-introduce $[{\rm Ca}^{2+}]_{\rm i}$, for example.

10. Cell types. The model knows five cell types: endocardial, LV and RV M-cell, epicardial, and ischemic. The RV M-cell and ischemic types were not present in Bernus' original model. The following table lists the values used by Bernus et al. of all parameters that differ between cell types. These differences result in different values for τ_{to}, to_∞, τ_X, and X_∞, which affect the transient outward current $I_{\rm to}$ and the delayed rectifier current $I_{\rm K}$. The other currents are the same for all cell types.

	epicardial	M cell(LV)	endocardial
$1/p$	1.0	1.7	2.8
$V_{\rm shift}$ (mV)	0	-4	-12
$g_{\rm to}$ (nS/pF)	0.4	0.35	0.13
$g_{\rm K}$ (nS/pF)	0.019	0.013	0.019

Other parameters: $C_{\rm m} = 2.0\,\mu{\rm F/cm}^2$ ($Cm = 153.4$ in the implementation), $S = 0.2\,\mu{\rm m}^{-1}$

The table of membrane parameters as a function of membrane potential has to accomodate for the extreme values of *V_shift*.

```
⟨ variables that are shared by the functions 10 ⟩ ≡
#include "membranes/bernus.d"
  typedef double table_t;
  static double p_to[BERNUS_NTYPES] = {1.0, 1.7, 2.8, 2.8, 1.7};
#if READ_TABLES
  static double V_shift[BERNUS_NTYPES] = {0.0, 0.0, 0.0, 0.0, 0.0};
#define VSHIFT_MIN 0.0
#define VSHIFT_MAX 0.0
#else
  static double V_shift[BERNUS_NTYPES] = {0.0, −4.0, −12.0, −12.0, −4.0};
#define VSHIFT_MIN (−16.0)      /* λ + μ may be larger than 1 */
#define VSHIFT_MAX 0.0
#endif
```

See also sections 20, 28, 32, 33, and 36.

This code is used in section 13.

11. These codes are used to refer to the cell types. The definitions are shared with the PRM code, therefore we put them in a separate file.

```
⟨ define cell types 11 ⟩ ≡
#define BERNUS_EPIC 0     /* LV+RV epicardial */
#define BERNUS_MCEL 1     /* LV or LV+RV midmyocardial */
#define BERNUS_ENDO 2     /* LV+RV endocardial */
#define BERNUS_ISCH 3     /* LV+RV endocardial ischemic */
#define BERNUS_RVMC 4     /* RV midmyocardial [volders99,verkerk05] */
#define BERNUS_NTYPES 5
```

This code is used in section 31.

12. ⟨ define cell type names 12 ⟩ ≡

```
#ifdef BERNUS_C
  char bernus_cname[BERNUS_NTYPES][20] = {"epic","mcel","endo","isch","rvmc"};
#else
  extern char bernus_cname[BERNUS_NTYPES][20];
#endif
```

This code is used in section 30.

13. Implementation. With the exception of a few constants defined before, the implementation of the model starts here. It consists of several functions, as outlined above. The functions share some variables, notably to pass values of constants. All functions are placed in a single C file, so that they can share variables without making these available to other parts of the program.

```
⟨ bernus.c 13 ⟩ ≡
  //␣$Id:␣bernus.web,v␣1.62␣2008/09/09␣22:45:23␣potse␣Exp␣$
#define BERNUS_C
  ⟨ Preprocessor definitions ⟩
  ⟨ Extended preprocessor code 53 ⟩
  ⟨ include files 22 ⟩
  ⟨ variables that are shared by the functions 10 ⟩
  ⟨ public functions 14 ⟩
```

14. The *bernus_info* function returns information about the membrane model to the main program. In the *ct* argument it provides a reasonable example of a celltype configuration. The *ifo* argument serves to inform the application about memory requirements and such. Both are prescribed by `membrane.web`.

```
⟨ public functions 14 ⟩ ≡
  void bernus_info(Membrane_cell_info *ct, Membrane_info *ifo, void *prm)
  {
    int i;
    bpar = prm;      /* keep a copy for ourselves */
    ct→mcode = 0;      /* model number, cannot know here */
    ct→ccode = BERNUS_ENDO;      /* celltype code */
    ct→param = Λ;      /* may be allocated by application */
    ifo→Nsvar = BERNUS_NSVAR;      /* number of status variables per cell */
    ifo→Ntypes = BERNUS_NTYPES;      /* number of cell types defined */
    ifo→Tname = malloc(ifo→Ntypes * sizeof(char *));
    for (i = 0; i < ifo→Ntypes; i++) ifo→Tname[i] = strdup(bernus_cname[i]);
    ifo→Nparam = 1;
    ifo→param = malloc(sizeof(int));
    ifo→param[0] = bpar→gkfac_param;
    ifo→info = &bernus_info;
    ifo→init = &bernus_init;
    ifo→infinite = &bernus_infinite;
    ifo→int_current = &bernus_int_current;
    ifo→get_currents = &bernus_get_currents;
    ifo→ccode = &bernus_ccode;
  }
```

See also sections 15, 16, 17, 18, and 19.

This code is used in section 13.

15. The function *bernus_int_current*( ) updates the status variables and calculates the current. It is to be used in the inner loop, so it should be optimized as much as possible.

The *Isd* argument is intended to pass the stimulation and diffusion currents to the function. In this model it is not used, but other models may use it to keep ion concentrations compatible with the electric charge and thus with the membrane potential [hund01].

```
⟨ public functions 14 ⟩ +≡
  double bernus_int_current(double vm, yyy_t *__restrict__ cell_status, Membrane_cell_info
          *ctype, double Isd, double dt, float *dtime, float simtime, long elm)
  {
    ⟨ local variables of bernus_int_current( ) 43 ⟩
    celltype = ctype→ccode;
    switch (celltype) {
    case BERNUS_EPIC: case BERNUS_MCEL: case BERNUS_ENDO: case BERNUS_ISCH: case BERNUS_RVMC:
      break;
    default: Error(1, "ccode␣%d␣is␣out␣of␣range", ctype→ccode);
    }
    ⟨ determine the cell type 44 ⟩
    ⟨ find the row in memtab 45 ⟩
    ⟨ update state variables 50 ⟩
    ⟨ check if the cell depolarized 47 ⟩
    ⟨ return ionic current 46 ⟩
  }
```

16. The computation of constants and tabulation of expensive functions is done in *bernus_init*( ). It also takes a pointer to the membrane parameter struct and makes a copy available to the other functions.

```
⟨ public functions 14 ⟩ +≡
  void bernus_init(double dt)
  {
    ⟨ local variables of the initialization function 38 ⟩
    static int first_time = 1;
    if (first_time) ⟨ identify 21 ⟩
    ⟨ calculate equilibrium potentials 7 ⟩
    ⟨ calculate potential-independent factors 34 ⟩
    ⟨ calculate the table 35 ⟩
    first_time = 0;
  }
```

17. The *bernus_infinite* function fills in the resting membrane potential and the corresponding y_∞ values.

```
⟨ public functions 14 ⟩ +≡
  void bernus_infinite(Membrane_cell_info *ctype, vm_t * Vm, yyy_t *yyy)
  {
    long i;      /* used to index yyy */
    int m, celltype;
    double fm, dm0, dm1;
    celltype = ctype→ccode;
    *Vm = bpar→vmrest[celltype];
    fm = (bpar→vmrest[celltype] − TAB_BOT) * inv_step;
    m = (int) fm;      /* cheap floor( ) */
    dm1 = fm − m;
    dm0 = 1.0 − dm1;      /* dm0, dm1 are used in interp_tab */
    yyy[GATE_M] = interp_tab(m, celltype, TC_INF_M);
    yyy[GATE_V] = interp_tab(m, celltype, TC_INF_V);
    yyy[GATE_F] = interp_tab(m, celltype, TC_INF_F);
    yyy[GATE_TO] = interp_tab(m, celltype, TC_INF_TO);
    yyy[GATE_X] = interp_tab(m, celltype, TC_INF_X);
    yyy[GATE_R] = interp_tab(m, celltype, TC_INF_R);
  }
```

18. The *bernus_get_currents* function returns the the individual ionic currents. It repeats a lot of code from *bernus_int_currents*, maybe this can be done cleaner. However, integration with *bernus_int_currents* does not seem a good idea, since that function should be as fast as possible and does not need to split the background currents into parts.

FIXME: test this (with `mmex.web`)

```
⟨ public functions 14 ⟩ +≡
  void bernus_get_currents(float vm, yyy_t *cell_status, Membrane_cell_info *ctype, float *ion)
  {
    int elm = 0;
    ⟨ local variables of bernus_int_current( ) 43 ⟩      /* borrowed */
    celltype = ctype→ccode;
    ⟨ determine the cell type 44 ⟩
    ⟨ find the row in memtab 45 ⟩
    ion[0] = (float) I_Na;
    ion[1] = (float) I_Ca;
    ion[2] = (float) I_to;
    ion[3] = (float) I_K;
    ion[4] = (float) interp_tab(rownr, celltype, TC_I);      /* total time-independent current */
    ion[5] = (float) vm;
    ion[6] = (float) interp_tab(rownr, celltype, TC_IK1);      /* individual... */
    ion[7] = (float) interp_tab(rownr, celltype, TC_ICAB);
    ion[8] = (float) interp_tab(rownr, celltype, TC_INAB);
    ion[9] = (float) interp_tab(rownr, celltype, TC_NAK);
    ion[10] = (float) interp_tab(rownr, celltype, TC_NACA);
  }
```

19. This function returns the cell type code corresponding to the given cell type name. It must be guaranteed that the returned codes are acceptable to the other functions.

```
⟨ public functions 14 ⟩ +≡
  int bernus_ccode(char *name)
  {
    int i;
    for (i = 0; i < BERNUS_NTYPES; i++) {
      if (strcmp(name, bernus_cname[i]) ≡ 0) return i;
    }
    Error(1, "I␣don't␣know␣about␣cell␣type␣\"%s\"", name);
    return −1;      /* keep the compiler happy */
  }
```

20. These variables store constants that are computed during initialization; they were discussed as part of the Introduction, in section ⟨ calculate equilibrium potentials 7 ⟩.

```
⟨ variables that are shared by the functions 10 ⟩ +≡
  static double E_Na[BERNUS_NTYPES], E_Ca[BERNUS_NTYPES], E_to[BERNUS_NTYPES],
      E_Kk[BERNUS_NTYPES];
```

21. ⟨ identify 21 ⟩ ≡

```
  printf("initializing␣Bernus␣membrane␣$Revision:␣1.62␣$\n");
  ⟨ show settings 27 ⟩
```

This code is used in section 16.

22. The implementation needs some well-known include files.

```
⟨ include files 22 ⟩ ≡
#include <limits.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>      /* (defines strdup as a macro invoking __strdup) */
#ifdef MATLAB_MEX_FILE
#undef strdup      /* provides its own version of strdup */
#endif
#include <time.h>
#include "igb/header.h"      /* IGB file header handling */
#include "util/endians.h"      /* Byte swapper and Endian test */
#include "membranes/membrane.h"
#include "membranes/membrane_internal.h"
#include "membranes/bernus_p.h"      /* needed before function declarations */
#include "membranes/bernus.h"
```

This code is used in section 13.

23. Settings. This one switches on a real r gating variable for the transient outward current, as in the full Priebe–Beuckelmann version of the model.

```
#define PRIEBE_BEUCKELMANN_TO 1
#define TAUPRIME 0 /* use τ'_X in Bernus membrane model */
```

24. Disables inactivation of the transient outward current, as hypothesized by Pascal van Dessel for a specific patient, October 2004. This requires `PRIEBE_BEUCKELMANN_TO` $\equiv 1$!

```
#define MUTATED_TO_INACTIVATION 0
```

25. These are for debugging. By settings these macros, we can make the model behave more closely like Bernus' own implementation. This is only useful for debugging; the differences are small eneough to ignore.

```
#define FULL_BERNUS_COMPAT 0
#define BERNUS_ROUND 0.5
#define BERNUS_STEP_OFF 1
```

26. If set to true, some tabulated values are read from file instead of computing them here.

```
#define READ_TABLES 0
```

27.

```
⟨ show settings 27 ⟩ ≡
  printf("PRIEBE_BEUCKELMANN_TO␣␣␣=␣%d\n", PRIEBE_BEUCKELMANN_TO);
  printf("FULL_BERNUS_COMPAT␣␣␣␣␣␣=␣%d\n", FULL_BERNUS_COMPAT);
  printf("BERNUS_ROUND␣␣␣␣␣␣␣␣␣␣␣␣=␣%f\n", BERNUS_ROUND);
  printf("BERNUS_STEP_OFF␣␣␣␣␣␣␣␣␣=␣%d\n", BERNUS_STEP_OFF);
  printf("READ_TABLES␣␣␣␣␣␣␣␣␣␣␣␣␣=␣%d\n", READ_TABLES);
  printf("MUTATED_TO_INACTIVATION␣=␣%d\n", MUTATED_TO_INACTIVATION);
```

This code is used in section 21.

28. Initialization. The initialization function computes a number of variables that remain constant during the simulation. Many of these are a function of the membrane potential; these variables are tabulated for several potential values. The limits of the table are determined by the expected range of membrane potentials. To these values we add `VSHIFT_MIN` and `VSHIFT_MAX`, respectively, which account for the lookup of a value $V_{\rm m} - V_{\rm shift}$ instead of $V_{\rm m}$ itself in the table, as is done for the computation of the Transient outward current.

```
#define TAB_BOT (−100.0 − VSHIFT_MAX)
#define TAB_TOP (90.0 − VSHIFT_MIN)
#define TAB_SIZE 500 /* Bernus uses 10000 but does not interpolate */
⟨ variables that are shared by the functions 10 ⟩ +≡
  static table_t Memtab[TAB_SIZE][BERNUS_NTYPES][TAB_COLS];
#if FULL_BERNUS_COMPAT
#define TAB_STEP ((double) (TAB_TOP − TAB_BOT)/(TAB_SIZE − BERNUS_STEP_OFF))
#else
#define TAB_STEP ((double) (TAB_TOP − TAB_BOT)/(TAB_SIZE − 1))
#endif
```

29. The columns of the table are specified with these macros. The macro `TAB_COLS` specifies the number of columns in the table, and should be one larger than the largest of these. The variables will be discussed later.

```
#define TC_INF_M 0 /* m_∞ */
#define TC_EXP_M 1 /* exp(−dt/τ_m) */
#define TC_INF_V 2 /* v_∞ */
#define TC_EXP_V 3 /* exp(−dt/τ_v) */
#define TC_FAC_D 4 /* g_Ca d_∞ f_Ca (V_m − E_Ca), no τ associated with d */
#define TC_INF_F 5 /* f_∞ */
#define TC_EXP_F 6 /* exp(−dt/τ_f) */
#define TC_FAC_R 7 /* r_∞(V_m − E_to) (Bernus version only) */
#define TC_INF_TO 8 /* to_∞ */
#define TC_TAU_TO 9 /* τ_to(p = 1) */
#define TC_INF_X 10 /* X_∞ */
#define TC_EXP_X 11 /* exp(−dt/τ_X) */
#define TC_I 12 /* I_K1 + I_Ca,b + I_Na,b + I_NaK + I_NaCa */
#define TC_INF_R 13 /* r_∞ for Priebe–Beuckelmann version */
#define TC_EXP_R 14 /* exp(−dt/τ_r) for Priebe–Beuckelmann version */
#define TC_IK1 15 /* I_K1 */
#define TC_ICAB 16 /* I_Ca,b */
#define TC_INAB 17 /* I_Na,b */
#define TC_NAK 18 /* I_NaK */
#define TC_NACA 19 /* I_NaCa */
#define TAB_COLS 20
```

30. The data type of the table is referred to with a newly defined type, so that we may change it easily. Ditto for the gating variables and the membrane potential. These types must be exported to the main program.

```
format table_t int
format yyy_t int
format byte int
format Membrane_cell_info int
format Membrane_info int
```

⟨ bernus.h 30 ⟩ ≡

```
//␣$Id:␣bernus.web,v␣1.62␣2008/09/09␣22:45:23␣potse␣Exp␣$␣
#include "membranes/bernus.d"
#define BERNUS_B_H 1 /* identify us */
⟨ define cell type names 12 ⟩
void bernus_info(Membrane_cell_info *ct, Membrane_info *ifo, void *prm);
void bernus_init(double dt);
void bernus_infinite(Membrane_cell_info *ctype, vm_t * Vm, yyy_t *yyy);
double bernus_int_current(double vm, yyy_t *cell_status, Membrane_cell_info *ctype, double
    Isd, double dt, float *dtime, float simtime, long elm);
void bernus_get_currents(float vm, yyy_t *cell_status, Membrane_cell_info *ctype, float *ion);
int bernus_ccode(char *name);
```

This code is cited in section 42.

31. This file is to be shared between C and PRM code.

⟨ bernus.d 31 ⟩ ≡

```
⟨ define cell types 11 ⟩
#define BERNUS_NSVAR 6 /* 6 including P–B r gating variable */
```

32. The storage for the parameters is allocated by the application, by defining a variable of type **Bernus_param**. A pointer to this struct is passed to *bernus_info*; the other functions will share it.

⟨ variables that are shared by the functions 10 ⟩ +≡

```
static Bernus_param *bpar;
```

33. ⟨ variables that are shared by the functions 10 ⟩ +≡

```
static double f_Ca[BERNUS_NTYPES];
```

34. Factor f_{Ca} [bernus02,priebe98]. *bernus.ci_Ca* is internal calcium concentration

⟨ calculate potential-independent factors 34 ⟩ ≡

```
{
  int t;
  for (t = 0; t < BERNUS_NTYPES; t++) {
    f_Ca[t] = KCa/(KCa + bpar→ci_Ca[t]); /* = 1.0/(1.0 + bpar→ci_Ca/KCa) */
  }
}
```

This code is used in section 16.

35. The table is filled in in a loop over all potential values. The contents of the table are discussed in the following sections. An overview of the table columns was given above.

```
⟨ calculate the table 35 ⟩ ≡
  {
    double vm, step = TAB_STEP;
    int tr, t;
    int skip = TAB_SIZE/100;      /* print about 100 lines in table file */
    ⟨ read some tabulated parameters from file 37 ⟩
    ⟨ open the table file 63 ⟩
    for (tr = 0; tr < TAB_SIZE; tr++) {
      vm = (double) TAB_BOT + step * tr;
      for (t = 0; t < BERNUS_NTYPES; t++) Memtab[tr][t][TC_I] = 0.0;
      ⟨ fill in this row 49 ⟩
      if ((tr % skip) ≡ 0) ⟨ print a line in the table file 64 ⟩
    }
    ⟨ close the table file 65 ⟩
    inv_step = 1.0/TAB_STEP;
    clip_max = TAB_TOP + VSHIFT_MIN − 3 * TAB_STEP;      /* allow for dm1 */
    clip_min = TAB_BOT + VSHIFT_MAX;
    printf("clip_max = %f, clip_min = %f, TAB_STEP = %f\n", clip_max, clip_min, TAB_STEP);
    printf("VSHIFT_MIN = %f, TAB_TOP = %f\n", VSHIFT_MIN, TAB_TOP);
    printf("VSHIFT_MAX = %f, TAB_BOT = %f\n", VSHIFT_MAX, TAB_BOT);
  }
```

This code is cited in section 8.

This code is used in section 16.

36. The inverted table step is often used in *bernus_int_current*( ).

```
⟨ variables that are shared by the functions 10 ⟩ +≡
  static double inv_step, clip_max, clip_min;
```

37. Mainly for debugging: Bernus' implementation of the model uses measured values, tabulated in binary files, for four parameters. We use the continuous approximations from the paper [bernus02], but in order to reproduce everything exactly we must use the tables too, since the approximations are, well... just approximations.

These data files contain **double**s written in Little-Endian order. We have to swap them if running on a Big-Endian machine like the SGI Origin.

```
⟨ read some tabulated parameters from file 37 ⟩ ≡
  {
#if READ_TABLES
    FILE *tauv_file, *vinf_file, *taux_file, *xinf_file;
    if (TAB_SIZE ≠ 10000 ∨ TAB_BOT ≠ −100.0 ∨ TAB_TOP ≠ 90.0)
      Error(1, "Table␣parameters␣must␣agree␣when␣using␣tables␣from␣file");
    if (¬(tauv_file = fopen("TAU_V10000", "r"))) Error(1, "cannot␣open␣a␣tabfile");
    if (¬(vinf_file = fopen("V_INF10000", "r"))) Error(1, "cannot␣open␣a␣tabfile");
    if (¬(taux_file = fopen("TAU_X10000", "r"))) Error(1, "cannot␣open␣a␣tabfile");
    if (¬(xinf_file = fopen("X_INF10000", "r"))) Error(1, "cannot␣open␣a␣tabfile");
    if (fread(ftab_vi, sizeof(double), 10000, vinf_file) ≠ 10000) Error(1, "trouble!");
    if (fread(ftab_tauv, sizeof(double), 10000, tauv_file) ≠ 10000) Error(1, "trouble!");
    if (fread(ftab_xi, sizeof(double), 10000, xinf_file) ≠ 10000) Error(1, "trouble!");
    if (fread(ftab_taux, sizeof(double), 10000, taux_file) ≠ 10000) Error(1, "trouble!");
    fclose(tauv_file);
    fclose(vinf_file);
    fclose(taux_file);
    fclose(xinf_file);
    if (big_endian()) {
      swap8(ftab_vi, 10000);
      swap8(ftab_tauv, 10000);
      swap8(ftab_xi, 10000);
      swap8(ftab_taux, 10000);
    }
    printf("tables␣from␣file␣used\n");
#else
    printf("no␣tables␣from␣file␣used\n");
#endif
  }
```

This code is used in section 35.

38. ⟨ local variables of the initialization function 38 ⟩ ≡

```
#if READ_TABLES
  double ftab_vi[10000], ftab_tauv[10000];
  double ftab_xi[10000], ftab_taux[10000];
#endif
```

See also sections 61 and 66.

This code is used in section 16.

39. Parameters. The following code defines a parameter definition file for the `prm` program. This file is used by the Bernus membrane module to define and initialize its parameter struct. It can also be used by an application to define an identical struct, which can be passed to *bernus_init* to set parameter values.

The `prm` program creates, among others, a file `bernus_p.c`, which we will include here.

The `prm` language is slightly adapted to make it usable in a `cweb` document; the C preprocessor does the translation using macros from `cweb_prm.h`. The `.prr` file must be explicitly preprocessed; gcc crashes if it reads this when called from `prm`.

```
⟨ bernus.prr 39 ⟩ ≡
  //␣$Id:␣bernus.web,v␣1.62␣2008/09/09␣22:45:23␣potse␣Exp␣$
#include "prm/include/PrM/cweb_prm.h"      /* translates to prm code */
#include "membranes/membrane.d"      /* defines array sizes etc */
#include "membranes/bernus.d"      /* defines array sizes etc */
  structure Bernus_param
    {
      member(g_K, type = float[BERNUS_NTYPES], default = {0.036, 0.013, 0.030, 0.030, 0.013},
          s_desc = "conductivity␣of␣delayed␣rectifier␣current␣for␣Bernus␣model",
          units = "nS/pF")
      member(g_to, type = float[BERNUS_NTYPES], default = {0.40, 0.35, 0.13, 0.13, 1.40},
          s_desc = "conductivity␣of␣I_to␣for␣Bernus␣model␣without␣r␣gate", units = "nS/pF")
      member(g_tor, type = float[BERNUS_NTYPES], default = {0.30, 0.26, 0.10, 0.10, 1.04},
          s_desc = "conductivity␣of␣I_to␣for␣Bernus␣model␣with␣r␣gate", units = "nS/pF")
      member(g_Na, type = float[BERNUS_NTYPES], default = 16.0,
          s_desc = "conductivity␣of␣fast␣Na␣current", units = "nS/pF")
      member(g_Ca, type = float[BERNUS_NTYPES], default = 0.064,
          s_desc = "conductivity␣of␣slow␣Ca␣current", units = "nS/pF")
      member(g_K1, type = float[BERNUS_NTYPES], default = 3.9,
          s_desc = "conductivity␣of␣inw.␣rect.␣K␣current", units = "nS/pF")
      member(g_Nab, type = float[BERNUS_NTYPES], default = 0.001,
          s_desc = "conductivity␣of␣Na␣background␣current", units = "nS/pF")
      member(g_Cab, type = float[BERNUS_NTYPES], default = 0.00085,
          s_desc = "conductivity␣of␣Ca␣background␣current", units = "nS/pF")
      member(g_NaK, type = float[BERNUS_NTYPES], default = 1.3,
          s_desc = "conductivity␣of␣Na/K␣pump", units = "nS/pF")
      member(g_NaCa, type = float[BERNUS_NTYPES], default = 1000.0,
          s_desc = "conductivity␣of␣Na/Ca␣pump", units = "nS/pF")
      member(ci_Ca, type = float[BERNUS_NTYPES], default = {0.0004, 0.0004, 0.0004, 0.0004,
          0.0004}, s_desc = "intracellular␣calcium␣concentration", units = "mM")
      member(ce_Ca, type = float[BERNUS_NTYPES], default = {2.0, 2.0, 2.0, 2.0, 2.0},
          s_desc = "extracellular␣calcium␣concentration", units = "mM")
      member(ci_Na, type = float[BERNUS_NTYPES], default = {10.0, 10.0, 10.0, 10.0, 10.0},
          s_desc = "intracellular␣sodium␣concentration", units = "mM")
      member(ce_Na, type = float[BERNUS_NTYPES], default = {138.0, 138.0, 138.0, 138.0, 138.0},
          s_desc = "extracellular␣sodium␣concentration", units = "mM")
      member(ci_K, type = float[BERNUS_NTYPES], default = {140.0, 140.0, 140.0, 140.0, 140.0},
          s_desc = "intracellular␣potassium␣concentration", units = "mM")
      member(ce_K, type = float[BERNUS_NTYPES], default = {4.0, 4.0, 4.0, 12.0, 4.0},
          s_desc = "extracellular␣potassium␣concentration", units = "mM")
      member(vmrest, type = float[BERNUS_NTYPES], default = {−90.272, −90.272, −90.272, −65.0,
          −90.272}, s_desc = "resting␣membrane␣potential␣for␣initialization", units = "mV")
      member(gkrange, type = float[2], default = {1.0, 1.0},
          s_desc = "extent␣of␣g_K␣modification␣by␣input␣file", units = "nS/pF")
      member(gkfac_param, type = int, default = 0,
          s_desc = "parameter␣number␣where␣gkfac␣is␣to␣come␣from")
    }
```

40. In order to create `"bernus_p.c"` we need a parameter file with at least one variable definition to make it acceptable to `prm`.

Applications, on the other hand, must include the parameter file without any variable definitions: `bernus.prs`, which is created from `bernus.prr` above.

⟨ `bernus.prm` 40 ⟩ ≡

```
/*␣$Id:␣bernus.web,v␣1.62␣2008/09/09␣22:45:23␣potse␣Exp␣$␣*/
#include "bernus.prs"
␣$variable␣dummy␣{␣$type=Bernus_param,␣$s_desc="module␣parameters"␣}␣
```

41. Gating variables and current computation. In the *bernus_int_current* function, we have to update the 6 gating variables and compute the new ionic current. The *interp_tab* macro interpolates linearly in column c of *memtab*. The *int_gate* macro integrates a gating variable over one time step; it uses the interpolation macro both for y_∞ and for τ_y. Arguments to *int_gate* are the column number of y in the status table, column number of y_∞ in *memtab*, and column number of τ_y in *memtab*.

```
#define interp_tab(m,t,c) (dm0 * Memtab[m][t][c] + dm1 * Memtab[m+1][t][c])
#define interp_tab_to(m,t,c) (dm0_to * Memtab[m][t][c] + dm1_to * Memtab[m+1][t][c])
        /* exception */
#define int_gate(y,m,t,i,j) {
        double y_inf;
        y_inf = interp_tab(m,t,i);
        y = y_inf - (y_inf - y) * interp_tab(m,t,j);      /* y_new = y_inf - (y_inf - y)e^{-dt/tau_y} */
      }
#define int_gate_to(y,m,t,i,j) {      /* to gate uses its own row and dm */
        double y_inf, tau;
        y_inf = interp_tab_to(m,t,i);
        tau = p_to[celltype] * interp_tab(m,t,j);      /* no V_shift for tau! */
        y = y_inf - (y_inf - y) * exp(-dt/tau);      /* was fexp */
      }
```

42. In ⟨ `bernus.h` 30 ⟩ we have declared that we need 6 status variables. The macros `GATE_M` etc. define which variable goes where. In this membrane model, all status variables are gating variables. The identifiers *gate_m*, *gate_v* etc. are left/right-value shortcuts to gating to the gating variables of the current cell.

```
#define gate_m (cell_status[GATE_M])
#define gate_v (cell_status[GATE_V])
#define gate_f (cell_status[GATE_F])
#define gate_to (cell_status[GATE_TO])
#define gate_X (cell_status[GATE_X])
#define gate_r (cell_status[GATE_R])
#define GATE_M 0      /* column in yyy where m is stored */
#define GATE_V 1      /* etc... */
#define GATE_F 2
#define GATE_TO 3
#define GATE_X 4
#define GATE_R 5
```

43. ⟨ local variables of *bernus_int_current*( ) 43 ⟩ ≡

```
int rownr;
int celltype;
double vhat;
double dm0, dm1;
double gkfac_c;
```

This code is used in sections 15 and 18.

44. Parameters λ and μ were removed because they cannot account for LV and RV M-cells; think of a different implementation for smoothly varying cell type. For example, a membrane model that reads in the fixed cell types, then sets up a 3-D array for each parameter, fills it in with the fixed-type values, and finally applies some smoothing operator. The membrane table should be abandoned to allow doing this with ion concentrations too. Such a membrane model is likely to be a lot less efficient. Storage of all parameters for all cells would take a lot of space too; perhaps it can be done at 1-mm resolution.

In the current implementation, only g_{K} is now variable. This is necessary to implement an APD gradient.

```
⟨ determine the cell type 44 ⟩ ≡
  {
    double gkscale = bpar→gkrange[1] − bpar→gkrange[0];
    if (bpar→gkfac_param) {
      gkfac_c = bpar→gkrange[0] + ctype→param[0] ∗ gkscale;
    }
    else {
      gkfac_c = 1.0;      /∗ or bpar→gkrange[0] ∗/
    }
  }
```

This code is used in sections 15 and 18.

45. *vhat* is clipped to a limited part of the table, allowing V_{shift} to be applied to it without having to check the limits and clip again when computing I_{to}.

```
⟨ find the row in memtab 45 ⟩ ≡
  {
    double fm;
    vhat = vm;      /∗ membrane potential ∗/
    if (vhat < clip_min) {
      Warning(2, "clipped␣vhat=%f␣to␣%f␣at␣%d", vhat, clip_min, elm);
      vhat = clip_min;
    }
    else if (vhat ≥ clip_max) {
      Warning(2, "clipped␣vhat=%f␣to␣%f␣at␣%d", vhat, clip_max, elm);
      vhat = clip_max;
    }
    else if (isnan(vhat)) {
      Error(1, "vhat␣is␣not-a-number␣at␣%d", elm);
    }
    fm = (vhat − TAB_BOT) ∗ inv_step;
#if FULL_BERNUS_COMPAT
    rownr = (int) (fm + BERNUS_ROUND);      /∗ cheap round( ) ∗/
    dm1 = 0.0;
    dm0 = 1.0;
#else
    rownr = (int) (fm);      /∗ cheap floor( ) ∗/
    dm1 = fm − rownr;
    dm0 = 1.0 − dm1;
#endif
    if (rownr < 0 ∨ rownr ≥ TAB_SIZE − 1)
      Error(1, "out␣of␣table␣range:␣m=%d,␣fm=%f\n", rownr, fm);      /∗ DEBUG temporary ∗/
  }
```

This code is used in sections 15 and 18.

46. Row `TC_I` of *memtab* specifies the sum of all static currents.

```
⟨ return ionic current 46 ⟩ ≡
  {
#if 0
    printf("vhat=%f␣m=%f␣v=%f␣f=%f␣to=%f␣X=%f␣r=%f␣I_Na=%f␣I_Ca=%f␣I_to=%f␣I_K=%f␣Is=%f\n",
        vhat, gate_m, gate_v, gate_f, gate_to, gate_X, gate_r, I_Na, I_Ca, I_to, I_K, interp_tab(m, celltype,
        TC_I));
#endif
    return (I_Na + I_Ca + I_to + I_K + interp_tab(rownr, celltype, TC_I));
  }
```

This code is used in section 15.

47. Accurate depolarization times are returned in the *dtime* argument. We know that a cell depolarized if the gating variable m reaches a value near unity. Since m will be reduced later, we need an extra variable to remember that the cell depolarized. For this purpose, the application is supposed to initialize *dtime* to a negative value. So we overwrite *dtime* only if it is negative.

If the cells are to be depolarized again, all *dtime* variables should be reset. This feature is somewhat inconvenient for modeling of reentry—which we didn't do so far. We can facilitate the implementation of reentry by developing a criterion for excitability, and returning a flag to the application to tell it whether the cell is excitable.

```
⟨ check if the cell depolarized 47 ⟩ ≡
  {
    if (*dtime < 0.0) {
      if (gate_m > 0.98) {
        *dtime = simtime;      /* set time and remember it depolarized */
      }
    }
  }
```

This code is used in section 15.

48. Go with the flow. The implementation of the currents was largely copied from program source provided by Olivier Bernus. However, he used tabulated functions that were read from file for the variables τ_v, τ_X, v_∞, and X_∞. At this time we use the continuous approximations given in the paper. Should check if the tabulated values are better.

In the following sections, the tabulated quantities are treated. Each section deals with one ionic current. The equation for this current is given, together with its ingredients. A macro is defined that, expanded within the *bernus_int_current* function, evaluates the current. In addition, a piece of code is defined that is used in *bernus_init* to fill in the associated table element(s), as well as a second piece of code, which performs the update of associated status variables.

49. The Fast Na^+ current.

$$I_{\mathrm{Na}} = g_{\mathrm{Na}}\, m^3\, v^2\, (V_{\mathrm{m}} - E_{\mathrm{Na}})$$

$$\alpha_m = \frac{0.32(V_{\mathrm{m}} + 47.13)}{1 - \exp[-0.1(V_{\mathrm{m}} + 47.13)]}$$

$$\beta_m = 0.08 \exp(-V_{\mathrm{m}}/11)$$

$$v_\infty = \frac{1}{2}\big[1 - \tanh(7.74 + 0.12 V_{\mathrm{m}})\big]$$

$$\tau_v = 0.25 + 2.24\, \frac{1 - \tanh[7.74 + 0.12 V_{\mathrm{m}}]}{1 - \tanh[0.07(V_{\mathrm{m}} + 92.4)]}$$

Note that there is no α_v and β_v. The gating variables m and v are computed from their steady-state values and time constants, as discussed in the Introduction. We tabulate these four quantities. In Bernus' implementation of the model v_∞ and τ_v are read from files containing the measured values. In addition, he tabulates $g\_Na * (vm - E\_Na)$

```
#define I_Na (bpar¬g_Na[celltype] * pow3(gate_m) * pow2(gate_v) * (vm - E_Na[celltype]))
#define I_Na_test (bpar¬g_Na[celltype] * pow3(gate_m) * gate_v * (vm - E_Na[celltype]))
⟨ fill in this row 49 ⟩ ≡
  {
    double α, β, τ;
    int t;
#if 1     /* Bernus model for ventricular myocardium */
    if (fabs(vm + 47.13) > 0.001) α = 0.32 * (vm + 47.13)/(1.0 - exp(-0.1 * (vm + 47.13)));
    else α = 3.2;
    β = 0.08 * exp(-vm/11.0);
#else     /* Purkinje cell according to DiFrancesco and Noble: */
    if (fabs(vm + 41) > 0.001) α = 0.2 * (vm + 41.0)/(1.0 - exp(-0.1 * (vm + 41.0)));
    else α = 2.0;
    β = 0.2 * exp(-vm/18.0);
#endif
    τ = 1.0/(α + β);     /* unit is ms */
    for (t = 0; t < BERNUS_NTYPES; t++) {
      Memtab[tr][t][TC_EXP_M] = exp(-dt/τ);
      Memtab[tr][t][TC_INF_M] = α * τ;
    }
#if READ_TABLES
    for (t = 0; t < BERNUS_NTYPES; t++) Memtab[tr][t][TC_INF_V] = ftab_vi[tr];
    τ = ftab_tauv[tr];
#else
    for (t = 0; t < BERNUS_NTYPES; t++) Memtab[tr][t][TC_INF_V] = 0.5 * (1 - tanh(7.74 + 0.12 * vm));
    τ = 0.25 + 2.24 * (1 - tanh(7.74 + 0.12 * vm))/(1.0 - tanh(0.07 * (vm + 92.4)));
#endif
    for (t = 0; t < BERNUS_NTYPES; t++) Memtab[tr][t][TC_EXP_V] = exp(-dt/τ);
  }
```

See also sections 51, 54, 56, 58, 59, 60, and 62.

This code is cited in section 61.

This code is used in section 35.

50. ⟨ update state variables 50 ⟩ ≡

```
int_gate(gate_m, rownr, celltype, TC_INF_M, TC_EXP_M);
int_gate(gate_v, rownr, celltype, TC_INF_V, TC_EXP_V);
```

See also sections 52, 55, and 57.

This code is used in section 15.

51. The Slow Ca^{2+} current.

$$I_{\mathrm{Ca}} = g_{\mathrm{Ca}}\, d_\infty\, f\, f_{\mathrm{Ca}}\, (V_{\mathrm{m}} - E_{\mathrm{Ca}})$$

$$\alpha_d = \frac{14.98\exp\{-0.5[(V_{\mathrm{m}} - 22.36)/16.68]^2\}}{16.68\sqrt{2\pi}}, \qquad \beta_d = 0.1471 - \frac{5.3\exp\{-0.5[(V_{\mathrm{m}} - 6.27)/14.93]^2\}}{14.93\sqrt{2\pi}}$$

$$\alpha_f = \frac{6.87\cdot 10^{-3}}{1+\exp[-(6.1546 - V_{\mathrm{m}})/6.12]}, \qquad \beta_f = \frac{0.069\exp[-0.11(V_{\mathrm{m}} + 9.825)] + 0.011}{1+\exp[-0.278(V_{\mathrm{m}} + 9.825)]} + 5.75\cdot 10^{-4}$$

$$f_{\mathrm{Ca}} = \frac{1}{1 + [\mathrm{Ca}^{2+}]_{\mathrm{i}}/0.0006}$$

where the factor $0.0006 = K_{\mathrm{Ca}}$ in the P-B model. The gating variable f is computed from its steady-state value and time constant, which are in turn computed from α and β, as discussed in the Introduction. Gating variable d is static: $d = d_\infty$, so we can tabulate all of I_{Ca} except for the gating variable f.

#define *I_Ca* (*interp_tab*(*rownr*, *celltype*, TC_FAC_D) ∗ *gate_f*)

⟨ fill in this row 49 ⟩ +≡

```
{
  double α, β, τ, a, b;
  int t;
  a = sqrt(2.0 ∗ M_PI);
  b = (vm − 22.36)/16.6813;
  α = (14.9859/(16.6813 ∗ a)) ∗ exp(−0.5 ∗ b ∗ b);
  b = (vm − 6.2744)/14.93;
  β = 0.1471 − ((5.3/(14.93 ∗ a)) ∗ exp(−0.5 ∗ b ∗ b));
  for (t = 0; t < BERNUS_NTYPES; t++) {
    Memtab[tr][t][TC_FAC_D] = bpar→g_Ca[t] ∗ α/(α + β) ∗ f_Ca[t] ∗ (vm − E_Ca[t]);
      /∗ g_Ca d_∞ f_Ca (V_m − E_Ca) ∗/
  }
  α = 6.872 · 10^−3/(1.0 + exp((vm − 6.1546)/6.1223));
  a = 0.0687 ∗ exp(−0.1081 ∗ (vm + 9.8255)) + 0.0112;
  b = 1.0 + exp(−0.2779 ∗ (vm + 9.8255));
  β = a/b + 5.474 · 10^−4;
  τ = 1.0/(α + β);
  for (t = 0; t < BERNUS_NTYPES; t++) {
    Memtab[tr][t][TC_EXP_F] = exp(−dt/τ);
    Memtab[tr][t][TC_INF_F] = α ∗ τ;
  }
}
```

52. ⟨ update state variables 50 ⟩ +≡

```
int_gate(gate_f, rownr, celltype, TC_INF_F, TC_EXP_F);
```

53. The Transient outward current. This current has two gating variables: the activation variable r and the inactivation variable *to*. Two versions of the current are implemented: the Bernus version and the full Priebe–Beuckelmann version. The current for the full version is

$$I_{\mathrm{to}} = g_{\mathrm{to}}\, r\, to\, (V_{\mathrm{m}} - E_{\mathrm{to}})$$

where for the Bernus version the activation variable $r = r_\infty$, i.e. a steady-state gating variable, while it is a real gating variable in the full version. The expressions for α_r, β_r, α_{to}, and β_{to} are the same in both versions.

$$\alpha_r = \frac{0.5266 \exp[-0.0166(V_{\mathrm{m}} - 42.2912)]}{1 + \exp[-0.0943(V_{\mathrm{m}} - 42.2912)]}, \qquad \beta_r = \frac{5.186 \cdot 10^{-5}\, V_{\mathrm{m}} + 0.5149 \exp[-0.1344(V_{\mathrm{m}} - 5.0027)]}{1 + \exp[-0.1348(V_{\mathrm{m}} - 5.186 \cdot 10^{-5})]}$$

$$\alpha_{\mathrm{to}} = \frac{5.612 \cdot 10^{-5}\, V_{\mathrm{m}} + 0.0721 \exp[-0.173(V_{\mathrm{m}} + 34.2531)]}{1 + \exp[-0.1732(V_{\mathrm{m}} + 34.2531)]}$$

$$\beta_{\mathrm{to}} = \frac{1.215 \cdot 10^{-4}\, V_{\mathrm{m}} + 0.0767 \exp[-1.66 \cdot 10^{-9}\, (V_{\mathrm{m}} + 34.0235)]}{1 + \exp[-0.1604(V_{\mathrm{m}} + 34.0235)]}$$

We also study a hypothesized mutation where inactivation fails completely. This is implemented by omitting *gate_to* from the equation. The variable *gate_to* is still computed.

⟨ Extended preprocessor code 53 ⟩ ≡

```
#if MUTATED_TO_INACTIVATION
#define I_to bpar→g_tor[celltype] * gate_r * (vm − E_to[celltype]) )
#elif PRIEBE_BEUCKELMANN_TO
#define I_to   (bpar→g_tor[celltype] * gate_r * gate_to * (vm − E_to[celltype]))
#else
#define I_to   (bpar→g_to[celltype] * interp_tab(rownr, celltype, TC_FAC_R) * gate_to * (vm − E_to[celltype]))
#endif
```

This code is used in section 13.

54. The time constant and steady-state value for the *to* gate involve extra parameters in order to deal with the different cell types.

$$\tau_{\rm to} = \frac{1}{p\alpha_{\rm to} + p\beta_{\rm to}}, \qquad to_\infty = \frac{\alpha_{\rm to}(V_{\rm m} - V_{\rm shift})}{\alpha_{\rm to}(V_{\rm m} - V_{\rm shift}) + \beta_{\rm to}(V_{\rm m} - V_{\rm shift})}$$

We will use this modification, which was introduced by Bernus, also in the full version, since we do need cell types in either case.

We will not tabulate $\exp(-dt/\tau_{\rm to})$ and to_∞ separately for each of the cell types: instead we will tabulate τ itself and to_∞ for the parameter values $p = 1$ and $V_{\rm shift} = 0$. Adaptation for M-cells and endocardial cells and computation of $\exp(-dt/\tau)$ can only be done within the loop, since we have continuous λ and μ parameters.

```
⟨ fill in this row 49 ⟩ +≡
  {
    double α, β, τ, a, b;
    int t;
    a = vm − 42.2912;
    α = 0.5266 ∗ exp(−0.0166 ∗ a)/(1.0 + exp(−0.0943 ∗ a));
    a = 0.5149 ∗ exp(−0.1344 ∗ (vm − 5.0027)) + 5.186 · 10^−5 ∗ vm;
    b = 1.0 + exp(−0.1348 ∗ (vm − 5.186 · 10^−5));
    β = a/b;
    τ = 1.0/(α + β);
    for (t = 0; t < BERNUS_NTYPES; t++) {
      Memtab[tr][t][TC_FAC_R] = α/(α + β);      /∗ r∞, for Bernus version ∗/
      Memtab[tr][t][TC_INF_R] = α ∗ τ;      /∗ r itself, for PB version ∗/
      Memtab[tr][t][TC_EXP_R] = exp(−dt/τ);      /∗ for PB version ∗/
    }
    a = vm + 34.2531;
    α = (0.0721 ∗ exp(−0.173 ∗ a) + 5.612 · 10^−5 ∗ vm)/(1.0 + exp(−0.1732 ∗ a));
    a = vm + 34.0235;
    β = (0.0767 ∗ exp(−1.66 · 10^−9 ∗ a) + 1.215 · 10^−4 ∗ vm)/(1.0 + exp(−0.1604 ∗ a));
    τ = 1.0/(α + β);
    for (t = 0; t < BERNUS_NTYPES; t++) {
      Memtab[tr][t][TC_TAU_TO] = τ;
      Memtab[tr][t][TC_INF_TO] = α ∗ τ;
    }
  }
```

55. Not simply *int_gate*(*gate_to*, TC_INF_TO, TC_EXP_TO); we have to take p and V_{shift} into account. For the lookup of to_∞ we use a shifted membrane potential *vs*. It should not be necessary to check the limits again. The τ variable is obtained by *int_gate_to* using the original row number.

```
⟨ update state variables 50 ⟩ +≡
  {
    double fm_to, dm0_to, dm1_to;
    int rownr_to;
    fm_to = (vhat − V_shift[celltype] − TAB_BOT) ∗ inv_step;
#if FULL_BERNUS_COMPAT
    rownr_to = (int) (fm_to + BERNUS_ROUND);      /∗ cheap round( ) ∗/
    dm1_to = 0.0;
    dm0_to = 1.0;      /∗ no interpolation ∗/
#else
    rownr_to = (int) fm_to;      /∗ cheap floor( ) ∗/
    dm1_to = fm_to − rownr_to;      /∗ used in int_gate_to ∗/
    dm0_to = 1.0 − dm1_to;      /∗ used in int_gate_to ∗/
#endif
    if (rownr_to < 0) Error(1, "This␣can't␣happen:␣rownr_to=%d␣(fm_to=%f␣vhat=%f␣Vshift=%f)",
          rownr_to, fm_to, vhat, V_shift[celltype]);
    if (rownr_to > TAB_SIZE − 2)
      Error(1, "This␣can't␣happen:␣rownr_to=%d␣(fm_to=%f␣vhat=%f␣␣Vshift=%f)", rownr_to, fm_to,
          vhat, V_shift[celltype]);
    int_gate_to(gate_to, rownr_to, celltype, TC_INF_TO, TC_TAU_TO);
#if PRIEBE_BEUCKELMANN_TO
    int_gate(gate_r, rownr, celltype, TC_INF_R, TC_EXP_R);
#endif
  }
```

56. The Delayed rectifier K^+ current.

$$I_{\mathrm{K}} = g_{\mathrm{K}}\, X^2\, (V_{\mathrm{m}} - E_{\mathrm{K}})$$

The expressions for X_∞ and τ_X have almost the same form, but several different parameters for M-cells.

$$X_\infty = \frac{0.988}{1+\exp(-0.861-0.0620V_{\mathrm{m}})}, \qquad X_{\infty,\mathrm{Mcell}} = \frac{0.972}{1+\exp(-2.036-0.0834V_{\mathrm{m}})}$$

$$\tau_X = 240\exp[-(25.5+V_{\mathrm{m}})^2/156] + 182[1+\tanh(0.154+0.0116V_{\mathrm{m}})] + \tau'_X$$

$$\tau'_X = 40[1-\tanh(160+2V_{\mathrm{m}})]$$

$$\tau_{X,\mathrm{Mcell}} = 380\exp[-(25.5+V_{\mathrm{m}})^2/156] + 166[1+\tanh(0.558+0.0169V_{\mathrm{m}})]$$

Note that τ'_X must also be added to the measured value of τ_X. The τ'_X parameter was added by Bernus solely to improve APD restitution curves; it is said not to affect the action potential shape.

#define I_K $(gkfac\_c * bpar\rightarrow g\_K[celltype] * pow2(gate\_X) * (vm - E\_Kk[celltype]))$

⟨ fill in this row 49 ⟩ +≡

```
{
  double tau_X, taup_X, tau_X_M, a;
  int t;
  a = exp(-pow2(25.5 + vm)/156);
  taup_X = 40 * (1 - tanh(160 + 2 * vm));
#if READ_TABLES
  for (t = 0; t < BERNUS_NTYPES; t++) Memtab[tr][t][TC_INF_X] = ftab_xi[tr];
  tau_X = ftab_taux[tr] + taup_X;
#else
  for (t = 0; t < BERNUS_NTYPES; t++)
    Memtab[tr][t][TC_INF_X] = 0.988/(1.0 + exp(-0.861 - 0.0620 * vm));
  tau_X = 240 * a + 182 * (1 + tanh(0.154 + 0.0116 * vm));
  if (TAUPRIME) tau_X += taup_X;
#endif
  Memtab[tr][BERNUS_ENDO][TC_EXP_X] = exp(-dt/tau_X);
  Memtab[tr][BERNUS_EPIC][TC_EXP_X] = exp(-dt/tau_X);
  Memtab[tr][BERNUS_MCEL][TC_INF_X] = 0.972/(1.0 + exp(-2.036 - 0.0834 * vm));
  tau_X_M = 380 * a + 166 * (1 + tanh(0.558 + 0.0169 * vm));
  Memtab[tr][BERNUS_MCEL][TC_EXP_X] = exp(-dt/tau_X_M);
  Memtab[tr][BERNUS_RVMC][TC_INF_X] = 0.972/(1.0 + exp(-2.036 - 0.0834 * vm));
  tau_X_M = 380 * a + 166 * (1 + tanh(0.558 + 0.0169 * vm));
  Memtab[tr][BERNUS_RVMC][TC_EXP_X] = exp(-dt/tau_X_M);
}
```

57. ⟨ update state variables 50 ⟩ +≡

```
{
  int_gate(gate_X, rownr, celltype, TC_INF_X, TC_EXP_X);
}
```

58. The Inward rectifier K^+ current. This may sound like a strange name for an outward (repolarizing) current; it's because "inward rectification" is the property that the outward current of these channels is blocked at high V_m [oudit04zj]. The inward rectifier current is generated by the Kir family of channels, and is involved in the LQT7 syndrome [oudit04zj]. It is stronger in the ventricles than in the atria.

This and the following currents have no direct time dependence; their dynamic behaviour depends only on the membrane potential. We have to tabulate only the sum of these currents. According to Olivier Bernus [personal communication] the E_{K1} in the equation for β_{K1} [bernus02] should read E_K just like the others; his implementation uses both, but their values are the same.

$$I_{K1} = g_{K1}\, K1_\infty\, (V_m - E_K)$$

$$\alpha_{K1} = \frac{0.1}{1 + \exp[0.06(V_m - E_K - 200)]}$$

$$\beta_{K1} = \frac{3\exp[2\cdot 10^{-4}(V_m - E_K + 100)] + \exp[0.1(V_m - E_K - 10)]}{1 + \exp[-0.5(V_m - E_K)]}$$

⟨ fill in this row 49 ⟩ +≡

```
{
  double α, β, v, iK1;
  int t;
  for (t = 0; t < BERNUS_NTYPES; t++) {
    v = vm − E_Kk[t];      /* same equilibrium potential as I_K */
    α = 0.1/(1.0 + exp(0.06 * (v − 200.0)));
    β = (3.0 * exp(0.0002 * (v + 100.0)) + exp(0.1 * (v − 10.0)))/(1.0 + exp(−0.5 * v));
    iK1 = bpar→g_K1[t] * α/(α + β) * v;
    Memtab[tr][t][TC_I] += iK1;
    Memtab[tr][t][TC_IK1] = iK1;
  }
}
```

59. The Ca^{2+} and Na^{+} background currents.

$$I_{\mathrm{Ca,b}} = g_{\mathrm{Ca,b}}\,(V_{\mathrm{m}} - E_{\mathrm{Ca}})$$

$$I_{\mathrm{Na,b}} = g_{\mathrm{Na,b}}\,(V_{\mathrm{m}} - E_{\mathrm{Na}})$$

```
⟨ fill in this row 49 ⟩ +≡
  {
    double ICab, INab;
    int t;
    for (t = 0; t < BERNUS_NTYPES; t++) {
      ICab = bpar→g_Cab[t] * (vm - E_Ca[t]);
      INab = bpar→g_Nab[t] * (vm - E_Na[t]);
      Memtab[tr][t][TC_I] += (ICab + INab);
      Memtab[tr][t][TC_INAB] = INab;
      Memtab[tr][t][TC_ICAB] = ICab;
    }
  }
```

60. The Na^+-K^+ pump.

$$I_{\mathrm{NaK}} = g_{\mathrm{NaK}}\, f_{\mathrm{NaK}}\, f'_{\mathrm{NaK}}$$

involves two factors, one depending on V_{m}, the other on the (fixed) ion concentrations.

$$f_{\mathrm{NaK}} = \frac{1}{1 + 0.1245\exp(-0.0037V_{\mathrm{m}}) + 0.0365\sigma\exp(-0.037V_{\mathrm{m}})}$$

$$f'_{\mathrm{NaK}} = \frac{1}{1 + (10/[\mathrm{Na}^+]_{\mathrm{i}})^{1.5}} \cdot \frac{[\mathrm{K}^+]_{\mathrm{e}}}{[\mathrm{K}^+]_{\mathrm{e}} + 1.5}$$

$$\sigma = \frac{1}{7}[\exp([\mathrm{Na}^+]_{\mathrm{e}}/67.3) - 1]$$

⟨ fill in this row 49 ⟩ +≡

```
{
  double fNaK, a, b, c, sigma;
  int t;
  for (t = 0; t < BERNUS_NTYPES; t++) {
    a = vm/RTonF;
    sigma = 0.1428571 * (exp(bpar→ce_Na[t]/67.3) - 1.0);
    fNaK = 1.0/(1.0 + 0.1245 * exp(-0.1 * a) + 0.0365 * sigma * exp(-a));
    a = KmNai/bpar→ci_Na[t];
    b = 1.0/(1.0 + sqrt(a * a * a));
    c = bpar→ce_K[t]/(bpar→ce_K[t] + KmKe);
    Memtab[tr][t][TC_I] += bpar→g_NaK[t] * fNaK * b * c;
    Memtab[tr][t][TC_NAK] = bpar→g_NaK[t] * fNaK * b * c;
  }
}
```

61. Found in Bernus' implementation, but not in the paper.

⟨ local variables of the initialization function 38 ⟩ +≡

```
double ksat = 0.1;      /* constants used in ⟨ fill in this row 49 ⟩ and for f_Ca */
double KmCa = 1.38;
double KCa = 0.0006;    /* half-maximum [Ca2+] binding concentration for I_Ca */
double KmKe = 1.5;
double KmNa = 87.5;
double KmNai = 10.0;
double eta = 0.35;
double RTonF = 8.3143 * 310.15/96.4867;    /* RT/F */
```

62. The $Na^{+}-Ca^{2+}$ exchanger.

$$I_{\mathrm{NaCa}} = g_{\mathrm{NaCa}}\, f_{\mathrm{NaCa}}$$

has one such factor

$$f_{\mathrm{NaCa}} = \left\{87.5^3 + [\mathrm{Na}^+]_\mathrm{e}{}^3\right\}^{-1} \cdot \left\{1.38 + [\mathrm{Ca}^{2+}]_\mathrm{e}\right\}^{-1} \cdot \left\{1 + 0.1\exp(-0.024V_\mathrm{m})\right\}^{-1} \cdot$$
$$\left\{[\mathrm{Na}^+]_\mathrm{i}{}^3[\mathrm{Ca}^{2+}]_\mathrm{e}\exp(0.013V_\mathrm{m}) - [\mathrm{Na}^+]_\mathrm{e}{}^3[\mathrm{Ca}^{2+}]_\mathrm{i}\exp(-0.024V_\mathrm{m})\right\}$$

```
#define pow2(x) ((x)*(x))
#define pow3(x) ((x)*(x)*(x))
⟨ fill in this row 49 ⟩ +≡
  {
    double a, b, c, INaCa;
    int t;
    for (t = 0; t < BERNUS_NTYPES; t++) {
      a = vm/RTonF;
      b = exp(eta*a);
      c = exp((eta-1.0)*a);
      a = bpar→g_NaCa[t]/((pow3(KmNa)+pow3(bpar→ce_Na[t]))*(KmCa+bpar→ce_Ca[t])*(1.0+ksat*c));
      INaCa = a*b*pow3(bpar→ci_Na[t])*bpar→ce_Ca[t] - a*c*pow3(bpar→ce_Na[t])*bpar→ci_Ca[t];
      Memtab[tr][t][TC_I] += INaCa;
      Memtab[tr][t][TC_NACA] = INaCa;
    }
  }
```

63. The table file. This is for debugging; the `membrane.dat` file will contain an extract from *memtab* in ascii format; it is comparable to `yinfini.dat` in Trudel's model.

⟨ open the table file 63 ⟩ ≡
```
{
  char *fname = "membrane_bernus.dat";
  if (¬(table_file = fopen(fname, "w"))) Error(1, "Impossible␣d'ouvrir␣le␣fichier␣%s.\n", fname);
}
```
This code is used in section 35.

64.
```
#define DEFAULT_CELL_TYPE 4    /* hackje (only for membrane table) */
```
⟨ print a line in the table file 64 ⟩ ≡
```
{
  int c;
  fprintf(table_file, "%.2f␣", vm);
  for (c = 0; c < TAB_COLS; c++) {
    fprintf(table_file, "%g␣", Memtab[tr][DEFAULT_CELL_TYPE][c]);
  }
  fprintf(table_file, "\n");
}
```
This code is used in section 35.

65. ⟨ close the table file 65 ⟩ ≡
```
{
  fclose(table_file);
}
```
This code is used in section 35.

66. ⟨ local variables of the initialization function 38 ⟩ +≡
```
FILE *table_file;
```

67. Bibliography.

[bernus02] O. Bernus, R. Wilders, C. W. Zemlin, H. Verschelde, and A. V. Panfilov. A computationally efficient electrophysiological model of human ventricular cells. *Am. J. Physiol. Heart Circ. Physiol.*, 282:H2296–H2308, 2002.

[hh52] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *J. Physiol.*, 117:500–544, 1952.

[hund01] Thomas J. Hund, Jan P. Kucera, Niels F. Otani, and Yoram Rudy. Ionic charge conservation and long-term steady state in the Luo–Rudy dynamic cell model. *Biophys. J.*, 81:3324–3331, 2001.

[oudit04zj] Gavin Y. Oudit, Rafael J. Ramirez, and Peter H. Backx. Voltage-regulated potassium channels. In Douglas P. Zipes and José Jalife, editors, *Cardiac Electrophysiology; From Cell to Bedside*, chapter 3. Saunders, Philadelphia, PA, 4th edition, 2004.

[priebe98] Leo Priebe and Dirk J. Beuckelmann. Simulation study of cellular electric properties in heart failure. *Circ. Res.*, 82:1206–1223, 1998.

[verkerk05] Arie O. Verkerk, Ronald Wilders, Eric Schulze-Bahr, Leander Beekman, Zahurul A. Bhuiyan, Jessica Bertrand, Lars Eckardt, Dongxin Lin, Martin Borggrefe, Günter Breithardt, Marcel M. A. M. Mannens, Hanno L. Tan, Arthur A. M. Wilde, and Connie R. Bezzina. Role of sequence variations in the human ether-a-go-go-related gene (HERG, KCNH2) in the Brugada syndrome. *Cardiovasc. Res.*, 68:441–453, 2005.

[volders99] Paul G. A. Volders, Karin R. Sipido, Edward Carmeliet, Roel L. H. M. G. Spätjens, Hein J. J. Wellens, and Marc A. Vos. Repolarizing K^+ currents I_{TO1} and I_{Ks} are larger in right than left canine ventricular midmyocardium. *Circulation*, 99:206–210, 1999.

68. Index.