Symbolic and Monte Carlo number keying error analysis

Harold Thimbleby
harold@thimbleby.net
Swansea University
United Kingdom

This file analyzes number keying errors, using both a symbolic approach and a Monte Carlo method.

To use for the first time: select everything, **Edit>Select All** and then choose **Evaluation>Evaluate Cells**, probably you can also use shift+return as a shortcut. Everything should work, and you will be asked whether you want to try running some slow calculations or to skip them; I recommend skipping them!

Once you have run everything, the data will be saved in a file, and you can recover it by executing the **Get** command, which you can find below. If you use **Get** in another session to restore data, don't forget that you must also do all initialization, else the rest of the code won't work properly!

If you are lucky, you will get lots of data and graph visualizations. Have fun.

Setup etc

This *Mathematica* notebook was developed using *Mathematica,* as follows:

```
In[1072]:= Print["Last run ", DateString@Date[],
             " using Mathematica " <> $Version]
```

```
Last run Sat 1 Aug 2009 02:25:22
  using Mathematica 6.0 for Mac OS X x86 (32-bit) (June 19, 2007)
```

How it works — overview

It's simplest to consider the Monte Carlo analysis first. Think of a number, then try keying it with certain probabilities of making keying errors. Treat that possibly erroneous key sequence like a typical gadget, and get a numeric value (which would be what a user would get using a typical device). The ratio of this value to the intended value will be *r*; thus there is an out by *r* error. Of course *r* may be 1, in which case there is no numerical error. We then classify the keyed number as syntactically valid (good) or invalid (bad). With a bit of tinkering, this allows us to plot graphs of the probability of an out by 10 error for various underlying probabilities of keying errors.

More precisely, for each number, the Monte Carlo analysis tries, for a range of probabilities (*e*=0.01 to 1.00), keying the number. This gives for each out by *r* error a vector of numbers, the frequencies those out by *r* errors occur in the simulation for each *e*. To be any use later in the analysis, these vectors are converted to polynomials in *e*.

Thinking of a random number is done as follows. Generate four decimal digits (from a uniform distribution), and convert to the form XX.XX. Remove leading and trailing zeros. This is now a number between 0.01 and 99.99 and in a valid form.

This *Mathematica* notebook also provides symbolic analyses. These are slower, and typically you won't want to wait around for a symbolic analysis of 0.01 to 99.99. The advantage is that the analysis is exact; here is how it is done.

Instead of choosing a single random number, for *every* valid number in a given format (e.g., 0.1 to 9.9), consider *every* way of keying it (rather than just one way, chosen probabalistically). For each way of keying it, work out the probability that the user would have keyed it that way. We know the underlying probability rate, *e*, and this calculation is purely symbolic. As with the Monte Carlo approach, it ends up with polynomials in *e*, but this time they are exact.

Assumptions

It could take forever if we model the user keying anything. Instead, when analyzing how the user keys a number, we only consider the user keying the correct number of keystrokes. We allow for the user prematurely terminating the number, however (because we treat termination as a keystroke with a certain probability), but we do not analyze if the user keys too much. In fact, if the user keys too much the digits they key will generally have less and less significance, if they follow a decimal point, and it's clear that if a keyed number is out by ten, keying more digits isn't going to stop it being out by ten. However, this assumption in our models means that we will miss some out by ten errors. Similarly, we do not define a number with too many keys as an invalid number. Perhaps ideally, we should have done both, but it would just make the analysis more complex without any obvious argument as to the benefits of better understanding what's going on.

Defining probabilities

Probability of hitting [0123456789.] in error is *p*; probability of hitting $ (representing the termination key, perhaps called ENTER) in error is *q*. We assume all digits/dot equiprobable.

We define *k*=*p*/*q* (*k* would be 2 for example if the user is twice as likely to press the wrong numeric key than end the number prematurely), and then for an overall error probability *e*=*p*+*q*, *e*=[0,1], we get:

In[1086]:=
```
Clear[e, p, q, k];
Solve[e == p + q && k == p / q, {q, p}] /. (a_ → b_) :→ (a = b);
Grid[{{Defer[p], "=", p}, {}, {Defer[q], "=", q}}] //
 TraditionalForm
```

Out[1088]//TraditionalForm=

$$p = \frac{e\,k}{k+1}$$

$$q = \frac{e}{k+1}$$

For my iPhone calculator, there are 11 digit keys and 4 ways to end a number, so we estimate *k*=7/4; on the Abbott *aimplus* pump, there are 11 digit keys and only 1 key ends a number, called [YES/ENTER] on the Abbott *aimplus*, so we estimate *k*=(11-1)/1:

In[1098]:=
```
k = 10;
```

Auxilliary functions *etc*

The obvious way to represent a keyed number is as a list of its keystrokes. A keystroke is coded as a number, with 0–9 being the digit keys 0–9, and decimal point being coded as 10, and the end of a number (which we call dollar) being coded as 11.

It's convenient to define names for the decimal and terminating codes, and a convenience function to convert one of these number lists into a printable string.

In[32]:=

```
dot = 10;
dollar = 11;
asString[list_] :=
  StringJoin @@ (Which[# == dot, ".", # == dollar, "$",
       True, ToString[#]] & /@ list);
```

For example,

In[706]:=

```
asString@{1, 2, 3, dot, 9, 0, dollar} // FullForm
```

Out[706]//FullForm=

```
"123.90$"
```

We need to know the value of a number and whether it is a syntax error. Notice that **value[]** provides a value like a typical calculator, even if the number has a syntax error.

In[36]:=

```
value[list_] := (*parse number; stops at end,
 or non-digit after first decimal point*)
 Module[{c, n = 0, len = Length[list], d},
  For[i = 1, i ≤ len, i++,
   c = list〚i〛;
   If[0 ≤ c && c ≤ 9, n = n * 10 + c, Break[]]
  ];
  If[c == dot,
   For[d = 10; i++, i ≤ len, i++,
    c = list〚i〛;
    If[0 ≤ c && c ≤ 9, n = n + c / d; d = d * 10, Break[]]
   ]
  ];
  N[n]
 ]
```

error[] returns true if the number is invalid by ISMP rules.

In[37]:=
```
error[list_] :=
 Module[{n, dots},
  n = TakeWhile[list, # ≠ dollar &];
  If[Length[n] == 0, Return[True]];
  dots = Count[n, dot];
  dots > 1 || If[dots == 0, First[n] == 0,
    First[n] == dot || Last[n] == dot || Last[n] == 0]
 ]
```

For example, here's a table of various numbers:

In[38]:=
```
Grid@Map[
  {asString[#], "=", value[#], If[error[#], " invalid", "ok"]} &,
  {{1, dot, 2}, {1, dot, 2, dot, 3},
   {1, dot, 2, dollar, 9, dot}, {dot, 2}, {0, dot, 1},
   {6, 0, dot, 0}, {1, 2, 0}, {0, 1, 2}, {1, dot, dollar}}]
```

Out[38]=
```
  1.2    =  1.2     ok
 1.2.3   =  1.2   invalid
1.2$9.   =  1.2     ok
  .2     =  0.2   invalid
  0.1    =  0.1     ok
 60.0    =  60.   invalid
  120    = 120.     ok
  012    =  12.   invalid
  1.$    =   1.   invalid
```

The dose error is the ratio of the intended and actual keyed doses. Call the ratio *t*; we need to track the probability of each event *t*, which reprsents an out by *t* error. To program that we keep an array *b* with *scale* bins, such that *b*[*i*] will accummulate the probability of out by *t* events where **Round[**scale*t**]**=*i*. Since *Mathematica* arrays have upper bounds, any *t*≥ *huge*/*scale* is mapped to *huge*. For all our graph plotting, we're only really interested in plots to *t*=10, so huge is set to 12*scale*. (For our purposes we just need a large enough value of *scale* to make the graphs look smooth.)

In[1082]:=
```
scale = 100;
huge = 12 scale;
makeArray[] := makeArray[0];
makeArray[n_] := Table[n, {huge}];
```

With a scale of 100, the bins will be 1/*scale*=0.01 wide.

Having got the bins, we may want to find values in them! Given lists of unsorted {*x*,*y*} data, interpolate to find the point closest to a given *x* or *y*. Naff algorithm, but it works reliably with few assumptions.

In[42]:=
```
xintercept[n_, list_] := Module[{lo, hi},
  lo = Last@Sort[Cases[list, {_, _?(# ≤ n &)}], #1⟦2⟧ < #2⟦2⟧ &];
  hi = First@Sort[Cases[list, {_, _?(# ≥ n &)}], #1⟦2⟧ < #2⟦2⟧ &];
  If[lo⟦2⟧ == n, {lo⟦1⟧, n},
   {lo⟦1⟧ + (hi⟦1⟧ - lo⟦1⟧) (n - lo⟦2⟧) / (hi⟦2⟧ - lo⟦2⟧), n}]
 ]
```

In[43]:=
```
yintercept[n_, list_] := Module[{lo, hi},
  lo = Last@Sort[Cases[list, {_?(# ≤ n &), _}], #1⟦1⟧ < #2⟦1⟧ &];
  hi = First@Sort[Cases[list, {_?(# ≥ n &), _}], #1⟦1⟧ < #2⟦1⟧ &];
  If[lo⟦1⟧ == n, {n, lo⟦2⟧},
   {n, lo⟦2⟧ + (hi⟦2⟧ - lo⟦2⟧) (n - lo⟦1⟧) / (hi⟦1⟧ - lo⟦1⟧)}]
 ]
```

Example:

In[44]:=
```
xintercept[4, {{1, 2}, {2, 3}, {4, 5}, {6, 7}}]
```

Out[44]=
```
{3, 4}
```

Generate data

The code works by generating all numbers of a particular correct syntactic form (say, 10 to 99), and for each such number (say 23) generating all possible ways of keying it. This is done by generating all base 12 numbers and converting them to lists — these are then all possible key sequences, usually called dj in the code. For each key sequence we work out its value and (given *p* and *q*) the probability that the user would have keyed it.

Each syntactic form keeps to arrays of bins: good and bad (or *g* and *b* in the code). It would be nice to track out by *t* errors, but that would be slow; instead we track out by "exactly" *t*, to within the precision given by the scale of the bins.

Good tracks the out by 'exactly' *t* errors that are not syntax errors, and *bad* tracks the out by *t* errors that are syntax errors. All the functions below then returns the pair of arrays (along with some other descriptive stuff) an appends to the vector *data*.

Make some constants, so we don't keep repeating these sums in inner loops:

In[1102]:=
```
ok = 1 - p - q;
poverten = p / 10;
```

In[53]:=
```
SetAttributes[update, HoldAll];
update[dj_, b_, g_, val_, prob_] :=
  (*
  dj is what was keyed,
  b and g are the bad and good bin arrays,
  val is the value of the intended number,
  prob is the probability of keying dj
  *)
  Module[
   {t = value[dj]}, (* t is the device'
    s idea of the value of the keyed number *)

   (* set t to the scaled out by t ratio so it
    can be used directly as an index into the bins;
   if the scaled t is out of range, make it huge *)
   t = If[t == 0 || (t = scale * Max[val / t, t / val]) ≥ huge,
     huge, Round[t]];

   (* if the keyed number is invalid,
   add to bad bin, else add to good bin *)
   If[error[dj], b⟦t⟧ += prob, g⟦t⟧ += prob]
  ];
```

Clear data to initialize for a new run…

In[982]:=
```
data = {};
```

Symbolic: 1 digit integer, 1-9

Having done all that, the easiest code to explain in detail is to analyze all possible ways of keying a single digit. Please see all the comments below; all further symbolic approaches use the same method.

In[983]:=
```
AppendTo[data,
  Module[{g, b, i, j, dj, prob, val, entries = 0},
   g = makeArray[]; (* generate empty bins *)
   b = makeArray[];
   Print@ProgressIndicator[Dynamic[i / 10]];
   For[i = 1, i < 10, i++, (* generate correct numbers 1 to 9 *)
    val = i;
    (* the value of a single digit is easy: it's itself *)
    entries++;
    For[j = 0, j < 12, j++,
     (* all possible ways j of keying it *)
     (* calculate the probability of keying this version *)
     prob =
      If[
       j == dollar, q, (* dollar is always incorrect *)
       If[j ≠ i, poverten,
        (* there are 10 ways of being unlike j apart from $ *)
        ok (* the digit keyed is correct,
        with probability ok *)
       ]
      ];
     dj = {j}; (* for consistency, convert j to {j} which
      is the format we use everywhere else below *)
     update[dj, b, g, val, prob](* update
      the good and bad bins *)
    ]
   ];
   {"good" → g, "bad" → b,
    "description" → "1 digit with no decimal point, 1-9",
    "size" → entries, "type" → "Partial"}
  ]
 ];
```

[redacted]

Symbolic: 2 digit integer, 10-99

In[984]:=
```
AppendTo[data,
  Module[{g, b, i, di, j, dj, prob, k, val, entries = 0},
   g = makeArray[];
   b = makeArray[];
   Print@ProgressIndicator[Dynamic[i / 90]];
   For[i = 10, i < 100, i++,
    di = IntegerDigits[i]; (* what user should key *)
    val = i;
    entries++;
    For[j = 0, j < 12^2, j++,
     dj = PadLeft[IntegerDigits[j, 12], 2];
     (* what they do key *)
     (* what is prob of typing dj? *)
     prob = 1;
     For[k = 1, k ≤ 2, k++,
      prob *=
       If[
        dj⟦k⟧ == dollar, q,
        If[dj⟦k⟧ ≠ di⟦k⟧, poverten,
         ok
        ]
       ]
     ];
     update[dj, b, g, val, prob]
    ]
   ];
   {"good" → g, "bad" → b,
    "description" → "2 digit integer, 10-99",
    "size" → entries, "type" → "Partial"}
  ]
 ];
```

Symbolic: 2 digits with a decimal point, 0.1-9.9 (obviously excluding 1.0, 2.0 which are invalid etc)

Unlike the previous examples, converting *i* to the numeric value isn't so trivial; we use value[] as defined above.

In[985]:=

```
AppendTo[data,
  Module[{g, b, i, di, j, dj, prob, k, val, entries = 0},
   g = makeArray[];
   b = makeArray[];
   Print@ProgressIndicator[Dynamic[i / 90]];
   For[i = 1, i < 10^2, i++,
    If[Mod[i, 10] == 0, Continue[]];
    (* easy way to ensure doesn't end in zero *)
    di = PadLeft[IntegerDigits[i], 2];
    (* what user should key *)
    di = {di⟦1⟧, dot, di⟦2⟧};
    val = value[di];
    entries++;
    For[j = 0, j < 12^3, j++,
     dj = PadLeft[IntegerDigits[j, 12], 3];
     (* what they do key *)
     (* what is prob of typing dj? *)
     prob = 1;
     For[k = 1, k ≤ 3, k++,
      prob *=
       If[
        dj⟦k⟧ == dollar, q, (* dollar is always unlike dj[k],
        as there is no $ in it *)
        If[dj⟦k⟧ ≠ di⟦k⟧, poverten,
         ok
        ]
       ]
     ];
     update[dj, b, g, val, prob]
    ]
   ];
   {"good" → g, "bad" → b,
    "description" → "2 digits with a decimal point, 0.1–9.9",
    "size" → entries, "type" → "Partial"}
  ]
 ];
```

Symbolic: 3 digits with a decimal point after second digit, 10.1-99.9 (obviously excluding 10-99 integers)

In[986]:=

```
AppendTo[data,
  Module[{g, b, i, di, j, dj, prob, k, t, val, entries = 0},
   g = makeArray[];
   b = makeArray[];
   Print@ProgressIndicator[Dynamic[i / 1000]];
   For[i = 101, i < 1000, i++,
    di = IntegerDigits[i]; (* what user should key *)
    If[di⟦3⟧ == 0, Continue[]];
    di = {di⟦1⟧, di⟦2⟧, dot, di⟦3⟧};
    val = value[di];
    entries++;
    For[j = 0, j < 12^3, j++,
     dj = PadLeft[IntegerDigits[j, 12], 3];
     (* what they do key *)
     (* what is prob of typing dj? *)
     prob = 1;
     For[k = 1, k ≤ 3, k++,
      prob *=
       If[
        dj⟦k⟧ == dollar, q, (* dollar is always unlike dj[k],
        as there is no $ in it *)
        If[dj⟦k⟧ ≠ di⟦k⟧, poverten,
         ok
        ]
       ]
     ];
     update[dj, b, g, val, prob]
    ]
   ];
   {"good" → g, "bad" → b, "description" →
     "3 digits with a decimal point after second digit,
      10.1-99.9", "size" → entries, "type" → "Partial"}
  ]
 ];
```

[redacted]

Symbolic: 1 to 4 digits with or without a decimal point, 0.01-99.99 (takes ages!)

This code, below, is disabled because it is so slow: it's going to take much longer than all of the other bits of code put together. With the **False** in the **If**, you can thus safely execute everything in this file, and it will work in a reasonable time (like, 10 minutes) — provided you answer the **ChoiceDialog** box question!

In[590]:=

```
If[ChoiceDialog[
   "Handling 0.01-99.99 symbolically is very slow.\n\nDo you
```

```
      really want to do it (and wait days perhaps)?",
    {"No" → False, "Yes" → True}],
  AppendTo[data,
   Module[{g, b, i, di, j, dj, k,
      prob, val, entries = 0, cases, keyPresses},
     g = makeArray[];
     b = makeArray[];
     Print@ProgressIndicator[Dynamic[i / 10 000]];
     For[i = 1, i < 10 000, i++,
      di = PadLeft[IntegerDigits[i], 4];
      (* remove trailing zeros after decimal point;
      and remove decimal point if two trailing zeros *)
      If[di⟦4⟧ ≠ 0,
       di = {di⟦1⟧, di⟦2⟧, dot, di⟦3⟧, di⟦4⟧},
       If[di⟦3⟧ ≠ 0,
        di = {di⟦1⟧, di⟦2⟧, dot, di⟦3⟧},
        If[di⟦1⟧ ≠ 0,
         di = {di⟦1⟧, di⟦2⟧},
         di = {di⟦2⟧}
        ]
       ]
      ];
      If[di⟦1⟧ == 0, di = Rest[di]]; (* delete any leading zero *)
      (* di is now a list of what user should key *)
      keyPresses = Length[di];
      cases = 12^keyPresses;
      val = value[di];
      entries++;
      For[j = 0, j < cases, j++,
       dj = PadLeft[IntegerDigits[j, 12], keyPresses];
       (* what is actually keyed *)
       (* what is prob of typing dj? *)
       prob = 1;
       For[k = 1, k ≤ keyPresses, k++,
        prob *=
         If[
          dj⟦k⟧ == dollar, q,
          If[dj⟦k⟧ ≠ di⟦k⟧, poverten,
           ok
          ]
         ]
       ];
       update[dj, b, g, val, prob]
      ]
     ];
     {"good" → g, "bad" → b, "description" →
       "upto 4 digits with or without a decimal point,
         0.01–99.99", "size" → entries, "type" → "All"}
    ]
   ]
  ];
```

Monte Carlo: 1 to 4 digits, 0.01-99.99

All the methods above systematically generate all possible numbers in the given range and format, then try all possible ways of keying them. This is slow! An alternative method is to use a Monte Carlo approach. Rather than generate all numbers, we generate numbers randomly, and we try keying them, but making random keying errors with the probabilities *p* and *q*. The longer the program is allowed to run, the more accurate its results. To generate numbers, there has to be a known *e*; it cannot be done symbolically. So we generate data for particular values of *e*, then use *Mathematica* to generate interpolation functions in terms of *e*; this gets us back to the exact same format of data we have for the purely symbolic approaches — except, instead of an expression in *e*, we have a expression like F[*e*], where F is some *Mathematica* function that fits the data.

In[561]:=
```
keyError[k_] :=
  RandomChoice[DeleteCases[{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, dot}, k]];
```

In[1028]:=
```
AppendTo[data,
  Module[{g, b, di, dj, steps = 100, eval = 0, prob,
    i, val, keyPresses, monte, t, experiments = 2000},

   Print@ProgressIndicator[
     Dynamic[monte / (experiments * steps) + eval / steps]];

   Print["Generate ", experiments,
    " random numbers, repeating with e = 0 to 1 in steps of ",
    N[1 / steps]];

   g = makeArray[Table[0, {steps + 1}]];
   b = makeArray[Table[0, {steps + 1}]];

   For[eval = 0, eval ≤ steps, eval++,
    Block[{e = eval / steps},
     For[monte = 1, monte < experiments, monte++,
      (* generate random number 0-99.9, with 0-1 decimal digits
          di=
        PadLeft[IntegerDigits[10*RandomInteger[{1,1000}]],4];
      *)

      (* generate random number 0-
       99.99 with 0-2 decimal digits *)
      di = PadLeft[IntegerDigits[10 * RandomInteger[{1, 1000}]],
        4];

      (* remove trailing zeros after decimal point;
      and remove decimal point if two trailing zeros *)
      If[di⟦4⟧ ≠ 0,
       di = {di⟦1⟧, di⟦2⟧, dot, di⟦3⟧, di⟦4⟧},
       If[di⟦3⟧ ≠ 0,
        di = {di⟦1⟧, di⟦2⟧, dot, di⟦3⟧},
        If[di⟦1⟧ ≠ 0,
         di = {di⟦1⟧, di⟦2⟧},
         di = {di⟦2⟧}
        ]
       ]
      ];
      If[di⟦1⟧ == 0, di = Rest[di]];
      (* delete any leading zero *)

      (* di is now a list of what user should key;
      try keying it to make dj *)
      keyPresses = Length[di];
      val = value[di];
      (* now try keying di *)
      dj = If[e == 0, di,
        If[e < 1,
         Table[RandomChoice[{p, q, ok} →
            {keyError[di⟦i⟧], dollar, di⟦i⟧}], {i, keyPresses}],
         Table[RandomChoice[{p, q} → {keyError[di⟦i⟧], dollar}],
          {i, keyPresses}],
        ]];
      t = value[dj]; (* t is the device'
       s idea of the value of the keyed number *)

      (* set t to the scaled out by t ratio so it
       can be used directly as an index into the bins;
      if the scaled t is out of range, make it huge *)
      t = If[t == 0 || (t = scale * Max[val / t, t / val]) ≥ huge,
        huge, Round[t]];

      (* if the keyed number is invalid,
      add to bad bin, else add to good bin *)
      If[error[dj], b⟦t, eval + 1⟧ += 1, g⟦t, eval + 1⟧ += 1]
     ]
    ]
   ];
   {"good" → g, "bad" → b,
    "description" → "Monte Carlo experiments,
       generating randomly upto 4 digits with or without a
       decimal point, 0.01-99.99",
    "size" → experiments, "type" → "All"}
  ]
 ];
```

[redacted]

Generate 2000 random numbers, repeating with e = 0 to 1 in steps of 0.01

Import/export data — generally a lot faster than regenerating it next time!

Save any data that has been worked out.

In[1032]:=
```
Module[{file = Close[OpenWrite["data.m"]]},
  Print["Data is stored in: ", Directory[], "/", file];
  Save[file, data];
  Print["Data last saved: ", DateString@Date[]]
 ];
```

```
Data is stored in: /Users/harold/data.m
```

```
Data last saved: Sat 1 Aug 2009 02:16:51
```

Recover data from data file:

In[635]:=
```
Clear[data];
Get["data.m"];
```

See what you have got:

In[1030]:=
```
Scan[Print["You have data defined for: ", "description" /. #] &,
  data];
```

```
You have data defined for: 1 digit with no decimal point, 1–9
```

```
You have data defined for: 2 digit integer, 10–99
```

```
You have data defined for: 2 digits with a decimal point, 0.1–9.9
```

```
You have data defined for: 3 digits with a decimal point after second digit, 10.1–99.9
```

```
You have data defined for:
 Monte Carlo experiments, generating randomly upto 4 digits with or
   without a decimal point, 0.01–99.99
```

Combine available results with probabilities

Since it's tedious to wait for all those results, the code below will combine whichever results have got values, and then average them weighted by how many values each range of data covers.

Data is a vector with elements in the format: {"good" → goodBins, "bad" → badBins, "description" → "e.g., 3 digits with a decimal point after second digit, 10.1-99.9", "size" → entries, "type" → "Partial"}. The goodBins and badBins will contain symbolic expressions in *e*, or if generated by a Monte Carlo process, they will be vectors of data. If vectors, the process below notices the **Head** of the data is **List** and then uses *Mathematica*'s Interpolation function to convert the vectors to functions in *e*.

At the end of the following code, the cumulative probabilities (i.e., normalized frequencies) are in the vectors *prevented* and *notPrevented* and now represented as pairs {*x*, *y*}, where *y* is the probability of an out by *x* error; note that *y* is a function of *e*.

In[1027]:=
```
Module[
 {dg, db, i, j, k, intj, intk, validData = 0, steps, types = {}},
 Scan[AppendTo[types, "type" /. #] &, data];
 If[types == {},
  MessageDialog["No data has been generated yet!"];
  Return[]
 ];
 If[MemberQ[types, "All"],
  (* choose one of them, if any *)
  (*Print["You have ",
    Count[types,"All"]," complete data sets"];*)
  i = {"Other data" → {}};
  Scan[If["All" == ("type" /. #),
     AppendTo[i, ("description" /. #) → #]] &, data];
  i = ChoiceDialog["You have some complete data sets.\n\nWhich
      complete sets do you want to analyze, if any?", i];
  If[i ≠ {},
   validData = "size" /. i;
   {dg, db} = {"good" /. i, "bad" /. i} / validData;
  ];
 ];
 If[validData == 0 && MemberQ[types, "Partial"],
  Print["You have ", Count[types, "Partial"],
   " partial data sets, which will be combined"];
  (* combine what defined data there are *)
  Scan[
   If["Partial" == ("type" /. #),
     Print["Using ", "description" /. #,
      " values covering ", "size" /. #, " samples"];
     t = {"good" /. #, "bad" /. #};
     If[validData > 0, {dg, db} += t, {dg, db} = t];
     validData += "size" /. #;
    ] &,
   data];
  {dg, db} /= validData;
 ];

 If[validData == 0,
  MessageDialog["You haven't selected any data to analyze!"],
  Print[validData, " values combined..."];
  prevented = makeArray[];
  notPrevented = makeArray[];
  j = k = 0.0;
  For[i = huge, i ≥ 1, i--,
   j += db⟦i⟧;
   k += db⟦i⟧ + dg⟦i⟧;
   {intj, intk} =
    (* if the Monte Carlo method was used,
    j and k will be vectors *)
    If[Head[j] === List,
     (* values must be interpolated *)
     steps = Length[j] - 1;
     (* MapThread[f[{##}][e]&, *)
     Interpolation[##, InterpolationOrder → 2][e] & /@
      Map[MapIndexed[Function[{x, y},
          {N[(y⟦1⟧ - 1) / steps], x}], #] &, {j, k}],
     {j, k}];
   prevented⟦i⟧ = {N@i / scale, intj};
   notPrevented⟦i⟧ = {N@i / scale, intk};
  ]
 ];

 (* the first few entries correspond to out by factors <
  1 (and will be zero) so they are deleted *)
 prevented = Drop[prevented, scale - 1];
 notPrevented = Drop[notPrevented, scale - 1];

 (* simplification makes
   everything later much faster later... *)
 prevented = Simplify@prevented;
 notPrevented = Simplify@notPrevented;
]
```

```
You have 4 partial data sets, which will be combined
```

```
Using 1 digit with no decimal point, 1–9 values covering 9 samples
```

```
Using 2 digit integer, 10–99 values covering 90 samples
```

```
Using 2 digits with a decimal point, 0.1–9.9 values covering 90 samples
```

```
Using 3 digits with a decimal point after second digit, 10.1–99.9
  values covering 810 samples
```

```
999 values combined...
```

Visualize out by 10 results

This is what the data looks like; you can see it is symbolic, using expressions in *e*. We now want to study out by 10 errors in particular.

In[988]:=
```
good = yintercept[10, prevented]⟦2⟧;
bad = yintercept[10, notPrevented]⟦2⟧;
{good, bad} // ColumnForm
```

Out[990]=

$$0. + \frac{397\,e}{1\,221} - \frac{673\,e^2}{4\,477} + \frac{5\,672\,e^3}{147\,741}$$

$$0. + \frac{349\,e}{407} - \frac{5\,014\,e^2}{13\,431} + \frac{8\,612\,e^3}{147\,741}$$

Red lines in graphs below represent conventional devices; green lines are error-blocking devices (and therefore have better out by 10 performance).

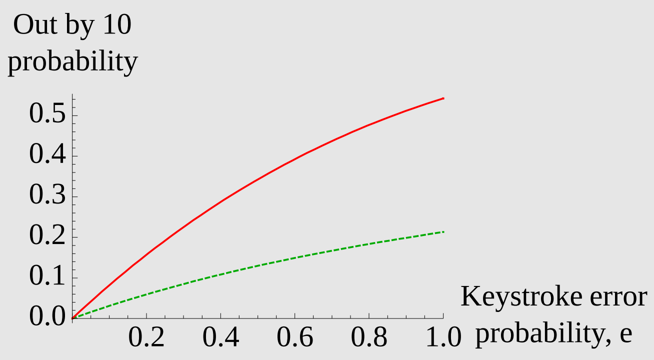
How do we want to visualize results? The **ImageSize→1000** parameter in **Plot** (if not commented out) is useful if you want to cut-and-paste the graphics to (say) PowerPoint, as it makes it bigger and of a consistent size. For presentations, we probably want to provide axes labels in the presentation tool rather than in *Mathematica*. Hence there is a flag **Presentation**. Set it to **False** if you want to live in *Mathematica* and get axes labels; set it to **True** if you want to cut-and-paste plots to other applications.

In[968]:=
```
Presentation = ChoiceDialog[
   "Do you want to have labeled graphs, or to have plain
     graphs at a large, standard size for presentations?",
   {"Axes labeled" → False, "No axes, standard size" → True}];
Print["Graphs formatted ", If[Presentation,
  "large, no axes labels", "with labelled axes"], "."]
```

```
Graphs formatted with labelled axes.
```

In[991]:=
```
Plot[
 {good, bad}, {e, 0, 1},
 AxesOrigin → {0, 0},
 PlotStyle →
  {Directive[Darker[Green], Dashing[{.01, .01}]], Red},
 PlotRange → {{0, 1}, Automatic},
 BaseStyle → {Thickness[.005], 14},
 Evaluate@Sequence[If[Presentation, ImageSize → 1000,
    AxesLabel → {"Keystroke error\nprobability, e",
      "Out by 10\nprobability"}]]
]
```

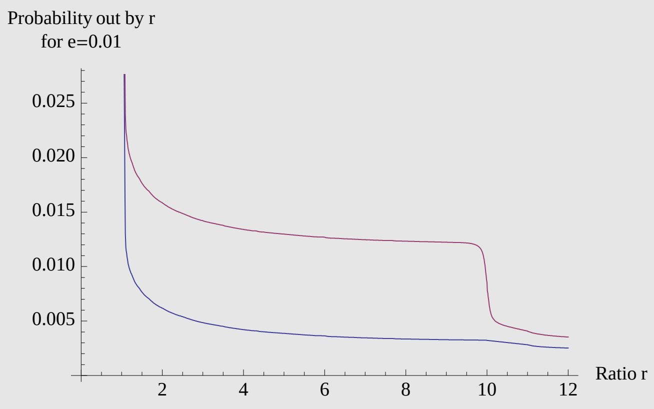
Out[991]=



In[992]:=

```
Block[{e = .01},
 ListPlot[{prevented, notPrevented},
  AxesOrigin → {0, 0}, Joined → True,
  Evaluate@Sequence[If[Presentation,
     ImageSize → 1000, AxesLabel → {"Ratio r",
       "Probability out by r\nfor e=" <> ToString[e]}]]]
]
```

Out[992]=



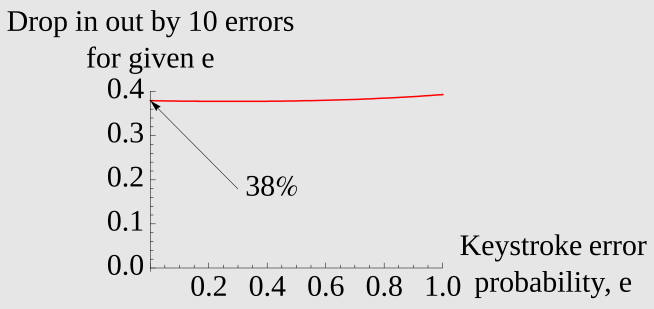
Note. *Mathematica* does not plot the Monte Carlo model well, as the randomness doesn't handle *e*→0 very well.

What effective improvement does it give? The graph below shows the saving as *e* tends to zero; it doesn't work well with Monte Carlo data because of randomness.

In[1043]:=

```
Module[{r = good / bad /. e → 0.001, xpos = .3},
 Plot[good / bad, {e, 0, 1},
  AxesOrigin → {0, 0},
  PlotStyle → {Darker[Green], Red},
  PlotRange → {{0, 1}, Automatic},
  BaseStyle → {Thickness[.005], 14},
  Epilog → {Thickness[.001], Arrow[{{xpos, r - .2}, {0, r}}],
    Text[" " <> ToString[Round[100 r, 1]] <> "%",
     {xpos, r - .2}, {-1, 0}]},
  Evaluate@Sequence[If[Presentation, ImageSize → 1000,
     AxesLabel → {"Keystroke error\nprobability, e",
      "Drop in out by 10 errors\nfor given e"}
    ]]
 ]
]
```

Out[1043]=

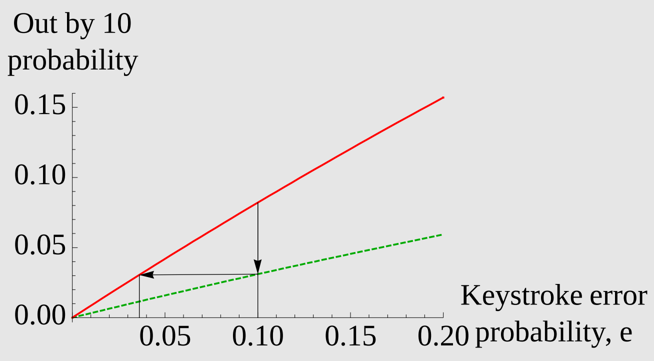


For a given value of *b* (say 0.01) we want to find the *e* such that bad[e]=good[b], so we can draw a nice graph. Although *Mathematica* has a **Solve** function, it doesn't work well on **InterpolationFunctions**, which result from the Monte Carlo analysis. Here, we use a simple binary search — it's just run often enough to be accurate enough.

In[994]:=
```
BinarySearch[f_, val_] :=
 Module[{lo = 0, hi = 1, h, n},
  Do[
   h = ((f /. e → lo) + (f /. e → hi)) / 2;
   If[h < val, lo = (lo + hi) / 2, hi = (lo + hi) / 2],
   {10}];
  N@lo
 ]
```

In[995]:=
```
Module[{b = 0.1, bb},
 bb = BinarySearch[bad, good /. e → b];
 Plot[{good, bad}, {e, 0, 2 b},
  AxesOrigin → {0, 0},
  PlotStyle →
   {Directive[Darker[Green], Dashing[{.01, .01}]], Red},
  PlotRange → {{0, 2 b}, Automatic},
  BaseStyle → {Thickness[.005], 14},
  Epilog → {
    Thickness[.002],
    Line[{{b, 0}, {b, bad /. e → b}}],
    Arrow[{{b, bad /. e → b}, {b, good /. e → b}}],
    Arrow[{{b, good /. e → b}, {bb, bad /. e → bb}}],
    Line[{{bb, 0}, {bb, bad /. e → bb}}]
   },
  Evaluate@Sequence[If[Presentation, ImageSize → 1 000,
     AxesLabel → {"Keystroke error\nprobability, e",
       "Out by 10\nprobability"}
    ]]
 ]
]
```

Out[995]=



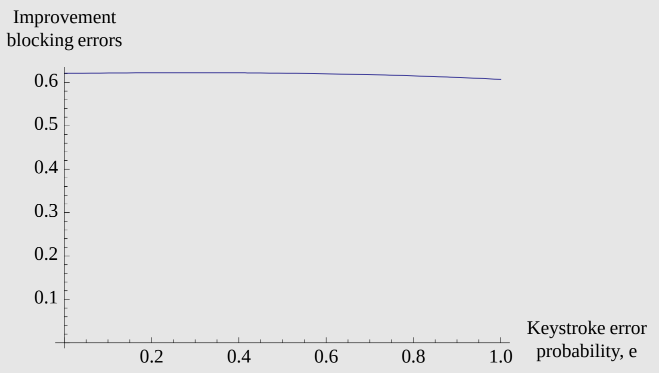
Visualize improvement

In[1044]:=
```
saving = 1 - good / bad;
monteCarlo[] := MemberQ[improvementData,
  InterpolatingFunction, Infinity, Heads → True]
```

For the Monte Carlo data, this will give a bumpy plot!

In[1046]:=
```
Plot[saving, {e, 0.001, 1},
 AxesOrigin → {0, 0},
 Evaluate@Sequence[If[Presentation, ImageSize → 1000,
    AxesLabel → {"Keystroke error\nprobability, e",
      "Improvement\nblocking errors"}
   ]]]
```

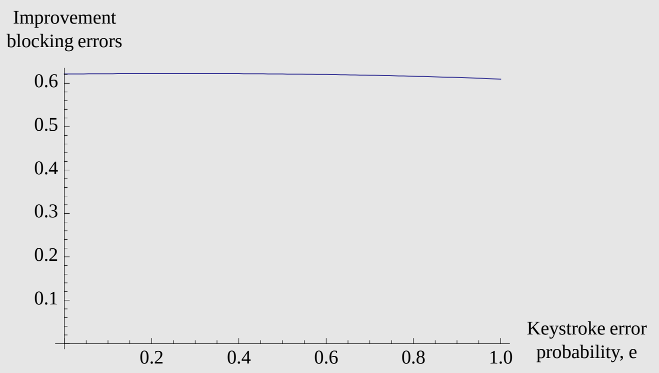
Out[1046]=



This smooths the plot — useful for Monte Carlo data:

In[1047]:=
```
ListPlot[MovingAverage[Table[saving, {e, 0.001, 1, 0.01}], 10],
 Joined → True, AxesOrigin → {0, 0}, DataRange → {0, 1},
 Evaluate@Sequence[If[Presentation, ImageSize → 1000,
    AxesLabel → {"Keystroke error\nprobability, e",
      "Improvement\nblocking errors"}
   ]]]
```

Out[1047]=



In[1059]:=
```
typicalSaving = If[monteCarlo[],
   MovingAverage[Table[saving, {e, 0, 1, 0.01}], 10][[1]],
   saving /. e → 0.001];
CellPrint[Cell["The significance of blocking " <>
   ToString[Round[100 improvement, .1]] <>
   "% of out by 10 errors...", "Subtitle"]]
```

The significance of blocking 62.1% of out by 10 errors...

Original figures below are taken from Vicente *et al* (2003).

We convert to per year (divide by 12), convert to UK population (60,943,912), based on US population (303,824,640). The Abbott pump represents (Vicente, correspondence) 75% of the US market, so we could legitimately scale by 4/3 to get representative rate for all PCA devices.

If typing error rates are causing a rate of **saving** bad doses that can be detected, then the detection rate will eliminate some fatalities:

In[1061]:=

```
italicize[n_List] := Map[italicize, n];
italicize[n_] := If[StringQ[n], Style[n, Italic], n];
embolden[n_List] := Map[embolden, n];
embolden[n_] :=
  If[StringQ[n], Style[n, Bold, 20, FontFamily → "Helvetica"], n];
display[d_, title_] :=
  Print@Text@Grid[{
      {embolden@title, SpanFromLeft, SpanFromLeft,
       SpanFromLeft, SpanFromLeft, SpanFromLeft},
      italicize@{"", "", "Estimates of true incidence", ,
        "Estimates of prob", SpanFromLeft},
      italicize@{"", "Reported deaths", "Low",
        "High", "Low", "High"},
      Prepend[d⟦1⟧, italicize@"Min"],
      Prepend[d⟦2⟧, italicize@"Max"]},
     Alignment →
      {{Center, Right, Right, Right, Right, Right}, Baseline},
     Dividers → {All, {1 → True, 3 → True, -1 → True}}];
vicente =
  {
   {5, 65, 417, 2.95 × 10^-6, 1.89 × 10^-5},
   {8, 104, 667, 4.72 × 10^-16, 3.03 × 10^-5}
  };
display[vicente, "Vicente original data 12 year, USA"];
display[vicentePA =
   Map[Round[# * {c, c, c, 1, 1} /. c → (1 / 12) / 303.824640, .001] &,
    vicente],
  "Fatalities: scaled per 1M population pa for one PCA"];
display[Map[Round[# * {typicalSaving, typicalSaving,
      typicalSaving, 0, 0}, .0001] &, vicentePA],
 "Prevented fatalities per 1M pa for one PCA device"]; display[
 Map[Round[# * {typicalSaving, typicalSaving, typicalSaving, 0, 0},
    .0001] &, 60.943912 * vicentePA],
 "Prevented fatalities UK pa for one PCA device"];
```

Vicente original data 12 year, USA

		Estimates of true incidence		*Estimates of prob*	
	Reported deaths	*Low*	*High*	*Low*	*High*
Min	5	65	417	2.95×10^{-6}	0.000 018 9
Max	8	104	667	4.72×10^{-16}	0.000 030 3

Fatalities: scaled per 1M population pa for one PCA

		Estimates of true incidence		*Estimates of prob*	
	Reported deaths	*Low*	*High*	*Low*	*High*
Min	0.001	0.018	0.114	0	0
Max	0.002	0.029	0.183	0	0

Prevented fatalities per 1M pa for one PCA device

		Estimates of true incidence		*Estimates of prob*	
	Reported deaths	*Low*	*High*	*Low*	*High*
Min	0.000 6	0.011 2	0.070 8	0	0
Max	0.001 2	0.018	0.113 6	0	0

Prevented fatalities UK pa for one PCA device

		Estimates of true incidence		*Estimates of prob*	
	Reported deaths	*Low*	*High*	*Low*	*High*
Min	0.037 8	0.681	4.313 3	0	0
Max	0.075 7	1.097 2	6.924	0	0

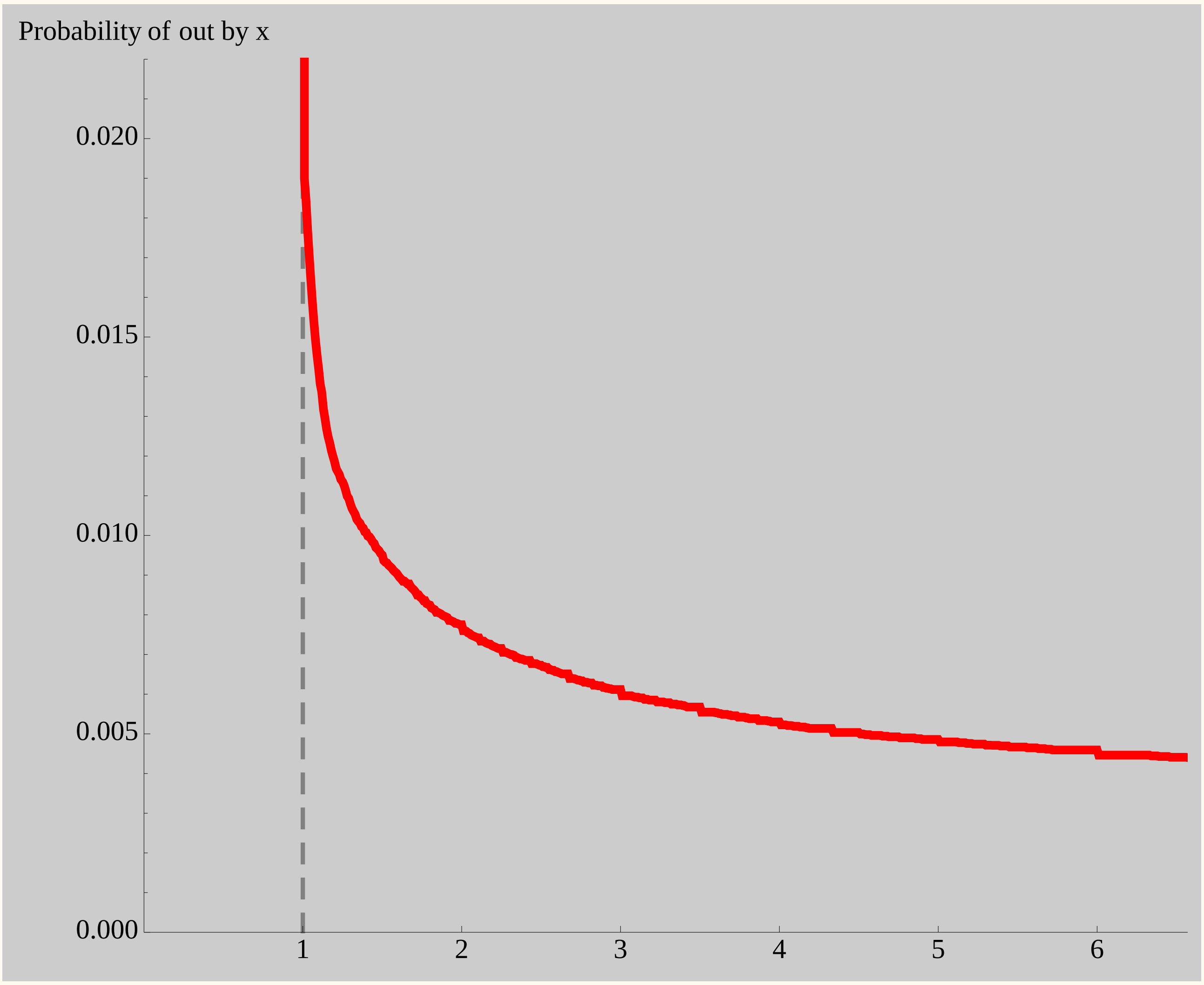
Visualizing the data another way ...

zapNullOptions[] takes a list of option values, and is a handy way to clear empty options.

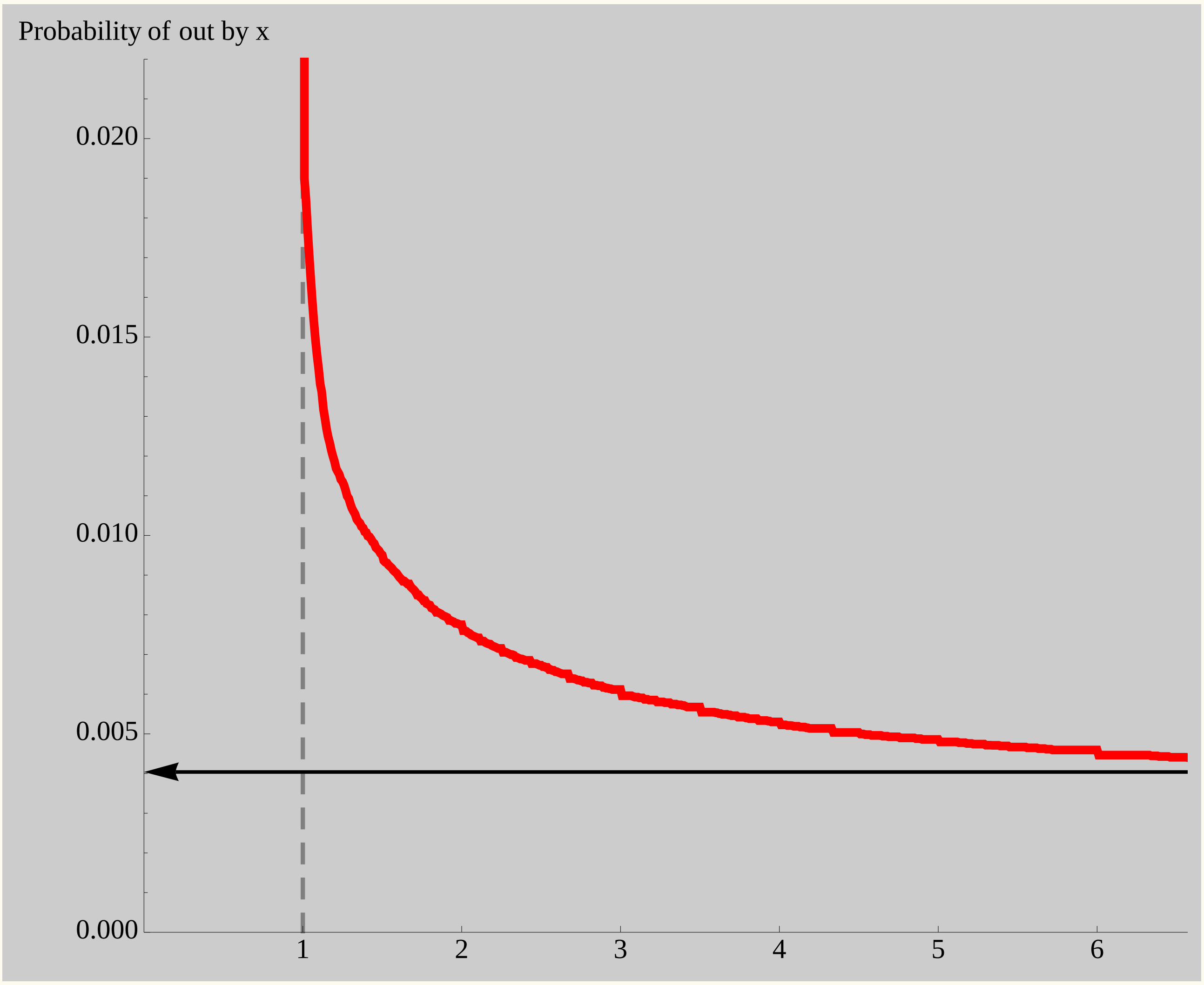
```
In[709]:=
zapNullOptions[n_] :=
  DeleteCases[n, False → _] /. (True → arg_) → arg;
```

explainPlot[], below, draws a sequence of graphs to help explain the data. Useful for PowerPoint talks!

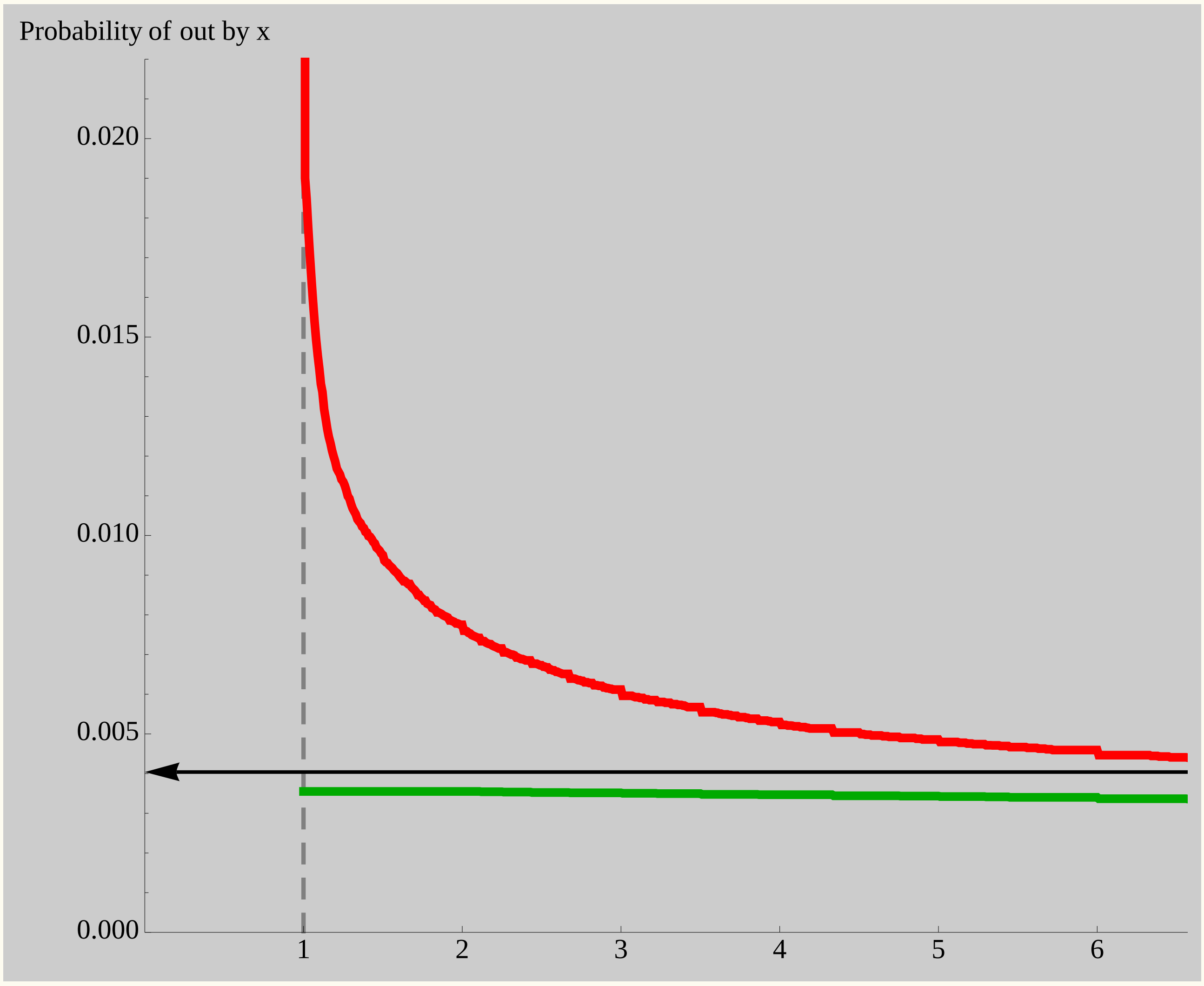
In[1070]:=
```
explainPlot[detail_] :=
  Module[{ratio = 10, icpt, nppt, fig, explain, saving},
   fig[n_] := ToString[Round[n, 0.001]];
   explain[t_, xy_] :=
    Module[{dh = 1},
     {Text[t, xy + {-0.1, -0.000 75}, {1, 1}],
      Arrow[{xy, {0, xy⟦2⟧}}]}
    ];
   icpt = yintercept[ratio, preventedate];
   nppt = yintercept[ratio, notpreventedate];
   saving = (nppt⟦2⟧ - icpt⟦2⟧) / nppt⟦2⟧;
   Print@ListPlot[zapNullOptions@
      {{{1, 0}, {1, 1}},
       detail > 1 → {{ratio, 0}, {ratio, 1}},
       detail > 2 → preventedate,
       notpreventedate
      },
     AxesLabel → {"Out by x", "Probability of out by x"},
     Joined → True,
     PlotStyle → zapNullOptions@
       {{Thickness[.002 5], Dashing[{.01, .01}], Gray},
        detail > 1 →
         {Thickness[.002 5], Dashing[{.01, .01}], Gray},
        detail > 2 → Darker[Green],
        Red},
     ImageSize → 1 000,
     PlotRange → {{0, 11}, {0, 0.022}},
     BaseStyle → {Thickness[.005], 14},
     AxesOrigin → {0.00, 0},
     Ticks → {Range[10], Automatic},
     AspectRatio → .5,
     Epilog → zapNullOptions@
       {Thickness[.002], Arrowheads[.02],
        detail > 3 →
         explain["Safer device, " <> fig[icpt⟦2⟧], icpt],
        detail > 1 → explain["Ordinary device, " <> fig[nppt⟦2⟧] <>
           "  (×" <> fig[ratio] <> " overdose)", nppt]
       }];
   CellPrint[Cell[
     "With prob " <> fig[nppt⟦2⟧] <>
      " a normal device is out by " <> ToString[ratio] <>
      "; with prevention it is probability " <>
      fig[icpt⟦2⟧] <> " it is ≥ " <> fig[ratio] <>
      " out. That is, for a ratio ≥ " <> fig[ratio] <>
      ", it is " <> fig[100 saving] <> "% safer.", "Text"]]
  ];
explainPlot /@ Range[4];
```



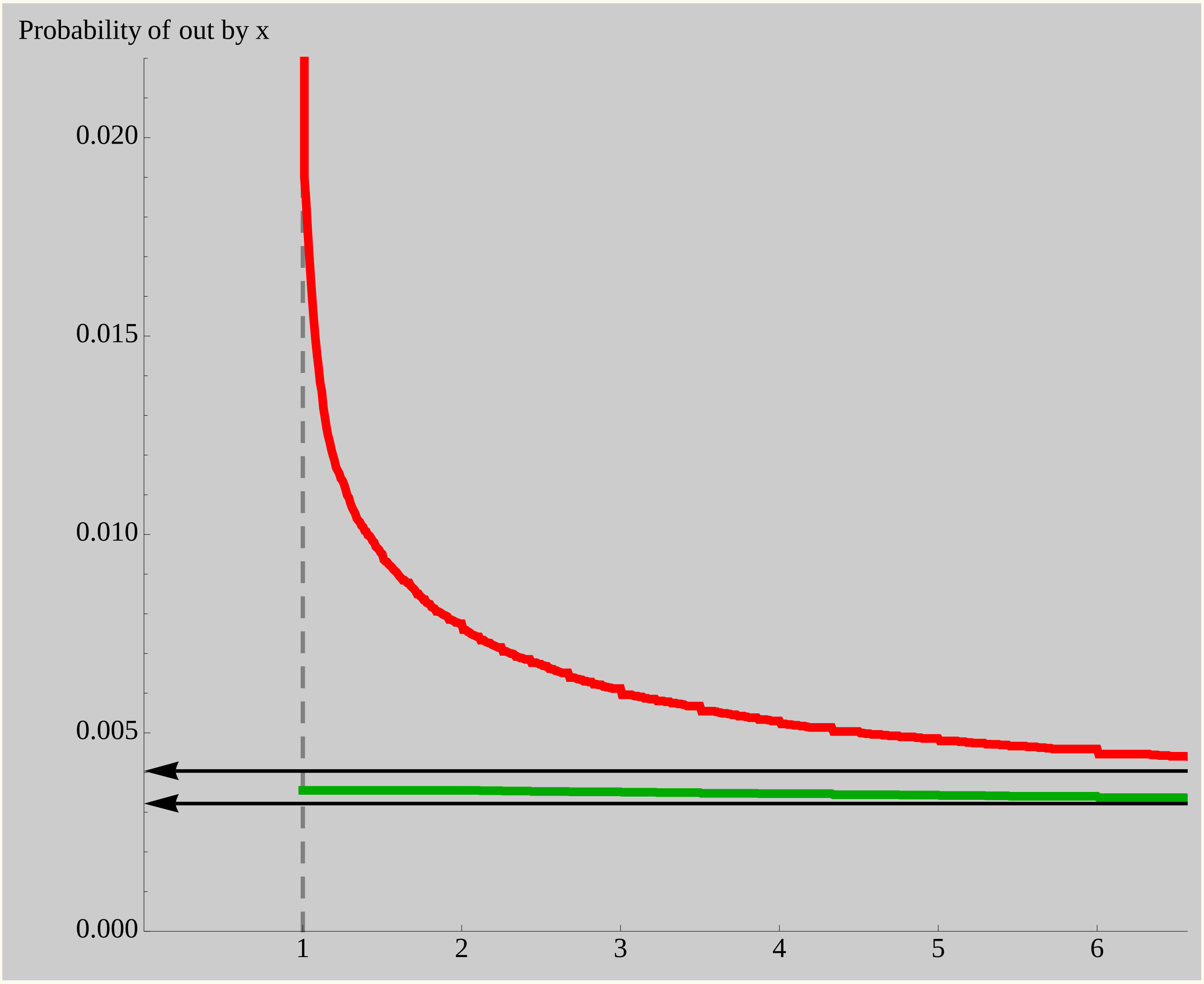
With prob 0.004 a normal device is out by 10; with prevention it is probability 0.003 it is ≥ 10. out. That is, for a ratio ≥ 10., it is 20.328% safer.



With prob 0.004 a normal device is out by 10; with prevention it is probability 0.003 it is ≥ 10. out. That is, for a ratio ≥ 10., it is 20.328% safer.



With prob 0.004 a normal device is out by 10; with prevention it is probability 0.003 it is ≥ 10. out. That is, for a ratio ≥ 10., it is 20.328% safer.



With prob 0.004 a normal device is out by 10; with prevention it is probability 0.003 it is ≥ 10. out. That is, for a ratio ≥ 10., it is 20.328% safer.

References

K. J. Vicente, K. Kada-Bekhaled, G. Hillel, A. Cassano & B. A. Orser, "Programming errors contribute to death from patient controlled analgesia: Case report and estimate of probability," *Canadian Journal of Anesthesiology*, **50**(4):328–332, 2003.

K. J. Vicente, Correspondence, *Canadian Journal of Anesthesiology*, **50**(8):856-857, 2003.