PARAMCALC: compute parameters to fit condition

Miguel V. S. Frasson
(mvsfrasson@gmail.com)

2026, jan. 27

Abstract

Sometimes one wants to get font size or letter spacing so that text fits a desired length, or some unit length so that a picture fits some desired size, for instance. This package implements macros to compute such parameters to fit a condition.

1 The purpose of this package

Suppose that one wants to create a nice logo with two lines of same size, first line "AWESOME" and second line "example of use for `paramcalc`". We look for font size of first line, so that the lines have the same width. This is the desired result:

AWESOME
example of use for paramcalc

Instead of setting font size by hand[1] and visually trying to get the desired condition, we want to compute such font size for line 1.

This can be acomplished with `paramcalc` with the following code (make sure you use scalable fonts[2]):

```
\newcommand{\mytext}{\Large example of use for paramcalc}
\newlength{\lineIIwd}
\settowidth{\lineIIwd}{\mytext}

\newlength{\lineIwd}
\paramcalc{12}{50}{\lineIIwd}[\lineIwd]{
  % compute \lineIwd as function of fontsize \x
  \settowidth{\lineIwd}{\fontsize{\x}{\x}\selectfont \textls*{AWESOME}}
}

Showing the result.
\begin{center}
  {\fontsize{\x}{\x}\selectfont \textls*{AWESOME}\par}
  \mytext
\end{center}
```

It you use `\x` in the document, you get the computed value 29.26342205668603.

[1] As width is proportional to font size, one could also compute font size by proportion, measuring texts of both lines.

[2] Computer Modern are not scalable, but Latin Modern (package `lmodern`), EB Garamond (package `ebgaramond`) and many others are.

2 Usage

To compute a desired parameter, the situation can be reduced to find a zero of a real function. In the example above, we wanted that the length $y(x)$ of first line as function of its font size x should match the length y_0 of second line, so that we need to find the zero of the function $f(x) = y(x) - y_0$.

The computation is done by `\paramcalc` macro.

`\paramcalc` `\paramcalc[`⟨*x macro*⟩`]{`⟨*x initial*⟩`}{`⟨*x final*⟩`}[`⟨ε⟩`]{`⟨*y desired*⟩`}[`⟨*y macro*⟩`]{`⟨*body*⟩`}`

The optional ⟨*x macro*⟩ defaults to `\x`, the optional ⟨*y macro*⟩ defaults to `\y` and both should be control sequences. ⟨*y macro*⟩ and ⟨*y desited*⟩ could be lengths, converted to numbers as pt.

⟨*body*⟩ should be code that computes ⟨*y macro*⟩ for an arbitrary value of ⟨*x macro*⟩, supposed to expand to a decimal value. The result is stored in ⟨*x macro*⟩ after its computation. If its value is in between ⟨*x initial*⟩ and ⟨*x final*⟩, convergence is ensured. Often[3] the functions involved are affine (for instance, a width is proportional a font size), and in this case convergence is achieved after just one step.

The convergence is achieved when $|\langle y\ macro\rangle - \langle y\ desired\rangle| < \varepsilon$. The value of ε, where ε is given in optional arg ⟨*epsilon*⟩ and defaults to 0.05. If it doesn't happen after a maximum number of steps, an error is signaled. Then, probably de desired x is outside initial interval.

There are situations where only integer values of x are suitable, like in the optional parameter n for `\textls[`⟨n⟩`]{`⟨*text*⟩`}`. In such cases, use `\intparamcalc`, with same arguments. The only difference is that ⟨*x macro*⟩ only assume integer values, but ⟨*y macro*⟩ still are supposed to assume real values.

`\intparamcalc` `\intparamcalc[`⟨*x macro*⟩`]{`⟨*x initial*⟩`}{`⟨*x final*⟩`}[`⟨ε⟩`]{`⟨*y desired*⟩`}[`⟨*y macro*⟩`]{`⟨*body*⟩`}`

Attention: The numerical method implemented in this package is not suitable when y assumes (too) discrete values or isn't a continuous function of x (it its where real-valued) like when y counts lines, for instance.

2.1 Package options

- `maxsteps` = ⟨n⟩ : sets maximum number of steps before giving up. Defaults to 20.
- `epsilon` = ⟨*epsilon*⟩ : sets the convegence criterium ε to ⟨*epsilon*⟩. Defaults to 0.05.

3 Examples

Creating a logo for the book "História Geral do Brasil", in three lines.

- The first two lines in font size 20pt of size;
- For the first line we compute letter spacing to fit size of 2nd line;
- For the third line, we compute font size, keeping same size.

[3] I could not figure out any real life example where the function was not affine.

This is achieved with the code:

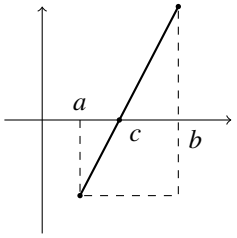
```
\begin{center}
  \fontsize{20}{22}\bfseries

  % base length \lIIwd: line 2 with \textls* and \bfseries
  \newlength{\lIIwd} \newlength{\auxwd}
  \settowidth{\lIIwd}{\textls*{GERAL \,DO}}

  % compute letter spacing \lIIs for 1st line: body compute \auxwd
  \intparamcalc[\lIIs]{0}{200}{\lIIwd}[\auxwd]{
    \settowidth{\auxwd}{\textls*[\lIIs]{HISTÓRIA}}}

  % compute font size \lIIIfs for 3rd line: body compute \auxwd
  \paramcalc[\lIIIfs]{20}{50}{\lIIwd}[\auxwd]{
    \settowidth{\auxwd}{\fontsize{\lIIIfs}{\lIIIfs}\selectfont \textls*{BRASIL}}}

  % typesetting the logo
  \textls*[\lIIs]{HISTÓRIA} \\ \textls*{GERAL \,DO} \\
  \fontsize{\lIIIfs}{\lIIIfs}\selectfont \textls*{BRASIL}
\end{center}
```

4 Numerical method implemented

To ensure convergense (with a root in $[a, b]$), we implement Illinois algorithm[4], an improvement over False Position method (*regula falsi*):

Assuming that $f : I \to \mathbb{R}$ is a continuous function, end-points $a, b \in I$, $f(a) \cdot f(b) < 0$, so f has a root in $[a, b]$.

Let c be the root of the secant line that connects $(a, f(a))$ and $(b, f(b))$.



so, by similarity of triangles,

$$\Delta = \frac{b-a}{f(b)-f(a)} = \frac{b-c}{f(b)} \implies c = b - f(b)\Delta. \tag{1}$$

The Illinois algorithm for False Position puts the following iteraction:

$$(a, f(a)) = \begin{cases} (c, f(c)), & \text{if } f(b)f(c) < 0, \\ (a, \frac{f(a)}{2}), & \text{otherwise} \end{cases} \tag{2}$$

$$(b, f(b)) = (c, f(c)) \tag{3}$$

[4] Dowell, M., Jarratt, P. A modified regula falsi method for computing the root of an equation. *BIT Numerical mathematics* **11**, 168–174 (1971). https://doi.org/10.1007/BF01934364

5 Code

Identidication of the package.

```
\NeedsTeXFormat{LaTeX2e}
\ProvidesPackage{paramcalc}[2026/01/27 v1.0 compute params to fit conditions]
```

We use LaTeX3 codebase.

```
\ExplSyntaxOn
```

There are two versions of the solving macros: real roots are admitted or only integer roots (like parameters for `\textls[\x]{text}`). We do a single implementation and the cases are regulated by a boolen.

```
\bool_new:N \g_PARC_int_bool
```

The iteraction will end when `f(c)` $< \varepsilon$.

Floating point variables for a, b, $f(a)$, $f(b)$, c, $f(c)$ and Δ:

```
\fp_new:N \l_PARC_a_fp
\fp_new:N \l_PARC_b_fp
\fp_new:N \l_PARC_fa_fp
\fp_new:N \l_PARC_fb_fp
\fp_new:N \l_PARC_x_fp
\fp_new:N \l_PARC_fx_fp
\fp_new:N \l_PARC_delta_fp
```

Also, a counter for steps taken, in case of divergence.

```
\int_new:N \l_PARC_steps_int
```

Defining package option for maximum number of steps.

```
\int_new:N \l_PARC_maxsteps_int
\int_set:Nn \l_PARC_maxsteps_int { 20 }
\keys_define:nn {paramcalc} { maxsteps .int_set:N = \l_PARC_maxsteps_int }
```

Defining package option for default epsilon (stop criterium).

```
\fp_new:N \l_PARC_epsilon_fp
\fp_set:Nn \l_PARC_epsilon_fp { 0.05 }
\keys_define:nn {paramcalc} { epsilon .fp_set:N = \l_PARC_epsilon_fp }
```

Passing options to package.

```
\DeclareOption*{ \keys_set:ne { paramcalc } { \CurrentOption } }
\ProcessOptions
```

`\PARC_paramcalc:NnnnnNn` The arguments of `\PARC_paramcalc:NnnnnNn` are

`#1` = macro that holds value for x (default to `\x` later on)
`#2` = a
`#3` = b
`#4` = ε
`#5` = y_0
`#6` = macro that stores computed $y(x)$ (default to `\y` later on)
`#7` = body of code that computes $y(x)$ and stores value in `#6`.

```
\cs_new:Nn \PARC_paramcalc:NnnnnNn
{
```

Store a from `#2` in `\l_PARC_a_fp` and set `#1` as the decimal representation of a, taking care if integer values must be used.

```
  \fp_set:Nn \l_PARC_a_fp { #2 }
  \bool_if:NTF \g_PARC_int_bool
```

```
  { \tl_set:Ne #1 { \fp_to_int:n { #2 } } }
  { \tl_set:Ne #1 { \fp_to_decimal:n { #2 } } }
```

Call body to get $y(x)$ in `#6`

```
  #7
```

Save $f(x) = y(x) - y_0$ in `\l_PARC_fa_fp`

```
  \fp_set:Nn \l_PARC_fa_fp { #6 - (#5) }
```

Do as before, now for b stored in `#3`:

```
  \fp_set:Nn \l_PARC_b_fp { #3 }
  \bool_if:NTF \g_PARC_int_bool
  { \tl_set:Ne #1 { \fp_to_int:n { #3 } } }
  { \tl_set:Ne #1 { \fp_to_decimal:n { #3 } } }
  #7
  \fp_set:Nn \l_PARC_fb_fp { #6 - (#5) }
```

In many situations, the function $f(x)$ is affine and the secant method converges after the first iteration, so it really doesn't matter if $f(a)f(b) < 0$, so we trigger just a warning if signs of $f(a)$ and $f(b)$ are the same.

```
  \fp_compare:nNnT { (\l_PARC_fa_fp) * (\l_PARC_fb_fp) } > { 0 }
  {
    \msg_warning:nn {paramcalc}
    {
      The ~values ~of ~\string#6 ~are
      ~{\fp_to_decimal:N \l_PARC_fa_fp} ~and
      ~{\fp_to_decimal:N \l_PARC_fa_fp}, ~both
      \fp_compare:nNnTF { \l_PARC_fa_fp } > 0 { ~above } { ~below }
      ~{\fp_to_decimal:n {#5}} ~so ~convergence ~is ~not ~ensured.\\
      Try ~to ~set ~initial ~values ~that ~surround ~the ~root.
    }
  }
```

Now the loop. Last iteraction is supposed to be stored in `\l_PARC_fb_fp`. Stop condition is `f(b)` $< \varepsilon$, with ε stored in `#4`.

```
  \fp_while_do:nNnn { abs(\l_PARC_fb_fp) } > { #4 }
  {
```

Computing Δ and c as in (1).

```
    \fp_set:Nn \l_PARC_delta_fp
      { (\l_PARC_b_fp - \l_PARC_a_fp) / (\l_PARC_fb_fp - \l_PARC_fa_fp) }
    \fp_set:Nn \l_PARC_c_fp
      { \l_PARC_b_fp - (\l_PARC_fb_fp) * (\l_PARC_delta_fp) }
```

Compute $f(c)$ and store it in `\l_PARC_fc_fp`.

```
    \bool_if:NTF \g_PARC_int_bool
    { \tl_set:Ne #1 { \fp_to_int:N \l_PARC_c_fp } }
    { \tl_set:Ne #1 { \fp_to_decimal:N \l_PARC_c_fp } }
    #7
    \fp_set:Nn \l_PARC_fc_fp { #6 - (#5) }
```

Now implement Illinois iteraction from (2)

```
    \fp_compare:nNnTF { (\l_PARC_fc_fp) * (\l_PARC_fc_fp) } < { 0 }
    {
      \fp_set:Nn \l_PARC_a_fp { \l_PARC_b_fp }
      \fp_set:Nn \l_PARC_fa_fp { \l_PARC_fb_fp }
```

```
    }
    {
      \fp_set:Nn \l_PARC_fa_fp { 0.5 * (\l_PARC_fa_fp) }
    }
```

and (3).

```
    \fp_set:Nn \l_PARC_b_fp { \l_PARC_c_fp }
    \fp_set:Nn \l_PARC_fb_fp { \l_PARC_fc_fp }
  }
```

Handling possible divergence if $f(a)f(b) > 0$. If it doesn't converge after `\l_PARC_maxsteps_int` steps, an error is signaled.

```
  \int_incr:N \l_PARC_steps_int
  \int_compare:nNnT { \l_PARC_steps_int } > { \l_PARC_maxsteps_int }
  {
    \msg_error:nn {paramcalc}
    {
      Failed ~to ~converge ~after ~\int_use:N \l_PARC_steps_int ~steps.\\
      Try ~to ~narrow ~initial ~values ~around ~the ~zero.
    }
  }
}
```

Recall that the user access the computed value from `#1`.

Now we define user functions that set `\g_PARC_int_bool` and call previous function.

Define `\paramcalc`.

```
\NewDocumentCommand{\paramcalc}{ O{\x} m m O{0.05} m O{\y} m}
{
  \bool_set:Nn \g_PARC_int_bool { \c_false_bool }
  \PARC_paramcalc:NnnnnNn #1 { #2 } { #3 } { #4 } { #5 } #6 { #7 }
}
%
% Define \texttt{\string\intparamcalc}.
%    \begin{macrocode}
\NewDocumentCommand{\intparamcalc}{ O{\x} m m O{0.05} m O{\y} m}
{
  \bool_set:Nn \g_PARC_int_bool { \c_true_bool }
  \PARC_paramcalc:NnnnnNn #1 { #2 } { #3 } { #4 } { #5 } #6 { #7 }
}
```

Finish LaTeX3 syntax.

```
\ExplSyntaxOff
```