In Memoriam "Orbit™"

This text is a **supportive document** to our "**Response to Miller's Critique on SEE++**" and provides a **technical evaluation of Orbit™ 1.8** original Macintosh source code as well as a comparison between **Orbit™ 1.8 software and Orbit™ model in SEE++**.

There is no doubt that the **biomechanical model contained in Orbit™ 1.8** is scientifically **well supported**, **established and realistically predicts eye motility** for normal "healthy" eyes and a large set of simulations of pathologies. But it is also a fact that the **Orbit™ 1.8 biomechanical model has not been updated since 1999**, certainly an eternity for a computer program. Moreover, recent physiological findings like **"Active Pulleys" are not included in Orbit™ 1.8**.

Orbit™ 1.8 was **implemented for the Macintosh "Old World" generation** (Macintosh II series, Quadra/Centris, PowerPC, G3 etc.), **only runs in emulation mode on Mac OS X** and **cannot run on the new Intel-based Macs**. Moreover, the **development environment** and **class library** that was used to implement Orbit™ **does not exist anymore**. This unfortunately must lead to the conclusion that the **Orbit™ 1.8 computer program is hopelessly outdated**.

We see **SEE++** as **replacement or extension to Orbit™** with **more clinical relevance** while providing **essential functionality similar to what Orbit™ offered**. The **SEE++** software **differentiates** between **biomechanical models and user interfaces** and therefore provides an **open, flexible and portable basis** for further development. Additionally, the "**SEE-KID" and "SEE-KID Active Pulley"** models have been developed in order to **incorporate new anatomic and physiological findings** from basic research. Compared to Orbit™, these models also use a **different mathematical approach for numerical optimization** in order to more reliably solve non-linear problems, a special and important task when simulating the statics of mechanical systems like the human eye.

When looking at the **source code of Orbit™**, an experienced Software-Engineer will get desperate and tear one's hair. **Code analysis of the Orbit™** mathematical model core reveals a total of **5123 lines of code, 2672 being comments only**, which are mainly uncommented variants of different calculations. This means that **more than 50% of the total source code is commented out** and meaningless. **Software quality** finds, amongst other things, to a major part its roots in **well structured and exactly designed implementation**. The implementation of Orbit™ could at best serve as deterrent example for "**how to fail on this topic**". It is **no matter of course** that highly qualified **researchers transform** their ideas and breaking research **results into well-formed maintainable software applications**. On the other hand, **we do not claim to carry out basic medical or physiological research**, however, we understand our work as interdisciplinary applied research that follows basic research as well as clinical practice, creating technically and socially useful solutions.

When it comes to **scientific computing**, computer programs are used to **construct mathematical models** and solution techniques to analyze and **solve scientific problems**. In practical use, this is typically the **application of computer simulation** and other forms of computation to problems in various scientific disciplines. Unfortunately, most problems in these fields **cannot be solved analytically** and **numerical methods need to be applied** to obtain approximations to a solution. **Orbit™ as well as SEE++** apply **numerical methods to find mechanical force equilibriums** for eye positions and innervations. Building on that, **both computer programs simulate binocular functions** and simulate the clinical **Hess-Lancaster test**.
When using numerical computing methods, **floating point arithmetic is a famous source for computational errors**. Once **errors** are generated, they **propagate through calculations** and some algorithms amplify these errors which then causes **numerical instability**. **Due to its implemenation, Orbit™ 1.8 is vulnerable to such errors**.

We would like to support our statements with some of the most obvious implementation problems that we found in the Orbit™ 1.8 source code:

1. **There are different implementations for the calculation of Listing's Torsion:**
   There is a function "*myListing*" defined in the "*sqlibrary.cp*" file that uses two constants "Degrees" and "Radians" for **conversion of degrees to radians and back**. These constants are defined as follows:

   ```
   const extended Radians = 180. / 3.14159;
   const extended Degrees = 3.14159 / 180.;
   ```

   In other cases, there are definitions of the **constant PI** with higher precision in "*squint.h*" and "*sqnt.h*" that are used for conversions of degrees to radians and vice versa:

   ```
   #define PI 3.14159265358979323846264338З
   #define DEG_PER_RAD (180.0 / ORBIT_PI)
   #define RAD_PER_DEG (ORBIT_PI / 180.0)
   ```

   It turns out that **Orbit™ uses two functions** "*myListing*" and "*_listing*" to **calculate Listing's Torsion**, "*myListing*" for angles in degrees and "*_listing*" for radians. However, **sometimes** "*myListing*" is used which results in **low precision calculation** of degrees to radians conversion, in **other situations** "*_listing*" is **directly called** with "*DEG_PER_RAD*" and "*RAD_PER_DEG*" **precalculated high precision** radian angles.
   In "*Fascia.cp*", the **difference between two Listing's Torsion values**, one in low and one in high precision is **calculated**, which leads to the **inconsistency** that for exactly the **same eye position**, this **difference is not 0**.
   One could now argue that these **differences are marginal**, but **in connection with an iterative optimization**, these **inconsistencies commulate** and result in **different model predictions**. Deciding **whether a solution is right or wrong**, compared to other models or calculations becomes a **challanging task** in this situation.
   Unfortunately, **if this "bug" is corrected**, **solutions cannot be compared** to the original Macintosh version of Orbit™ 1.8 due to different model predictions that are displayed in the mechanical stateviewer dialog of Orbit™.

2. **Orbit™ uses float precision numbers for finding 3D eye positions but extended precision numbers anywhere else:**
   For calculating 3D eye positions from 2D eye positions when the fixing eye is pathologic, **Orbit™** uses a slightly modified implementation of the **Newton minimization algorithm taken from the book "Numerical Recipes in C"**. The **default implementation uses for all numbers float precision** and so does Orbit™. However, **all other model code in Orbit™ uses the "extended"** floating point type. The **"extended" type is a non-standard Macintosh specific data type**, which offers more precision in floating point calculations than the "double" type. **On Intel platforms** (also on the new Intel-Macs) **no such data type exists** due to the different processor architecture.

   ```
   extended g_pos[3];
   static void POS_POS(int n, float x[], float f[]){
     ...
     ...
     ...
     extended rot_deg[4]; /* theta, phi, psi, cyclo */
     rot_deg[0] = x[1];
     rot_deg[1] = x[2];
     rot_deg[2] = myListing(x[1], x[2], psi0);
     // see (1) for problems with myListing
     ...
     ...
     ...
     f[1] = rot_deg[0];  // implicit extended to float conversion
     f[2] = rot_deg[1];  // possible loss of significant digits
     g_tor = rot_deg[2];

     f[1] -= g_pos[0];   // implicit extended to float conversion
     f[2] -= g_pos[1];   // possible loss of significant digits
   ```

```
} /*POS_POS*/

void find_inner_eye_positions(...) {
  ...
  ...
  ...
  int n = 2;
  float x[3], x_int[3];
  x[1] = g_pos[0];    // implicit extended to float conversion
  x[2] = g_pos[1];    // possible loss of significant digits
  ...
  ...
  ...
  int check = 0;
  newt(x, n, &check, POS_POS);
  ...
  ...
  ...
} /*find_inner_eye_positions*/
```

3. **Orbit™ passes most parameters internally via strings:**
   To our **big surprise**, we discovered that **Orbit™ almost randomly uses format strings to cut or round intermediate calculation results**. Here are some examples:

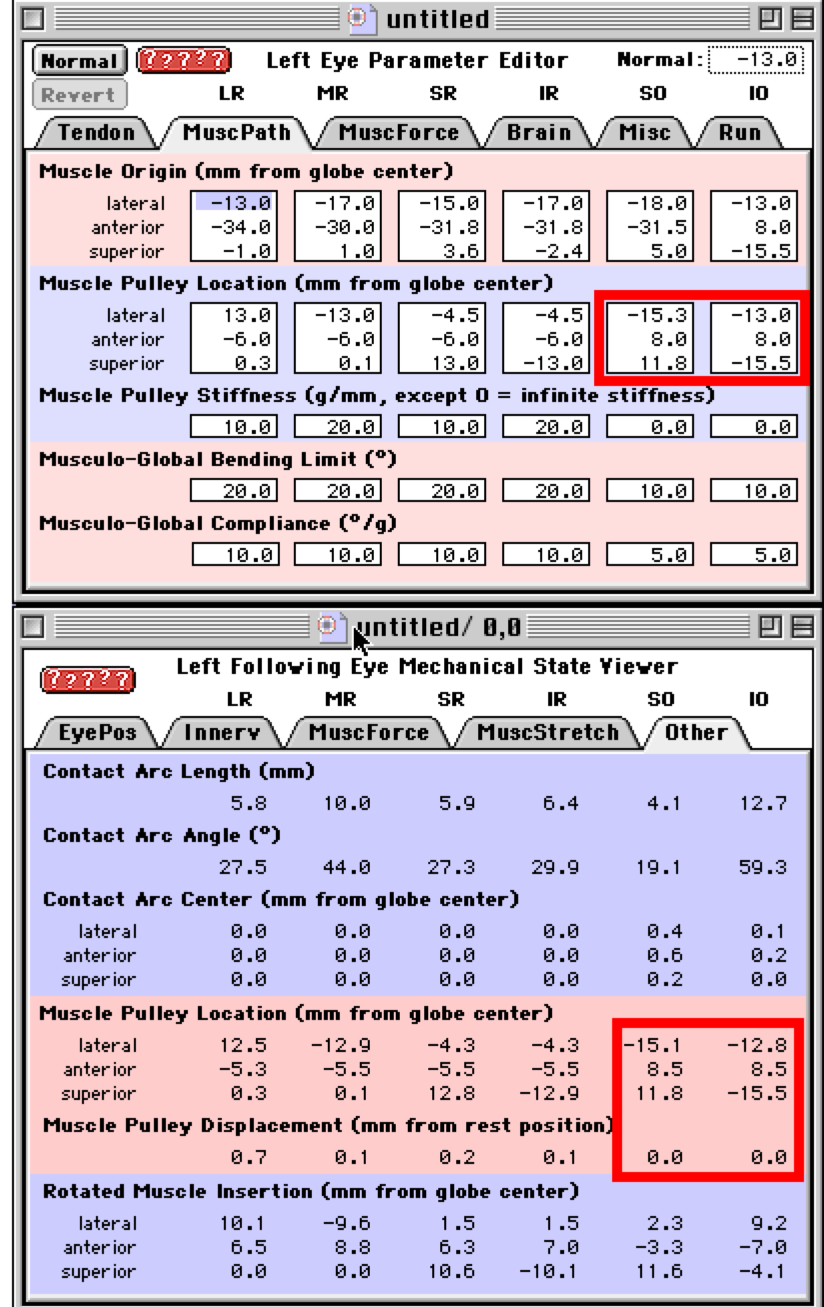
```
//Definitions from squint.h
#define PR_POS_L "%20.15lf %20.15lf %20.15lf %20.15lf %20.15lf %20.15lf\n"
#define PR_EXC_L "%20.15lf %20.15lf %20.15lf %20.15lf %20.15lf %20.15lf\n"

static void FLIP(char *in_pos_line, char *out_pos_line) {
  extended pos[3];
  char temp[256];
  extended ok=0.0;

  asign(pos, 3, "", in_pos_line, "", -1000., 1000., ok, 1.);   // asign copys eye position from string to
pos array
// note: sometimes this has to accept missing values
  if(pos[0] != MIS_VAL) {
    pos[0] = - pos[0];
    pos[2] = - pos[2]; // torsion
  }

  sprintf(temp, "%-6.2f %-6.2f %-6.2f ", pos[0], pos[1], pos[2]);
  sprintf(out_pos_line, "pos: %s\n", temp);
}/*FLIP*/

void Execute_INN_POS(...){
  ...
  ...
  ...
  switch(parameters.ruleType) {
  case INN:
    sprintf(inn_pos_string, PR_EXC_L, outExc[0], outExc[1], outExc[2], outExc[3], outExc[4], outExc[5]);
    if( outPtr[0]) {
      sprintf(outPtr[0], "inn: %s", inn_pos_string);
      MovePtr(0);
    } break;

  case POS:
    sprintf(inn_pos_string, PR_POS_L, rot_deg[0], rot_deg[1], rot_deg[2], trans[0], trans[1], trans[2]);
    if( outPtr[0]) {
      sprintf(outPtr[0], "pos: %s", inn_pos_string);
      MovePtr(0);
    }
  break;
  }
  ...
  ...
  ...
}/*Execute_INN_POS*/
```

The question in this case is, **why does Orbit™ use the extended data type** when it truncates intermediate calculation results (e.g. when flipping eye positions in the "FLIP" function above) to **2 digits after the comma**. Another question is, why does function **"FLIP" use 2 comma digits** in its result when **"Execute_INN_POS" uses 15 comma digits**. In **both cases double precision** calculations would have been **sufficient**. We will **not comment on performance issues** and bad coding style, because we think that this **speaks for itself**. Interestingly, in the

implementation of **"FindPosition" and "FindInnervations" functions no rounding takes place** and intermediate values are calculated with full extended precision.

4. **Orbit™ shows different convergence behavior when using consistent (extended) data types:**
   The **convergence behavior of Orbit™** is **highly fragile**. **Small changes** in the source code (modification of precision as explained in (1), consistent usage of data types as explained in (2), consistent precision in rounding values as explained in (3)) immediately **lead to different model predictions**. Especially when the **fixing eye is pathological** and the torsional value of the fixing eye needs to be evaluated using the Newton optimizer, simulation results vary or **loss of convergence** results.
   These problems **clearly do not qualify Orbit™** to represent a **solid basis** for extensions or development of new models. Due to the fact that we **cannot offer the source code** of Orbit™ for download (for legal reasons) and **only fragments of the source code would not help** to illustrate this issue, it is not possible for us to give a short example.

5. **Orbit™ obviously moves "Pulleys" of superior and inferior obliques:**
   Up to our knowledge this is **highly suspicious of being a bug**, since the **simulation of pulleys** does **not apply to the obliques**. The default geometric configuration in Orbit™ for the **superior oblique** is to use the **pulley as representation for the trochlea**, wheras for the **inferior oblique**, **origin and pulley are identical**, since the inferior oblique does not have a pulley. Our example shows the Orbit™ **left eye parameter editor** as well as the **mechanical stateviewer** for a normal "healthy" left eye in primary position.



Under **"Muscle Pulley Displacement"**, Orbit™ **correctly shows 0** for both oblique muscles. However, under **"Muscle Pulley Location"**, **coordinates** for inferior and superior obliques are **different** compared **to the original geometrical default values** shown in the left eye parameter editor. Consequently, there are **two possibilities**, either **Orbit™ moves trochlea and non-existing pulleys** (inferior oblique) but **displays the wrong pulley displacement** magnitudes, or **Orbit™ does not move trochlea and inferior oblique pulley** but **coordinates are invalid**. In either case, this is an **inconsistency** and speaking in Miller's words, "**causes worry and doubt**".

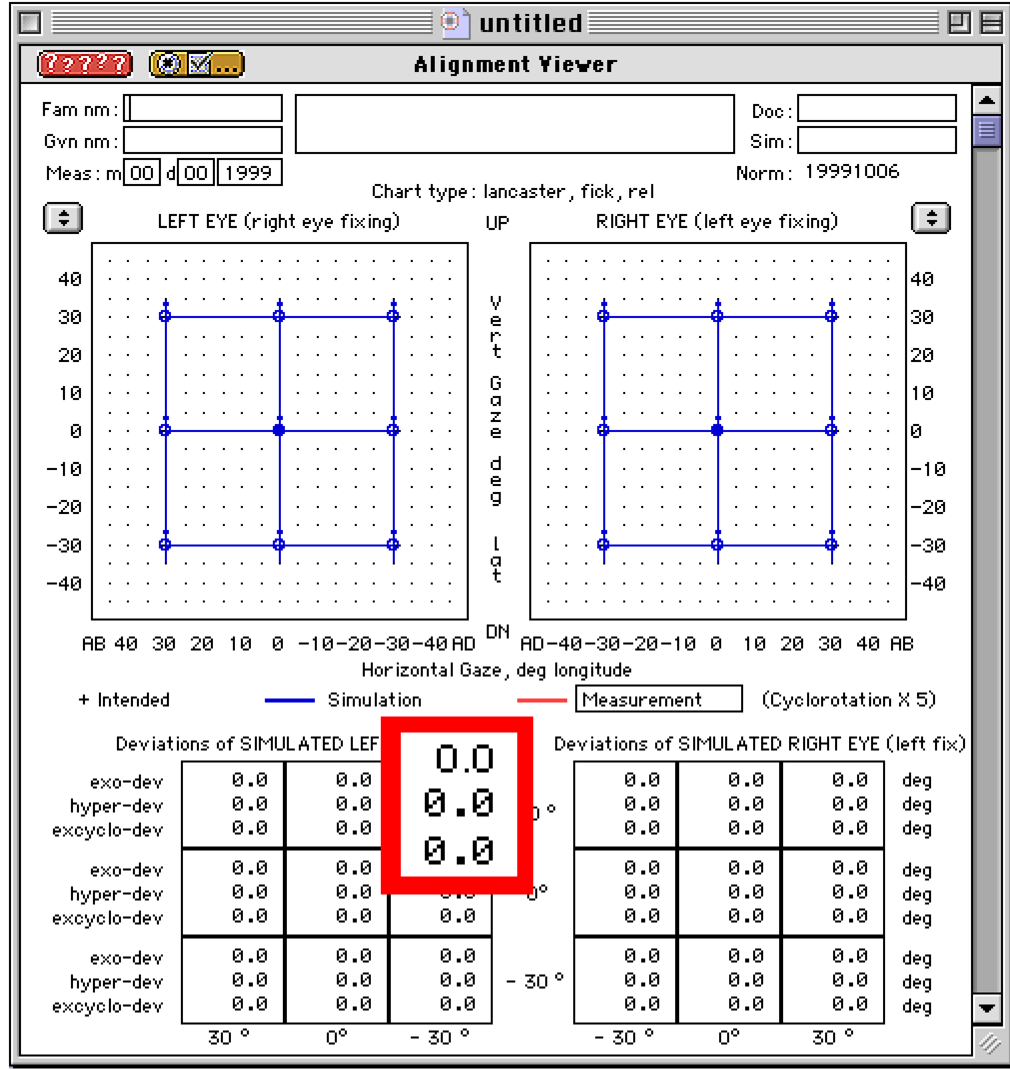
6. **3D display (Live Eyes and Graphic Eyes) in Orbit™ incorrectly display insertion lines of eye muscles:**
   Since we **did not review the 3D display** source code, we **insistently hope** that insertion lines as seen in the 3D view are **only for show and not part of the biomechanical model**.

The **animation above** shows that obviously the **insertion of a muscle in Orbit™ is not attached to the globe** but rather "floats" on the globe. Looking at the **red arrows** in the animation, it can be seen that the **distance between the cornea and the insertion line** of the **right medial rectus increases when the right eye is abducted**, conversely for the **left inferior rectus** the **distance to the cornea decreases in adduction** of the left eye. Assuming that **muscles are tightly coupled to the globe** at their insertions, the situation shown in the animation above does **not constitute a realistic simulation and causes worry and confusion**.

7. **The support of Orbit™ for creating movies is simply ridiculous:**
   Creating **movies in Orbit™ 1.8** is a **time-consuming and difficult job** for an ordinary computer user. The **user interface support** for creating movies is **catastrophic and beyond any reasonable solution**. In the **User's Manual of Orbit™ 1.8**, the following **courageous explaination** can be found under the topic "*How do I make an Orbit Movie?*":
   "*To use Movie Player to make a folder of PICS into a movie, the PICT files must all be named with the same characters followed by a positive number that tells Movie Player the order in which to string them together (eg: faden01, faden02, ...). You will probably need to rename the files produced by Orbit.*"
   This **cannot be seen as a feature** of the program but rather looks like a **complicated and clumsy workaround**.

8. **Minor suggestion: Where is the "Mechanical Stateviewer for Inner Eye"? Even "incorrectly implemented" SEE++ has one!**

9. **Minor GUI glitch:**
A minor glitch in the **textual display of the simulated deviations** can be seen in the following image:



The **first number of the horizontal deviation** (exo-dev) is **formatted differently**. This is **negligible**, but compared to Miller's critique on SEE++ **needs to be mentioned** for the sake of completeness.

Differences between Orbit™ and the Orbit™ model in SEE++

The **above list** shows the **most important bugs** we found in the **Orbit™ 1.8** software system. In the following we want to give a **short overview** of the **main differences between the Orbit™ 1.8** software system and the **Orbit™ model in SEE++**. In order to be able to **compare simulation results** of the Orbit™ model in SEE++ with the Orbit™ 1.8 software system running on Macintosh computers, **we did NOT correct the bugs** described in **(1)** and **(5)** in the Orbit™ model of SEE++.
The following list describes the most important differences:

1. **SEE++ uses different signs for duction and torsion than Orbit™:**
In the **Orbit™ software system**, **adduction** is defined as **being negative**, **abduction is positive**, **intorsion is negative** and **extorsion is positive**, respectively. **In SEE++** (and also in the Orbit™ model of SEE++) the signs for duction and torsion are inverted, namely **adduction** is **positive**, **abduction** is **negative**, **intorsion** is **positive** and **extorsion** is **negative**. Since the **convention of signs** does **not have any influence on simulation results** we decided to use a **common clinical convention** which defines **positive angles towards the nose** for all three degrees of freedom.

2. **SEE++ uses Quaternions to represent rotations and eye positions:**
   Orbit™ uses **rotation matrices and radian angles** to represent rotations and eye positions. **SEE++ uses Quaternions** which have some advantages over other representations of rotations:

   - Quaternions don't suffer from gimbal lock, unlike euler angles.
   - They can be represented as 4 numbers, in contrast to the 9 numbers of a rotations matrix.
   - The conversion to and from axis/angle representation is trivial.
   - Smooth interpolation between two quaternions is easy (in contrast to axis/angle or rotation matrices).
   - After a lot of calculations on quaternions and matrices, rounding errors accumulate, so quaternions need to be normalized and rotation matrices need to be orthogonalised, but normalising a quaternion is a lot less troublesome than orthogonalising a matrix.
   - Similar to rotation matrices, you can just multiply 2 quaternions together to receive a quaternion that represents both rotations.

3. **Orbit™ gaze selector always shows points in the coordinate system of the following eye:**
   The **SEE++ system contains** a so-called "**gaze pattern dialog**" as an **equivalent to the Orbit™ gaze selector**. However, in Orbit™ the **points selected in this dialog** are **always in the coordinate system of the following eye**, whereas in **SEE++ points** used as **intended gaze positions** for the simulation are **defined in the coordinate system of the fixing eye** (with a different sign in duction). Moreover, the **gaze pattern dialog in SEE++** can also be used to **enter patient-measured data**, which are **additionally displayed in Hess diagrams** and defined in the **coordinate system of the following eye**. Orbit™ uses a "**measured gaze editor**" for **entering patient-measured data**, which is used in a **completely different way than the Orbit™ gaze selector**. Since we think it is a **bad idea** to enter **intended and measured** gaze positions in a **completely different manner**, **SEE++ uses** the **same input procedure for both** types of gaze positions.

4. **SEE++ consistently uses double precision for floating point calculations:**
   As already explained in **(2)** and **(4)** in the **above list**, **Orbit™ uses** the **"extended" data type for most calculations**. Since **SEE++ is designed** to be run on **Intel platforms** (Microsoft Windows operating system) where **no such data type exists**, **SEE++ consistently uses double precision** for all floating point calculations. The **"extended" data type** on Macintosh is **somewhat historic**, comes from the **old Mac generation** and in case this type is used in implementations, unfortunately makes it **impossible to identically port software** between platforms.
   Most modern systems handle floating point following the **IEEE-695 standard**. However, there are **still portability issues**. Most processors use **64 bits of precision** when computing floating point values. However, the widely used **Intel x86 series** of processors compute **temporary values** using **80 bits** of precision, as do **most instances of the Motorola 68k** series. Some **other processors**, such as the **PowerPC**, provide **fused multiply-add instructions** which perform a multiplication and an addition using high precision for the intermediate value. Optimizing **compilers will generate such instructions** based on **sequences of C/C++ operations**.
   For **almost all programs**, these **differences do not matter**. However, for **programs which do intensive floating point operations**, the **differences can be significant**. It is **possible to write floating point loops** which **terminate on one sort of processor but not on another**.
   Unfortunately, there is **no rule of thumb that can be used to avoid these problems**. Most **compilers** provide an **option to disable** the **use of extended precision** for intermediate values

(for GNU cc, the option is `-ffloat-store'). However, this merely **shifts the portability problem elsewhere**.

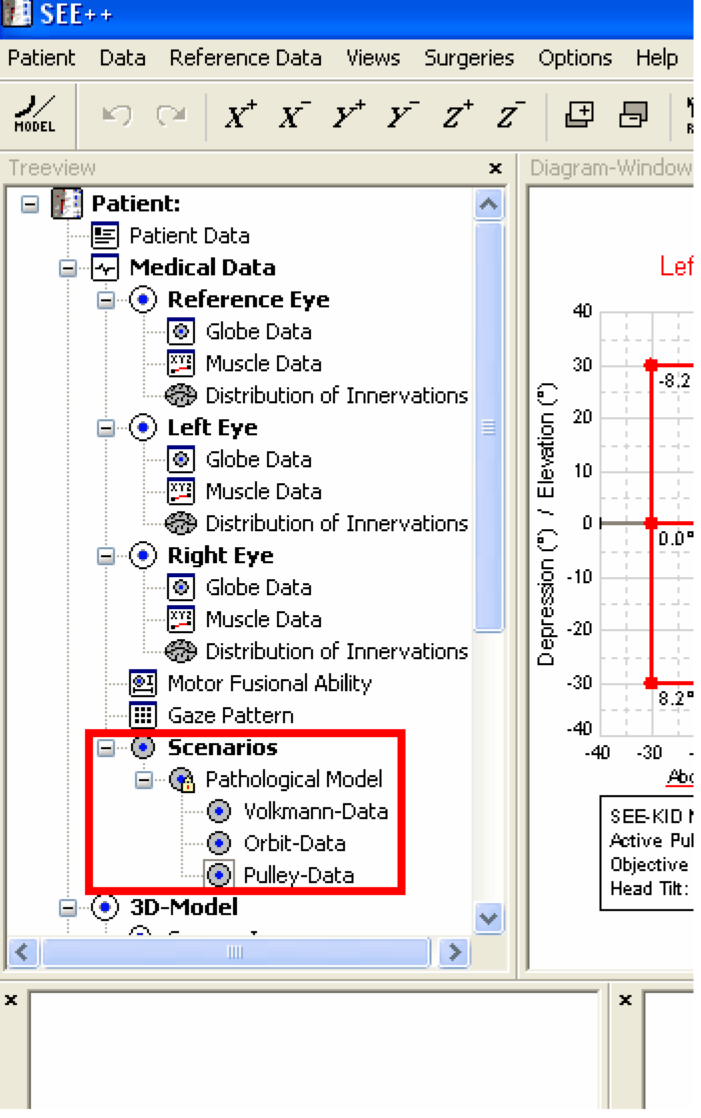
When **comparing simulation results** between **Orbit™ 1.8 Macintosh** version and the **SEE++** implementation of Orbit™, **differences only** occur when **numerical optimization** for some gaze position **reaches the precision limit of double floating point** precision. In this case, either the **resulting gaze position will show a small difference** (we observed differences up to 0.1 degrees), or **in very special cases**, **loss of convergence occurs** and SEE++ will not find a solution for this gaze position.

Another **portability problem** is the **calculation of trigonometric functions**, usually provided by the **runtime environment** that comes with each compiler on each operating system. Due to **different architectures**, it **cannot be guaranteed** that runtime functions **behave identically**, of course this will never be the case when **"extened" data type is used on Mac** platforms.

5. **SEE++ uses different "normal" eye parameters than Orbit™:**

   In SEE++ **"normal" eye parameter values** have been used **according to Volkmann and Miller**, however **Miller revised** his **normal eye parameter values** in Orbit™ 1.8 **again**. **Differences are rather marginal**, but when **comparing simulation results**, normal eye values must certainly be **the same for both programs**. **SEE++ therefore includes default scenarios** for normal eye data as "Volkmann Data" (no pulleys), "Pulley Data" (default values from Volkmann and Miller) and "Orbit Data" (latest version of Miller's "Orbit_Norm" file).

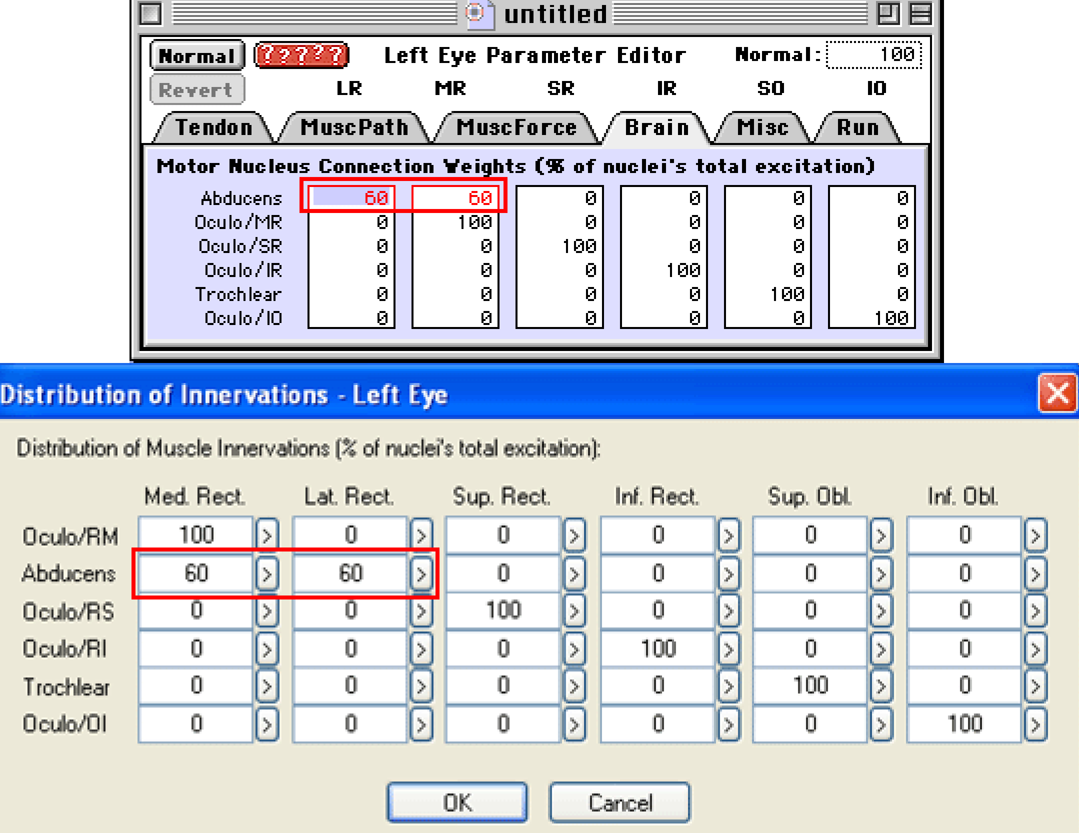
   In the **picture below** these **default scenarios are shown** and the **respective scenario needs to be selected** in order to load the correct normal eye data **prior to simulation and comparison** of results.



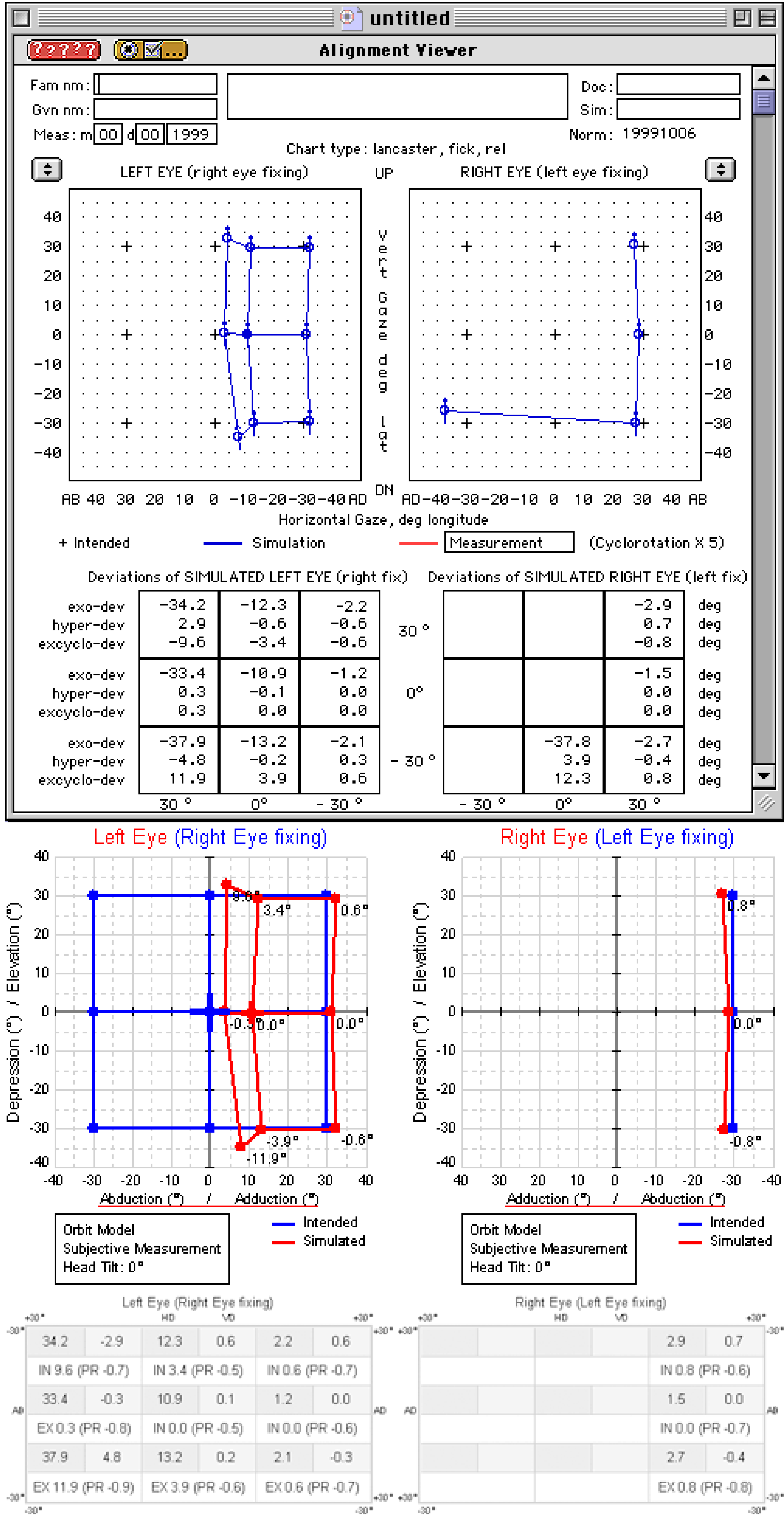
6. **Comparing simulation results between Orbit™ 1.8 Macintosh and Orbit™ implementations in SEE++:**

We would like to briefly show a comparision of a simulation between the two programs Orbit™ 1.8 and Orbit™ implementation within SEE++. For simulation we modified distribution of innervations of the abducens nerve to innervate medial and lateral rectus muscles of a left eye with a magnitude of 60%.

Below the parameter values that were changed for this simulation are shown in Orbit™ and SEE++:



The **simulation results** show that the calculated eye positions are **exactly the same in both programs** (see textual deviation angles), however, **SEE++ omits one fixation point** due to **loss of convergence explained in (4)**. This is the **rare case**, more **frequently deviation angles will show up to 0.1 degrees of difference** between both programs.



	30°	0°	-30°		-30°	0°	30°	
exo-dev	-34.2	-12.3	-2.2	30°			-2.9	deg
hyper-dev	2.9	-0.6	-0.6				0.7	deg
excyclo-dev	-9.6	-3.4	-0.6				-0.8	deg
exo-dev	-33.4	-10.9	-1.2	0°			-1.5	deg
hyper-dev	0.3	-0.1	0.0				0.0	deg
excyclo-dev	0.3	0.0	0.0				0.0	deg
exo-dev	-37.9	-13.2	-2.1	-30°		-37.8	-2.7	deg
hyper-dev	-4.8	-0.2	0.3			3.9	-0.4	deg
excyclo-dev	11.9	3.9	0.6			12.3	0.8	deg

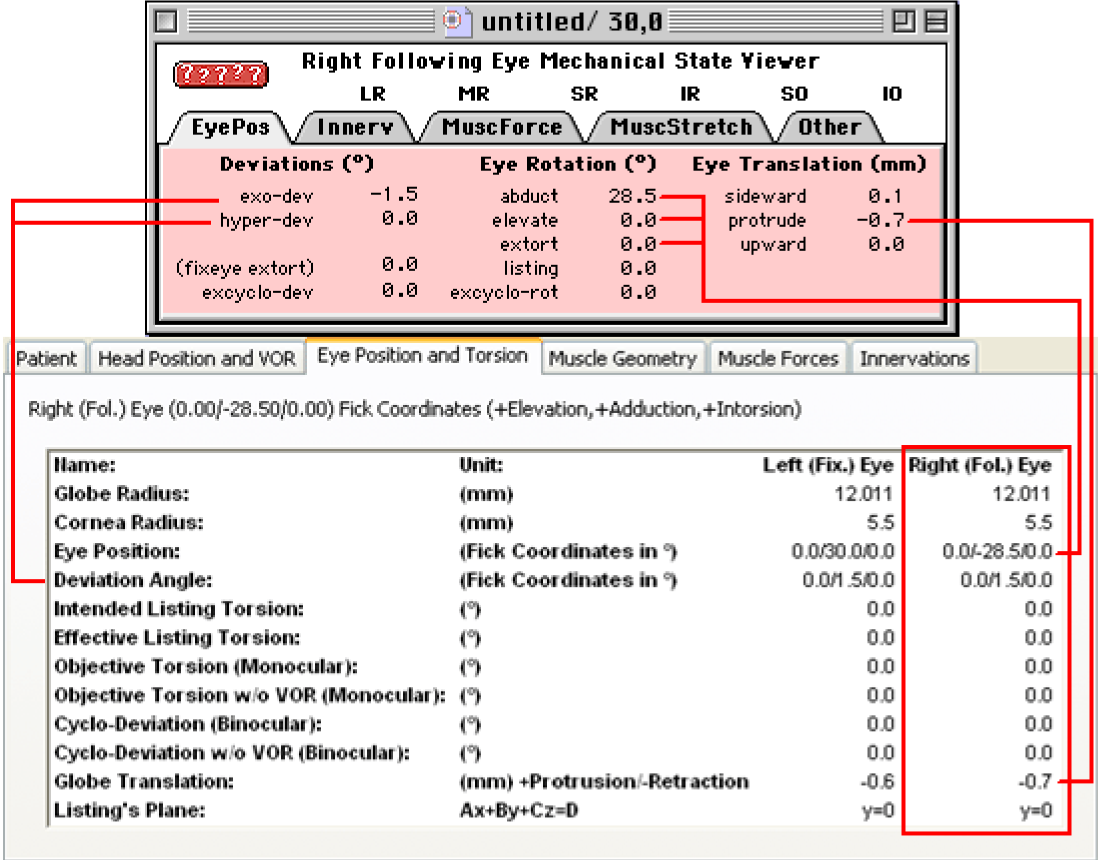
Left Eye (Right Eye fixing)

34.2	-2.9	12.3	0.6	2.2	0.6
IN 9.6 (PR -0.7)		IN 3.4 (PR -0.5)		IN 0.6 (PR -0.7)	
33.4	-0.3	10.9	0.1	1.2	0.0
EX 0.3 (PR -0.8)		IN 0.0 (PR -0.5)		IN 0.0 (PR -0.6)	
37.9	4.8	13.2	0.2	2.1	-0.3
EX 11.9 (PR -0.9)		EX 3.9 (PR -0.6)		EX 0.6 (PR -0.7)	

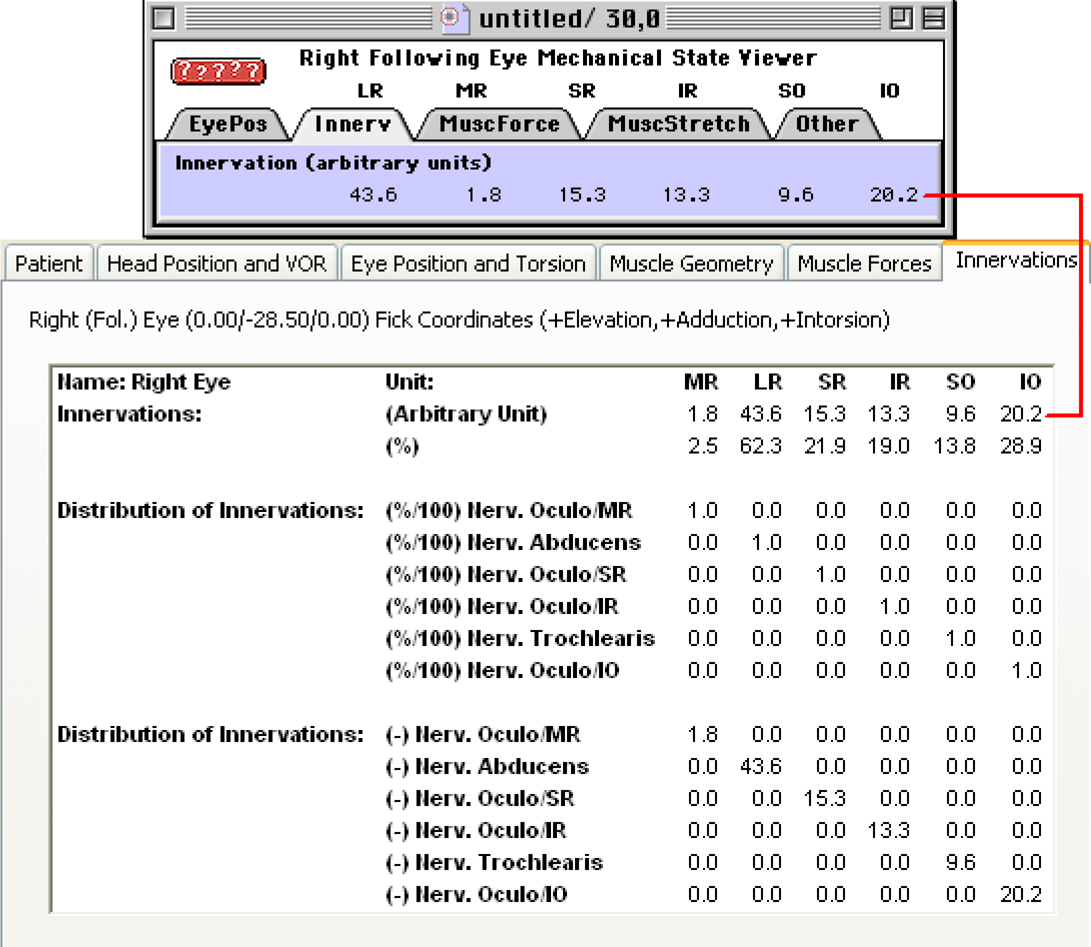
Right Eye (Left Eye fixing)

				2.9	0.7
				IN 0.8 (PR -0.6)	
				1.5	0.0
				IN 0.0 (PR -0.7)	
				2.7	-0.4
				EX 0.8 (PR -0.8)	

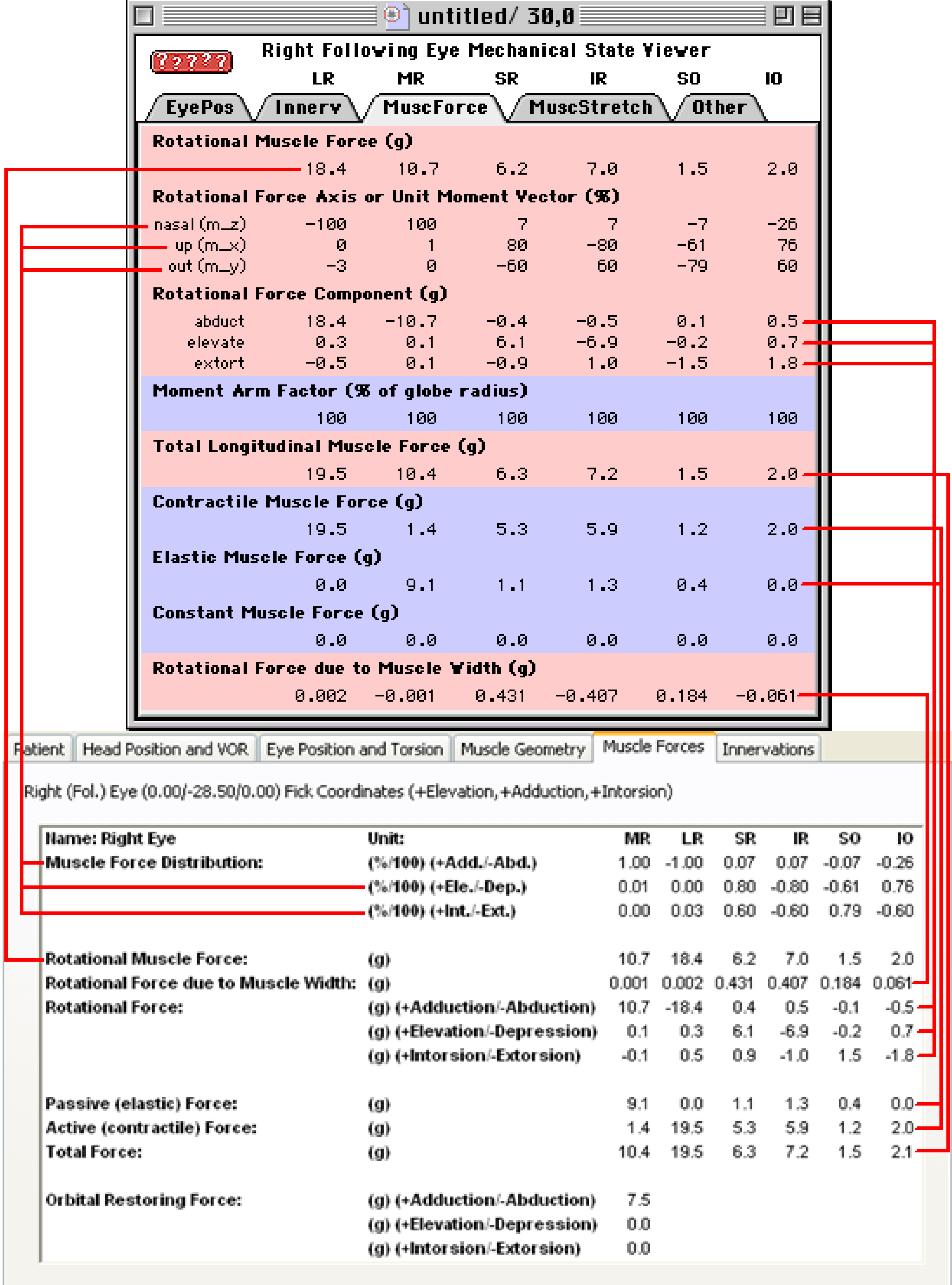
In order to **verify that both implementations are identical**, **stateviewer** values can be **compared**, and it can be seen that **all calculated model values for eye positions are the same**. Below the **comparsion of the fixation** of the **left eye at 30 degrees abduction** is shown, consequently the **right eye will move into adduction**. Please also note that **duction and torsion have opposite signs** in SEE++ compared to Orbit™.



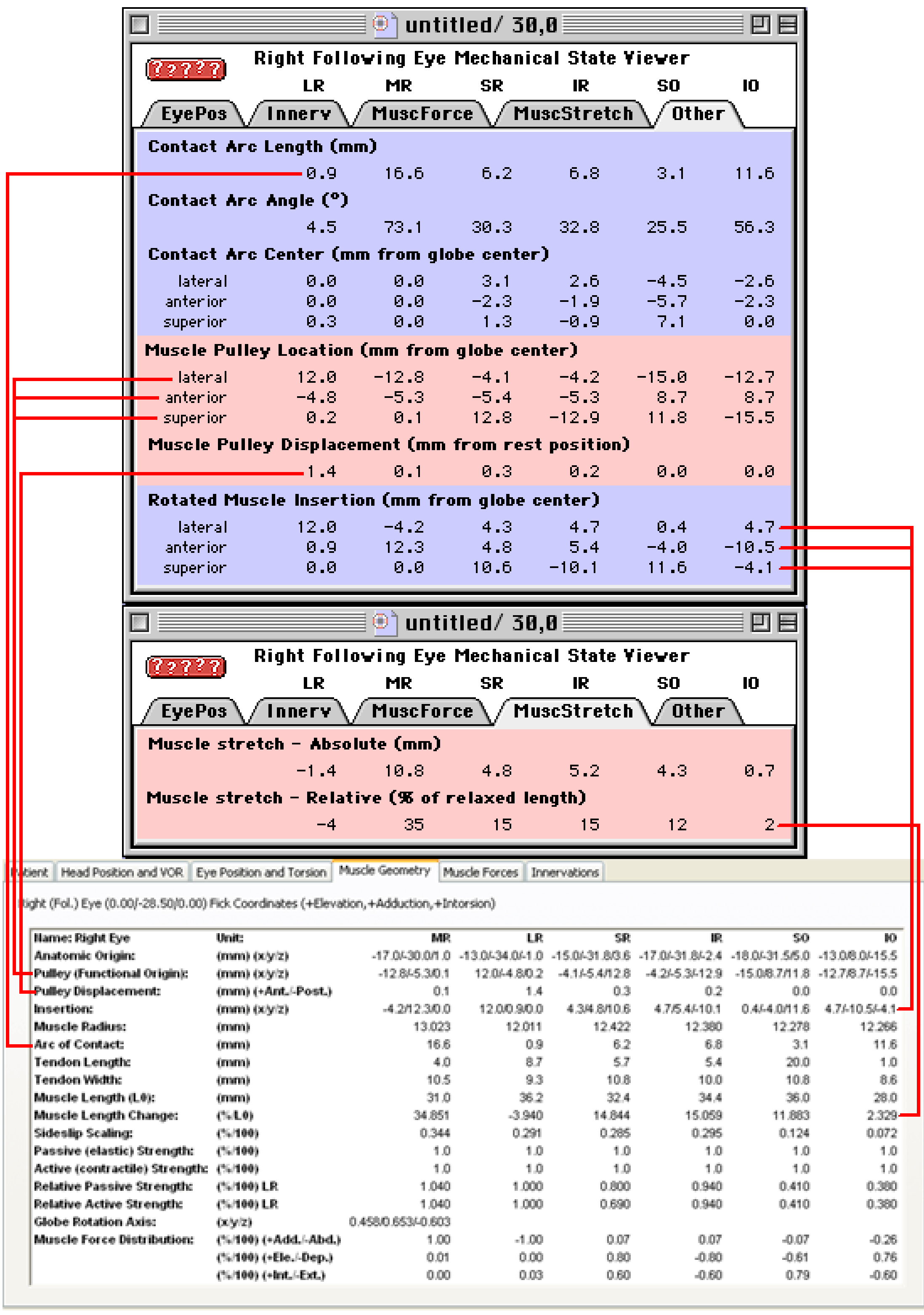
Innervations in both programs show identical values, however, in SEE++ **medial and lateral rectus** muscles are **swapped**:



Muscle forces are almost identical, **inferior oblique** total longitudinal muscle force **differs (0.1g difference)** due to **floating point inconsistencies** as explained in **(4)**:



Muscle geometry shows identical values for all parameters in both programs:



Name: Right Eye	Unit:	MR	LR	SR	IR	SO	IO
Anatomic Origin:	(mm) (x/y/z)	-17.0/-30.0/1.0	-13.0/-34.0/-1.0	-15.0/-31.8/3.6	-17.0/-31.8/-2.4	-18.0/-31.5/5.0	-13.0/8.0/-15.5
Pulley (Functional Origin):	(mm) (x/y/z)	-12.8/-5.3/0.1	12.0/-4.8/0.2	-4.1/-5.4/12.8	-4.2/-5.3/-12.9	-15.0/8.7/11.8	-12.7/8.7/-15.5
Pulley Displacement:	(mm) (+Ant./-Post.)	0.1	1.4	0.3	0.2	0.0	0.0
Insertion:	(mm) (x/y/z)	-4.2/12.3/0.0	12.0/0.9/0.0	4.3/4.8/10.6	4.7/5.4/-10.1	0.4/-4.0/11.6	4.7/-10.5/-4.1
Muscle Radius:	(mm)	13.023	12.011	12.422	12.380	12.278	12.266
Arc of Contact:	(mm)	16.6	0.9	6.2	6.8	3.1	11.6
Tendon Length:	(mm)	4.0	8.7	5.7	5.4	20.0	1.0
Tendon Width:	(mm)	10.5	9.3	10.8	10.0	10.8	8.6
Muscle Length (L0):	(mm)	31.0	36.2	32.4	34.4	36.0	28.0
Muscle Length Change:	(%L0)	34.851	-3.940	14.844	15.059	11.883	2.329
Sideslip Scaling:	(%/100)	0.344	0.291	0.285	0.295	0.124	0.072
Passive (elastic) Strength:	(%/100)	1.0	1.0	1.0	1.0	1.0	1.0
Active (contractile) Strength:	(%/100)	1.0	1.0	1.0	1.0	1.0	1.0
Relative Passive Strength:	(%/100) LR	1.040	1.000	0.800	0.940	0.410	0.380
Relative Active Strength:	(%/100) LR	1.040	1.000	0.690	0.940	0.410	0.380
Globe Rotation Axis:	(x/y/z)	0.458/0.653/-0.603					
Muscle Force Distribution:	(%/100) (+Add./-Abd.)	1.00	-1.00	0.07	0.07	-0.07	-0.26
	(%/100) (+Ele./-Dep.)	0.01	0.00	0.80	-0.80	-0.61	0.76
	(%/100) (+Int./-Ext.)	0.00	0.03	0.60	-0.60	0.79	-0.60

In case of sceptisism, we encourage the reader to **download both programs and gather own experiences** (if one can find a Macintosh that is old enough to run Orbit™ 1.8). Unfortunately, as **requested by Miller**, it is **impossible to specifiy** in which **situations differences occur** due to the **nature of floating point arithmetics and processor architectures**. However, in all our **numerous experiments**, we **never found differences greater that 0.1** degrees in resulting Hess-Lancaster simulation results.

In case that some reader reached this point of this document, we would be happy to hear your opinion and would like to receive feedback to see-kid@uar.at **.**

Last Updated (Wednesday, 06 June 2007)