Lab 6: Optimisation

MCHA4400

Semester 2 2023

Introduction

In this lab, you will derive gradients and Hessians of cost functions for use in optimisation using analytical derivatives and compare the performance against forward-mode and reverse-mode automatic differentiation. Then, using the Laplace approximation, you will complete the implementation of a second-order iterated EKF for the ballistic radar tracking problem from Lab 5.

The lab should be completed within 4 hours. The assessment will be done in the lab at the end of your enrolled lab session(s). Once you complete the tasks, call the lab demonstrator to start your assessment.

The lab is worth 5% of your course grade and is graded from 0–5 marks.

LLM Tip

You are strongly encouraged to explore the use of Large Language Models (LLMs), such as GPT, PaLM or LLaMa, to assist in completing this activity. Bots with some of these LLMs are available to use via the UoN Mechatronics Slack team, which you can find a link to from Canvas. If you are are unsure how to make the best use of these tools, or are not getting good results, please ask your lab demonstrator for advice.

1 Optimisation (1 mark)

In this task, you will implement the analytical gradient and Hessian for the Rosenbrock function (see Figure 1) and use both Newton and SR1 trust-region methods to find the minimum. The Rosenbrock function is given by

$$f(x_1, x_2) = (1 - x_1)^2 + 100(x_2 - x_1^2)^2. \tag{1}$$

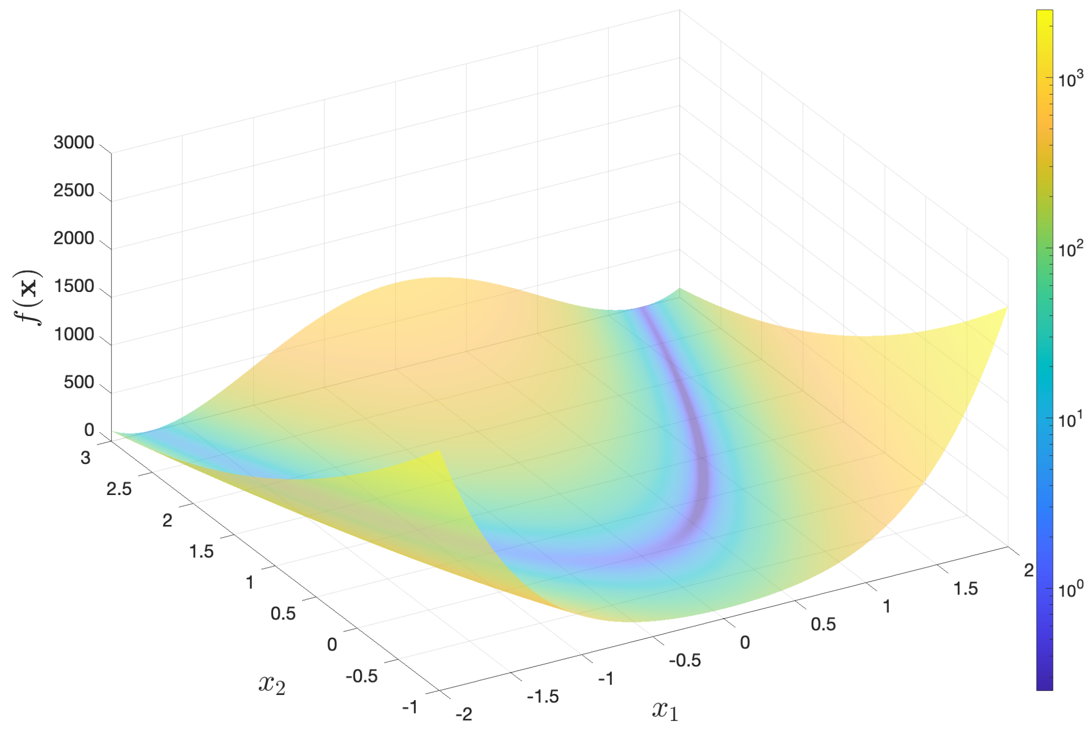

Figure 1: Rosenbrock function

Tasks

a) Implement the analytical gradient and Hessian in the `RosenbrockAnalytical` functor in `src/rosenbrock.cpp`.

Info

A functor (function object) is a C++ class that acts like a function. An instance of a functor can be called using the same syntax as a function, due to the implementation of `operator()`. In our case, it is convenient to bundle the Rosenbrock function and its derivatives into closely related functions within the same functor.

b) Build the application and ensure the unit tests in `test/src/rosenbrock.cpp` pass.

Terminal

```
nerd@basement:~/MCHA4400/lab6$ cmake -G Ninja -B build -DCMAKE_BUILD_TYPE=Debug && cd build
nerd@basement:~/MCHA4400/lab6/build$ ninja
[Lots of failing unit tests, pay attention to the relevant ones only]
[Rosenbrock benchmark results]
```

Tip

If the non-relevant unit tests are too distracting, you can temporarily move them out of the `test/src` directory and move them back in as needed for the following parts.

c) Switch to a release build, run `ninja` and review the benchmark results for the `Rosenbrock gradient only` and `Rosenbrock gradient and Hessian` cases. Compare the performance of forward-mode autodiff, reverse-mode autodiff, and the analytical expressions for evaluating the cost function gradient and Hessian.

Info

The performance benchmark is only meaningful in a release build, with assertions and bounds checking disabled and full compiler optimisations enabled. You can switch to a release build and run the benchmarks as follows:

Terminal

```
nerd@basement:~/MCHA4400/lab6/build$ cd ..
nerd@basement:~/MCHA4400/lab6$ cmake -G Ninja -B build -DCMAKE_BUILD_TYPE=Release && cd build
nerd@basement:~/MCHA4400/lab6/build$ ninja
[Lots of failing unit tests, pay attention to the relevant ones only]
[Rosenbrock benchmark results]
```

You can switch back to a debug build by running the above commands again with `Release` replaced with `Debug`.

d) In the `main` function, call the Newton trust region method `funcmin::NewtonTrust`, implemented in `src/funcmin.hpp`, to obtain the minimum of the Rosenbrock function.

e) Print the following to the console, with the verbosity of `funcmin::NewtonTrust` set to 3:

Terminal

```
nerd@basement:~/MCHA4400/lab6/build$ ninja; ./lab6
Note: The above command runs lab6 even if there are build errors or the unit tests fail. If you have
    ↪ build errors, this will run the stale executable.
[Ninja-ing intensifies]
[Output from tests and benchmarks]
Initial x =
10
10

f = 810081

g =
360018
-18000

H =
116002  -4000
 -4000    200

Running optimisation
Iter =      0, Cost =   8.10e+05, Newton decr^2 =   3.60e+05, Delta =   1.00e+00
Iter =      1, Cost =   5.04e+05, Newton decr^2 =   5.11e+05, Delta =   2.00e+00
Iter =      2, Cost =   1.51e+05, Newton decr^2 =   2.23e+05, Delta =   4.00e+00
[More output from optimisation iterations]
CONVERGED: Newton decrement below threshold in 38 iterations

Final x =
1
1

f = 6.48755e-20

g =
 3.45915e-09
-1.48592e-09

H =
 802 -400
-400  200
```

f) Replace the call to `funcmin::NewtonTrust` with `funcmin::SR1Trust` and re-run the optimisation. Note the number of iterations needed to converge and the quality of the Hessian approximation at the solution.

2 Log of a Gaussian (2 marks)

In the iterated EKF, the k^{th} measurement correction seeks the state value that maximises the posterior density $p(\mathbf{x}_k|\mathbf{y}_{1:k})$, given the measurement likelihood $p(\mathbf{y}_k|\mathbf{x}_k)$ and the prior density $p(\mathbf{x}_k|\mathbf{y}_{1:k-1})$,

$$\begin{aligned}
\boldsymbol{\mu}_{k|k} &= \arg\max_{\mathbf{x}_k} p(\mathbf{x}_k|\mathbf{y}_{1:k}) \\
&= \arg\max_{\mathbf{x}_k} \frac{p(\mathbf{y}_k|\mathbf{x}_k)\, p(\mathbf{x}_k|\mathbf{y}_{1:k-1})}{p(\mathbf{y}_k|\mathbf{y}_{1:k-1})} \\
&= \arg\max_{\mathbf{x}_k} p(\mathbf{y}_k|\mathbf{x}_k)\, p(\mathbf{x}_k|\mathbf{y}_{1:k-1}) \\
&= \arg\max_{\mathbf{x}_k} p(\mathbf{y}_k|\mathbf{x}_k)\, \mathcal{N}(\mathbf{x}_k; \boldsymbol{\mu}_{k|k-1}, \mathbf{P}_{k|k-1}).
\end{aligned}$$

In practice, it is more convenient to minimise the negative log of the above cost, i.e., let

$$\mathcal{V}(\mathbf{x}_k) = -\log p(\mathbf{y}_k|\mathbf{x}_k) - \log \mathcal{N}(\mathbf{x}_k; \boldsymbol{\mu}_{k|k-1}, \mathbf{P}_{k|k-1}). \tag{2}$$

Then, under the Laplace approxmation, the posterior mean and covariance are given by

$$\boldsymbol{\mu}_{k|k} = \arg\min_{\mathbf{x}_k} \mathcal{V}(\mathbf{x}_k),$$

$$\mathbf{P}_{k|k} = \left(\left. \frac{\partial^2 \mathcal{V}}{\partial \mathbf{x}_k^2} \right|_{\mathbf{x}_k = \boldsymbol{\mu}_{k|k}} \right)^{-1}.$$

The optimisation to find $\boldsymbol{\mu}_{k|k}$ and $\mathbf{P}_{k|k}$ requires the gradient and Hessian,

$$\mathbf{g}_k = \frac{\partial \mathcal{V}}{\partial \mathbf{x}_k} = -\frac{\partial}{\partial \mathbf{x}_k} \log p(\mathbf{y}_k|\mathbf{x}_k) - \frac{\partial}{\partial \mathbf{x}_k} \log \mathcal{N}(\mathbf{x}_k; \boldsymbol{\mu}_{k|k-1}, \mathbf{P}_{k|k-1}), \tag{3}$$

$$\mathbf{H}_k = \frac{\partial^2 \mathcal{V}}{\partial \mathbf{x}_k^2} = -\frac{\partial^2}{\partial \mathbf{x}_k^2} \log p(\mathbf{y}_k|\mathbf{x}_k) - \frac{\partial^2}{\partial \mathbf{x}_k^2} \log \mathcal{N}(\mathbf{x}_k; \boldsymbol{\mu}_{k|k-1}, \mathbf{P}_{k|k-1}), \tag{4}$$

respectively.

Info

The cost (2), gradient (3) and Hessian (4) are implemented in `Measurement::costJointDensity`, which rely on a descendant class to implement `Measurement::logLikelihood`, which evaluates the log likelihood and its derivatives for a particular sensor type. The optimisation problem itself is solved in `Measurement::update`.

Since the prior is Gaussian, we therefore need a function to evaluate the log of a Gaussian distribution and the first two derivatives of that function. In addition, if the measurement likelihood is also Gaussian, i.e.,

$$p(\mathbf{y}_k|\mathbf{x}_k) = \mathcal{N}(\mathbf{y}_k; \mathbf{h}(\mathbf{x}_k), \mathbf{R}),$$

for some mean function $\mathbf{h}(\cdot)$ and covariance $\mathbf{R}$, then we also need to be able to compute the derivatives, $\frac{\partial}{\partial \mathbf{x}_k} \log \mathcal{N}(\mathbf{y}_k; \mathbf{h}(\mathbf{x}_k), \mathbf{R})$ and $\frac{\partial^2}{\partial \mathbf{x}_k^2} \log \mathcal{N}(\mathbf{y}_k; \mathbf{h}(\mathbf{x}_k), \mathbf{R})$.

Info

To enable the possibility of computing the derivatives of the log of a Gaussian measurement likelihood function using automatic differentiation, the `Gaussian` class has been templated to take an arbitrary `Scalar` type, such as `double` (default), `autodiff::real`, `autodiff:dual`, `autodiff:dual2nd` or `autodiff::val`. It is now entirely implemented in `src/Gaussian.hpp`.

Tasks

a) The log of a Gaussian probability density function is given by

$$\log \mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \mathbf{P}) = \log \left((\det 2\pi \mathbf{P})^{-\frac{1}{2}} \exp\left(-\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu})^\mathsf{T} \mathbf{P}^{-1} (\mathbf{x} - \boldsymbol{\mu}) \right) \right). \tag{5}$$

A naïve implementation of (5) is already provided, but suffers from numerical issues. Complete a numerically robust implementation of (5) and its first two derivatives as the three `Gaussian::log` functions within `src/Gaussian.hpp`.

Tip

Since the members of `Gaussian` are dependent on the `Scalar` type, the compiler may need some extra hints by way of the `typename` and `template` keywords to help deduce if particular tokens are types or templates, respectively. For example, if you need an upper triangular view of `S_`, which is a dependent type `Eigen::MatrixX<Scalar>`, the `template` keyword may be required, i.e.,

```
S_.template triangularView<Eigen::Upper>().solve(...)
```

This is needed because type of `S_` is not known until the template is instantiated for a particular `Scalar` template argument, and prior to instantiation, the compiler cannot otherwise parse the syntax of the template.

b) Run `ninja` and ensure the unit tests in `test/src/GaussianLogLikelihood.cpp` pass.

c) Re-implement `Gaussian::marginal` and `Gaussian::conditional` based on your Lab 5 solution and ensure the unit tests in `test/src/GaussianMarginal.cpp` and `test/src/GaussianConditional.cpp` pass.

3 Iterated EKF (2 marks)

In this task you will implement the log-likelihood function for the RADAR sensor from Lab 5. The log-likelihood function is given by

$$\log p(y_{\text{rng}} \mid \mathbf{x}) = \log \mathcal{N}(y_{\text{rng}}; h_{\text{rng}}(\mathbf{x}), \sigma_{\text{rng}}^2), \tag{6}$$

where $h_{\text{rng}}(\cdot)$ and σ_{rng} are defined in Lab 5.

Tasks

a) Re-implement `StateBallistic::dynamics` from your Lab 5 solution and ensure the unit tests in `test/src/StateBallistic.cpp` pass.

b) Re-implement `src/ballistic_plot.cpp` based on your Lab 5 solution.

c) The gradient and Hessian of the measurement log likelihood, (6), can be computed using one of the following three methods (you only need to complete one method for this lab):

i) **Analytical derivatives:** Implement $\mathbf{h}(\mathbf{x})$ and its first two derivatives in the three `MeasurementRADAR::predict` member functions within `src/MeasurementRADAR.cpp` and then implement (6) and its first two derivatives in the three `Measurement::logLikelihood` member functions within `src/Measurement.cpp`. This can be achieved as follows:

- In `test/src/MeasurementRADAR.cpp`, uncomment the unit tests that cover all three `MeasurementRADAR::predict` member functions.
- In `src/MeasurementRADAR.cpp`, complete the implementation of the two-argument member function `MeasurementRADAR::predict`, which also computes the Jacobian matrix $\mathbf{J}$

such that

$$J_{ij} = \frac{\partial h_i}{\partial x_j},$$

which you previously implemented in Lab 5.

- In `src/MeasurementRADAR.cpp`, complete the implementation of the three-argument member function `MeasurementRADAR::predict`, which also computes the Hessian tensor **H** such that

$$H_{ijk} = \frac{\partial^2 h_i}{\partial x_j \partial x_k}.$$

- In `src/Measurement.cpp`, complete the implementation of the two-argument member function `Measurement::logLikelihood`, which also computes the gradient vector **g** such that

$$g_i = \frac{\partial}{\partial x_i} \log \mathcal{N}(\mathbf{y}; \mathbf{h}(\mathbf{x}); \mathbf{R}).$$

  Note that this implementation is intentionally in the base class `Measurement`, and not the `MeasurementRADAR` class.

- In `src/Measurement.cpp`, complete the implementation of the three-argument member function `Measurement::logLikelihood`, which also computes the Hessian matrix **H** such that

$$H_{ij} = \frac{\partial^2}{\partial x_i \partial x_j} \log \mathcal{N}(\mathbf{y}; \mathbf{h}(\mathbf{x}); \mathbf{R}).$$

  Note that this implementation is intentionally in the base class `Measurement`, and not the `MeasurementRADAR` class.

- In `src/MeasurementRADAR.cpp`, set the `DERIVATIVE_MODE` define to `DERIVATIVE_ANALYTICAL`.

ii) **Forward-mode autodifferentiation:** Use `Scalar`-templated implementations, `MeasurementRADAR::predictImpl<Scalar>` and `MeasurementRADAR::logLikelihoodImpl<Scalar>` and `Gaussian<Scalar>::log`, to set up a chain of functions to perform forward-mode autodifferentiation to compute the gradient and Hessian of the log likelihood function (6).

Hint

Review the gradient and Hessian autodiff implementations from the forward-mode and reverse-mode Rosenbrock functors, and those provided in the demo unit tests given in Lab 1.

This can be achieved as follows:

- In `src/MeasurementRADAR.cpp`, complete the implementation of the two-argument member function `MeasurementRADAR::logLikelihood`, which also computes the gradient vector $\mathbf{g}$ such that

$$g_i = \frac{\partial}{\partial x_i} \log \mathcal{N}(\mathbf{y}; \mathbf{h}(\mathbf{x}); \mathbf{R}).$$

Hint

The forward-mode function `gradient` requires a *callable* type as its first argument, i.e., almost anything that can be called like a function using `()`. This includes functions, functors, lambda expressions, function pointers and member function pointers, but not member functions themselves. To compute forward-mode gradients on a member function, we could wrap it in a lambda expression, or use `std::bind` or use a member function pointer. As an example of the latter, if the class `C` has a template member function called `f<>()` that we want to compute the gradient w.r.t. its first argument, then can do the following:

```cpp
double C::f_grad_fwd(const Eigen::VectorXd & x, Eigen::VectorXd & g) const
{
    Eigen::VectorX<autodiff::dual> xdual = x.cast<autodiff::dual>();
    autodiff::dual fdual;
    g = gradient(&C::f<autodiff::dual>, wrt(xdual), at(this, xdual), fdual);
    return val(fdual);              // cast return value to double
}
```

Note that the keyword `this` appears as the first argument in `at()` because under the hood, the member function call `obj.f(x)` is implemented similarly to a non-member function call `f(obj, x)`.

- In `src/MeasurementRADAR.cpp`, complete the implementation of the three-argument member function `MeasurementRADAR::logLikelihood`, which also computes the Hessian matrix $\mathbf{H}$ such that

$$H_{ij} = \frac{\partial^2}{\partial x_i \partial x_j} \log \mathcal{N}(\mathbf{y}; \mathbf{h}(\mathbf{x}); \mathbf{R}).$$

Hint

Review the Hessian autodiff implementations from the forward-mode Rosenbrock functors, and those provided in the demo unit tests given in Lab 1.

- In `src/MeasurementRADAR.cpp`, set the `DERIVATIVE_MODE` define to `DERIVATIVE_FORWARD_AUTODIFF`.

iii) **Reverse-mode autodifferentiation:** Use `Scalar`-templated implementations, `MeasurementRADAR::predictImpl<Scalar>` and `MeasurementRADAR::logLikelihoodImpl<Scalar>` and `Gaussian<Scalar>::log`, to set up a chain of functions to perform reverse-mode autodifferentiation to compute the gradient and Hessian of the log likelihood function (6). This can be achieved as follows:

- In `src/MeasurementRADAR.cpp`, complete the implementation of the two-argument member function `MeasurementRADAR::logLikelihood`, which also computes the gradient vector $\mathbf{g}$ such that

$$g_i = \frac{\partial}{\partial x_i} \log \mathcal{N}(\mathbf{y}; \mathbf{h}(\mathbf{x}); \mathbf{R}).$$

Hint

The reverse-mode function `gradient` requires an `autodiff::var` variable that contains the expression tree built from a forward pass through the function. For example, if the class `C` has a template member function called `f<>()` that we want to compute the gradient w.r.t. its first argument, then can do the following:

```
double C::f_grad_rev(const Eigen::VectorXd & x, Eigen::VectorXd & g) const
{
    Eigen::VectorX<autodiff::var> xvar = x.cast<autodiff::var>();
    autodiff::var fvar = f(xvar);   // Build expression tree
    g = gradient(fvar, xvar);       // Evaluate derivatives from tree
    return val(fvar);               // cast return value to double
}
```

Note that the keyword `this` appears as the first argument in `at()` because under the hood, the member function call `obj.f(x)` is implemented similarly to a non-member function call `f(obj, x)`.

- In `src/MeasurementRADAR.cpp`, complete the implementation of the three-argument member function `MeasurementRADAR::logLikelihood`, which also computes the Hessian matrix $\mathbf{H}$ such that

$$H_{ij} = \frac{\partial^2}{\partial x_i \partial x_j} \log \mathcal{N}(\mathbf{y}; \mathbf{h}(\mathbf{x}); \mathbf{R}).$$

Hint

Review the Hessian autodiff implementations from the reverse-mode Rosenbrock functors, and those provided in the demo unit tests given in Lab 1.

- In `src/MeasurementRADAR.cpp`, set the `DERIVATIVE_MODE` define to `DERIVATIVE_REVERSE_AUTODIFF`.

Foreshadowing

Although not required for this lab, it may be helpful to implement all three methods above so that you are familiar with the process of computing derivatives both analytically and automatically through a non-trivial implementation.

d) Ensure the unit tests in `test/src/MeasurementRADAR.cpp` relevant to the derivative method chosen above pass.

e) Remove the early `return` statement within `src/main.cpp` at the end of Problem 1.

f) Build and run the application and ensure you get the following output:

Terminal

```
nerd@basement:~/MCHA4400/lab6/build$ ninja && ./lab6
[Ninja-ing intensifies]
[All tests pass]
[Benchmark results]
[Output from Problem 1]
Reading data from ../data/estimationdata.csv
Found 501 rows within ../data/estimationdata.csv

Initial state estimate
mu[0] =
 14000
  -450
0.0005
P[0] =
4.84e+06        0        0
       0    10000        0
       0        0    1e-06
Run filter with 500 steps.
[k=0] Processing event:.............done
[k=1] Processing event:.......done
[k=2] Processing event:.........done
[More output]
[k=499] Processing event:....done

Final state estimate
mu[end] =
    2185.02
   -152.733
0.000982087
P[end] =
    216.725     10.7622 0.000163737
    10.7622     1.57177  2.5506e-05
0.000163737  2.5506e-05 4.79731e-10
```

Ensure the figure generated matches Figure 2.

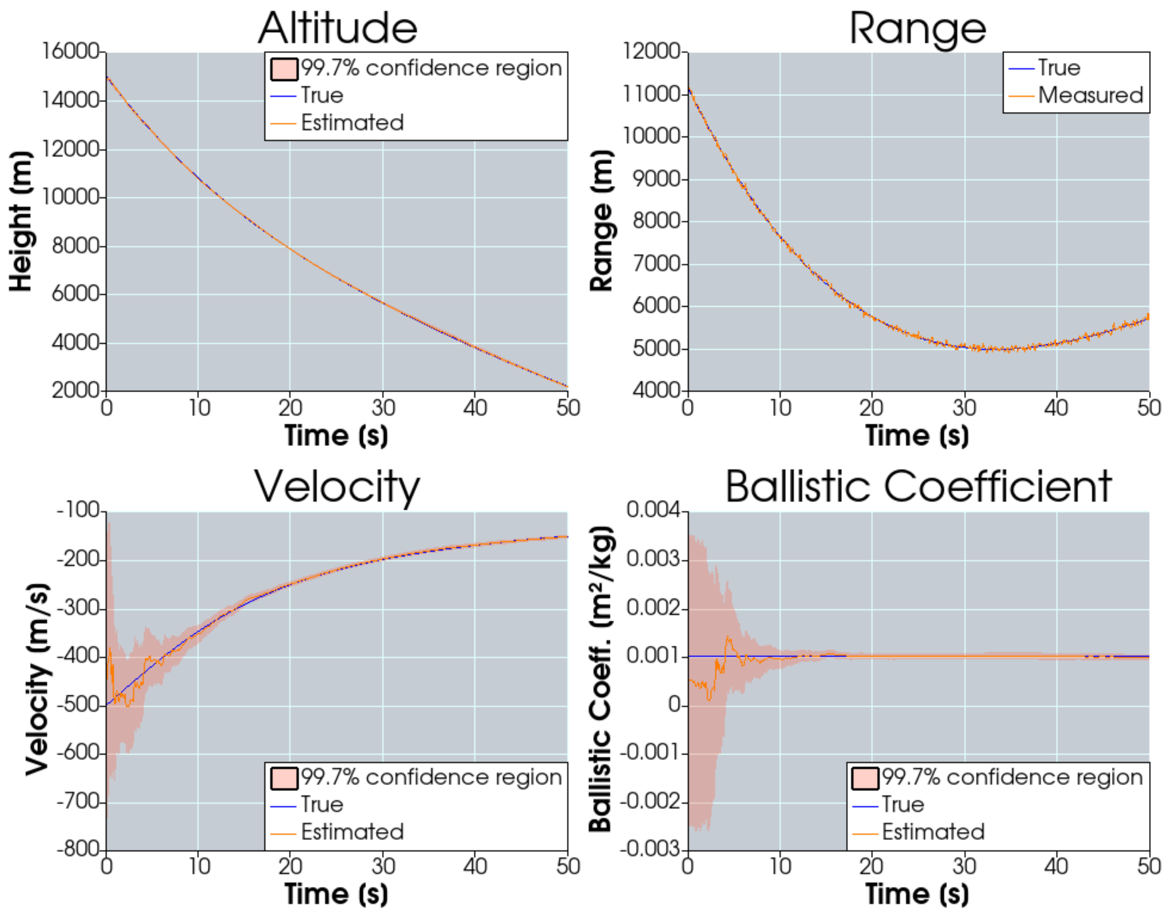

Figure 2: Expected output plot from a working iterated EKF