Imperial College London
Department of Computing

Bundle Adjustment with PyOP2

Paul Gribelyuk

August 2013

Supervised by Professor Paul H J Kelly

Submitted in part fulfilment of the requirements for the degree of
Master of Science in Computing of Imperial College London

Abstract

We present a general approach for solving 'bundle adjustment', a problem which is central to scene understanding algorithms in comupter vision, and compare performance characteristics with existing approaches.

In the context of computer vision in autonomous robotic systems, bundle adjustment is the search for a large set of parameter blocks (bundles) which optimally fit incoming sensor data. In a typical scenario, data is presented as a series of camera image frames. A feature extraction algorithm produces a set of 'landmarks' from each frame, which may overlap with other frames. Since both robot motion and measurement equipment inevitably inject uncertainty into the measurement of those landmarks, an optimization algorithm is used to reconcile data discrepancies.

The formulation of this optimization algorithm is framed as the minimization of squared-errors and can be solved by a variety of methods. Commonly, this is a non-linear least-squares (NLS) problem amenable to gradient descent or Gauss-Newton minimization techniques.

In this paper, we take a similar approach, and consider opportunities for performance improvement via an existing code generation framework, PyOP2. We looked at a variety of data sizes to analyze the degree of performance gains obtained versus three existing popular approaches.

I would like to thank my supervisor, Professor Paul Kelly for his guidance, and time in completing this project. I would also like to thank Professor Andrew Davidson and Renato Salas-Moreno who helped me understand the finer points in computer vision and steer me in the right direction.

Contents

List of Tables

List of Figures

1. Introduction

We analyze the general setting of the bundle adjustment problem from the computer vision context and investigate techniques for speeding up the steps involved in the calculation, using automatic differentiation as well as heterogeneous compilation tools. We use the Sympy Python package [24] for symbolic representation and automatic differentiation of positional parameters, vectors, motion functions, and error vectors and PyOP2 [20] for executing kernels used. We show that many of the steps can be executed in parallel, and we outline a domain specific language which enhances programmability of these problems. Lastly we compare popular bundle adjustment solvers such as ceres-solver from Google [1], the g2o package [13], and the iSAM package [12].

1.1. Motivation

Robot scene recognition researches have historically relied on filtering techniques, such as the Extended Kalman Filter [19], Particle Filters [26], and Rao-Blackwellised Filters [11], to solve the many steps of a larger process known as Simultaneous Localization and Mapping (SLAM). In the SLAM setting, a robot is placed in an unknown environment with sensors measuring locations of surrounding landmarks as it navigates this new environment. Prior to each time step, the robot stores a prior probability distribution for each landmark as well as its own current and past locations. As it obtains new measurements of the surrounding environment, it produces a posterior distribution for each landmark. In the Kalman Filtering framework, properties of Bayesian probability laws are are used to make each update with an implied Gaussian distribution for errors. Particle Filtering methods also implement Bayes' Law but, instead choose Monte Carlo Simulation to generate an estimate of the probability distribution.

In contrast to these methods, bundle adjustment emerged from photogrammetry research in the 60s and 70s, mainly with military and geographical applications. Computations on collected image data were done offline and were typically very time consuming. In the past 10 years, advances in computing hardware and novel architectures (e.g. multicore, simultaneous multithreading, vectorized instructions, general purpose GPU computing, etc.), have narrowed the gap between filtering and bundle adjustment techniques allowing researchers to seriously consider bundle adjustment frameworks as an alternative to Kalman Filtering in online robot vision problems. For example, Salas-Moreno et. al. [21] use this approach to optimize and reconstruct object scenes. Furthermore, algorithms to solve sparse linear systems have also evolved, with tools such as METIS, PetSc, and Eigen able to take advantage of sparsity considerations to efficiently parallelize computations and make the computations easier to program. These developments, when properly utilized, can help robotic systems solve very large scene recognition problems in real-time. As of this writing, the only notable attempt we have found at applying techniques from the high performance computing toolbox towards speeding up bundle adjustment is the paper on multicore bundle adjustment by Wu et. al. [27]. We believe there is more work to be done to more easily expose the necessary functionality in a bundle adjustment solver without burying the user with unnecessary abstractions or hardware-specific implementation details.

For this purpose, we leverage PyOP2, a framework for performing finite element computations over unstructured meshes. The architecture of PyOP2 modularizes domains of expertise without sacrificing performance. We show that it can be used to efficiently perform bundle adjustment computations. We use Python as a staging language for our analysis.

1.2. Objectives

We aim to explore the computational challenges of performing bundle adjustment in scene recognition problems and propose a set of routines to speed up these computations using available frameworks. Benchmarking on available datasets, varying from 900 poses and 1900 landmarks to 500,000 poses and 2,100,000 landmarks, shows the efficiency of using our approach in comparison to available packages. These benchmarks should also help guide research into software optimization tools with a view towards accelerating vision applications.

1.3. Outline

The following report is organized in the following way:

- Chapter 2 covers the mathematical and theoretical background of bundle adjustment, which lays the groundwork for understanding the computational steps taken during the implementation. It also provides a simple example of bundle adjustment to give the reader a flavor of the various quantities involved.

- Chapter 3 summarizes previous work in this field. Since interest in novel techniques to solve bundle adjustment has intensified in the last 10 years, we outline the approaches taken by g2o, ceres-solver, and iSAM. Although there are other libraries, these are chosen for their performance and clean code base.

- Chapters 4 and 5 cover the PyOP2 kernels used, as well as the use of Theano for automatic differentiation when constructing the error function and the Jacobian and Hessian matrices. Here, we also outline the computational complexity in the problem and explain how our approach overcomes these obstacles. We present results compared to other bundle adjustment implementations.

- Chapter 6 produces the conclusion of our work, summarizing what we covered in this thesis. It also provides a guide to further research that can (and should) be conducted in this field.

2. Background

In this chapter, we outline the structure of the bundle adjustment problem in the context of an autonomous vehicle which uses sensors to navigate in a potentially unknown environment. The etymology of the term refers to the ‘bundles‘ of light rays which originate at each feature and terminate at the camera’s lens. The goal is to optimally determine the coordinates of measured landmarks as well as its own current and past positions. In the process of moving around the environment and collecting data, error inevitably creeps into the estimation as both motion and measurement cannot be calculated precisely. First, we produce an expectation of landmark and pose positions using update rules governed by physics. Next, we evaluate new measurements of these positions, which produces an array of errors. Since our goal is to minimize these simultaneously, we construct a functional from these errors (which themselves are a function of the parameters we are trying to find optimal values for), and look for ways to minimize it. It is very important to note that our functional has a non-linear relationship to the parameters we are trying to estimate. Thus, we must rely on iterative methods to approach the solution.

We also cover the PyOP2 framework, which is instrumental to our implementation. This framework is an extension of work started by Giles et. al. at Oxford University [18] to allow domain specialists in fluid dynamics to write C++ applications and run them on heterogeneous architectures. PyOP2, in contrast, uses a Python front-end, which allows the user to more easily set up and run finite element schemes. It also uses and builds upon automated kernel generation tools available through the FENICS framework [14].

2.1. Bundle Adjustment

We aim to outline the mathematical underpinnings of bundle adjustment methods in this section and to lay the groundwork for the choices for our implementation. We will begin with the geometry of camera (more generally: sensor) measurements and a background on how incoming image data is used to obtain information about specific landmarks in the scene. After that, we will discuss the dynamics of robot motion and its relationship to the expectation of landmark positions from different poses. We use the output of this calculation to calculate the ‘error‘, given specific camera measurements. Next, we review the theory of non-linear least squares, especially sparse matrix techniques, which underly the search for optimal bundle parameters. This iterative approach is elucidated in the final subsection.

2.1.1. Preprocessing

We begin by considering the two-dimensional image which represents a three-dimensional scene. To determine specific landmarks present in the scene, the system first employs a feature recognition algorithm, for example, a blob detector. Typically, an image is represented by a mapping $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. In the case of the ‘Laplacian of Gaussian‘ feature detector, a convolution between the image and a Gaussian kernel is performed:

$$L(x, y; t) = g(x, y; t) \star f(x, y)$$

where $g(x, y; t) = \frac{1}{2\pi t} \exp\{-(x^2 + y^2)/(2t)\}$ is the Gaussian. Next, the Laplace operator $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial}{\partial y^2}$ is applied to $L(x, y; t)$ exposing extrema for dark and light ‘blobs‘ present in the image. The variance parameter t also acts as a scale factor, with smaller values picking up smaller features in an image, while larger values of t only output larger features. Alternatively, an *edge detection* algorithm, such as the Canny Edge Detector [7], uses a multistep approach:

- Blur the image by convolving with a Gaussian filter
- Compute edge strength by applying Sobel operators on the smoothed

image:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

The edge strength at each pixel is then $\sqrt{G_x^2 + G_y^2}$ and the edge direction is $\theta = \tan^{-1}\left(\frac{G_y}{G_x}\right)$.

- A pixel is determined to be an edge if it has a maximum gradient compared to its neighbors and that gradient meets a minimum threshold

- A final 'hysteresis' step is used to identify edges which might not meet threshold gradients, but do lie next to pixels which do

These techniques typically parallelize well on GPUs, since there are minimal read or write conflicts in the data. More advanced techniques, using scale and rotation invariance [16] allow for matching under more general camera and robot transformations. These involve robust algorithms to hash and store feature descriptors for quick comparison and retrieval. As a practical example, the OpenCV package [4] provides a function to calculate Canny edges:

Listing 2.1: Using OpenCV for Canny edge detection

```
using namespace cv;
// first apply a 3x3 blur to de-noise the image
blur( src_grayscale, edges_only_dst, Size(3,3) );
// Canny(...) uses a low and high thresholds and Sobel size
Canny(edgesMat, edges_only_dst, low, low*ratio, sobel_size );
dst = Scalar::all(0);
src.copyTo( dst, edges_only_dst);
```

This code snippet does the work to produce results in figure [?]. Once the system has identified landmarks using one of the previously mentioned techniques, determining which common landmarks are shared by multiple images is known as the *correspondence problem* in computer vision. Researchers in this field apply a variety of tools such as *normalized cross cor-*

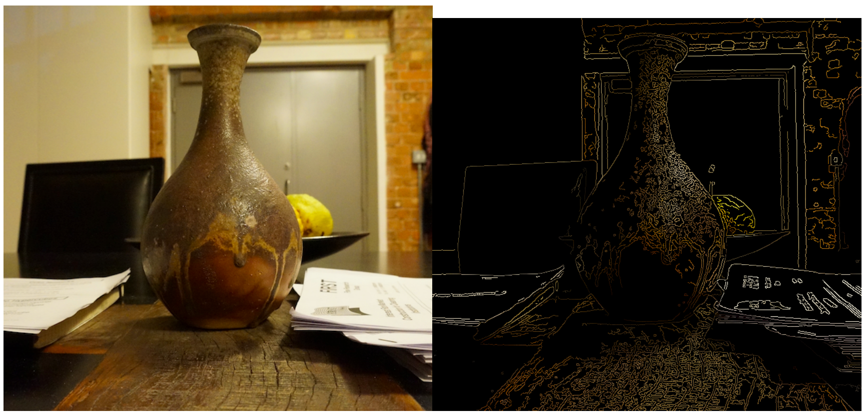
Figure 2.1.: Canny Edge Detection in Action

relation between a template $t(x, y)$ and a sub-image $f(x, y)$:

$$\frac{1}{n}\sum_{x,y}\frac{(f(x,y)-\bar{f})(t(x,y)-\bar{t})}{\sigma_f\sigma_t}$$

to determine the probability of that sub-image matching the template. Generally, this is a difficult problem in computer vision with an active research community.

2.1.2. Modelling the Camera

We now take a look at the way the camera measures landmarks. Specifically, the camera's orientation and position, with respect to a global coordinate frame, determine our expectation of where in that global frame the landmark is located. Since we are interested in minimizing the error between our estimated landmark position and the measurement, it is important to convert the pixel locations (which is our input from the camera) to global coordinate locations. The camera can be considered to have a position $\mathbf{x}$, which is a combination of a rotation R and a translation $\mathbf{t}$ (using the notation used by Strasdat et. al [23]). The rotation matrix R can be represented in many ways. In, two dimensions,

$$R_{2D} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

uniquely describes the counterclockwise rotation of any vector $\mathbf{x} \in \mathbb{R}^2$ by θ radians about the origin. However, in three dimensions, multiple representations are possible. For example, the Rodrigues rotation formula describes 3D rotations by θ degrees about a unit-length axis vector $\mathbf{u} \in \mathbb{R}^3$:

$$R_{3D} = I\cos\theta + \sin\theta[\mathbf{u}]_\times + (1-\cos\theta)\mathbf{u}\otimes\mathbf{u}$$

Alternatively, one can think of applying rotations γ, β, α (yaw, pitch, and roll) about the $x-$, $y-$, and $z-$ axes respectively:

$$\begin{aligned} R_{3D} &= R_x(\gamma)R_y(\beta)R_z(\alpha) \\ &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned}$$

This is known as a the Euler angles method.

Thus, a point, $\mathbf{y}$ seen in the local coordinates of the camera image has a global coordinate location

$$R\mathbf{y} + \mathbf{t}$$

With this background, the camera can be modeled in a variety of different ways. One of the most common and also simplest models is the *pinhole model* A 3D world coordinates point $\mathbf{P} = (X, Y, Z)$ has a camera coordinate point $\mathbf{p} = (x, y)$, thus, after adjusting for focal length f and horizontal and vertical scale parameters k and l, we have:

$$x = kf\frac{X}{Z} \qquad \text{and} \qquad y = lf\frac{Y}{Z}$$

Further parameters can be added to correct for lens distortion (fisheye vs barrel). Wu et. al. [27] model these parameters as a scalar factor of the pixel coordinates $\mathbf{p}$:

$$r(\mathbf{p}) = 1 + k_1|\mathbf{p}|^2 + k_2|p|^4$$

2.1.3. Modelling Motion

As discussed in the previous section, it is possible to describe the position and pose of a camera in global coordinates via a 3-dimensional position

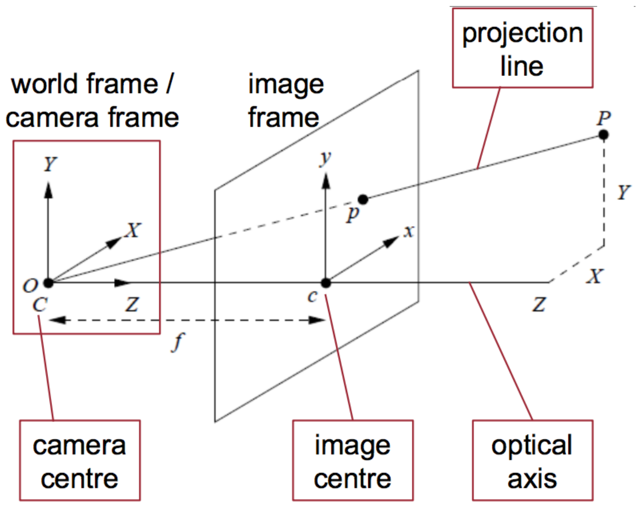

Figure 2.2.: Pinhole Camera Setup

vector (x, y, z) and a 3-dimensional orientation vector (corresponding to pitch, roll, and yaw) $(\gamma,\ \alpha,\ \beta)$. A 2D motion operator describing a rigid object moving from pose $a = (x_1, y_1, \theta_1)$ to pose $b = (x_2, y_2, \theta_2)$ is first a translation, then a rotation, thus the naive Python class representing this is (note: SE2 is the Special Euclidean group for 2D rigid motion):

Listing 2.2: Python code for a 2D motion operator [?]

```
class SE2(object):
    def __init__(self, x, y, theta):
        self.t = np.array([x, y]).reshape(2,1)
        self.theta = theta
        self.r = self.R(theta)

    @classmethod
    def R(cls, theta):
        return np.array([[cos(theta), sin(theta)], [-sin(theta)
            , cos(theta)]])

    @classmethod
    def normalize(cls, theta):
        return ((theta + np.pi) % (2*np.pi) - np.pi)

    def __rmul__(self, otherSE2):
        return otherSE2.__mul__(self)

    def __mul__(self, otherSE2):
        new_t = self.t + self.r.dot(otherSE2.t)
        new_theta = normalize2(self.theta + otherSE2.theta)
        new_R = self.R(new_theta)
        return SE2(new_t[0], new_t[1], new_theta)

    def __repr__(self):
        return "<x=%f␣y=%f␣theta=%f>" % (self.t[0], self.t[1],
            self.theta)
```

Thus, when given an input which represents motion between two poses, we can estimate the new position by applying this motion operator to the old position. Then, when we obtain a measurement of the new position, we can begin to build the error function which will be important in solving bundle adjustment. The error function can be defined in a variety of ways, but the most common is:

$$\mathbf{e} = \hat{\mathbf{z}} - \mathbf{z}$$

where $\hat{\mathbf{z}}$ is our estimate from applying $T_{a\to b}a$ and $\mathbf{z} = b$ in this case. Note, that similar calculations are carried out for landmark locations, since a landmark $l = (x_l, y_l)$ observed in one pose, would be expected to appear at position $T_{a\to b}l$ when viewed in pose b. Next, to estimate the global error of our estimate for every position and of every landmark, we take the sum of squares (SSE):

$$SSE = \sum_i \mathbf{e}_i^T \mathbf{e}_i$$

This model can be further adapted by taking into consideration the known measurement variance for each error term, Ω_i^{-1}. Now the total error becomes:

$$SSE = \sum_i \mathbf{e}_i^T \Omega_i \mathbf{e}_i$$

The quadratic form is not chosen by accident. It can be shown that this form is the best unbiased estimator available if we assume a Gaussian distribution of measurement errors. Furthermore, it is the solution to the maximum log-likelihood function. Our goal in the next section will be to consider methods of solving this problem.

To recap, we have defined a model which allows us to produce parameter estimates (either landmark positions, or robot poses, or otherwise). This framework can be generalized by more generally defining an estimation function (similar to `__rmul__` in listing [?]) and forming the SSE as before.

2.1.4. Non-linear Least Squares

We have created the quadratic form comprised of error terms, the SSE function, and we are trying to ‘adjust‘ the input bundles of parameters to minimize this function. To do this, we first note that this function has a non-linear dependence on the bundle parameters, which means that an iterative method is necessary. Such a method starts with an initial guess, $\mathbf{x}_0$ and linearizes the parameter space locally using first-order terms from the Taylor series expansion:

$$\begin{aligned} SSE(\mathbf{x}_0 + \Delta\mathbf{x}) &= \sum_i \mathbf{e}_i((\mathbf{x}_0)_i + \Delta\mathbf{x}_i)^T \Omega_i \mathbf{e}_i((\mathbf{x}_0)_i + \Delta\mathbf{x}_i) \\ &= \sum_i (\mathbf{e}_i + \mathbf{J}_i)^T \Omega_i (\mathbf{e}_i + \mathbf{J}_i) \\ &= \sum_i \mathbf{e}_i^T \Omega_i \mathbf{e}_i + 2\mathbf{e}_i \Omega_i \mathbf{J}_i \Delta\mathbf{x}_i + \Delta\mathbf{x}_i^T \Omega_i \mathbf{J}_i \Delta\mathbf{x}_i \end{aligned}$$

Since $\mathbf{e}_i : \mathbb{R}^3 \to \mathbb{R}^3$, we have denoted $\mathbf{J}_i$ to be the Jacobian matrix comprising of terms $\frac{\partial(\mathbf{e}_i(x,y,\theta))_j}{\partial x_k}$. With this linearized form, we are looking for a direction $\Delta\mathbf{x}$ which leans towards the smallest SSE value. Assuming that $\mathbf{H} = \mathbf{J}^T \Omega \mathbf{J}$, the value of $\Delta\mathbf{x}$ which attains this will occur when the following is solved:

$$(\Delta\mathbf{x}^*)^T \mathbf{H} \Delta\mathbf{x}^* = -2\mathbf{e}\Omega\mathbf{J}\Delta\mathbf{x}^* \quad \Longrightarrow \quad \mathbf{H}\Delta\mathbf{x}^* = -\mathbf{J}^T \Omega \mathbf{e}$$

This way, we obtain the next estimate from the previous one as $\mathbf{x}_1 = \mathbf{x}_0 + \Delta\mathbf{x}^*$ This *second-order method* is commonly known as *Gauss-Newton* and converges quickly when our initial guess, $\mathbf{x}_0$ is already close to the optimal value. Note that we could have also applied a different technique, known as *gradient descent*, which involves computing the Jacobian immediately and following the update rule.

$$\mathbf{x}_1 = \mathbf{x}_0 - \lambda \nabla \mathbf{e}(\Delta\mathbf{x})$$

This *first-order* update rule has quick convergence when the initial guess is bad, but takes many steps to converge at the end. Thus, the well-known *Levenberg-Marquardt* algorithm merges these two approaches by modifying the Gauss-Newton formulation as follows:

$$(\mathbf{H} + \lambda \mathbf{W})\Delta \mathbf{x}^* = -\mathbf{J}^T \Omega \mathbf{e}$$

where $\lambda \in \mathbb{R}$ is a step-size parameter we get to adjust at each iteration, and $\mathbf{W}$ is a diagonal matrix having the diagonal components of $\mathbf{H}$ but can also be chosen to be the identity matrix. We will delve deeper into this by analyzing the structure of the matrices we are trying to work with, namely $\mathbf{J}$ and $\mathbf{H}$. Specifically, with a toy example such as that listed in figure [**?**], we see a set of edges (feature measurements) connecting vertices (images), and camera calibrations (which are linked to landmark measurements, only from specific poses). Notice that camera calibrations are connected to poses (K1

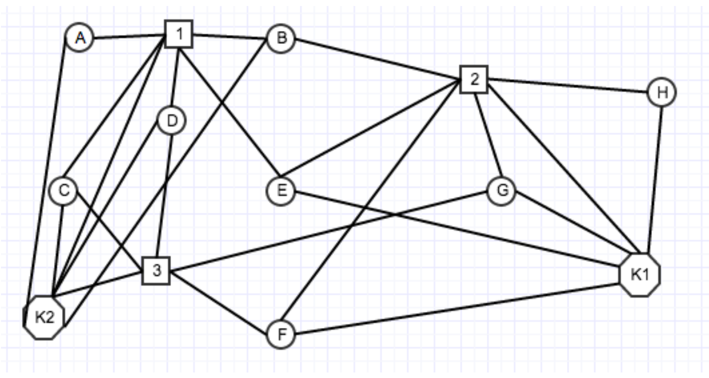

Figure 2.3.: Toy Bundle Adjustment Graph

is connected to 2, and the landmarks observed from 2, while K2 is connected to 1 and 3), and thus also to (some of) the landmarks seen (measured) from those poses. In general poses can be connected to other poses (via odometry measurements), and landmarks to other landmarks. The corresponding Jacobian matrix will be sparse, as it will only contain entries for which there are edges in the graph. We have provided a view of the sparsity structure of the Jacobian $\mathbf{J}$ and the Hamiltonian $\mathbf{H} = \mathbf{J}^T\mathbf{J}$ in figure 2.4. Although this is a slightly unrealistic setup, it shows the general structure of the Hessian matrix we are trying to solve. Various methods exist for solving these sparse systems, and they are listed here:

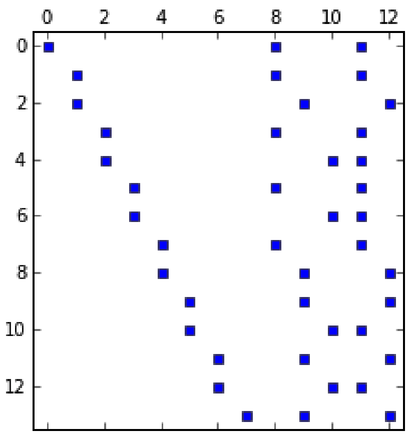
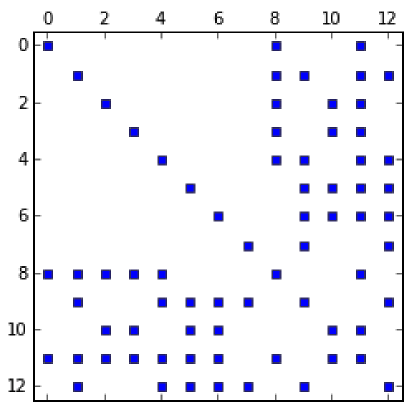
Figure 2.4.: Sparsity of Bundle Adjustment Matrices (non-zero values displayed): Jacobian (left), and Hamiltonian (right)

- Schur complement - produces a reduced camera system by solving first for poses, then for landmarks
- Cholesky decomposition - solves $\mathbf{H} = \mathbf{L}^T\mathbf{L}$, where $\mathbf{L}$ is lower-diagonal; performance depends heavily on sparsity of system
- Conjugate Gradient - an iterative solver for large sparse positive definite linear systems
- GMRES - generalized minimal residual method

The majority of bundle adjustment literature has relied on either Schur complement or Cholesky decomposition with good results. We used Golub et. al. [10] as reference for these methods.

The Schur complement 'trick' uses the sparsity structure of the resulting Hessian matrix. Essentially, we can reduce the previously stated linearized system in block-form:

$$\mathbf{H} = \begin{bmatrix} U & W \\ W^T & V \end{bmatrix} \quad \text{and} \quad \Delta\mathbf{x} = \begin{bmatrix} \Delta\mathbf{a} \\ \Delta\mathbf{b} \end{bmatrix} \quad \text{and} \quad -\mathbf{J}\Omega\mathbf{e} = \begin{bmatrix} \epsilon\mathbf{a} \\ \epsilon\mathbf{b} \end{bmatrix}$$

Then, by premultiplying the equation:

$$\mathbf{H}\Delta\mathbf{x} = \mathbf{J}\Omega\mathbf{e}$$

by

$$\begin{bmatrix} I & -WV^{*-1} \\ 0 & I \end{bmatrix}$$

we obtain a reduced system, whereby we can first solve the equation for $\mathbf{a}$ and then back-substituted to solve for $\mathbf{b}$.

In Cholesky decomposition, the goal is to decompose the matrix into the form $\mathbf{H} = \mathbf{L}\mathbf{L}^T$. Specifically, one frames the following equation:

$$A = \begin{bmatrix} a_{11} & \mathbf{A}_{21}^T \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 \\ \mathbf{L}_{21} & \mathbf{L}_{22} \end{bmatrix} * \begin{bmatrix} l_{11} & \mathbf{L}_{21}^T \\ 0 & \mathbf{L}_{22} \end{bmatrix}$$

and then note that for this to hold, $l_{11} = \sqrt{a_{11}}$, $\mathbf{L}_{21} = \frac{1}{l_{11}}\mathbf{A}_{21}$, leaving for us to solve

$$\mathbf{A}_{22} - \mathbf{L}_{21}\mathbf{L}_{21}^T = \mathbf{L}_{22}\mathbf{L}_{22}^T$$

which is once again a Cholesky factorization problem with matrix of size $n-1$. This is a recursive process which step-wise creates $\mathbf{L}$ by moving along the diagonal of the Hessian [10]. An intelligent re-ordering of the columns and rows in a sparse matrix can greatly reduce the amount of extra computations, known as *fill-in*. Since the re-ordering is in general an NP-hard problem, these methods can have variable success.

2.1.5. Putting It All Together

We have shown how the pieces of the bundle adjustment puzzle fit together, starting from the measurements/observations of the robot via a camera, and finishing with a large sparse linearlized system, whose solution gives us an optimal parameter update towards the final solution. This solution provides the robot with the highest likelihood coordinates for environment landmarks (a.k.a. a map) as well as its own current and past positions. In an on-line setting, a robot exploring its environment would continuously solve this system with new landmarks and poses being added at each step. Since the computation time is bounded by solving the Hessian, compute-time will grow at least $\mathcal{O}(n)$ with the number of bundles, and potentially as poorly as $\mathcal{O}(n^3/3)$ for poorly structured matrices. For example, Spielman and Teng [22] found certain sparse systems solvable in $\mathcal{O}(n^{1.31})$. Thus researchers usually apply a rolling window to incoming data to limit complexity growth. In an off-line setting, which where we our focus lies, we read in an entire graph structure from an existing database, and perform the relevant computations. Problems of this sort would arise when learning very large environments and computational techniques which take advantage of distributed computing platforms play a leading role.

2.2. PyOP2

We aim to give a background for the PyOP2 framework and the types of problems it specializes in solving. We will cover the use of Python as a language for problem specification. A judicious use of operator overloading allows researchers to specify the problem domain concisely without destroying the mathematical structure, which an optimizing compiler can use to reduce computational load. Most typically PyOP2 is very efficient at handling unstructured mesh applications, typical in fluid dynamics and mechanical engineering applications. However, we will to use it to set up and solve bundle adjustment computations, since both problems deal with solving sparse systems generated by graph structures.

2.2.1. Python as a Domain Specific Language

With the strong uptake of the Python programming language in the scientific community, many libraries have been developed which wrap up existing high performance computing packages and provide an easy-to-program interface. The language itself is unsuitable for CPU intensive computations due to the slow nature of dynamic dispatch, it's loose (read non-existent) type system, and the global interpreter lock (GIL) which prevents the use of multithreading. However, it is an ideal staging language, allowing for easy set-up of objects, which can then be sent to more performant compute routines. Figure 2.5 illustrates a small selection of the selection of Python libraries and tools available to the scientific community.

2.2.2. Finite Element Methods and How they Relate to Bundle Adjustment

PyOP2 was built for accelerating unstructured mesh computations in finite element problems. These problems typically start as a differential equation

Figure 2.5.: Scientific Python Ecosystem

with a boundary condition, e.g.:

$$\begin{aligned} \nabla^2 u &= f \quad \text{for} \quad x \in \Omega \\ u &= g(x) \quad \text{for} \quad x \in \partial\Omega \end{aligned}$$

The solution can then be thought of as a linear combination of mutually orthogonal basis functions $v_i(x)$ and the goal is to find coefficients α_i such that we best approximate $u(x) = \sum_i \alpha_i v_i(x)$. The corresponding linear algebra problem is obtained by first considering the weak form:

$$\int_\Omega \nabla \cdot u \nabla v ds = \int_\Omega f v ds \equiv \phi(u, v)$$

Constructing a matrix $L_{ij} = \int_\Omega v_i v_j ds$ and $M_{ij} = \int_\Omega \nabla v_i \nabla v_j ds$ leads to the equivalent formulation of the PDE:

$$\mathbf{Lu} = \mathbf{Mb}$$

where $\mathbf{u} = (u_1, \ldots, u_n)^T$ and $\mathbf{b} = (\int_\Omega f v_1 ds, \ldots \int_\Omega f v_n ds)$.

At this stage, the similarities between the bundle adjustment matrix formulation and finite element method problems still seem somewhat murky, although they are both formulated as a sparse matrix problem. Unfortunately, after further investigation into these similarities, we were not able to extend the link. Specifically, the weak form is obtained under strong assumptions about differentiability of the solution, something that is not present in the bundle adjustment problem. However, finite elements are also a form of a graph problem. To see this, note that the basis functions $v_j(x)$ in 2D are continuous surfaces in some small sub-domain and 0 elsewhere. Thus, they represent edges in the graph, while the graph vertices can be thought of as the boundaries between adjacent surfaces. In finite element parlance, the surfaces are called 'elements' or 'facets'. A visualization of the surfaces is shown in figure 2.6. Although finite element solvers are not

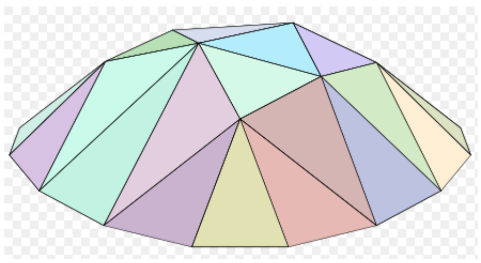
Figure 2.6.: 2D Elements of a Finite-Element Mesh

directly applicable for bundle adjustment, tools which efficiently schedule computational work where data is loosely coupled via a graph structure, can be used. Since nearby landmarks are typically observed from the same robot pose, the induced computational work can be scheduled in a similar way to those on a finite element mesh.

2.2.3. Code Generation

PyOP2 (with tools from FEniCS) provides a powerful framework for modeling differential equations, and solving the resultant finite element problem. The framework uses UFL (Unified Form Language) to automatically generate 'kernels' (main code block for the OP2 solver to iterate over). PyOP2 can be configured to run on a variety of back-end parallel architectures, including GPUs (OpenCL and CUDA), CPU clusters communicating via MPI, and others. The user is not responsible for knowing architecture-specific routines and can concentrate of working in the problem domain. The following listing is taken from example code for the advection problem and illustrates the techniques.

Listing 2.3: Code Generation Example

```
from pyop2.ffc_interface import compile_form
from ufl import *
p = TrialFunction(T)
q = TestFunction(T)
M = p * q * dx
adv, = compile_form(M, "adv")
```

At this point, the `adv` variable is of type `op2.Kernel` and contains code representing discretized calculation for:

$$\int p(x)q(x)dx$$

The kernel `adv` contains autogenerated C-code which is next passed to the `op2.par_loop()` function (along with variables representing input values) which distributes work while exploiting the sparsity structure. In our work, we aim to produce a similar structure for bundle adjustment problems

We have hopefully presented the prior knowledge necessary for the reader to proceed with the rest of this report. We have presented the context in which bundle adjustment is relevant for computer vision and have shown a graph-theoretic formulation of the problem, and the resulting sparse non-linear linear algebra problem. We have also outlined some sparse solving techniques, which may be interesting when considering which solver to use in a given setting. Next, we side-stepped to consider finite element methods, another *variational problem*, which has been well studied, and shows mathematical similarities to bundle adjustment. We have hopefully presented our intuition behind choosing PyOP2 as an execution platform for bundle adjustment. In the remaining chapters, we will show what work has already been done in this field, and which software is available.

3. Related Work

After being developed by Gauss, least-squares has seen a variety of applications, most popularly, in solving for parameters in statistical models (e.g. regression). Bundle adjustment was originally attempted by D.C. Brown et. al. in the late 1950s [5]. Through the 60s and 70s, techniques focused on reduction techniques for sparse matrices to make the problem solvable. In recent years interest in this field of computer vision has grown as computing power has allowed for this approach to become competitive with Kalman Filtering and other probabilistic techniques.

3.1. Early Bundle Adjustment

D.C Brown et al [5] approached the problem by representing the pose parameters in terms of landmark parameters, thus eliminating the need to solve for them. Brown solved the resulting system of pose parameters with classical Gaussian elimination on the then-state-of-the-art computers. This approach relies on having a *dense* system, one in which landmarks are visible from most poses. However, realistic situations usually have individual poses capturing a small percentage of all landmarks as a camera (or robot) moves around in a large environment. This prompted researches to look for ways to reduce this inherently sparse minimization problem into a smaller dense form throughout the 60s and 70s. For example, *recursive partitioning*, as summarized by Brown in his overview of the state-of-the art techniques available in 1976 [6], is a technique which explicitly considers the zero blocks of the matrix to reduce the size of the linear algebra problem. After some reconfiguration, one is left with a system that can once again undergo further simplification and what is left is a significantly smaller system which can be readily solved by Gaussian elimination. As Brown notes, this solution neglects a measurement error model as well as assumes a very rigid block arrowhead sparsity structure which does not take loop closures into account (the possibility that the robotic system re-visits a scene previously seen).

3.2. Modern Methods

In 1999, Triggs et. al. conducted a thorough review of the mathematical literature available on bundle adjustment. Since then, several notable software package have come out which solve this problem or some variant thereof:

- SBA - A generic sparse solver for Bundle Adjustment problems written in C with MATLAB and command-line extensions; This library was first released in 2004 by Lourakis et. al. [15], and focused on providing a basic implementation using LAPAK libraries as a Linear Algebra back-end when applying the Schur complement trick during the solution of the sparse linearized system.
- Multicore BA - Wu et. al. [27] created this library to exploit parallelism in some of the computations being done during bundle adjustment; This is the only library which explicitly attempts to make performance improvements on a different computer architecture; They report speedup gains of approximately 10-13 times versus single-threaded code and 2-3 times versus multi-threaded code.
- Ceres-Solver - A joint collaboration between Google and Washington University created by Agrawal et. al. [2]; Similar to g2o, the also follow an extensible architecture, allowing the user to fully specify a general minimization problem; The library also uses vectorized instruction sets to speed up computations as well as automatic differentiation of user-specified functions to derive optimizations based on a reduced number of computations.
- SAM and iSAM - A C++ library developed by Kaess et. al. [12]; They use QR factorization of the sparse Jacobian matrix to minimize the quantity $||J\mathbf{x}-\mathbf{b}||^2$; Their incremental approach uses Givens rotations to prevent re-solving of the QR problem in future instances when new data points arrive; This approach works well in an on-line setting but still relies on back-solving a lower diagonal matrix at each iteration.

- g2o - A C++ project to solve general hypergraph optimization problems developed by Kummerle et. al. [13]; The library follows an extensible architecture, which allows the freedom to select different solvers and specify different measurement and error functions for the SLAM problem; The authors look for performance gains by performing vectorized instructions in the called matrix libraries.

These software packages, all released within the past 10 years, underline the increased interest seen in this field and its application to robot vision. They all use differing data formats depending on what is most convenient for the approach. Thus, testing and benchmarking comparisons between different packages has been difficult to come by, as each researcher would have to replicate a large chunk of an existing library (especially when the data reading operations are rooted deeply within the object hierarchy, as in g2o). We will look at two simulated 2D datasets, Intel campus, and Manhattan city grid, to evaluate performance and resource utilization. When comparing iSAM (in bulk mode) and g2o on the Manhattan dataset (3500 vertices and 5598 edges), g2o performed approximately 1.5 times faster than iSAM (0.25 seconds versus 0.4 seconds).

4. Program Design

We cover our approach at architecting a bundle adjustment solver and describe the efficiencies we seek to achieve by employing smart code generation and automatic differentiation tools like PyOP2 and Theano. There are many classes of bundle adjustment problems, ranging from a simple 2D pose-only dataset, with constraints constituting odometry readings between successive poses to complex scenarios with 3D landmarks coupled with different camera calibrations used across scenes. In all cases, we begin by either loading the data into memory or streaming it from an input pipe. A streaming architecture forces one to consider memory issues such as a growing the relevant data structures with new data. Techniques that allow for memory-mapped data to be brought in quickly and only when required have been efficiently implemented with the HDF5 data model [25] as well as with `numpy` memory-mapped arrays (`numpy.memmap`).

After providing the program with access to data, we will show the following steps in our implementation:

- Build error function $\mathbf{e} = \hat{\mathbf{z}} - \mathbf{z}$: this step involves bringing in both the estimate for the measurement, and the measurement itself and performing a differencing operation, which is simple in Euclidean space but can become more involved with rotation groups; Furthermore, different *types* of vertices (landmarks, poses, etc.) can have different measurement, and therefore error, functions.

- Compute the Jacobian $\mathbf{J}$: This step has to be carefully constructed to only perform computations for those vertices where they are linked by an edge; We will borrow techniques from finite element methods to perform this efficiently; here, automatic differentiation is applicable.

- Compute the Hessian $\mathbf{H} = \mathbf{J}^T \Omega \mathbf{J}$ or the Levenberg-Marquardt equivalent $\mathbf{H} + \lambda \mathbf{W}$ (assembling Ω is a straightforward step since it is block-diagonal with data directly provided from input).

- Compute the product $\mathbf{J}^T\Omega\mathbf{e}$: this constitutes the right-hand-side of the non-linear equation we are trying to solve.
- Pass the left-hand-side sparse matrix and the right-hand-side matrix to the solver.
- Compare the new error measure with the old error, adjust λ accordingly, and check any stopping conditions to terminate the non-linear least squares search.

4.1. Data

Bundle adjustment literature has done a thorough job of describing the methodology used to solve for pose coordinates, landmark coordinates, and even camera coordinates. There has, however, been less emphasis on a systematic standardization of data formats as each available package provides data in non-compatible layouts. We have found the g2o data format to be the most general formulation, allowing the user to specify a wide range of graph types.

Generally, data arrives as plain text with rows for vertices and edges. The vertex label (e.g. SE2, QUART, etc) determines the type and the data items in that row. Specifically, an SE2 vertex will contain:

```
VERTEX_SE2 15 3.15454 3.89159 1.53144
```

representing the vertex index along with estimated parameters for that position x, y, θ. A row containing an edge (measurement) also comes with a label, allowing the program to correctly associate the appropriate error function calculation at each edge. The other items in the row of edge data provide indices of the vertices that edge connects as well as the measurement calculations and other fixed parameters (e.g. the measurement variance matrix for that edge Ω).

We used the `pandas` package [17], which uses `numpy` arrays for contiguous, efficient, and fast data storage. The advantage that `pandas` brings is quick named column and row selection (see listing A.3). Furthermore, it supports streaming data from regular or HDF files via iterators and visualization with output in 4.1:

Listing 4.1: Visualizing The Intel Dataset

```
from ba_data import INTEL_G2O
import matplotlib.pyplot as plt
vertices, edges = quickload(INTEL_G2O)
plt.plot(vertices['dim1'], vertices['dim2'])
```

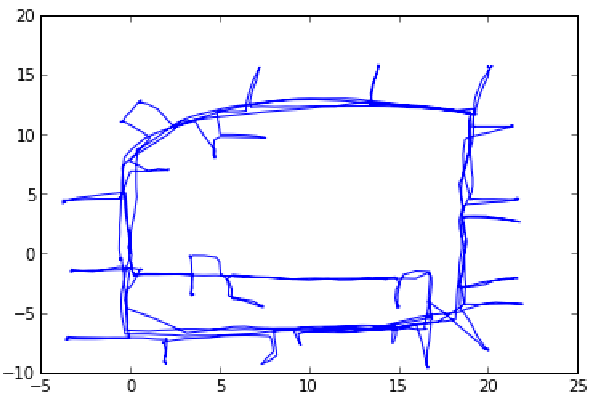

Figure 4.1.: Plot X and Y coordinates of Intel data from g2o

Similarly, the Manhattan dataset is show in figure 4.2.

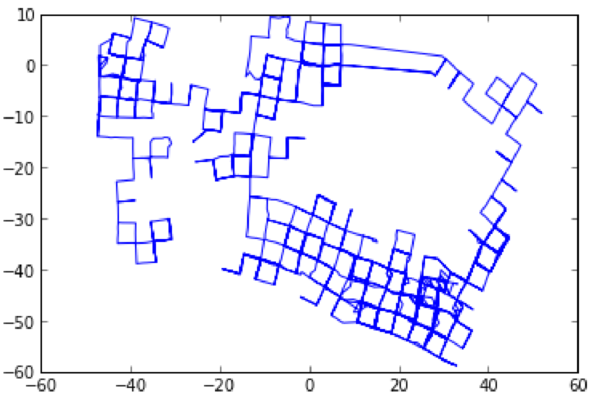

Figure 4.2.: Plot X and Y coordinates of Manhattan data from g2o

4.2. Building the Error Function

We present our approach to creating the error function in bundle adjustment problems. This function is a mapping from parameters (edges e_i, each having a few degrees of freedom, d_i) to errors:

$$\mathbf{e} : \mathbb{R}^{\sum_i d_i} \to \mathbb{R}^{\sum_i d_i}$$

As an example of the dimensionality involved, a 2D dataset has 3 dimensions per edge for measurements between poses and 2 for those between landmarks. A relatively small dataset with 10000 pose-to-pose only vertices would imply an error vector in $\mathbb{R}^{20000}$. Since each edge contains measurement information between vertices, we can accelerate these calculations by employing PyOP2 in its construction. For that, it is necessary to instantiate the data structures PyOP2 interacts with. The framework mimics mathematical concepts of *sets* and *map*, which are used to create indirection between the data and the computation. We present a small example in listing 4.2.

Listing 4.2: Example PyOP2 Code for Building Bundle Adjustment Graph

```
import numpy as np
from pyop2 import op2

def identity(num, dim):
        return np.asarray([i/2 for i in range(dim*num)], dtype=
            np.uint32)

NUM_POSES = 3
NUM_POSE_CONSTRAINTS = 2

poses = op2.Set(NUM_POSES, 'poses')
pose_constraints = op2.Set(NUM_POSE_CONSTRAINTS, '
    pose_constraints')

# constraint 1: pose 0 --> pose1; constraint 2: pose1 --> pose2
constraint_pose_data = np.asarray([0, 1, 1, 2], dtype=np.uint32
    )
# all constraints map to themselves
constraint_constraint_data = identity(NUM_POSE_CONSTRAINTS, 2)

constraints_to_poses = op2.Map(pose_constraints,
                                   poses,
                                   2,
                                   constraint_pose_data,
                                   'poses_constraints')
```

```

constraint_to_constraint = op2.Map(pose_constraints,
                                   pose_constraints,
                                   2,
                                   constraint_constraint_data,
                                   'constraint_to_constraint')
```

In mathematical terms, we have created *maps* which tell PyOP2 how to create a correspondence to data when iterating over the *set* of pose constraints. When the PyOP2 runtime is given a kernel to execute, it relies on these maps to partition the iteration space into disjoint subspaces so as to minimize data contention. Note that we build the `constraint_to_constraint` mapping with dimensionality 2 because each constraint actually maps to 2 dimensions: x, y (in the case of Euclidean parameters). In the case of finite elements, PyOP2 uses coloring on the elements, but other techniques have been studied since at least 1970. In our production code, we can encapsulate these procedures in a data structure and populate the data appropriately. These structures are next passed to a PyOP2 C-style execution kernel, which represents a computation to be done at each iteration. Since PyOP2 performs the iteration space tiling to maximize parallelism on a pre-specified backend, this approach provides portable performance without needing to modify for various execution platforms.

Although the above example is simplistic, it illustrates the approach we take in building the graph in PyOP2 for efficient execution. At each step, we iterate over the measurements (or 'constraints' or 'edges' in graph theory parlance) and produce calculations over poses or landmarks ('vertices'). The `op2.Map` construct allows for this level of indirection since it maps a specific measurement to a specific number of poses which that edge acts upon. Because this is quite general, it is equally possible to define computations over hyper-edges and hyper-vertices (where edges connect more than two vertices), although bundle adjustment problems do not usually warrant the use of these structures.

The error function, as described earlier, is a high-dimensional vector, which is better represented as chunks, $\mathbf{e} = (\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n)^T$, with each chunk $\mathbf{e}_i \in \mathbb{R}^{d_i}$ where d_i is the dimensionality of the i^{th} measurement. If $\mathbf{e}_i$ measures distances on the SE2 manifold, $d_i = 3$. Our implementation iterates over these edges, but requires data from the corresponding poses (or landmarks). To do so, we first encapsulate the available measurement data into an `op2.Dat` with a specified ‘constraint-to-pose‘ `Map` built-in. We then follow a two-stage approach:

- First, compute the estimate by iterating over the measurements, retrieving the two corresponding poses via the `Map`, and write the result to an `estimate` variable
- Next, iterate once more over the measurements, this time performing a trivial `IdentityMap` over them and applying the user-specified error function at both estimates and the available observation data

This parallelizes the construction of the error function, which is further used in two places. First, we evaluate the success of the bundle adjustment algorithm but continuously monitoring the cost function:

$$SSE = \sum_i \mathbf{e}_i^T \Omega_i \mathbf{e}_i$$

or more generally:

$$SSE_\delta = \sum_i \rho_\delta(\mathbf{e}_i^T \Omega_i \mathbf{e}_i) \quad \text{where} \quad \rho_\delta(x) = \left\{ \begin{array}{ccc} x^2 & \text{for} & |x| < \delta \\ 2\delta|x| - \delta^2 & & \text{otherwise} \end{array} \right)$$

This *robustified* total error gives less weight to extreme outliers (those lying further than δ away from the measurement) while maintaining the nice properties of convexity, and thus, not decreasing the chance of converging to the global minimum. Since Ω_i are inputs, we can provide a reference to them via PyOP2 `Dat` objects, defined over measurements (since each one corresponds solely to the inverse-variance of measurement data and has no direct connection to pose or landmark data). The point of data contention is the writing of the SSE variable by different processes are they traverse the measurement iteration space. PyOP2 provides a type of data carrier, `op2.Global`, which is shared among all the processes. Thus, our kernel will be specified as in listing 4.3:

Listing 4.3: Computing the Total Sum of Squares Error

```
        F = op2.Global(dim=1, data=0.0, dtype=np.float64)
        total_error_code = """
        void total_error(double e[2], double omega_block[4],
            double * sse)
        {
            *sse += (e[0]*omega_block[0] + e[1]*omega_block[2])
                 * e[0] +
                           (e[0]*omega_block[1] + e[1]*
                                omega_block[3]) * e[1];
        }
        """

        total_error = op2.Kernel(total_error_code, 'total_error
            ')
        op2.par_loop( total_error, pose_constraints,
                      e(op2.IdentityMap, op2.READ),
                      F(op2.INC))
```

4.3. Building the Jacobian Blocks

So far, we have an error vector and a measure of the total error of input data, but to create a favorable update to the position parameters we are estimating, we also construct the Jacobian matrix. In the simple case of 2D coordinates and plain Euclidean estimates, a typical measurement error **e** between poses **p** and **q** will look as follows:

$$\begin{pmatrix} e_x \\ e_y \end{pmatrix} = \hat{\mathbf{z}} - \mathbf{z} = \begin{pmatrix} \hat{z_x} \\ \hat{z_y} \end{pmatrix} - \begin{pmatrix} q_x - p_x \\ q_y - p_y \end{pmatrix}$$

The $\hat{\mathbf{z}}$ are constant data, so the Jacobian block for this measurement will have the following structure:

$$\mathbf{J}_e = \left(\begin{array}{cc:cc} \frac{\partial e_x}{\partial p_x} & \frac{\partial e_x}{\partial p_y} & \frac{\partial e_x}{\partial q_x} & \frac{\partial e_x}{\partial q_y} \\ \frac{\partial e_y}{\partial p_x} & \frac{\partial e_y}{\partial p_y} & \frac{\partial e_y}{\partial q_x} & \frac{\partial e_y}{\partial q_y} \end{array}\right) = \left(\begin{array}{cc:cc} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{array}\right)$$

Of course, for non-Euclidean parameter spaces, the cross-terms also come into play, and we automatic differentiation to to more quickly obtain these Jacobian blocks. We make a generic approach in generating these blocks by the use of *automatic differentiation.* There are a number of packages across various programming languages, which can be used for this purpose. For example, Theano [3] allows the user to construct a symbolic representation of their problem, and then performs optimizing graph transformations on the resultant execution graph, following by fast generated C code (for the CPU) or CUDA/OpenGL code (for GPUs). Some of the functionality overlaps with PyOP2, though provides less support for more complicated parallelization techniques such as graph partitioning. Since we are only concerned with differentiation among a small set of variables, as well as with smooth functions, we selected to use the Sympy module for this task. The overriding reason for this is the in-program generation of C code the Sympy provides, which we can directly use to populate the PyOP2 kernel we use to construct the Jacobian blocks. For example, in SE2, the estimation functions is:

$$\mathbf{q} - \mathbf{p} = \begin{pmatrix} (q_x - p_x)\cos p_\theta + (q_y - p_y)\sin p_\theta \\ -(q_x - p_x)\sin p_\theta + (q_y - p_y)\cos p_\theta \\ ((p_\theta - q_\theta + \pi) \bmod 2\pi) - \pi \end{pmatrix}$$

between poses and the usual Euclidean one between a pose and a landmark. Thus, the Jacobian block for a measurement between poses $\mathbf{p}$ and $\mathbf{q}$ will look as follows:

$$\begin{aligned} \mathbf{J}_e &= \left(\begin{array}{ccc:ccc} \frac{\partial e_x}{\partial p_x} & \frac{\partial e_x}{\partial p_y} & \frac{\partial e_x}{\partial p_\theta} & \frac{\partial e_x}{\partial q_x} & \frac{\partial e_x}{\partial q_y} & \frac{\partial e_x}{\partial q_\theta} \\ \frac{\partial e_y}{\partial p_x} & \frac{\partial e_y}{\partial p_y} & \frac{\partial e_y}{\partial p_\theta} & \frac{\partial e_y}{\partial q_x} & \frac{\partial e_y}{\partial q_y} & \frac{\partial e_y}{\partial q_\theta} \\ \frac{\partial e_\theta}{\partial p_x} & \frac{\partial e_\theta}{\partial p_y} & \frac{\partial e_\theta}{\partial p_\theta} & \frac{\partial e_\theta}{\partial q_x} & \frac{\partial e_\theta}{\partial q_y} & \frac{\partial e_\theta}{\partial q_\theta} \end{array}\right) \\ &= \left(\begin{array}{ccc:ccc} \cos w_\theta & \sin w_\theta & (q_x - p_x)\sin w_\theta - (q_y - p_y)\cos w_\theta & -\cos w_\theta & -\sin w_\theta & 0 \\ -\sin w_\theta & \cos w_\theta & -(q_x - p_x)\cos w_\theta + (q_y - p_y)\sin w_\theta & \sin w_\theta & -\cos w_\theta & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 \end{array}\right) \end{aligned}$$

We generate a similar structure programmatically with Sympy as in listing A.4. Furthermore, this allows the user to specify a custom-made measurement function, without dropping to a low-level language to do so.

4.4. Building the Hessian

Since we aim to ultimately solve the non-linear least-squares problem:

$$(\mathbf{H} + \lambda\mathbf{W})\Delta\mathbf{x} = \mathbf{J}^T\Omega\mathbf{e} \quad \text{where} \quad \mathbf{H} \equiv \mathbf{J}^T\Omega\mathbf{J}$$

for $\Delta\mathbf{x}$, the next step is to build the left-hand-side of that equation. At this stage, we have computed the Jacobian blocks and the inverse-covariance blocks, Ω_i (encapsulated in a reference via the PyOP2 `Dat` type), in the previous step. Although the Hamiltonian matrix has dimensions solely dependent on pose and landmark data, we chose the constraints `Set` as our iteration space, allowing us to bring in pose data related to those consraints via the `contraint_to_pose Map` defined earlier, and populate the matrix that way. Specifically, the Hamiltonian construction kernel is displayed in listing 4.4

Listing 4.4: The Hamiltonian Matrix Kernel

```
hamil_sparsity = op2.Sparsity((poses ** 2, poses ** 2), (constraints_to_poses, constraints_to_poses), 'hamil_sparsity')
    hamil_mat = op2.Mat(hamil_sparsity, np.float64, 'hamil_mat')

    poses_hamiltonian_code = """
    void poses_mat_hamiltonian( double H[2][2], double J[8], int i, int j)
    {
        int block1 = 4*i;
        int block2 = 4*j;
        H[0][0] += J[block1 + 0]*J[block2 + 0] + J[block1 + 2]*J[block2 + 2];
        H[0][1] += J[block1 + 0]*J[block2 + 1] + J[block1 + 2]*J[block2 + 3];
        H[1][0] += H[0][1]; // symmetry
        H[1][1] += J[block1 + 1]*J[block2 + 1] + J[block1 + 3]*J[block2 + 3];

         if ( i == 0 && j == 0 ) {
                H[0][0] *= 2.;
                H[0][1] *= 2.;
                H[1][0] *= 2.;
                H[1][1] *= 2.;
            }
    }
    """

    poses_mat_hamiltonian = op2.Kernel(poses_hamiltonian_code, 'poses_mat_hamiltonian')

    op2.par_loop( poses_mat_hamiltonian, pose_constraints(2,2),
                hamil_mat((constraints_to_poses[op2.i[0]],
                    constraints_to_poses[op2.i[1]]), op2.INC),
                jacobian_blocks(op2.IdentityMap, op2.READ) )
```

A small, but important consideration has to be made for the initial constraint, one that dictates the initial pose. Thus, we added extra code to update the upper left block of the Hamiltonian appropriately. Without this extra condition, the Hamiltonian would prove to be ill conditioned. Similarly, we also update the calculation for the right-hand-side of the non-linear least squares problem. However, if we assume the initial measurement error is zero (after all, a famous physicist once said that everything is relative), we can safely neglect extra code there.

4.5. Solvers

As we mentioned in the background review, a variety of sparse linear solvers exist to tackle our linearized bundle adjustment problem. In PyOP2, we invoke the built-in solvers with the following code:

Listing 4.5: Invoking a Sparse Solver in PyOP2

```
    solver = op2.Solver(linear_solver = 'gmres')
    solver.solve(hamil_mat, x, rhs_vec)
```

Most modern bundle adjustment packages exploit the sparsity of the linearized system at the expense of compromising code flexibility. PyOP2 gives one the freedom to experiment with different solvers, so long as the back-end is implemented. Thus, the introduction of new solving techniques doesn't mean having to rearchitect the library. However, PyOP2, as of this writing, does not support solving via the Schur complement, a trick described in chapter **??** and successfully used to reduce dimensionality of bundle adjustment problems by first solving for poses and afterwards for landmarks. Thus, we expect our implementation to suffer when there is a roughly equal balance between the two types of vertices. However, this is generally not the case. In closed environments, pose data tends to accumulate as the autonomous vehicle measures many of the same landmarks repeatedly, while in open environments, each new pose will likely correspond to a (potentially large) handful of new landmarks. Current research focuses on considering more and more landmarks from each pose, thus heavily weighing the balance towards landmarks. Thus, we believe we are not at a great disadvantage for not being able to use Schur.

4.6. Putting It All Together

Finally, we discuss our approach to solving the full bundle adjutment problem. We have shown how we set up the problem, using symbolic differentiation and code generation techniques to populate the kernel strings which feed PyOP2. Our full solution of a single step, as in listing 4.5, produces a value for $\Delta \mathbf{x}$, which is an update to our initial parameter estimates. To progress towards the optimal parameters, we first update the parameters with the calculation:

$$\mathbf{x}_{t+1} \leftarrow \mathbf{x}_t + \Delta \mathbf{x}$$

This is another calculation in which we can employ PyOP2 kernels for speedup. Since $\mathbf{x}$ is defined over vertices (either poses or landmarks or both), our kernel does not need to employ extra indirection to make this update. In fact, this step is trivially parallelized without the need for graph partitioning techniques. As a result, we leave out the code from this report. More sophisticated Levenberg-Marquardt techniques call for a dynamic updating of the λ parameter, which would involve keeping two sets of $\mathbf{x}$ data.

Next, we once again re-evaluate our `total_error` kernel to described earlier to measure the impact of the calculated update. Typical conditions which are used to terminate the looping include asserting a maximum loop count, a minimum error reduction, as well as an overall error target. Note that it is important to consider the total error in the context of the total number of vertices.

In summary, the steps taken are as follows:

1. Load data from file or other source and set up preliminary `Sets` `Maps` and `Dats` including the Ω blocks.

2. Evaluate the total error, SSE, and if it is sufficiently small, exit and report the optimal parameter values, else proceed to [3].

3. Pre-process the Jacobian blocks.

4. Calculate the righ-hand-side kernel including the estimate and error vectors.
5. Re-evaluate the Hamiltonian kernel.
6. Solve the linearized system resulting from steps [4] and [5].
7. Loop back to step [2].

5. Evaluation

To assess the viability of the methods discussed in the previous chapter, we will discuss how our approach fares in solving the bundle adjustment problem and how it scales. We will discuss the resource usage of our implementation in comparison to modern bundle adjustment packages. Specifically, we track our performance against g2o and iSAM packages. We use two datasets to show the scaling properties of these packages as well as ours. We note that these packages were written in C++, and g2o also uses AVX to accelerate performance.

5.1. Analyzing Our Implementation

Our implementation follows a modular layout which allows for easy performance testing and tuning. The steps outlined in chapter 4 were taken, specifically the incremental building up of the components related to a large sparse non-linear minimization problem solved by linearizing the components and solving iteratively. There are two steps to set up the problem, reading the data from file into memory, and two substages, one to set up PyOP2 `Set` and `Map` objects, and another to set up PyOP2 `Dat` and `Mat` objects. These steps are inefficient, but need only to be run once to set up PyOP2 kernels for repeated use in later stages. They involve passing pose and constraint data into PyOP2 `Dat`s and formatting strings. Furthermore, Sympy-generated C-code is used in this stage to produce the kernel for both the error and the Jacobian of the error. This stage is the slowest part of our program due to the time spent executing Python code, which is typically 10-100 times slower than C or C++. Future implementations can consider the *pypy* interpreter, which has been shown to perform 6-10 times faster than Python. We first present the sparsity patterns of the Hamiltonian matrix obtained from the two 2D datasets we considered. They are shown in figures 5.1 and 5.2.

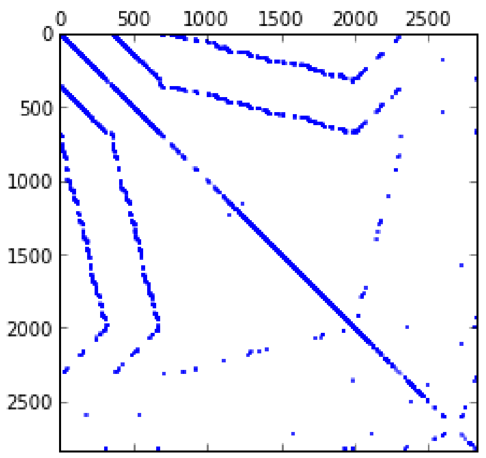

Figure 5.1.: Intel Dataset Hamiltonian Sparsity (non-zero elements displayed)

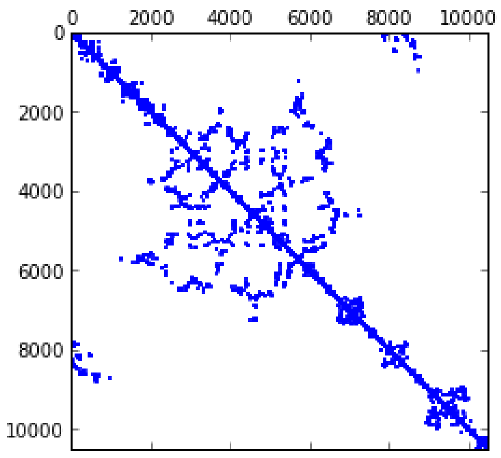

Figure 5.2.: Manhattan Dataset Hamiltonian Sparsity (non-zero elements displayed)

Next, we present some of the running times measured on an Intel 2GHz IvyBridge i7 MacBook Air with 8GB of 1600MHz DDR3 RAM. The compiler is an LLVM clang compiler with `-O3` optimization flats set. We used the sequential (MPI) backend for PyOP2, which allows for a more direct comparison to other bundle adjustment packages. Below, we show the result for running our bundle adjustment code over the Intel dataset, by printing the `dict` of timing, in seconds, we recorded.

```
{'error': 0.0017781257629394531,
 'estimate': 0.0017139911651611328,
 'jacobian': 0.0019482135772705077,
 'lhs': 0.0081790447235107425,
 'per_iter_setup': 0.00072622299194335938,
 'rhs': 0.0018266201019287109,
 'solve': 0.0023548126220703123,
 'sse': 0.0017397403717041016}
preprocess: 0.002824
generate kernels: 0.227442
setup data: 0.011783
total time per iteration: 0.020267
total time: 0.101334
```

We have included the individual steps taken (effectively the time to run each `op2.par_loop(...)`), as a guide. The labels are defined as follows:

- error: calculating the vector of measurement errors
- estimate: calculating the vector of estimate distances between each pair of constrained poses
- jacobian: applying the derivative calculation at each constraint
- lhs: this is the left-hand-side of the least squares equation, which involves repeatedly updating the Hamiltonian matrix with $\mathbf{J}^T\Omega\mathbf{J}$ values
- per_iter_setup: this is the zero-ing out of some arrays at each iteration
- rhs: this is the right-hand-side of the least squares equation, involving the updating of $\mathbf{J}^T\Omega\mathbf{e}$ values
- solve: this step dispatches the lhs and rhs to be solved by the PetSc solver
- sse: sum-of-squares calculation to measure the error
- preprocessing and kernel generation are also shown

In this case, generating the kernels is relatively expensive, taking more than 2X the time the rest of the program uses. As we shall see with the next dataset, this cost is constant. The biggest computational cost which seems to arise is, as we expected, the calculation of the left-hand-side, the Hamiltonian. These are expensive because of the relatively large number of operations the kernel performs. Multiplying 3 3x3 matrices takes 54 multiplication and 54 addition operations, and since we iterate over the constraints in both rows and columns, we are updating the Hamiltonian over 4 such blocks each time.

We also show the results from running on a larger dataset, the Manhattan dataset with 3500 poses and 5598 constraints:

```
{'error': 0.0020958423614501954,
 'estimate': 0.0022215366363525389,
 'jacobian': 0.0029551506042480467,
 'lhs': 0.019606590270996094,
 'per_iter_setup': 0.0005340576171875,
 'rhs': 0.0020139694213867189,
 'solve': 0.0038761615753173826,
 'sse': 0.0018854141235351562}
preprocess: 0.006884
generate kernels: 0.232810
setup data: 0.031318
total time per iteration: 0.035189
total time: 0.175944
```

Lastly, we present the table of timings using `cProfile` for the Manhattan dataset:

RunningTime	Percentage	Symbol Name
0.129	38.51%	{select.select}
0.072	21.49%	{_instant_module_.wrap_lhs__}
0.039	11.64%	{imp.load_module}
0.028	8.36%	{pyop2.op_lib_core.build_sparsity}
0.021	6.27%	{posix.read}
0.008	2.39%	{posix.fork}
0.007	2.09%	pyop2/petsc_base.py:195(solve)
0.006	1.79%	{_instant_module_wrap_jacobian_block__}
0.005	1.49%	sympy/core/cache.py:78(wrapper)
0.005	1.49%	{isinstance}
0.004	1.19%	{built-in __new__ of type object}
0.004	1.19%	{numpy.core.multiarray.array}
0.003	0.90%	sympy/core/basic.py:1772(_preorder_traversal)
0.002	0.60%	balib.py:102(identity_map)
0.002	0.60%	subprocess.py:650(__init__)

The predominant expenditure of time comes from the `select.select` Python system call, which interfaces with the Unix `select` call to communicate data via MPI. Unfortunately, this masks a large part of the profiling as each `op2.par_loop` is in essence making MPI calls. `Instant` and PyOP2 calls comprise a majority of the rest of the execution costs, which is reassuring to know that slower Python calls are being avoided. The real power behind PyOP2 is the ability to run over a variety of architectures and to generate efficient code for those architectures at compile time. The examples we have illustrated use a sequential backend, and spend some time in the MPI communication phase. On larger datasets, and with other backends, this cost can be partially offset.

5.2. Comparisons With Existing Software

We will look at g2o and iSAM packages as representative samples of existing bundle adjustment solutions. We obtained the timings by running them on the same hardware (2GHz Intel i7) as we used in the previous section. We found that these packages do not make full use of the resources available. Specifically, g2o was found to use only 1-core of the 2-core machine (4 logical cores via hyperthreading). However, after examining the codebase, the authors did indirectly make use of vectorized calculations by bit-aligning the data structures where necessary. Similarly iSAM is also limited to running on a single core. However, both packages rely on either Eigen (a templated C++ matrix library) or Suite-Sparse (another popular C++ matrix library), which both, in turn, call highly tuned BLAS functions. The package runs on the Manhattan dataset in approximately 415ms (according to the console output). We show a representative breakdown of where the package spent execution time (using the Instruments profiler in Mac OS X):

RunningTime	Percentage	Symbol Name
219.0ms	45.4%	non-virtual thunk to isam::Slam::jacobian()
130.0ms	26.9%	isam::CholeskyImpl::factorize()
70.0ms	14.5%	cholmod_factorize
48.0ms	9.9%	cholmod_analyze_p2
5.0ms	1.0%	cholmod_solve
57.0ms	11.8%	isam::SparseSystem::operator=()
32.0ms	6.6%	isam::SparseMatrix:: SparseMatrix()
25.0ms	5.1%	non-virtual thunk to isam::Slam::weighted_errors()

The system spends about 50% of the time computing Jacobian blocks, and another 27% on Cholesky factorization and solving the system. Our implementation has ameliorated some of the cost of calculating the numerical Jacobian, by obtaining the analytical derivative using Sympy.

Next, we profile g2o. This package has been seen to run comparably fast compared to iSAM on the Intel dataset and 50% faster on the Manhattan dataset. The Instruments profiler recorded the following timing costs for this package:

RunningTime	Percentage	Symbol Name
65.0ms	26.2%	EdgeSE2::read(std::istream&)
17.0ms	6.8%	VertexSE2::read(std::istream&)
11.0ms	4.4%	readLine()
11.0ms	4.4%	ParameterContainer::read()
8.0ms	3.2%	OptimizableGraph::addEdge()
4.0ms	1.6%	istream::operator>>()
53.0ms	21.3%	BlockSolver<>::solve()
11.0ms	4.4%	BlockSolver<>::buildSystem()
3.0ms	1.2%	SparseOptimizer::computeActiveErrors()
3.0ms	1.2%	BlockSolver<>::buildStructure()
4.0ms	1.6%	EdgeSE2::computeError()
15.0ms	6.0%	loadStandardSolver()
11.0ms	4.4%	loadStandardTypes()
8.0ms	3.2%	OptimizableGraph:: OptimizableGraph()
5.0ms	2.0%	SparseOptimizer:: SparseOptimizer()
4.0ms	1.6%	SparseOptimizer::initializeOptimization(int)

Of the total 250ms spent in running the g2o package on the Manhattan dataset, it is interesting to note that almost 50% of the time is spent on data IO operations. This is a symptom of the design choice made by the authors to strongly couple the data loading and processing operations. Thus, the user is forced to write a `load()` function for a new vertex or edge type and implement the calculations needed to update measurement error function, Jacobian, and Hamiltonian calculation. We speculate that this design choice maybe have been made to garner a speed improvement, however.

5.3. Remarks

We have shown that PyOP2 can allow the structuring and solving of bundle adjustment problems without the sacrifice in performance that usually comes with Python. Our code uses Python as a staging language, Sympy for representing the mathematical formulation of measurement and error functions, and PyOP2 for elegant graph representation and fast execution. It is important to note that our implementation solves 2D bundle adjustment, while the packages we have compared to also perform 3D solutions, and iterative solvers. In practice, these features would not be difficult to implement with our design, but care would have to be taken to create the appropriate indirection between poses and landmarks, especially when building the Hamiltonian, since constraints connecting different vertex types may have different dimensions. In practice, this could mean a larger memory overhead to accommodate the larger pose type.

Our implementation was faster than iSAM and marginally faster than g2o. We consider this a mixed result. First, we note that transitioning to a larger dataset showed a relative speed improvement compared to other packages, suggesting that even larger problems will have even greater benefit. Second, we did not test the ceres-solver package due to data and time limitations. In the next section, we state closing remarks and suggest potential avenues for further research.

6. Conclusion

In this thesis, our goal was to investigate application of the PyOP2 framework in other contexts, specifically, in robot vision. The field of robot vision is broad and requires a wide variety of disciplines to work together. We focused on the subset of problems in robot vision which deal with optimizing measurements obtained from autonomous exploration of an environment. We presented a viable solution. Our implementation is far from complete. Realistic SLAM problems require considerations for camera calibrations. In the rest of this chapter, we state concluding remarks along with an outline for future work.

6.1. Results

We have presented our findings on an intuitive approach to building and solving bundle adjustment problems of arbitrary size. We borrowed techniques for addressing large parallel finite element calculations, as both problems, although disparate, can be represented by (hyper)graphs and use sparse linear algebra algorithms at their core. We showed that the PyOP2 framework developed by the Software Optimization group at Imperial College has broader applicability, reaching into scene recognition. The larger result here is the performance portability of bundle adjustment in being able to seamlessly generate highly performant code across current and future computer architectures without a re-write and without sacrificing performance. As we discuss in the next section, interesting graph problems are also being tackled by computational neuroscientists as well as data scientists analyzing neuronal and social graphs, respectively, where highly sparse systems with a large degree of local connectivity, also know as *small world networks*, also persist.

There are also downsides to this approach. For one, the implementation phase can be protracted due to limited debugging visibility. First, symbolic problem definition does not allow the user to inspect data values incrementally as they are evaluated lazily. Second, code generation implies that the implementer's code is only a representation of the code being run and not the code itself. As with all parallel programming tools, PyOP2 does not allow for easy introspection of program execution, and allows for limited modularity. Both of these points make unit tests difficult to construct. Lastly, PyOP2 was made with finite elements in mind and some operations, such as reading from sparse matrix structures to update other `Dat` variables is not supported. Further support for more fine grained data manipulation support would also be helpful in easing ease-of-use for other applications.

6.2. Future Work

We have touched on some computational bottlenecks in bundle adjustment problems and provided some methods of solving them. Using PyOP2 provides a framework in which kernels can be executed and matrix equations can be solved across various architectural backends. Next steps in this research should first expand our current implementation to more complicated scenarios such as 3D poses and landmarks, as well as camera calibrations. It would also be interesting to cover more general kernel generation tools, similar to what UFL already does for finite element problems. We believe it is important to continue to explore the sparsity structure of resulting Hessian matrices in bundle adjustment problems. Also, due to time constraints, we neglected to test the performance of one of the potentially more interesting bundle adjustment solvers, the Multicore Bundle Adjustment solver by Wu et. al. [27]. They cited significant speedup from running GPU-enabled code and a comparison to PyOP2 would give further insight into either the efficiency of the PyOP2 framework (if it's faster) or potential for future improvements (if it's slower). Most recently, engineers at Facebook have shown how to perform scalable computations on a trillion-edge graph [8]. A great future project would be to create a back-end implementation for Giraph in PyOP2. This could open the door for solving very large scale sparse system problems including bundle adjustment.

Lastly, other minimization techniques have been largely overlooked in this field, justifiably so due to the efficiency of modern matrix solvers. However, Duckett [9] provides an example of using *genetic algorithms* to search the high dimensional solution space for optimal parameters and solve SLAM, although benchmark comparisons with bundle adjustment were not done. It may also be worth exploring whether genetic algorithms are efficient to finding bundle adjustment solutions. Other techniques from machine learning, such as support vector machines and graphical models, should also be used instead of solely relying on direct linear algebra methods.

Bibliography

[1] Agarwal, S., and Mierle, K. *Ceres Solver: Tutorial & Reference.* Google Inc., 2011.

[2] Agarwal, S., and Mierle, K. CERES SOLVER : TUTORIAL.

[3] Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-farley, D., and Bengio, Y. Theano : A CPU and GPU Math Compiler in Python. In *Scipy 2010* (2010), no. Scipy, pp. 1–7.

[4] Bradski, G., and Kaehler, A. *Learning OpenCV.* 2008.

[5] Brown, D. C. A solution to the general problem of multiple station analytical stereotriangulation. *Technical Report RCA-MTP Data Reduction Technical Report*, 43 (1958), 114.

[6] Brown, D. C. The Bundle Adjustment - Progress and Prospects. *Interantional Archives of Photogrammetry 21*, 3 (1976), 1–33.

[7] Canny, J. A computational approach to edge detection. *IEEE transactions on pattern analysis and machine intelligence 8*, 6 (June 1986), 679–98.

[8] Ching, A. Scaling Apache Giraph to a trillion edges. *https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920* (2013).

[9] Duckett, T. A Genetic Algorithm for Simultaneous Localization and Mapping. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA'2003)* (Taipei, Taiwan, 2003), pp. 434–439.

[10] Golub, G. H., and Loan, C. F. V. *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)(3rd Edition)*, vol. 208-209. The Johns Hopkins University Press, 1996.

[11] Grisetti, G., Stachniss, C., and Burgard, W. Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters. *IEEE Transactions on Robotics 23*, 1 (Feb. 2007), 34–46.

[12] Kaess, M., Member, S., and Ranganathan, A. iSAM : Incremental Smoothing and Mapping. *IEEE Transactions on Robotics* (2008), 1–14.

[13] Kummerle, R., Grisetti, G., Strasdat, H., Konolige, K., and Burgard, W. g2o: A general framework for graph optimization. 2011.

[14] Logg, A., Wells, G. N., and Book, T. F. *Automated Solution of Differential Equations by the Finite Element Method*, vol. 84 of *Lecture Notes in Computational Science and Engineering.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[15] Lourakis, M. I. a., and Argyros, A. a. SBA: A Software Package for Generic Sparse Bundle Adjustment. *ACM Transactions on Mathematical Software 36*, 1 (Mar. 2009), 1–30.

[16] Lowe, D. G. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision 60*, 2 (Nov. 2004), 91–110.

[17] McKinney, W. Pandas: Python for Data Analysis, 2013.

[18] Mudalige, G. R., Giles, M. B., Bertolli, C., and Kelly, P. H. J. OP2 : An Active Library Framework for Solving Unstructured Mesh-based Applications on Multi-Core and Many-Core Architectures. In *Innovative Parallel Computing* (2012), no. 2.

[19] Peter, M. *Stochastic Models , Estimation , and Control*, vol. 1. Academic Press Inc. (London) LTD., 1979.

[20] Rathgeber, F., Markall, G. R., Mitchell, L., Loriant, N., Ham, D. A., Carlo Bertolli, and Kelly, P. H. J. PyOP2: A

High-Level Framework for Performance-Portable Simulations on Unstructured Meshes. In *SC Companion: High Performance Computing, Networking Storage and Analysis* (2012), pp. 1116–1123.

[21] Salas-Moreno, R., Newcombe, R., Strasdat, H., Kelly, P., and Davison, A. SLAM++: Simultaneous Localisation and Mapping at the Level of Objects.

[22] Spielman, Daniel A; Teng, S.-H. Solving Sparse, Symmetric, Diagonally-Dominant Linear Systems in Time 0(m1.31). In *FOCS '03 Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science* (2003), IEEE Computer Society, p. 416.

[23] Strasdat, H. *Local Accuracy and Global Consistency for Efficient Visual SLAM.* Ph.d., Imperial College, London, 2012.

[24] Team, S. D. SymPy: Python library for symbolic mathematics, 2013.

[25] The HDF Group. Hierarchical data format version 5, 2013.

[26] Thrun, S. Particle Filters in Robotics. In *Proceedings of Uncertainty in AI* (2002).

[27] Wu, C., Agarwal, S., Curless, B., and Seitz, S. M. Multicore bundle adjustment. *CVPR 2011*, x (June 2011), 3057–3064.

A. Some Code

In this section, I provide some of the code used in the examples in this thesis. Some pieces of code rely on other software libraries. They should be obvious from the first few lines of each listing.

Listing A.1 is an example of setting up a simple bundle adjustment problem with 2 camera calibrations k_1, k_2, 3 poses $1, 2, 3$, and 8 landmarks a, b, c, d, e, f, g, h. We build the matrix structure naively only to demonstrate the sparsity nature of the resulting Jacobian and Hamiltonian matrices.

Listing A.1: Building Toy Bundle Adjustment Matrices

```python
import numpy as np
import scipy
import scipy.sparse as sp
import scipy.sparse.linalg as la
from collections import OrderedDict
from operator import itemgetter

data = OrderedDict()

data['a'] = [1]
data['b'] = [1, 2]
data['c'] = [1, 3]
data['d'] = [1, 3]
data['e'] = [1, 2]
data['f'] = [2, 3]
data['g'] = [2, 3]
data['h'] = [2]

cameras = OrderedDict()

cameras['k2'] = {'features': ['a', 'b', 'c', 'd'], 'images':
    [1, 3]}
cameras['k1'] = {'features': ['e', 'f', 'g', 'h'], 'images':
    [2]}
```

```

def unique_params(d):
    u = set(item for sublist in d.itervalues() for item in
        sublist)
    return sorted(u)

def offsets(d):
    off = {}
    key_iter = d.iterkeys()
    start_key = key_iter.next()
    off = {start_key: 0}
    cumulative = len(d[start_key])
    for k in key_iter:
        off[k] = cumulative
        cumulative += len(d[k])
    return off, cumulative

def showJacobian():
    N = unique_params(data)
    M = len(data.keys())
    off, C = offsets(data)
    key_idx = {k: v for (k, v) in zip(data.keys(), range(M))}
    val_idx = range(M, M + len(N), 1)

    Poses = scipy.sparse.dok_matrix((C, M), dtype=float)
    Params = scipy.sparse.dok_matrix((C, len(N)), dtype=float)
    for k, v in data.iteritems():
        row_mask = range(off[k], off[k] + len(v))
        col_idx = data.keys().index(k)
        Poses[row_mask, col_idx] = np.random.normal()+5
        for i in range(len(v)):
            Params[off[k]+i, N.index(v[i])] = np.random.normal
                ()+5

    Cams = scipy.sparse.dok_matrix((C, len(cameras)), dtype=
        float)
    for k, v in data.iteritems():
        for i, cam in enumerate(cameras.values()):
            indexes = list(set(cam['images']) & set(v))
            print [v.index(x) + off[k] for x in indexes],
                indexes
            Cams[[v.index(x) + off[k] for x in indexes], i] =
                np.random.normal()+5

    J = sp.hstack([sp.hstack([Poses, Params]), Cams])

    fig = figure()
    ax1 = fig.add_subplot(111)
    ax1.spy(J.todense(), markersize=5)
    plt.show()

def showHamiltonian()
    Hcsr = Jcsr.transpose() * Jcsr
    fig = figure()
    ax1 = fig.add_subplot(111)
    ax1.spy(Hcsr.todense(), markersize=5)
    plt.show()
```

Listing A.1 shows an example of how we load data from delimited files (as is typical of input data) and process the information for a 2D poses-only bundle adjustment problem. Using the `pandas.DataFrame` data structure to store edges and vertices separately provides an efficient way of handling the information we process: for vertices, the parameters x, y, θ and for edges, the parameters $dx, dy, d\theta$ as well as the structure of the inverse measurement covariance matrix Ω, which has 6 degrees of freedom:

$$\begin{bmatrix} \Omega_{00} & \Omega_{01} & \Omega_{02} \\ \Omega_{01} & \Omega_{11} & \Omega_{12} \\ \Omega_{02} & \Omega_{12} & \Omega_{22} \end{bmatrix}$$

Listing A.2: Loading g2o Data Into pandas.DataFrame

```
import numpy as np
import pandas as pd
import csv

EDGE_COLS = ['index', 'x_coord', 'y_coord', 'theta_coord', '
    omega_0_0', 'omega_0_1', 'omega_0_2', 'omega_1_1', '
    omega_1_2', 'omega_2_2']
VERTEX_COLS = ['index', 'x', 'y', 'theta']

def drop_empty(row):
    return [item for item in row if item != '']

def loadFromFile(filepath):
        vertices = []
        edges = []
        with open(filepath, 'r') as fd:
                reader = csv.reader(fd, delimiter='␣')
                for row in reader:
                        if 'VERTEX' in row[0].upper():
                                vertices.append(np.float64( drop_empty(row[1:]) ))
                        elif 'EDGE' in row[0].upper():
                                edges.append(np.float64( drop_empty(row[1:]) ))

        edges_dataframe = pd.DataFrame( edges, columns = EDGE_COLS )
        vertices_dataframe = pd.DataFrame( vertices, columns = VERTEX_COLS)

        return edges_dataframe, vertices_dataframe
```

Listing A.3: Retrieving g2o Data

```
import pandas as pd
def quickLoad(filepath, delim='␣'):
    max_num_cols = 30
    df = pd.read_csv(filepath,
                     delimiter=delim,
                     names=[str(x) for x in range(max_num_cols)]
                    ).dropna(axis=1, how='all')

    edges_mask = (~pd.isnull(df)).all(axis=1)
    edges = df.ix[edges_mask]
    vertices = df.ix[~edges_mask].dropna(axis=1)

    vertices.rename(columns={'0': 'label', '1': 'index'}, inplace=True)
    vertices.rename(columns={ str(i) : 'dim%d'%(i-1) for
                              i in range(2, len(vertices.columns)) }, inplace=True)

    edges.rename(columns={'0' : 'label', '1': 'from_v', '2': 'to_v' }, inplace=True)
    edges.rename(columns={ str(i) : 'meas%d'%(i-1) for
                                i in range(3, len(edges.columns)) },
                                     inplace=True)

    return vertices, edges
```

Here, we present a code listing which demonstrates the power of Sympy to symbolically differentiate mathematical expressions. The resultant objects are then used in listing A.5 to produce kernel code passed to PyOP2 for parallel execution.

Listing A.4: Automatic Differentiation with Sympy

```
from sympy import *

def e_x(p_x, p_y, q_x, q_y, p_theta, q_theta):
        """ symbolic x-component of error in SE2 """
        return (q_x - p_x)*cos(p_theta) + (q_y - p_y) * sin(
            p_theta)

def e_y(p_x, p_y, q_x, q_y, p_theta, q_theta):
        """ symbolic y-component of error in SE2 """
        return -(q_x - p_x)*sin(p_theta) + (q_y - p_y) * cos(
            p_theta)

def e_theta(p_x, p_y, q_x, q_y, p_theta, q_theta):
        """ symbolic theta-component of error in SE2 note that
          this should only be used for differentiation as Sympy
          cannot differentiate the Mod() function
        """
        return p_theta - q_theta
```

Listing A.5: C Code Generation Using Sympy

```
from sympy.utilities.codegen import codegen

jacobian_code = {}
for err_dim in ['e_x', 'e_y', 'e_theta']:
    for dim in ['p_x', 'p_y', 'p_theta', 'q_x', 'q_y', 'q_theta
        ']:
        deriv = 'd\%s_d\%s'\%(err_dim, dim)
        error_func = eval(err_dim)
        generated_code = codegen((deriv, diff(error_func(p_x,
            p_y, p_theta, q_x, q_y, q_theta), eval(dim))), 'C',
             deriv)[0][1]
        start = jacobian_code('return')+7
        end = jacobian_code(';')
        jacobian_code[deriv] = generated_code[start:end]
```

For generating the right-hand-side and left-hand-side of the sparse least squares problem we provide listings A.6 and A.7.

Listing A.6: C Code Generation Using Sympy

```
def generateRHSCode(name):
    """ pass in a name and get back a PyOP2 kernel which computes the vector:
        J-transpose * Omega * e
        This is general enough to allow situations where we have hypergraphs,
        i.e. the number of 'vertices' is not equal 2 per 'edge'
    """

    # code for omega * e; do this first because it is independend of poses
    omega_ij = [(i, j) for i in xrange(CONSTRAINT_DIM) for j in xrange(CONSTRAINT_DIM) if j >= i]
    omega_err = []
    for i in xrange(CONSTRAINT_DIM):
        row = []
        for j in xrange(CONSTRAINT_DIM):
            idx = omega_ij.index((j,i)) if i > j else omega_ij.index((i,j))
            row.append( 'omega[\%d]␣*␣err[\%d]' \% (idx, j) )
        omega_err.append('omega_times_err[\%d]␣=␣\%s;' \% (i, '␣+␣'.join(row)))

    code = []
    for i in xrange(CONSTRAINT_DIM):
        row = []
        for j in xrange(POSES_DIM):
            row.append( 'J[\%d*i␣+␣\%d]␣*␣omega_times_err[\%d]' \% (CONSTRAINT_DIM*POSES_DIM, j*CONSTRAINT_DIM + i, j))
        code.append( 'rhs_vector[i][\%d]␣-=␣\%s;' \% (i, '␣+␣'.join(row)) )
    rhs_code = """
    void \%(name)s(double err[\%(c_dim)d], double J[\%(j_dim)d], double omega[\%(o_dim)d], double * rhs_vector[\%(p_dim)d])
    {
        double omega_times_err[\%(c_dim)d];
        \%(omega_err)s
        int i = 0;
        for ( ; i < \%(poses_per_constraint)d; ++i ) {
            \%(code)s
        }
    }
    """ \% { 'name' : name, 'poses_per_constraint' : POSES_PER_CONSTRAINT, 'j_dim' : CONSTRAINT_DIM * POSES_DIM * POSES_PER_CONSTRAINT, 'j_subblock_dim' : CONSTRAINT_DIM * POSES_DIM, 'o_dim' : OMEGA_DOF, 'p_dim' : POSES_DIM, 'c_dim' : CONSTRAINT_DIM, 'omega_err' : '\n'.join( omega_err ), 'code' : '\n'.join(code) }

    if _PRINT_CODE:
        print rhs_code
    return op2.Kernel( rhs_code, name)
```

Listing A.7: Generating the Hamiltonian Kernel

```
def generateHamiltonianCode(name, lm_param):
    """ pass in a name string and a float representing the Levenberg-Marquardt parameter
        for H + lm_param * D
        the Hamiltonian is a NUM_POSES*POSES_DIM by NUM_POSES*POSES_DIM square matrix but
        is calculated by iterating over constraints, since that is how the jacobian blocks
        are obtained;
    """
    JBLOCK_SIZE = POSES_DIM * CONSTRAINT_DIM

    posdef_partial = partial( normal_to_posdef, dim=CONSTRAINT_DIM )

    block_code = ['j_t_omega[\%d]␣=␣\%s;' \% (i*CONSTRAINT_DIM + j, dotproduct('J', i, 1, 'omega', j, CONSTRAINT_DIM, CONSTRAINT_DIM, idxFunc2=posdef_partial, prefix1='\%d*j+'\%(JBLOCK_SIZE))) for i in xrange(POSES_DIM) for j in xrange(POSES_DIM)]

    update = ['H[\%d][\%d]␣+=␣\%s;' \% (i, j, dotproduct('j_t_omega', i*CONSTRAINT_DIM, 1, 'J', j, CONSTRAINT_DIM
        , CONSTRAINT_DIM, prefix2='\%d*i+'\%(JBLOCK_SIZE))) for
         i in xrange(POSES_DIM) for j in xrange(POSES_DIM)]

    hamiltonian_code = """
    void \%(name)s(double J[\%(j_dim)d], double omega[\%(o_dim)
        d], double H[\%(p_dim)d][\%(p_dim)d], int i, int j)
    {
        double j_t_omega[\%(j_t_omega_dim)d];
        \%(jacT_times_omega)s
        \%(update)s
    }
    """ \% { 'name' : name,'j_t_omega_dim' : CONSTRAINT_DIM *
        POSES_DIM, 'p_dim' : POSES_DIM, 'c_dim' :
        CONSTRAINT_DIM, 'o_dim' : OMEGA_DOF, '
        poses_per_constraint' : POSES_PER_CONSTRAINT, '
        jacT_times_omega' : '\n'.join( block_code ), 'update' :
         '\n'.join( update ), 'j_dim' : JBLOCK_SIZE *
        POSES_PER_CONSTRAINT }

    lm_kernel = generateHamiltonianDiagonalCode(name + '_lm',
        lm_param)
    if _PRINT_CODE:
        print hamiltonian_code
    return op2.Kernel(hamiltonian_code, name), lm_kernel
```